МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

О.С. Зеленський

В.С. Лисенко

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

на C++

Навчальний посібник

Кривий Ріг

2023

Навчальний посібник "Об’єктно-орієнтоване програмування на С++" призначений для поглибленого вивчення студентами об’єктно-орієнтованого програмування на мові С++. У С++ об’єкти реалізуються за допомогою використання класів, що дозволяють програмісту визначати нові типи даних і, таким чином, є потужною та зручною частиною С++. Навчальний посібник адресований студентам, слухачам магістратури, аспірантам, викладачам. Може бути використаний як самовчитель.

Автори: Зеленський О.С., Лисенко В.С. – Кривий Ріг: Державний університет економіки і технологій, 2023.-215 с.

Рецензенти:

А.І. Купін, д.т.н, професор, завідувач кафедри комп’ютерних систем та мереж, Криворізький національний університет.

І.О. Музика, к.т.н, доцент кафедри комп’ютерних систем та мереж, декан факультету інформаційних технологій Криворізький національний університет.

В.Б. Хоцкіна, к.т.н., доцент кафедри інформатики і прикладного програмного забезпечення, Державний університет економіки і технологій.

Рекомендовано Вченою радою Державного університету економіки і технологій

Протокол № 9 від 23.02.2023 р.

ЗМІСТ

ВСТУП

Навчальний посібник "Об’єктно-орієнтоване програмування на С++" призначений для поглибленого вивчення студентами об’єктно-орієнтованого програмування на мові С++. У С++ об’єкти реалізуються за допомогою використання класів, що дозволяють програмісту визначати нові типи даних і, таким чином, є потужною та зручною частиною С++.

На конкретних прикладах розглянуті алгоритми та програми по основним розділам курсу. У кожному розділі наводиться реалізація прикладів та виведення відповідних результатів. Значна увага приділяється роботі з класами (інкапсуляція, спадкування, поліморфізм, перевантаження операторів тощо), динамічним структурам даних, контейнерам, алгоритмам, обробці виключних ситуацій.

У окремих розділах приведені типові контейнери, а також алгоритми для їх обробки.

Зміст навчального посібника охоплює наступні розділи:

1. Інкапсуляція та приховування інформації.
2. Класи і підкласи.
3. Спадкування класів.
4. Поліморфізм.
5. Перевантаження операторів.
6. Обробка виключних ситуацій.
7. Рядки.
8. Шаблони класів.
9. Модульні програми.
10. Контейнерні класи.
11. Алгоритми.

Приклади, приведені в посібнику розроблені для режиму Console (тип проекту Win32 Console Application) з використанням сучасного стандарту С++ (ISO/IEC 14882). Даний стандарт застосовується у мовах програмування Visual C++, починаючи з 1998 року. У Visual C++ 2002-2022 років підтримується тільки сучасний стандарт і лише в Visual C++ 1998 року підтримується попередній та сучасний стандарти програмування.

Навчальний посібник адресований студентам, слухачам магістратури, аспірантам, викладачам. Може бути використаний як самовчитель.

РОЗДІЛ 1. ІНКАПСУЛЯЦІЯ ТА ПРИХОВУВАННЯ ІНФОРМАЦІЇ

1.1 Визначення та використання класів

Об’єктно-орієнтоване програмування (ООП) дозволяє програмісту об’єднувати дані та їхній обробний програмний код в один блок.

У C++ об’єкти реалізуються за допомогою використання класів, що дозволяють програмісту визначати нові типи даних і, таким чином, є потужною і зручною частиною C++ . Класи відрізняють C++ від C.

Займаючись процедурним програмуванням ви створюєте нові функції і спрямовуєте процес обчислення через їхню послідовність. Тут ви розділяєте дані і функції, що їх опрацьовують.

На відміну від такого підходу, ООП змушує вас мислити в термінах об’єктів і взаємодії між ними. Об’єкт містить деякі дані і методи (процедури) їх опрацювання.

Навіщо використовувати ООП?

Відповідь на це питання не проста. ООП дозволяє більш точно моделювати проблему, для рішення якої пишеться програма.

Крім того, об’єкти незалежні, їх легко відлагоджувати, змінювати і використовувати. І, нарешті, якщо ваші об’єкти добре сконструйовані, ви можете використовувати повторно більш значну частину програми, чим у процедурному програмуванні.

Об’єкт C++ – абстрактний опис якоїсь сутності. Сутністю може бути запис про студента або екранне вікно. Об’єкт містить у собі всі дані, необхідні для того щоб описати сутність і функції, що маніпулюють цими даними. Ідеальний об’єкт знає про себе все, включаючи те, як ввести, вивести й опрацьовувати дані, що зберігаються в ньому.

У C++ об’єкти вводяться через поняття класів. Клас є абстрактним типом даних, визначуваним користувачем, і є моделлю реального об'єкту у вигляді даних і функцій для роботи з ними. Дані класу називаються *полями*, а функції класу – *методами.* У літературі зустрічаються відповідно терміни "дані-члени" і "функції-члени". Надалі ми використовуватимемо обидві ці термінології. Поля і методи називаються *елементами класу.* Опис класу виглядає таким чином:

```
class <ім’я>{
[ private: ]
<опис прихованих елементів>
public:
<опис доступних елементів>
} ; // Опис закінчується крапкою з комою
```

Специфікатори доступу **private** та **public** управляють видимістю елементів класу. Елементи, описані після службового слова **private**, видимі тільки усередині класу. Цей вид доступу прийнятий в класі за умовчанням. Інтерфейс класу описується після специфікатора **public**. Дія будь-якого специфікатора розповсюджується до наступного специфікатора або до кінця класу. Можна задавати декілька секцій **private** та **public**, порядок їх проходження значення не має.

1.2. Поля і методи класів

1.2.1 Поля і методи класів

Поля класу:

– можуть мати будь-який тип, окрім типу цього ж класу (але можуть бути вказівками або посиланнями на цей клас);

– можуть бути описані з модифікатором const, при цьому вони ініціалізуються тільки один раз (за допомогою конструктора) і не можуть змінюватися;

– можуть бути описані з модифікатором static (пояснення приводиться нижче).

Як приклад створимо клас, моделюючий персонаж комп'ютерної гри. Для цього потрібно задати його властивості (наприклад, кількість щупалець, силу або наявність гранатомета) і поведінку.

```
class monstr
{
 int health, ammo;
 public:
 monstr(int he = 100, int am = 10)
       {health = he; ammo = am;}
 void draw (int x, int y, int scale, int position);
 int get_health(){return health;}
 int get_ammo(){return ammo;}
};
```

У цьому класі два приховані поля – **health** та **ammo**, набути значень яких ззовні можна за допомогою методів **get_health()** і **get_ammo()**. Всі методи класу мають безпосередній доступ до його прихованих полів. У приведеному класі міститься три визначення методів і одне оголошення (метод **draw**). Якщо тіло методу визначене усередині класу, він є *вбудованим* (**inline**). Як правило, вбудованими роблять короткі методи. Вбудовуємі функції можуть збільшити швидкість виконання програми, але непродумане їх використання може призвести до збільшення модуля до гігантських розмірів.

Якщо усередині класу записано тільки оголошення (заголовок) методу, сам

метод має бути визначений у іншому місці програми за допомогою операції доступу до зони видимості (::):

```
void monstr::draw(int x,int y,int scale,int position)
{
 /* тіло методу */
}
```

Метод можна визначити як вбудований і поза класом за допомогою директиви **inline**. У кожному класі є хоч би один метод, ім'я якого збігається з ім'ям класу. Він називається конструктором і викликається автоматично при створенні об'єкту класу. Конструктор призначений для ініціалізації об'єкту. Автоматичний виклик конструктора дозволяє уникнути помилок, зв'язаних з використанням неініціалізованих змінних.

1.2.2 Опис об’єктів

Конкретні змінні типу «клас» називаються *екземплярами класу, або об'єктами.* Час життя і видимість об'єктів залежить від виду і місця їх опису та підкоряється загальним правилам C++:

monstr Vasia; // *Об'єкт класу monstr з параметрами за умовчанням*
monstr Super(200, 300); // *Об'єкт з явною ініціалізацією*
monstr stado[100]; // *Масив об'єктів з параметрами за умовчанням*
monstr* beavis = new monstr (10); // *Динамічний об'єкт*
//(другий параметр задається за умовчанням)
monstr &butthead = Vasia; // *Посилання на об'єкт*

При створенні кожного об'єкту виділяється пам'ять, достатня для зберігання всіх його полів, і автоматично викликається конструктор, що виконує їх ініціалізацію. При виході об'єкту з області дії він знищується, при цьому автоматично викликається деструктор (деструктори описані далі).

Доступ до елементів об'єкту аналогічний доступу до полів структури. Для цього використовуються операція "." (крапка) при зверненні до елементу через ім'я об'єкту або операція "**->**" при зверненні через вказівку, наприклад:

```
int n = Vasia.get_ammo();
stado[5].draw(1,2,3,4);
cout << beavis->get_health();
```

Звернутися таким чином можна тільки до елементів із специфікатором **public**. Отримати або змінити значення елементів із специфікатором **private** мож-

на тільки через звернення до відповідних методів. У нашому прикладі метод **get_health()**, що має специфікатор **public**, повертає значення поля **health** в секції **private**.

Можна створити константний об'єкт, значення полів якого змінювати забороняється. До нього повинні застосовуватися тільки константні методи:

```
class monstr
{
 ...
 int get_health() const {return health;}
};

const monstr A(0,0); // Константний об'єкт
cout << A.get_health();
```

Константний метод:

– оголошується з ключовим словом **const** після списку параметрів;
– не може змінювати значення полів класу;
– може викликати тільки константні методи;
– може викликатися для будь-яких (не тільки константних) об'єктів.

Рекомендується описувати як константні ті методи, які призначені для набуття значень полів.

1.2.3 Вказівка this

Кожен об'єкт містить свій екземпляр полів класу. Методи класу знаходяться в пам'яті в єдиному екземплярі і використовуються всіма об'єктами спільно, тому необхідно забезпечити роботу методів з полями саме того об'єкту, для якого вони були викликані. Це забезпечується передачею у функцію прихованого параметра **this**, в якому зберігається константна вказівка на об'єкт, який викликав функцію. Вказівка **this** неявно використовується усередині методу для посилань на елементи об'єкту. У явному вигляді ця вказівка застосовується в основному для повернення з методу вказівки (**return this**;) або посилання (**return *this**;) на об'єкт. Для ілюстрації використання вказівки **this** додамо в приведений вище клас **monstr** новий метод, що повертає посилання на найбільшого (поле **health**) з двох монстрів, один з яких викликає метод, а інший передається йому як параметр (метод потрібно помістити в секцію **public** опису класу):

```
monstr & the_best(monstr &M)
{
if( health > M.health) return *this;
return M;
```

```
}
```

... monstr Vasia(50), Super(200);
// *Новий об'єкт* Best *ініціалізувався значеннями полів* Super:
monstr Best = Vasia.the_best(Super);

Вказівку **this** можна також застосовувати для ідентифікації поля класу у тому випадку, коли його ім'я збігається з ім'ям формального параметра методу. Інший спосіб ідентифікації поля використовує операцію доступу до зони видимості:

```
void cure(int health, int ammo)
{
 this->health += health; // Використання this
 monstr::ammo += ammo; // Використання операції ::
}
```

1.3 Інкапсуляція та приховування інформації

1.3.1. Приховані дані

Мета об’єктно-орієнтованого програмування – відокремити реалізацію об’єкта від його використання. Це можна зробити за допомогою приховування даних.

Приховування даних перешкоджає прямому доступу до даних-членів класу з боку програм, що використовують цей об’єкт. Використання механізму приховування даних забезпечує такі переваги:

– користувачі класу ізолюються від реального представлення даних;
– розроблювачі класу можуть змінювати представлення і джерело надходження даних, не змінюючи структури програми, в котрій цей клас використовується.

Для одержання цих переваг необхідно володіти поняттям видимості членів класу. Видимість дає можливість визначати, до яких членів класу можна одержати доступ. Якщо всі дані-члени класу приховані, як це робиться звичайно, ви повинні визначити ряд функцій-членів, використовуваних для доступу, зміни й опрацювання даних-членів. Ці функції-члени, доступні для користувача, називаються загальним інтерфейсом класу.

Щоб створити клас, використовуючи приховані дані, треба спочатку зрозуміти, як робити члени класу прихованими або видимими поза класом.

Головна мета класу – сховати якнайбільше інформації. Це накладає деякі обмеження на використання даних або коду усередині класу.

Існує три види використання класу:

– самий клас;

– звичайні користувачі;
– похідні класи.

Кожне оголошення усередині класу неявно визначає привілеї доступу, у залежності від того, у якій секції воно з'являється. Існує три привілеї доступу (специфікатора доступу):

1. private;
2. public;
3. protected.

Перші два специфікатори були розглянуті раніше.
Кожна секція в класі починається з приведених вище ключових слів.
Приклад:

```
class Apt
{
 int aa;
 void s();           // приватний по умовчанню
 public:
 double q; long n;   // загальнодоступні
 protected:
 int mm; float r(int);    // захищені
 private:
 char t; int ll();        // приватні
};
```

Будь-яке оголошення, що з'являється до ключового слова керування доступом, вважається приватним по умовчанню.

1.3.2. Загальнодоступні і приватні члени класу

Приватні члени класу мають найбільш обмежений доступ. Тільки сам клас або класи, оголошені як дружні (**friend**) мають доступ до приватних членів. Похідні об'єкти не мають доступу до членів батьківського класу. Дружні класи і похідні об'єкти розглядаються нижче.

Приклад:

```
class Rut
{
 long value;
 void f();
```

```
};

void main()
{
 Rut ob;              //створення об'єкта Rut
 long a = ob. value; //помилка доступу
 ob. f();            //помилка доступу
}
```

Специфікатор **public** забезпечує доступ до полів і методів.
Приклад:

```
class M
{
 public:
 int variable;
 void f();
};
void M::f(){}              //нічого не робить

void main()
{
 M c;
 int i = c. variable;     //все вірно
 c. f();                  //все вірно
}
```

Приклад: У цьому прикладі описується клас, який описує сферу

```
#include <iostream>
using namespace std;

#define M_PI        3.14159265358979323846

class Sf
{
 float r;
 public:
 float x, y, z;
 Sf(float xc, float yc, float zc, float rk)
 {
  x = xc; y = yc; z = zc; r = rk;
```

```
 }
 ~Sf(){}

 float Get_r();
 float cub()
 {
  return (r*r*r*4/3*M_PI);
 }
 float kv();
};

inline float Sf::kv()
{
 return(4*M_PI*r*r);
}

float Sf::Get_r()
{
 return r;
}

void main()
{
 Sf S(1.0, 2.0, 3.0, 4.0);
 cout << S. kv() << "\n" << S.Get_r();
}

//Виведення:
//201.062
//4.0
```

В даному прикладі обчислюється об'єм і площа кулі з використанням відповідних вбудованих функцій **kv()** і **cub()**. Доступ до елементів об'єкту можливий, якщо вони мають специфікатор **public**. Отримати або змінити значення елементів із специфікатором **private** можна тільки через звернення до відповідних методів. У нашому випадку функція **Get_r()** повертає значення змінної **r** (радіусу кулі), що має специфікатор доступу **private**. Функція **Sf** – функція конструктор має таке ж ім’я як і клас, використовується для одночасного створення й ініціалізації змінних класу. Ця функція не повертає значень.

Деструктор також має ім’я класу, але містить перед ім’ям тильду (~). Функція-деструктор не приймає аргументів і не повертає значень. У нашому прикладі функція **~Sf ()** нічого не робить, що характерно для більшості деструкторів. Їхнє

основне призначення – звільнення пам’яті, що була виділена для об’єкта класу. Деструктор може виконувати й інші дії, наприклад, такі як виведення останніх значень даних-членів класу. Це зручно при налагодженні. Якщо функція-деструктор не робить ніяких дій, то немає потреби її створювати. Як тільки змінна типу клас перестає існувати, виділена для об’єкта пам’ять повертається системі. Аналогічно, якщо конструктор не виконує ніяких дій, то немає потреби його створювати.

1.3.3. Захищені члени класу

Коли ви визначаєте клас, що у свою чергу використовується як базовий для інших класів, можете зробити його члени доступними тільки для функцій похідних класів за допомогою слова **protected**.

Розглянемо ієрархію об’єктів, приведених на рис. 1.

Ієрархія, подана на рисунку1 може бути виражена таким кодом:

```
class A
{
 protected:
 int val;
};

class B: public A
{
 public:
 void FB();
};

class C: public B
{
 public:
 void FC();
};
```

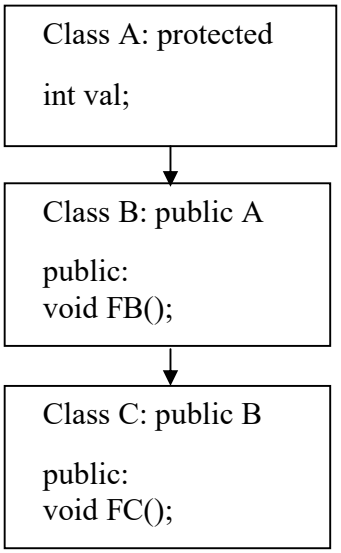

Рис. 1. Ієрархія класів.

Властивість захищеності поширюється по дереву доти, поки похідна класу повідомляє свій базовий клас загальнодоступним. При цьому будь-яка функція-член у класах **C** або **B** може одержати доступ до захищеного члена даних **val** базового класу **A**. Наприклад, цілком допустимий такий код:

```
void B::FB()
{
 val = 0;
}

void C::FC()
{
 val = 0;
}
```

Якщо функція-член приймає вказівку або посилання на інший об’єкт, то правила вже інші. Розглянемо модифікований клас **C**:

```
class C: public B
{
 public:
 void FC(A&);
};

void C::FC(A&a)
{
 val = 0;         //доступ дозволений
 a.val = 0;       /*тут використовується посилання A&:
```

```
порушення прав доступу, може бути використаний
загальнодоступний інтерфейс класу A.*/
}
```

Доступ до **val** через посилання на об'єкт класу **A** не дозволений, незважаючи на те, що функція розташована усередині класу, породженого від **A**, що має доступ до захищених членів класу.

Докладно про захищені члени класу буде розглянуто в розділі 3 "Спадкування класів". Можна створювати об'єкти класу або вказівку на об'єкти класу в основній або динамічній пам'яті.

Приклад:

```
class Co
{
 public:
 int GetVal() {return 1;}
};

void main()
{
Co ct;
Co*pt;           //вказівка на об'єкт в основній пам'яті
Co *ptt=new Co; //вказівка на об'єкт у динамічній пам'яті
pt = ptt;
GetVal(); // не в області видимості: невідома функція
ct.GetVal();  // нормально
pt->GetVal();     // нормально
ptt->GetVal();    // нормально
delete ptt; // вивільнення динамічної пам'яті
}
```

1.3.4. Організація загального інтерфейсу

Використання загальної (загальнодоступної) і приватної видимості членів класу дає можливість сховати члени класу від функцій, що не входять у цей клас. Загальні члени класу визначають то, як функції, що не є членами цього класу, можуть використовувати даний клас. Ці члени класу називаються загальним інтерфейсом класу.

Добре розроблений інтерфейс звичайно містить у собі функції-члени, що виконують такі дії:

- ініціалізують змінну типу даного класу. Конструктори завжди мають тип **public**;
- звільняють пам'ять, яка використовується змінною класу. Деструктори зав-

жди мають тип **public**;

- проводять початкову установку значень приватних змінних-членів класу;
- обчислюють значення приватних змінних-членів класу;
- виконують реальну роботу об’єкта.

Приклад: У цьому прикладі показана версія програми **Sf** із розширеним загальним інтерфейсом. У програмі реалізується повний набір загальних функцій-членів, що нададуть функції **main()** доступ до приватних даних-членів.

```
#include <iostream>
using namespace std;

#define M_PI        3.14159265358979323846

class Sf
{
 private: //це не обов’язково - по умовчанню private

 float r, x, y, z;
 float cub() {return (r*r*r);}
 float kv() {return (r*r);}
 public:

 Sf(float xk, float yk, float zk, float rk)
 { x = xk; y = yk; z = zk; r = rk;}
 ~Sf(){ cout << "\n Кінець \n";}

 float ob(){return (cub()*4*M_PI/3);}
 float pl(){return (kv()*4*M_PI);}

 void Setx(float newx){x = newx;}
 void Sety(float newy){y = newy;}
 void Setz(float newz){z = newz;}
 void Setr(float newr){r = newr;}

 float Getx() {return x;}
 float Gety() {return y;}
 float Getz() {return z;}
 float Getr() {return r;}
};

void main()
{
 Sf s(1.0, 2.0, 3.0, 4.0);
 cout << "x = " << s.Getx() << "\n";
 cout << "Площа = " << s.pl() << "\n";
```

```
 s.Setx(5.0); s.Setr(8.0);
 cout << "x = " << s.Getx() <<"\n";
 cout << "Площа = " << s. pl() << "\n";
}

Результат: x = 1
Площа = 201,062
x = 5
Площа = 804. 248
```

1.4 Конструктори і деструктори

Як вже наголошувалося конструктор призначений для ініціалізації об'єкту і викликається автоматично при його створенні. Основні властивості конструкторів:

1. Конструктор *не повертає значення,* навіть типу **void**. Не можна отримати вказівку на конструктор.
2. Клас може мати *декілька конструкторів* з різними параметрами для різних видів ініціалізації (при цьому використовується механізм перевантаження).
3. Конструктор, що викликається без параметрів, називається *конструктором за умовчанням.*
4. *Параметри конструктора* можуть мати будь-який тип, окрім цього ж класу. Можна задавати значення параметрів за умовчанням. Їх може містити тільки один з конструкторів.
5. Якщо програміст не вказав жодного конструктора, компілятор створює його *автоматично.* Такий конструктор викликає конструктори за умовчанням для полів класу і конструктори за умовчанням базових класів (див. розділ "Спадкування"). У разі, коли клас містить константи або посилання, при спробі створення об'єкту класу буде виведена помилка, оскільки їх необхідно ініціалізувати конкретними значеннями, а конструктор за умовчанням цього робити не вміє.
6. *Конструктори не успадковуються.*
7. Конструктори не можна описувати з модифікаторами **const**, **virtual** і **static**.
8. Конструктори глобальних об'єктів викликаються до виклику функції **main**. Локальні об'єкти створюються, як тільки стає активною зона їх дії. Конструктор запускається і при створенні тимчасового об'єкту (наприклад, при передачі об'єкту з функції).
9. Конструктор викликається, якщо в програмі зустрілася яка-небудь з синтаксичних конструкцій:

ім’я_класу ім’я_об’єкту [(список параметрів)];
//*Список параметрів не має бути порожнім*
ім’я_класу (список параметрів);
//Створюється об'єкт без імені (список може бути порожнім)

ім'я класу ім’я_об’єкту = вираз;
//Створюється об'єкт без імені і копіюється

Приклади:

monstr Super(200, 300), Vasia(50), Z;
monstr X = monstr(1000);
monstr Y = 500;

У першому операторові створюються три об'єкти. Значення не вказаних параметрів встановлюються за умовчанням. У другому операторові створюється безіменний об'єкт із значенням параметра health = 1000 (значення другого параметра встановлюється за умовчанням). Виділяється пам'ять під об'єкт **X**, в яку копіюється безіменний об'єкт. У останньому операторові створюється безіменний об'єкт із значенням параметра health = 500 (значення другого параметра, це відноситься і до конструкторів, які є методами). Виділяється пам'ять під об'єкт **Y**, в яку копіюється безіменний об'єкт. Така форма створення об'єкту можлива в тому випадку, якщо для ініціалізації об'єкту допускається задати один параметр.

Методи можуть приймати аргументи по умовчанню. Це відноситься і до конструкторів, які є методами. Якщо в конструкторі **Sf** приймаються по умовчанню координати y = 2.0, z = 2.5, r = 1, тоді конструктор приймає вид:

```
Sf(float xk, float yk=2.0, float zk=2.5, float rk=1);
{x = xc; y = yc; z = zc; r = rk;}
```

Тепер можна створити змінну класу в такий спосіб:

```
Sf (1.0); // використання всіх аргументів по умовчанню;
Sf (1.0,1.1);          //явно вказується координата y;
Sf (1.0,1.1,2.1,4.0); //скасовуються всі параметри по умовчанню.
```

Ви можете мати більш одного конструктора: це називається перепризначенням (перегрузкою) конструктора. Перегрузка дає функції можливість, використовуючи те ж саме ім’я, виконувати різноманітні дії. Використання того ж самого імені для функцій, що мають різноманітне число аргументів, або аргументи мають різноманітні типи, називається перегрузкою функції.

Приклад: Цей приклад демонструє загальний випадок використання перегружених функцій конструкторів. У даному випадку один конструктор ініціалізує усі дані-члени, інший конструктор залишає всі дані-члени непроініціалізованими.

```
#include <iostream>
using namespace std;
```

```
#define M_PI        3.14159265358979323846

class Sf
{
 public:
 float r, x, y, z;
 Sf(){}      //нічого не робить
 Sf(float xk, float yk, float zk, float rk)
 { x = xk; y = yk; z = zk; r = rk;}
 float kv() {return r*r*4*M_PI;}
 };

void main()
{
 Sf l (1, 2, 3, 4);
 Sf t;
 cout << "x = " << l.x << "\n";
 t=l;
 cout << t. kv();
}

//Виведення:
//X = 1
//201. 062
```

Як приклад класу з декількома конструкторами удосконалимо описаний раніше клас **monstr**, додавши в нього поля, заданий колір (**skin**) та ім'я (**name**):

```
enum color {red, green, blue};

class monstr
{
 int health, ammo;
 color skin;
 char *name;

public:
 monstr(int he = 100, int am =10);
 monstr(color sk);
 monstr(char * nam);
 int get_health(){return health;}
```

```
 int get_ammo(){return ammo;}
};
// ------------------------------------------------
monstr::monstr(int he, int am)
{
 health = he; ammo = am; skin = red; name = 0;
}
// ------------------------------------------------
monstr::monstr(color sk)
{
 switch (sk)
  {
    case red:
         health    = 100;
         ammo = 10;
         skin = red;
         name = 0;
         break;
    case green:
         health = 100;
         ammo = 20;
         skin = green;
         name = 0;
         break;
    case blue:
         health = 100;
         ammo = 40;
         skin = blue;
         name = 0;
         break;
  }
}

// ------------------------------------------------

monstr::monstr(char* nam)
{
 name = new char [strlen(nam) + 1];
// До довжини рядок додається 1 для зберігання нуль-символа
 strcpy(name, nam);
 health = 100; ammo =10; skin = red;
}
```

```
// -----------------------------------------

monstr* m = new monstr ("Ork");
monstr Green (green);
```

Перший з приведених вище конструкторів є конструктором за умовчанням, оскільки його можна викликати без параметрів. Об'єкти класу **monstr** тепер можна ініціалізувати різними способами, необхідний конструктор буде викликаний залежно від списку значень в дужках. При завданні декількох конструкторів слід дотримуватись тих же правил, що і при написанні переобтяжених функцій – у компілятора має бути можливість розпізнати потрібний варіант.

Існує ще один спосіб ініціалізації полів в конструкторі (окрім використаного в приведеній вище програмі привласнення полям значень формальних параметрів) – за допомогою списку ініціалізаторів, розташованих після двокрапки між заголовком та тілом конструктора:

```
monstr::monstr(int he, int am):
health (he), ammo (am), skin (red), name (0){}
```

Поля перераховуються через кому. Для кожного поля в дужках указується значення, що ініціалізується, яке може бути виразом. Без цього способу не обійтися при ініціалізації полів-констант, полів-посилань і полів-об'єктів, В останньому випадку буде викликаний конструктор, відповідний вказаним в дужках параметрам.

Конструктор не може повернути значення, щоб повідомити про помилку під час ініціалізації. Для цього можна використовувати механізм обробки виключних ситуацій, який буде розглянутий нижче.

Деструктор, як вже наголошувалося, – це особливий вид методу, що застосовується для звільнення пам'яті, займаної об'єктом. Деструктор викликається автоматично, коли об'єкт виходить з області видимості:

– для *локальних* об'єктів – при виході з блоку, в якому вони оголошені;
– для *глобальних* –як частина процедури виходу з **main**();
– для *об'єктів, заданих через вказівки,* деструктор викликається неявно при використанні оператору delete.

Деструктор:

– не має аргументів та значення, яке повертається;
– не може бути оголошений як **const** або **static**;
– не успадковується.

Якщо деструктор явним чином не визначений, компілятор автоматично створює порожній деструктор. Описувати в класі деструктор явним чином потрібний у разі, коли об'єкт містить вказівки на пам'ять, що виділяється динамічно – інакше при знищенні об'єкту пам'ять, на яку посилалися його поля-вказівки, не буде

помічена як вільна. Вказівку на деструктор визначити не можна. Якщо об'єкт містить вказівку на масив, скажімо поле char* name та ініціалізував в конструкторі, як name = new char[10], тоді деструктор повинен виглядати таким чином:

```
monstr::~monstr() {delete [ ] name;}
```

Деструктор можна викликати явним чином, наприклад:

```
Monstr* m;
m —>~monstr();
```

Це може знадобитися для об'єктів, яким за допомогою перевизначеного оператору **new** виділялася конкретна адреса пам'яті. Без необхідності явно викликати деструктор об'єкту не рекомендується.

Завдання

Завдання, розміщені в цьому розділі, окрім ознайомлення з класами містять інформацію про динамічні структури даних, перевантаження операторів тощо. Отже, відповідні завдання охоплюють декілька розділів та даються студентам по мірі проходження кожного з них.

Перелік варіантів завдань:

Варіант 1

Описати клас, який реалізовує стек. Написати програму, що використовує цей клас для моделювання Т-подібного сортувального вузла на залізниці. Програма повинна розділяти на два напрями потяг, що складається з вагонів двох типів (на кожен напрям формується потяг з вагонів одного типу). Передбачити можливість формування потягу з файлу та з клавіатури.

Варіант 2

Описати клас, який реалізовує бінарне дерево, що володіє можливістю додавання нових елементів, видалення тих, що існують, пошуку елементу по ключу, а також послідовного доступу до всіх елементів.

Написати програму, що використовує цей клас для представлення англо-російського словника. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу. Передбачити можливість формування словника з файлу та з клавіатури.

Варіант 3

Побудувати систему класів для опису плоских геометричних фігур: круг, квадрат, прямокутник. Передбачити методи для створення об'єктів, переміщення на площині, зміни розмірів і обертання на заданий кут.

Написати програму, що демонструє роботу з цими класами. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класів.

Варіант 4

Побудувати опис класу, що містить інформацію про поштову адресу організації. Передбачити можливість зміни складових частин адреси, створення та знищення об'єктів цього класу.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 5

Скласти опис класу для представлення комплексних чисел. Забезпечити виконання операцій додавання, віднімання та множення комплексних чисел.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 6

Скласти опис класу для об'єктів-векторів, які задаються координатами в тривимірному просторі. Забезпечити операції додавання та віднімання векторів з отриманням нового вектора (суми або різниці), обчислення скалярного добутку двох векторів, довжини вектора, косинуса кута між векторами.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 7

Скласти опис класу прямокутників зі сторонами, паралельними осям координат. Передбачити можливість переміщення прямокутників на площині, зміни розмірів, побудови найменшого прямокутника, що містить два задані прямокутники, та прямокутника, що є загальною частиною (перетином) двох прямокутників.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 8

Скласти опис класу для визначення одновимірних масивів цілих чисел (векторів). Передбачити можливість звернення до окремого елементу масиву з контролем виходу за межі масиву, можливість завдання довільних меж індексів при створенні об'єкту і виконання операцій поелементного додавання та віднімання масивів з однаковими межами індексів, множення та ділення всіх елементів масиву на скаляр, виведення на екран елементу масиву по заданому індексу і всього масиву.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 9

Скласти опис класу для визначення одновимірних масивів рядків фіксованої довжини. Передбачити контроль виходу за межі масиву, можливість звернення до окремих рядків масиву по індексах, виконання операцій поелементного зчеплення двох масивів з утворенням нового масиву, злиття двох масивів з виключенням елементів, що повторюються, а також виведення на екран елементу масиву по заданому індексу і всього масиву.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 10

Скласти опис класу многочленів від однієї змінної, що задаються ступенем многочлена і масивом коефіцієнтів. Передбачити методи для обчислення значення многочлена для заданого аргументу, операції додавання, віднімання та множення многочленів з отриманням нового об'єкту-многочлена, виведення на екран опису многочлена.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 11

Скласти опис класу одновимірних масивів рядків, кожен рядок задається довжиною та вказівкою на виділену для нього пам'ять. Передбачити контроль виходу за межі масивів, можливість звернення до окремих рядків масиву по індексах, виконання операцій поелементного зчеплення двох масивів з утворенням нового масиву, злиття двох масивів з виключенням елементів, що повторюються, а також виведення на екран елементу масиву і всього масиву.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 12

Скласти опис класу, що забезпечує представлення матриці довільного розміру з можливістю зміни числа рядків і стовпців, виведення на екран підматриці будь-якого розміру і всієї матриці.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 13

Написати клас для ефективної роботи з рядками, що дозволяє форматувати і порівнювати рядки, зберігати в рядках числові значення та повертати їх. Для цього необхідно реалізувати:

- перевизначені оператори привласнення і конкатенації;
- операції порівняння і приведення типів;
- перетворення в число будь-якого типу;
- виведення формату рядка.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 14

Описати клас "домашня бібліотека". Передбачити можливість роботи з довільним числом книг, пошуку книги за якою-небудь ознакою (наприклад, по авторові або по року видання), додавання книг в бібліотеку, видалення книг з неї, сортування книг по різних полях.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 15

Описати клас "записник". Передбачити можливість роботи з довільним числом записів, пошуку запису за якою-небудь ознакою (наприклад, по прізвищу, даті народження або номеру телефону), додавання і видалення записів, сортування по різних полях.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 16

Описати клас "студентська група". Передбачити можливість роботи із змінним числом студентів, пошуку студента за якою-небудь ознакою (наприклад, по прізвищу, даті народження або номеру телефону), додавання і видалення записів, сортування по різних полях.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 17

Описати клас, що реалізовує тип даних "дійсна матриця" і роботу з ними. Клас повинен реалізовувати наступні операції над матрицями:

• додавання, віднімання, множення, ділення (+, - *, /) (множення і ділення як на іншу матрицю, так і на число);

• комбіновані операції привласнення (+=, -=, *=, /==);

• операції порівняння на рівність/нерівність;

• операції обчислення зворотної і транспонованої матриці, операцію піднесення до ступеня;

• методи обчислення детермінанта і норми;

• методи, що реалізовують перевірку типу матриці (квадратна, діагональна, нульова, одинична, симетрична, верхня трикутна, нижня трикутна);

• операції введення/виведення в стандартні потоки.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 18

Описати клас "множина", що дозволяє виконувати основні операції – додавання і видалення елементу, перетин, об'єднання і різницю множин.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант 19

Описати клас, що реалізовує стек. Написати програму, що використовує цей клас для відшукання проходу по лабіринту.

Лабіринт представляється у вигляді матриці, що складається з квадратів. Кожен квадрат або відкритий, або закритий. Вхід в закритий квадрат заборонений. Якщо квадрат відкритий, то вхід в нього можливий з боку, але не з кута. Кожен квадрат визначається його координатами в матриці. Після відшукання проходу програма друкує знайдений шлях у вигляді координат квадратів.

Варіант 20

Описати клас "наочний покажчик". Кожна компоненту покажчика містить слово і номери сторінок, на яких це слово зустрічається. Кількість номерів сторінок, що відносяться до одного слова, від одного до десяти. Передбачити можливість формування покажчика з клавіатури і з файлу, виведення покажчика, виведення номерів сторінок для заданого слова, видалення елементу з покажчика.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

РОЗДІЛ 2. КЛАСИ І ПІДКЛАСИ

2.1. Конструктор копіювання

Конструктор копіювання – це спеціальний вид конструктора, який одержує єдиний параметр – вказівку на об'єкт цього ж класу:

T::T(const T&) { ... / * Тіло конструктора*/ }
де T – ім’я класу.

Цей конструктор викликається в тих випадках, коли новий об'єкт створюється шляхом копіювання існуючого об’єкту:

- при описі нового об'єкту з ініціалізацією іншим об'єктом;
- при передачі об'єкту у функцію за значенням;
- при поверненні об'єкту з функції.

Якщо програміст не вказав жодного конструктора копіювання, компілятор створює його автоматично. Такий конструктор виконує по елементне копіювання полів. Якщо клас містить вказівки або посилання, це, швидше за все, буде неправильним, оскільки і копія, і оригінал вказуватимуть на одну і ту ж область пам'яті.

Запишемо конструктор копіювання для класу **monstr**. Оскільки в ньому є поле **name**, що містить вказівку на рядок символів, конструктор копіювання повинен виділяти пам'ять під новий рядок і копіювати в неї початкову:

```
monstr::monstr(const monstr &M)
{
 if (M.name)
 {
   name = new char [strlen(M.name) + 1];
   strcpy(name, M.name);
 }
 else name = 0;
     health = M.health; ammo = M.ammo; skin = M.skin;
 }
…
monstr Vasia (blue);
monstr Super = Vasia; //Працює конструктор копіювання
monstr *m = new monstr ("Ork");
monstr Green = *m; // Працює конструктор копіювання
```

Будь-який конструктор класу, що приймає один параметр якого-небудь іншого типу, називається *конструктором перетворення,* оскільки він здійснює перетворення з типу параметра в тип цього класу.

Правила написання конструкторів класів, що входять в ієрархію, описані в розділі 3 "Спадкування класів".

2.2 Вкладені класи

Клас, оголошений усередині іншого класу, називається вкладеним. Він є членом охоплюючого класу, і його визначення може бути відкрите (**public**) або закрите (**private**). Рівень вкладеності не обмежується.

Ім'я вкладеного класу має бути унікальне в охоплюючому класі, але може збігатися з іншими іменами поза класом.

Доступу за умовчанням до приватних компонентів охоплюючого класу вкладений клас не має, як і охоплюючий клас – до приватних компонентів вкладеного. Обійти заборону допомагає механізм дружніх стосунків, наприклад

```
class External
{
 // Inner доступна приватна частина External
 friend class Inner;

  class Inner     // вкладений клас
  {
   // External доступна приватна частина Inner
   friend class External;
  };
};
```

Ні вкладений клас, ні охоплюючий не можуть звертатися до методів один одного безпосередньо. Як і в звичайних класах, необхідно оголосити об'єкт, який викликає потрібний метод. Об'єкт охоплюючого класу може передаватися методу вкладеного класу як аргумент.

```
void External::Inner::MethodInner(const External &t)
{
  //...
  // виклик методу охоплюючого класу
  memInner = t.MethodExternal();
  //...
}
```

Метод вкладеного класу **MethodInner()** отримує посилання на об'єкт зовнішнього класу і звичайним способом викликає метод **MethodExternal()**.

Якщо вкладений клас оголошений як **public**, то його можна використовувати як тип за всією програмою. Його ім'я слід писати з префіксом – ім'ям охоплюючого класу:

```
External::Inner *pointer;
```

Якщо вкладений клас оголошений в закритій частині охоплюючого класу, то він доступний тільки членам охоплюючого класу і його друзям. В цьому випадку компоненти вкладеного класу зазвичай роблять відкритими – тоді немає потреби і оголошувати іншому охоплюючий клас, наприклад

```
class External
{
    //...
    // Inner доступна приватна частина External
    friend class Inner;
    structure Inner
    { /* всі елементи доступні в External */ };
    //...
};
```

Усередині методів вкладеного класу ключове слово **this** є вказівкою на поточний об'єкт вкладеного класу.

Методи вкладеного класу можна реалізувати безпосередньо усередині класу. Якщо ж методи вкладеного класу визначаються поза класом, визначення необхідно ставити поза самим зовнішнім з охоплюючих класів – в області глобальної видимості. Ім'я методу у такому разі повинне мати префікси; кількість префіксів дорівнює рівню вкладеності класів.

В області глобальної видимості поза охоплюючим класом можна визначити і сам вкладений клас. C++ вирішує це робити, якщо в охоплюючому класі задати оголошення класу, наприклад:

```
class A
{
  //...
  class B;    // оголошення вкладеного класу
  //...
};

class A::B   // зовнішнє визначення вкладеного класу
{
  //...
};
```

Доступність визначеного таким чином класу залежить від того, в якій частині охоплюючого класу знаходиться оголошення, – якщо воно приватне, то і визначення є приватним в охоплюючому класі.

2.3 Статичні елементи класу

За допомогою модифікатору **static** можна описати статичні поля і методи класу. Їх можна розглядати як глобальні змінні або функції, доступні тільки в межах області класу.

2.3.1 Статичні поля

Статичні поля застосовуються для зберігання даних, загальних для всіх об'єктів класу, наприклад, кількості об'єктів або посилання на той, що розділяється всіма об'єктами ресурс. Ці поля існують для всіх об'єктів класу в єдиному екземплярі, тобто не дублюються. Нижче перераховані особливості статичних полів:

1. Пам'ять під статичне поле виділяється один раз при його ініціалізації незалежно від числа створених об'єктів (і навіть при їх відсутності) і ініціалізується за допомогою операції доступу до зони дії, а не операції вибору (визначення має бути записане поза функціями):

```
class A
{
 public:
 static int count; // Оголошення у класі
};

…
int A::count;    // Визначення у глобальній області
                 // За умовчанням ініціалізується нулем

//Приклад ініціалізації довільним значенням
// int A::count = 10;
```

2. Статичні поля доступні як через ім'я класу, так і через ім'я об'єкту:

```
A *a, b;
cout << A::count << a->count << b.count;
// Буде виведено одне й теж
```

3. На статичні поля розповсюджується дія специфікаторів доступу, тому статичні поля, описані як **private**, не можна змінити за допомогою операції доступу до зони дії, як описано вище. Це можна зробити тільки за допомогою статичних методів (див. далі).

4. Пам'ять, яку займає статичне поле, не враховується при визначенні розміру об'єкту за допомогою операції **sizeof**.

2.3.2 Статичні методи

Статичні методи призначені для звернення до статичних полів класу. Вони можуть звертатися безпосередньо тільки до статичних полів і викликати тільки інші статичні методи класу, тому що їм не передається прихована вказівка **this**. Звернення до статичних методів проводиться так само, як до статичних полів – або через ім'я класу, або, коли хоча б один об'єкт класу вже створений, через ім'я об'єкту.

```
class A
{
 static int count; // Поле count – приховане
 public:
 static void inc_count(){ count++: }
};

A::int count; // Визначення в глобальній області
void f()
{
 A a;
// a.count++ – неможна, поле count приховане
// Зміна поля за допомогою статичного методу:
a.inc_count(); // або A::inc__count();
}
```

Статичні методи не можуть бути константними (**const**) та віртуальними (**virtual**).

2.4 Дружні функції і класи

Іноді бажано мати безпосередній доступ ззовні до прихованих полів класу, тобто розширити інтерфейс класу. Для цього служать дружні функції і дружні класи.

2.4.1 Дружня функція

Дружні функції застосовуються для доступу до прихованих членів класу і є альтернативою методам. Метод, як правило, використовується для реалізації властивостей об'єкту, а у вигляді дружніх функцій оформляються дії, що не представ-

ляють властивості класу, але що концептуально входять в його інтерфейс і потребують доступу до його прихованих полів.

Нижче перераховані правила опису і особливості дружніх функцій.

1. Дружня функція оголошується *усередині класу,* до елементів якого їй потрібен доступ, з ключовим словом **friend**. Як параметр їй повинні передаватися об'єкт або посилання на об'єкт класу, оскільки вказівка **this** їй не передається.

2. Дружня функція може бути звичайною функцією або методом іншого раніше створеного класу. На неї не розповсюджується дія специфікаторів доступу, місце розміщення її оголошення в класі байдуже.

3. Одна функція може бути дружньою відразу з декількома класами.

Приклад :

```
class Person
{
int secret;                    // закрита частина класу
friend void Spouse(Person &);  // не член класу
public:
Person(){ secret = 10;}
};

void Spouse(Person &p)
{
 ++p.secret;
 cout << p.secret;
}

void main()
{
Person ff;
Spouse(ff);
}
```

У даному прикладі функція **Spouse()** є звичайною функцією і не належить жодному класу. Цій функції передається посилання на об'єкт класу **Person**.

Приклад:

```
class monstr; // Попереднє оголошення класу
class hero
{
 public:
 void kill(monstr &);
…
}
```

```
class monstr
{
…
 friend int steal_ammo(monstr &);
 friend void hero:: kill(monstr &);
// Клас hero має бути визначений раніше
};
…
 int steal_ammo(monstr &M){return --M.ammo;}
 void hero::kill(monstr &M){M.health = 0; M.ammo = 0;}
```

У даному прикладі приведений опис двох функцій, дружніх класу **monstr**. Функція **kill** є методом класу **hero**, а функція **steal_ammo** не належить жодному класу.

Використання дружніх функцій потрібно по можливості уникати, оскільки вони порушують принцип інкапсуляції і, таким чином, ускладнюють відлагодження та модифікацію програми.

2.4.2 Дружній клас

Якщо всі методи якого-небудь класу повинні мати доступ до прихованих полів іншого, весь клас оголошується дружнім за допомогою ключового слова **friend**. У приведеному нижче прикладі клас **mist** оголошується дружнім класу hero:

```
class hero
{
 friend class mist;
};

class mist
{
 void f1();
 void f2();
};
```

Функції **f1()** і **f2()** є дружніми по відношенню до класу **hero** (хоча і описані без ключового слова **friend**) і мають доступ до всіх його полів. Оголошення **friend** не є специфікатором доступу і не успадковується.

Завдання

Даний розділ тісно пов’язаний з динамічними структурами даних де розглядаються вкладені класи. Варіанти завдань треба виконати з використанням вкладених класів.

РОЗДІЛ 3. СПАДКУВАННЯ КЛАСІВ

3.1 Спадкування класів

Механізм спадкування класів дозволяє будувати ієрархії, в яких похідні класи отримують елементи батьківських, або базових, класів і можуть доповнювати їх або змінювати їх властивості. При великій кількості ніяк не зв'язаних класів управляти ними стає неможливо. Спадкування дозволяє впоратися з цією проблемою шляхом впорядковування і ранжирування класів, тобто об'єднання загальних для декількох класів властивостей в одному класі і використання його як базового. Класи, що знаходяться ближче до початку ієрархії, об'єднують в собі найбільш загальні риси для всіх класів, що знаходяться нижче. У міру просування вниз за ієрархією класи набувають все більше конкретних рис. Множинне спадкування дозволяє одному класу володіти властивостями двох та більше батьківських класів.

При описі класу в його заголовку перераховуються всі класи, що є для нього базовими. Можливість звернення до елементів цих класів регулюється за допомогою *ключів доступу* **private**, **protected** і **public**:

class ім'я : [private | protected | public] базовий_клас
{ тіло класу };

Якщо базових класів декілька, вони перераховуються через кому. Ключ доступу може стояти перед кожним класом, наприклад:

```
class A { ... };
class B { ... };
class C { ... };
class D: A, protected B, public C { ... };
```

За умовчанням для класів використовується ключ доступу **private**, а для структур – **public**. До цих пір ми в основному розглядали *специфікатори доступу* **private** і **public**. Для будь-якого елементу класу може також використовуватися специфікатор **protected**, який для одиночних класів, що не входять в ієрархію, рівносильний **private**. Різниця між ними виявляється при спадкуванні, що можна бачити з приведеної таблиці:

Як видно з таблиці, **private** елементи базового класу в похідному класі недоступні незалежно від ключа. Звернення до них може здійснюватися тільки через методи базового класу. Елементи **protected** при спадкоємстві з ключем **private** стають в похідному класі **private**, в решті випадків права доступу до них не змінюються.

Ключ доступу	Специфікатор у базовому класі	Доступ в похідному класі
private	private	немає
	protected	private
	public	private
protected	private	немає
	protected	protected
	public	protected
public	private	немає
	protected	protected
	public	public

Доступ до елементів **public** при спадкуванні стає відповідним ключу доступу. Якщо базовий клас успадковується з ключем **private**, можна вибірково зробити деякі його елементи доступними в похідному класі, оголосивши їх в секції **public** похідного класу за допомогою операції доступу до зони видимості:

```
class Base
{
…
 public: void f();
};

class Derived : private Base
{
…
 public:
 Base::void f ();
};
```

Як правило, похідний клас оголошується зі специфікатором доступу базового класу **public**.

Приклад.

```
class A
{
 int a;
 protected:
 void aa(int u) {a = u;}
};
```

```
class B: public A
{
 int b;
 public:
 void bb(int u);
};

void B::bb(int u)
{
 b = u;
 aa(u); //використання захищеного члена-функції
}
```

Аргументи, передані в базовий клас.

При створенні екземпляру похідного класу часто приходиться мати справу з конструкторами, що потребують передачі параметрів.

Приклад передачі параметрів.

```
class A
{
 int a, b, c;
 public:
 A(int x, int y, int z) (a = x; b = y; c = z;)
};

class B:public A
{
 int val;
 public:
 B(int d):A(d, d+1, d+5) {val = d;}
 B(int d, int e);
};

B::B(int d, int e):A(d, e, 13)
{
 val = d + e;
}
```

Будь-які аргументи, передані в базовий клас, потрібно визначити до того, як виконується тіло конструктора, оскільки параметри передаються конструктору базового класу до того, як виконається конструктор похідного класу. Тут сказано потрібно, а не повинно. Якщо ви дійсно хочете передавати ініціалізовані параметри базовому класу, то не використовуйте їх доти, поки не проініціалізуєте.

Приклад передачі неініціалізованих параметрів базовому класу.

```
class Next
{
 public:
 char* string;
 Next(char * cp){string = cp;}
};

class Last:public Next
{
 int val; char name[30];
 public:
 Last(int d);
};
// виклик базового класу з нульовою вказівкою
Last::Last(int d):Next((char *) 0)
{
 sprintf(name, "%d", d);
 //ініціалізується змінна базового класу
 string = name;
 val = d;
}
```

При створенні екземпляра класу викликається його конструктор. Якщо клас є похідним, то повинен бути викликаний конструктор базового класу. Спочатку будується базовий клас, потім похідний.

Якщо базовий клас є похідним, то процес повторюється доти, поки не буде досягнутий кореневий клас. Дерево спадкування, наведене на рис. 3.1, ілюструє це:

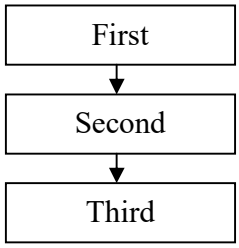

Рис. 3.1 Дерево спадкування

```
class First {...};
class Second: public First    {... };
class Third: public Second    {... };
```

При створенні екземпляра класу **Third** конструктори викликаються в такому порядку:

```
First:: First();
Second:: Second();
Third:: Third();
```

На відміну від конструкторів деструктори викликаються в зворотньому порядку:

```
Third:: ~Third();
Second:: ~Second();
First:: ~First();
```

У C++ класи розробляються для досягнення визначених цілей. Частіше усього програміст починає з нечітко обкресленої ідеї, що поступово в міру дозрівання проекту поповнюється деталями. Іноді справа закінчується двома класами, схожими один на одного. Щоб не дублювати коди в класах варто розбити їх на дві частини, помістивши загальну в батьківський клас, а ту, що відрізняється помістити в похідний.

Приклад двох класів, що мають загальні параметри:

```
#include <iostream>
using namespace std;

class A
{
 int color;
 int width, height;
 public:
 A(int, int, int);
 int GetColor() {return color;}
 int GetWidth() {return width;}
 int GetHeight() {return height;}
};

A::A(int c, int w, int h)
{
 color = c;
 width = w;
 height = h;
}

class B: public A
{
 int shel;
```

```
 public:
 B(int, int, int, int);
 int GetShel() {return shel;}
};

class C: public A
{
 int drawers, mater;
 public:
 C(int, int, int, int,int);
 int GetDrawers() {return drawers;}
 int GetMater() {return mater;}
};

B::B(int c, int w, int h, int s): A(c, w, h)
{shel = s;}

C::C(int c, int w, int h, int d, int m): A(c, w, h)
{drawers = d; mater = m;}

void main()
{
 B d(1, 2, 3, 4);
 C c(1, 2, 3, 4, 5);
 cout << c.GetMater(); // Результат - 5.
}
```

Породжуючи один клас від іншого, можна легко прийти до такої ситуації, коли в декількох класах використовуються змінні і функції з однаковими іменами.

Приклад.

```
class A
{
 public:
 int foo() { return 1;}
};

class B: public A
{
 public:
 int foo() { return 2;}
};

void main()
{
```

```
 A a;
 B b;
 cout<<a.foo()<<"\n";
 cout<<b.foo()<<"\n";
}
```

Клас B має дві функції **foo()**. Одна успадкована від **A**, інша – власна. При пошуку потрібної функції компілятор перегляне дерево спадкування починаючи з дерева **B** та використовує "найближчу" до класу функцію.

У прикладі результат відповідно 1 та 2.

У даному випадку ім'я в базовому класі переоголошується в похідному класі. Це одне й те ж, що оголошення змінних у складових операторних блоках: кожний блок має свою версію змінних і внутрішні імена подавлюють зовнішні.

Але є і відмінності. У C++ можна примусити компілятора "бачити" за межами поточної області видимості Це робиться за допомогою оператора **дозволу видимості**. Загальна форма цього оператору така:

<ім'я класу> :: <ідентифікатор із класу>

Для того, щоб використовувати оператор дозволу видимості, модифікуємо клас **B** в такий спосіб:

```
class B: public A
{
 public:
 int foo() { return A::foo();}
};
```

Тепер виклик функції **B::foo()** приведе до виклику **foo()** класу **A**. Оператор дозволу видимості використовується не тільки усередині функції класу, але і при виклику функції:

```
void main()
{
 B b;
 B * b1;
 int x = b. A::foo();
 int y = b1-> A::foo();
}
```

Використання дозволу видимості після оператора "."

```
class A
{
 public:
```

```
 int value;
};

class B: public A
{
 public:
 int count;
};

void main()
{
 B b;
 int i = b.count;
 int j = b.B::count; // зайво, але правильно
 int k = b.value;
 int l = b.A::value; // зайво, але правильно
}
```

3.2 Множинне спадкування

У природі живі істоти мають двох батьків, що забезпечує більшу розмаїтість нащадка, покращує адаптування і виживання.

Приклад використання двох батьків (рис. 3.2):

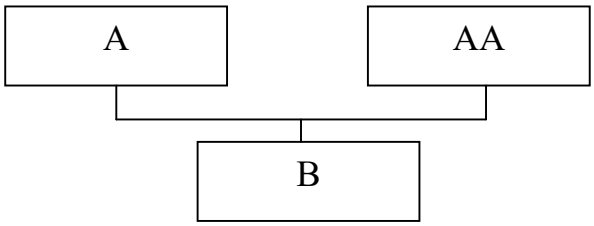

Рис.3.2 Спадкування нащадком двох класів-батьків

```
#include <iostream>
using namespace std;

#define M_PI        3.14159265358979323846

class A
{
 int a;
 public:
 A(int x) {a = x;}
 float pl1() {return (M_PI*a*a);}
};

class AA
{
```

```
 int aa;
 public:
 AA(int x) {aa = x;}
 float pl2() {return (M_PI*aa*aa);}
};
class B: public A, public AA
{
 int bb;
 public:
 B(int, int, int);
 float pl3() {return (M_PI*bb*bb);}
};

B::B(int a, int b, int c):A(a), AA(b)
{ bb = c;}

void main()
{
 B ob(1, 2, 3);
 cout << ob.pl1()<<"\n"<<ob.pl2()<<"\n"<<ob.pl3();
}
```

Розглянемо приклад спадкування з класами, що з’являються двічі (рис. 3.3):

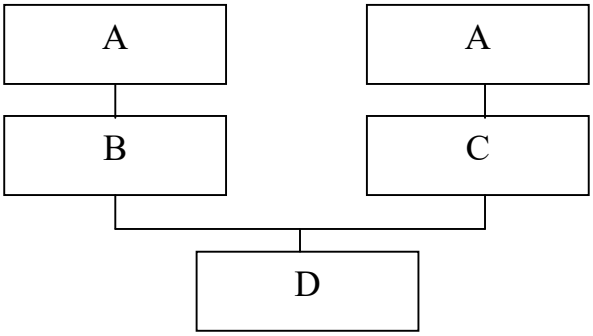

Рис. 3.3. Спадкування з класами, що з’являються двічі

```
class A
{
 public:
 int val;
};

class B: public A { };
class C: public A { };
class D: public B, C
{
```

```
 public:
  int value() {return val;}
};
```

Компілятор не розуміє на яку копію **val** посилатися, тому для усунення неоднозначності треба у функції **D::value** застосувати оператор дозволу видимості:

```
int value() {return C::val;}
```

Функція поза класом **D** також могла б одержати доступ до кожної змінної **val**, використовуючи оператор дозволу видимості:

```
void main()
{
 D d;
 int v = d.B::val;
 D *ob = new D;
 Int w = ob->B::val;
 delete ob;
}
```

У даному випадку немає ініціалізації змінної **val**. Зробити це самостійно за допомогою іншого конструктора.

3.3. Типовий приклад спадкування

У даному розділі приведено приклад спадкування від базового класу **DatabaseObject**, що вміщує інформацію про автора, назву роботи та дату її виконання. Клас **Paintings** є похідним від даного класу, в ньому зберігається інформація про видатних художників та їх праці (тип картини, її розміри тощо). Клас **Music** також є похідним від базового класу **DatabaseObject**. В ньому зберігається інформація про видатних музикантів, та додається ключ (ля-мінор, до-мажор і т.і.). Якщо необхідно розширити відомості щодо видатних музикантів та їх праць, необхідно створити вже клас похідний від **Music**, який збереже властивості базового, та за допомогою перевантаження функції **Display()** надасть можливість переглянути інформацію базового класу та нову інформацію, сформовану у похідному класі. У нашому випадку додаються інші відомості щодо видатних музикантів (камерна музика, кількість музикантів тощо).

Розглянемо дерево спадкування наданого набору класів (рис. 3.4).

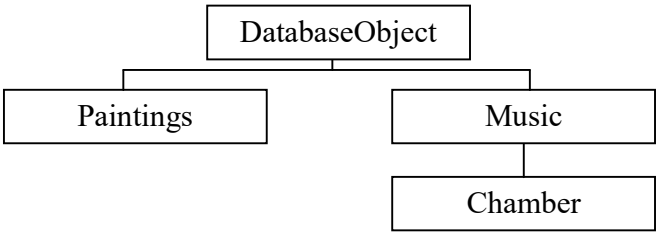

Рис. 3.4. Ієрархія класів даних

Всі загальні характеристики розташовуються у верхньому вузлі дерева.

Кожний елемент містить ім’я автора, назву праці та дату. Інша інформація, як засіб відображення відомостей про об’єкт змінюється від класу до класу.

Приклад. **Програма використання спадкування в ієрархії класів.**

```
#include <iostream>
using namespace std;

class DatabaseObject
{
 char author [50];
 char title [50];
 char date [50];
 public:
 DatabaseObject (char*, char*, char*);
 void Display ();
};

class Painting: public DatabaseObject
{
 int width, height;
 public:
 Painting (char*, char*, char*, int, int);
 void Display ();
};

class Music: public DatabaseObject
{
 char key [40];
 public:
 Music (char*, char*, char*, char*);
 void Display ();
};
```

```
class Chamber: public Music
{
 int number_of_musicians;
 public:
 Chamber (char*, char*, char*,char*, int);
 void Display ( );
};

DatabaseObject::DatabaseObject (char* who, char* what,
char* when)
{
 strcpy (author, who);
 strcpy (title, what);
 strcpy (date, when);
}

void DatabaseObject::Display ( )
{
 printf ("\n\nАвтор : %s", author);
 printf ("\nНазва : %s", title);
 printf ("\nДата : %s", date);
}

Painting::Painting (char* author, char* title, char* date, int
w, int h):
DatabaseObject (author, title, date)
{
 width = w;
 height = h;
}

void Painting::Display ( )
{
 DatabaseObject::Display ();
 printf ("\nТип : картина");
 printf ("\nрозмір: ширина = %d, висота = %d", width, height);
}

Music::Music (char* author, char* title, char* date,
char* k):
DatabaseObject (author, title, date)
{strcpy (key, k);}
```

```
void Music::Display ()
{
  DatabaseObject::Display ();
  printf ("\nТип : музика");
  printf ("\nКлюч : %s", key);
}

Chamber::Chamber (char* author, char* title, char*
date, char* key, int size):
Music (author, title, date, key)
{
 number_of_musicians = size;
}

void Chamber::Display ( )
{
 Music::Display ();
 printf ("\nІнші: камерна музика, %d музикан-
та",number_of_musicians);
}

void main ( )
{
    Music symphony ("Людвіг ван Бетховен",
                "Дев'ята симфонія",
                "1824",
                "ля-мінор");
    Painting painting ("Леонардо да Вінчі",
                "Мона Ліза",
                "1503",
                 24, 36);
    Chamber opus ("Вольфганг Амадей Моцарт",
                "Hoffmeister",
                "1786",
                "до-мажор",
                 4);
symphony.Display();
painting.Display();
opus.Display();
}
```

У результаті роботи програми буде згенерований такий звіт:

```
Автор    : Людвіг ван Бетховен
Назва    : Дев'ята симфонія
Дата     : 1824
Тип : Музика
Ключ     : ля-мінор

Автор    : Леонардо да Вінчі
Заголовок : Мона Ліза
Дата     : 1503
Тип : Картина
Розмір   : ширина = 24, висота = 36

Автор    : Вольфрганг Амадей Моцарт
Заголовок : Hoffmeister
Дата     : 1788
Тип : Музика
Ключ     : до-мажор
Інші: Камерна музика, 4 музиканта
```

Завдання

Перелік варіантів завдань:

Варіант 1

Створити базовий клас Car (машина), що характеризується торговою маркою (рядок), числом циліндрів, потужністю. Визначити методи перепризначення і зміни потужності. Створити похідний клас Lorry (вантажівка), що характеризується також вантажопідйомністю кузова. Визначити функції перепризначення марки та зміни вантажопідйомності.

Варіант 2

Створити клас Pair (пара чисел); визначити методи зміни полів і порівняння пар: пара p1 більше пари p2, якщо (first.p1 > first.p2) або (first.p1 == first.p2) та (second.p1 > second.p2). Визначити клас-спадкоємець Fraction з полями: ціла частина і дробова частина числа. Визначити повний набір методів порівняння.

Варіант 3

Створити клас Liquid (рідина), що має поля назви і щільності. Визначити методи перепризначення та зміни щільності. Створити похідний клас Alcohol (спирт), який має міцність. Визначити методи перепризначення та зміни міцності.

Варіант 4

Створити клас Pair (пара чисел); визначити методи зміни полів та обчислення добутку чисел. Визначити похідний клас Rectangle (прямокутник) з полями-сторонами. Визначити методи обчислення периметра та площі прямокутника.

Варіант 5

Створити клас Man (людина), з полями: ім'я, вік, стать і вага. Визначити методи перепризначення імені, зміни віку та зміни ваги. Створити похідний клас Student, який має поле року навчання. Визначити методи перепризначення та збільшення року навчання.

Варіант 6

Створити клас Triad (трійка чисел); визначити методи зміни полів і обчислення суми чисел. Визначити похідний клас Triangle з полями-сторонами. Визначити методи обчислення кутів та площі трикутника.

Варіант 7

Створити клас Triangle з полями-сторонами. Визначити методи зміни сторін, обчислення кутів, обчислення периметра. Створити похідний клас Equilateral (рівносторонній), який має поле площі. Визначити метод обчислення площі.

Варіант 8

Створити клас Triangle з полями-сторонами. Визначити методи зміни сторін, обчислення кутів, обчислення периметра. Створити похідний клас Rightangled (прямокутний), який має поле площі. Визначити метод обчислення площі.

Варіант 9

Створити клас Pair (пара чисел); визначити методи зміни полів і обчислення добутку чисел. Визначити похідний клас Rightangled з полями-катетами. Визначити методи обчислення гіпотенузи та площі трикутника.

Варіант 10

Створити клас Triad (трійка чисел); визначити метод порівняння тріад. Визначити похідний клас Date з полями: рік, місяць і день. Визначити повний набір методів порівняння дат.

Варіант 11

Створити клас Triad (трійка чисел); визначити метод порівняння тріад. Визначити похідний клас Time з полями: година, хвилина і секунда. Визначити повний набір методів порівняння моментів часу.

Варіант 12

Реалізувати клас-оболонку Number для числового типу float. Реалізувати методи додавання та ділення. Створити похідний клас Real, в якому реалізувати метод піднесення до довільного ступеня, і метод для обчислення логарифму числа.

Варіант 13

Створити клас Triad (трійка чисел); визначити методи збільшення полів на 1. Визначити похідний клас Date з полями: рік, місяць і день. Перевизначити методи збільшення полів на 1 і визначити метод збільшення дати на n днів.

Варіант 14

Реалізувати клас-оболонку Number для числового типу double. Реалізувати методи множення та віднімання. Створити похідний клас Real, в якому реалізувати метод, що обчислює корінь довільного ступеня, і метод для обчислення числа π в даному ступені.

Варіант 15

Створити клас Triad (трійка чисел); визначити методи збільшення полів на 1. Визначити похідний клас Time з полями: година, хвилина, секунда. Перевизначити методи збільшення полів на 1 і визначити методи збільшення на n секунд і хвилин.

Варіант 16

Створити базовий клас Pair (пара цілих чисел) з операціями перевірки на рівність і перемножування полів. Реалізувати операцію віднімання пар по формулі (a, b) – (c, d) = (a – b, c – d). Створити похідний клас Rational; визначити нові операції додавання (a, b) + (c, d) = (ad + bc, bd) і ділення (a, b)/(c, d) = (ad, bc); перевизначити операцію віднімання (a, b) – (c, d) = (ad – bc, bd).

Варіант 17

Створити клас Pair (пара чисел); визначити метод перемножування полів і операцію додавання пар (a, b) + (c, d) = (a + b, c + d). Визначити похідний клас Complex з полями: дійсна і уявна частини числа. Визначити методи множення (a, b) x (c, d) = (ac – bd, ad + bc) і віднімання (a, b) – (c, d) = (a – b, c – d).

Варіант 18

Створити клас Pair (пара цілих чисел); визначити методи зміни полів і операцію додавання пар (a, b) + (c, d) = (a + b, c + d). Визначити клас-спадкоємець Long з полями: старша частина числа і молодша частина числа. Перевизначити операцію додавання та визначити методи множення і віднімання.

Варіант 19

Створити клас Triad (трійка чисел) з операціями складання з числом, множення на число, перевірки на рівність. Створити похідний клас vector3d, що задається трійкою координат. Мають бути реалізовані: операція додавання векторів, скалярний добуток векторів.

Варіант 20

Створити клас Pair (пара цілих чисел); визначити метод множення на число і операцію додавання пар (a, b) + (c, d) = (a + b, c + d). Визначити клас-спадкоємець Money з полями: гривни і копійки. Перевизначити операцію додавання та визначити методи віднімання і ділення грошових сум.

РОЗДІЛ 4. ПОЛІМОРФІЗМ

4.1. Віртуальні функції

Поліморфізм – це властивість коду C++ поводитися по-різному, у залежності від ситуації, що виникає в момент виконання.

Поліморфізм реалізується за допомогою віртуальних функцій.

```
class A
{
 public:
 virtual void Display() {puts("\nКлас A");}
};

class B: public A
{
 public:
 virtual void Display() {puts("\nКлас B");}
};

void show (A* a) {a->Display();}

void main()
{
 A * a = new A;
 B * b = new B;
 a->Display();          // використання функції A::Display()
 b->Display();          // використання функції B::Display()
 show(a);                    // використання функції A::Display()
 show(b);                    // використання функції B::Display()
 delete a,b;
}
```

Поліморфне поводження функції **Display()** у класах **A** та **B** не очевидно, якщо розглядати тільки функцію **main()**. Воно розглядається тільки у функції **show()**, у котрій неможливо знати, яка саме функція **A->Display()** або **B->Display()** буде викликатися. Якби не було слова **virtual**, то робота коду виглядала б таким чином:

```
show(a);        // використання A::Display()
show(b);        // використання A::Display()
```

У цьому випадку звертання до функції призводить до виклику **A::Display()**.

Оголошення функції віртуальною не означає, що вона обов'язково буде перевизначатися в похідному класі. Щоб поширити поліморфне поводження функції униз по дереву спадкування, кожний похідний клас повинен оголосити ту саму функцію віртуальною.

У попередньому прикладі віртуальна функція **B::Display()** викликалася динамічно під час виконання функції **show()**. Отже, функція **B::Display()**, так сказати, віртуально перевизначила функцію **A::Display()**.

Функція, оголошена в похідному класі перевизначає функцію в базовому класі тільки тоді, коли має те ж ім'я і працює з тією ж кількістю і типом аргументів, що і віртуальна функція базового класу. Якщо вони відрізняються хоча б одним аргументом, то функція в похідному класі є цілком новою і перевизначена не буде. Розглянемо дерево спадкування на рис. 4.1.

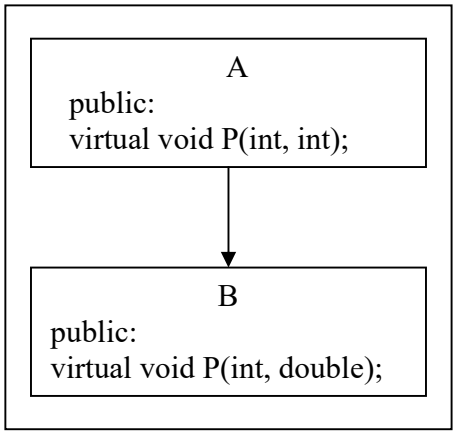

Рис. 4.1. Дерево спадкування

Дереву спадкування на рис.4.1. відповідає такий код:

```
class A
{
 public:
 virtual void P(int, int);
};

class B:public A
{
 public:
 virtual void P(int, double);
};

void A::P(int a, int b) { printf("\n a = %d, b = %d", a, b);}
void B::P(int a, double b) { printf("\n a = %d, b = %lf", a, b);}
void show(A* a) { a-> P(3, 5);}   // завжди викликає A::P(int, int)
```

```
void main()
{
 A* a = new A;
 B* b = new B;
 show(a);     // викликає A::P(int, int)
 show(b);     // викликає A::P(int, int)
 delete a,b;
}
```

4.2 Абстрактні класи

Класи, що знаходяться у верхній частині ієрархії, часто мають одну або декілька віртуальних функцій. Ці класи створені для більш узгодженого користувацького інтерфейсу в ієрархії.

У свою чергу узгоджений інтерфейс відчиняє дорогу інтенсивного використання пізнього зв'язування і полегшує роботу програміста по обслуговуванню об'єктів під час виконання програми.

C++ дозволяє обмежити використання "порожніх класів", оголошуючи їх абстрактними. Абстрактні класи містять хоча б одну чисту віртуальну функцію.

Для цього в C++ використовується спеціальний запис:

`virtual void print() = 0;` // чиста віртуальна функція, що не можна плутати з оголошенням порожньої віртуальної функції.

`virtual void print() {};` // порожня віртуальна функція

Абстрактні класи створюються для потреб похідних класів. Це означає, що абстрактні класи знаходяться поруч із коренем ієрархії.

Приклад:

```
#include <iostream>
using namespace std;

class A
{
 public:
 virtual void PrintOn() = 0; //чиста віртуальна функція,
                             //яка робить клас A абстрактним.
};

class B: public A
{
 public:
 virtual void PrintOn() {puts("Привіт\n");}
};
```

```
class C: public A
{
public:
virtual void PrintOn() {puts("Окей\n");}
};

void show(A* a)
{
 a->PrintOn();
}

void main()
{
B *b = new B;
C *c= new C;
show(b);      //використовується B::PrintOn()
show(c);      //використовується C::PrintOn()
delete b,c;
}
```

Клас **A** використовується винятково в якості базового для інших класів. Функція **show(A*)** у момент виконання базується на типі переданого їй об'єкта, визначає, яку використовувати функцію.

Клас, похідний від абстрактного, теж може бути абстрактним.

Якщо клас перевизначає будь-яку віртуальну функцію свого базового класу новою чистою віртуальною функцією, то він теж абстрактний. C++ потребує перевизначити у похідних класах усі чисті віртуальні функції. Відсутність перевизначеня чистої віртуальної функції призводить до помилки компіляції. Такий фрагмент коду ілюструє деякі проблеми, що виникають, якщо не перевизначити у похідному класі всі чисті віртуальні функції базового класу. Розглянемо просте дерево на рисунку 4.2.

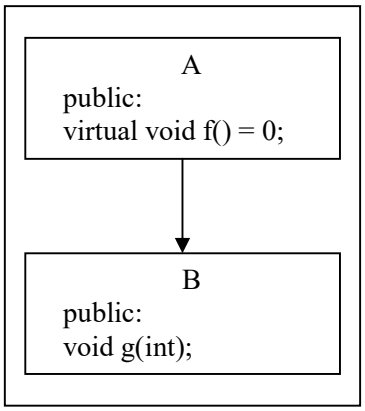

Рис. 4.2. Похідний клас, що не перевизначає чистої віртуальної функції базового класу

На рис.4.2. показано, що клас **B**, який породжений від **A**, не перевизначає чисту віртуальну функцію **A::f()**. От як виглядає відповідний код:

```
class A
{
 int value;
 public:
 virtual void f() = 0;
};

class B: public A
{
 int value;
 public:
 void g(int i){value = i;}
};
```

Якби чисті віртуальні функції поводилися також, як і оголошені віртуальні, то клас **B** успадкував би **A::f()**, оскільки клас **B** не перевизначає **A::f()**.

Припустимо, що новий клас **C** породжений від **B**. Якби клас **C** оголошував віртуальну функцію **f()**, то вона не була б доступна не через вказівку **A***, ні через **B***, оскільки ланцюжок віртуальних функцій перервався в класі **B**. Виклик **C::f()** через **A*** привело б до виклику чистої віртуальної функції **A::f()**, що призвело б до помилки під час виконання. Щоб уникнути таких проблем, у похідних класах варто перевизначити чисті віртуальні функції (навіть якщо вони знову перевизначаються чистими віртуальними функціями).

4.3. Приклади поліморфізму

Поліморфізм не тільки робить програму простіше, але і придає їй гнучкість і стійкість. Головна мета поліморфізму – заміщення функції базового класу функцією похідного класу. Гнучкість проявляється в тому, що при універсальній роботі з функціями базового класу (скажімо, контроль даних), окремі функції, які працюють з цими даними, можуть перевизначатися у похідних класах за рахунок віртуальності.

Приведемо приклад використання віртуальних функцій, що ілюструє могутність поліморфізму та демонструє дві важливі задачі, які виконує поліморфізм: це повне заміщення функції, а також вставка віртуальної функції з метою додавання інформації з базового класу.

У нашому прикладі головною задачею є пошук об’єму тіл, відомих з стереометрії, це призма та піраміда. Приведемо ієрархію класів (рис. 4.3):

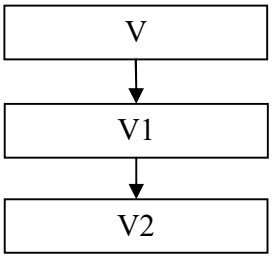

Рис. 4.3. Ієрархія класів даних

Так, клас **V** буде задавати загальний інтерфейс по введенню та контролю вхідних даних, а класи **V1** та **V2**, які є похідними від нього будуть мати лише одну віртуальну функцію, яка буде розраховувати об'єм, відповідно призми та піраміди.

Приведемо текст даного прикладу. Для виведення даних скористуємося функцією **recode()**, яка переводить символьні данні з таблиці кодування Windows у таблицю кодування DOS.

```
#include <iostream>
using namespace std;

char* recode(char *TXT);

class V
{
 protected:

 double x;
 double y;
 double z;

 public:

    // Чиста віртуальна функція f1

    virtual double f1() = 0;

   //Функція, яка викликає віртуальну функцію.(велика) - f2
   double f2()
   {
    // Ініціалізація змінних для обчислення об'єму тіла
        // багатогранника з врахуванням їх контролю

    while(1)
    {
     cout<<recode("Введите длину основания фигуры ");
     cin>>x;
     cout<<recode("Введите ширину основания фигуры ");
```

```
          cin>>y;
          cout<<recode("Введите высоту фигуры           ");
          cin>>z;

          if (x<=0||y<=0||z<=0)
          {
            cout<<recode("Расчет не возможен! Повторите ввод дан-
ных\n");
          }
           else break;
         }

          return f1();
       }
     };

     //Клас, що реалізує функцію пошуку об’єму призми
     //(основа - прямокутник);
     class V1: public V
     {
        public:
        virtual double f1()
        {
         return x*y*z;
        }
     };


     // Клас, що реалізує функцію пошуку об’єму піраміди
     // (основа - прямокутник);
     class V2: public V1
     {
        public:
        double f1()
        {
          return 1./3*V1::f1();
        }
     };

     void main()
     {
       double ob1,ob2;
       V1 prizma;
       V2 piramida;

       ob1 = prizma.f2();
       cout<<recode("Объем призмы ")<<ob1<<"\n";

       ob2 = piramida.f2();
```

```
  cout<<recode("Объем пирамиды ")<<ob2<<"\n";
}

char* recode(char *TXT)
{

     static char buf[257];
     strcpy(buf,TXT);
     unsigned int i, kol;
     kol = strlen(buf);
     for(i=0;i<kol;i++)
     {
          if (buf[i]>='A'&&buf[i]<='п')buf[i]-=64;
          if (buf[i]>='р'&&buf[i]<='я')buf[i]-=16;
    }

    return buf;
}
```

Так у класі **V** функція **f2()** є загальною і забезпечує повний інтерфейс введення та контролю даних. У ній надається можливість ввести довжину, ширину та висоту фігури, та якщо одне з введених значень буде менше або дорівнювати 0, користувачу надається можливість повторити введення даних.

Ця "велика" функція викликає "маленьку" віртуальну функцію, яка є чистою віртуальною функцію та поки що не виконує ніяких дій.

У похідному класі **V1** виконується реалізація віртуальної функції **f1()** базового класу, яка повертає об’єм призми, розрахований як x*y*z.

У похідному класі **V2**, який спадкується від класу **V1**, демонструється доповнення віртуальної функції **f1()** відносно класу **V1**. При реалізації даної функції об’єм піраміди розраховується як 1/3 частини об’єму призми:

```
return 1./3*V1::f1();
```

У функції реалізації програми **main()** створюємо два об’єкти ***prizma*** класу **V1** та ***piramida*** класу **V2**. Після цього визиваємо лише одну функцію базового класу **f2()**, яка забезпечує загальний інтерфейс введення даних та визиває функцію пошуку об’єму **f1()**, яка є віртуальною. Таким чином, за адресою будуть виконуватись віртуальні функції похідних класів, відповідно **V1** та **V2** у яких і будуть розраховуватися об’єми відповідних фігур.

Результат виконання програми буде виглядати наступним чином:

```
Введите длину основания фигуры    3
Введите ширину основания фигуры 4
Введите высоту фигуры                    5
Объем призмы 60
Введите длину основания фигуры    3
```

```
Введите ширину основания фигуры 4
Введите высоту фигуры                    5
Объем пирамиды 20
```

4.4. Внутрішнє представлення об’єктів і таблиця методів

Для кожного класу, що містить хоч би один віртуальний метод, компілятор створює *таблицю віртуальних методів* (**vtbl**), в якій для кожного віртуального методу записана його адреса в пам'яті. Адреси методів містяться в таблиці в порядку їх опису в класах. Адреса будь-якого віртуального методу має в **vtbl** один і той же зсув для кожного класу в межах ієрархії.

Кожен об'єкт містить приховане додаткове *поле посилання* на **vtb1**, зване **vptr**. Воно заповнюється конструктором при створенні об'єкту (для цього компілятор додає в початок тіла конструктора відповідні інструкції). На етапі компіляції посилання на віртуальні методи замінюються на звернення до **vtbl** через **vptr** об'єкту, а на етапі виконання у момент звернення до методу його адреса вибирається з таблиці. Таким чином, виклик віртуального методу, на відміну від звичайних методів і функцій, виконується через додатковий етап отримання адреси методу з таблиці. Це декілька уповільнює виконання програми.

Рекомендується робити віртуальними деструктори для того, щоб гарантувати правильне звільнення пам'яті з-під динамічного об'єкту, оскільки в цьому випадку у будь-який момент часу буде вибраний деструктор, відповідний фактичному типу об'єкту. Деструктор передає операції **delete** розмір об'єкту, що має тип **size_t**. Якщо об'єкт, що видаляється, є похідним і в ньому не визначений віртуальний деструктор, розмір об'єкту, який передається, може виявитися неправильним.

Чіткого правила, по якому метод слід робити віртуальним, не існує. Можна тільки дати рекомендацію оголошувати віртуальними методи, для яких є вірогідність, що вони будуть перевизначені в похідних класах. Методи, які у всій ієрархії залишаться незмінними або ті, якими похідні класи користуватися не будуть, робити віртуальними немає сенсу. З іншого боку, при проектуванні ієрархії не завжди можна передбачити, яким чином розширюватимуться базові класи (особливо при проектуванні бібліотек класів), а оголошення методу віртуальним забезпечує гнучкість і можливість розширення.

Для пояснення останньої тези уявимо собі, що виклик методу **draw** здійснюється з методу переміщення об'єкту. Якщо текст методу переміщення не залежить від типу переміщуваного об'єкту (оскільки принцип переміщення всіх об'єктів однаковий, а для малювання викликається конкретний метод), перевизначати цей метод в похідних класах немає необхідності, і він може бути описаний як невіртуальний. Якщо метод **draw** віртуальний, метод переміщення зможе без перекомпіляції працювати з об'єктами будь-яких похідних класів – навіть тих, про які при його написанні нічого відомо не було.

Віртуальний механізм працює тільки при використанні вказівок або поси-

лань на об'єкти. Об'єкт, що визначений через вказівку або посилання і містить віртуальні методи, називається *поліморфним.* В даному випадку поліморфізм полягає в тому, що за допомогою одного і того ж звернення до методу виконуються різні дії залежно від типу, на який посилається вказівка в кожен момент часу.

Розглянемо **приклад** спільного використання віртуальних і невіртуальних функцій та розміщення об'єктів в пам’яті:

```
#include <iostream>
using namespace std;

class A
{
 int a;
 public:
 void f() {puts("Функція A::f()");}
 virtual void g() {puts("Функція A::g()");}
};

class B: public A
{
 int b;
 public:
 void f() {puts("Функція B::f()");}
 void g() {puts("Функція B::g()");}
};
void do_(A& a)
{
 a.f();
 a.g();
}
void main()
{
 A a;
 B b;
 do_(a);
 do_(b);
}
```

Результатом програми буде таке виведення:

```
Функція A::f()
Функція A::g()
Функція A::f()
Функція B::g()
```

Звертання до **do_(A&)** з об'єктом **B** призводить до виклику функцій **A::f()** і **B::g()**. Це відбувається тому, що при виклику функції **do_(b)** посилання **B&** неявно перетвориться в **A&**. Компілятор бачить, що **A::f()** – невіртуальна функція і генерує пряме звертання до **A::f()**. Виклик функції **a.g()** у **do_(A&)** опрацьовується інакше, оскільки компілятор виявляє, що **g()** у класі **A** оголошена віртуальною. Це призводить до генерації коду, що включає механізм опрацювання віртуальних функцій. Розміщення об'єктів у пам'яті показано на рисунку 4.4.

Під час виконання код використовує таблицю по вказівці **vprt** в об'єкті класу **B**, щоб визначити яку функцію викликати при звертанні до **g()**, і знаходить **B::g()**. Поле **vptr** класу B знаходиться в тій же позиції, що і **vptr** класу **A**, так що **do_(A&)** одержує доступ до коректної функції, незалежно від того, який тип об'єкта передається в **do_(A&)**. Проте зверніть увагу на те, що **vptr** для об'єкта класу **A** і **vptr** для об'єкта класу **B** посилаються на різні таблиці **vtbl**.

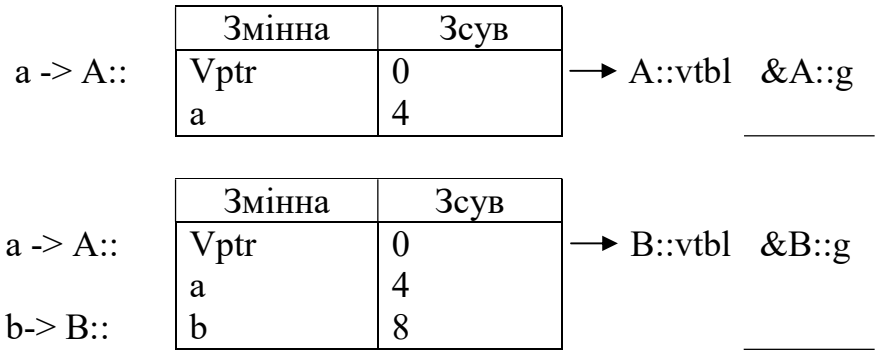

Рис. 4.4. Розміщення двох об'єктів в пам'яті

Приклад розміщення об'єктів в пам'яті:

```
#include <iostream>
using namespace std;

class A
{
 public:
 int a;
 A(int a){this->a = a;}
 virtual void PrintOn() {puts("Okey - A");}
 static int dd(){ return 10;}
};

class B: public A
{
 public:
```

```
 B(int b):A(100){this->b = b;}
 int b;
 virtual void PrintOn() {puts("Okey - B");}
};

class C: public A
{
 public:
 C(int c):A(500){this->c = c;}
 int c;
 virtual void PrintOn() {puts("Okey - C");}
};

void show(A* a)
{
 a->PrintOn();
}

void main()
{
 B *b = new B(400);
 C *c= new C(600);

 show(b);        //використовується B::PrintOn()
 show(c);    //використовується C::PrintOn()
 A*a = b;
 a->PrintOn();
 cout <<" --" << ((B*)a)->b <<"\n";
 a = c;
 a->PrintOn();
// розміри
   cout <<"size a - "<< sizeof(*a) <<"  size b - "
   <<sizeof(*b)<<" "<<"size c - "<< sizeof(*c) <<"\n";
 int* in = (int*)b;
 cout << *in << "   " <<*(in + 1)<<"  "<<*(in + 2);
 cout << endl;
 in = (int*)c;
 cout << *in << "   " <<*(in + 1)<<"  "<<*(in + 2);
 cout << "\n"<<A::dd();
 delete b,c;
}
```

Результат виконання програми буде виглядати наступним чином:

```
Okey - B
Okey - C
Okey - B
 --400
Okey - C
size a - 8  size b - 12 size c - 12
4290624   100  400
4290672   500  600
10
```

Завдання

Завдання отримуються з розділу 3 "Спадкування класів" та доповнюються віртуальними функціями.

РОЗДІЛ 5. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ

5.1 Загальні відомості

C++ дозволяє перевизначити дію більшості операторів так, щоб при використанні з об'єктами конкретного класу вони виконували задані функції. Це дає можливість використовувати власні типи даних точно так, як і стандартні. Позначення власних операторів вводити не можна. Можна перенавантажувати будь-які оператори, що існують в C++, за виключенням:

```
*    ?    ###    sizeof
```

Перевантаження операторів здійснюється за допомогою методів спеціального вигляду *{ функцій-операцій)* і підкоряється наступним правилам:

– при перевантаженні операторів зберігаються кількість аргументів, пріоритети операторів і правила асоціації (справа наліво або зліва направо), які використовуються в стандартних типах даних;
– для стандартних типів даних перевизначати оператори не можна;
– функції-оператори не можуть мати аргументів за умовчанням;
– функції-оператори успадковуються (за виключенням =);
– функції-оператори не можуть визначатися як **static**.

Функцію-оператор можна визначити трьома способами:
– методом класу (реалізація усередині класу);
– дружньою функцією класу;
– звичайною функцією;

У двох останніх випадках функція повинна приймати хоч би один аргумент, що має тип класу, вказівки або посилання на клас.

Функція-оператор містить ключове слово **operator**, за яким слідує знак оператора, який треба перевизначити.

тип operator оператор (список параметрів) { тіло функції }

5.2 Перевантаження унарних операторів

Унарна функція-оператор, яка визначається усередині класу, має бути представлена за допомогою нестатичного методу без параметрів. Якщо функція визначається поза класом (дружня або звичайна), вона повинна мати один параметр типу класу. При використанні звичайної функції (поза класом) потрібно враховувати доступність змінних полів. Для змінних поля із специфікатором **private** необхідне використання відповідного методу класу із специфікатором **public**.

Приклад:

```
class Pereg
{
 int h;
 public:
 Pereg (int h)
 {this->h = h;}

 int geth()
 { return h; }

 void set_h (int h)
 {
  this->h = h;
 };

// перевантаження (префіксного) інкремента
 Pereg& operator ++()
{++h; return *this;}

// перевантаження (префіксного) декремента - дружня функція
friend Pereg& operator --(Pereg& per);

// перевантаження постфіксного інкрімента
 Pereg& operator ++(int)
{h++; return *this;}

// перевантаження постфіксного декремента
 Pereg& operator --(int)
 {h--; return *this;}

};

// дружня функція
Pereg& operator --(Pereg& per)
{--per.h; return per;}


// звичайна функція (глобальна функція)
bool operator  !(Pereg& per)
{
```

```
    if(per.geth()) return false;
    return true;
}
```

Операції постфіксного інкремента і декремента повинні мати перший параметр типу **int**. Він використовується тільки для того, щоб відрізнити їх від префіксної.

5.3 Перевантаження бінарних операторів та операторів присвоювання

Бінарна функція-оператор, яка визначається *усередині класу,* має бути представлена за допомогою нестатичного методу з параметрами, об'єкт, що при цьому викликав її, вважається за перший операнд. Якщо функція визначається *поза класом,* вона повинна мати два параметри типу класу.

Оператор присвоювання визначений в будь-якому класі за умовчанням як поелементне копіювання. Цей оператор викликається кожного разу, коли одному існуючому об'єкту присвоюється значення іншого. Якщо клас містить поля, пам'ять під яких виділяється динамічно, необхідно визначити власний оператор привласнення. Щоб зберегти семантику присвоювання, функція-оператор повинна повертати посилання на об'єкт, для якого вона викликана, і приймати як параметр єдиний аргумент – посилання на присвоєний об'єкт.

Приклад:

```
class Pereg2: public Pereg
{
 int h;
 public:
 char* name;

Pereg2(int h):Pereg(h)
{
  this->h = h;
}

Pereg2(int h, char* nam):Pereg(h)
{
  this->h = h;
  if(nam == 0)name = 0;
  else
   {
     name = new char [strlen(nam)+1];
```

```
      strcpy(name,nam);
    }
}

int geth() {return h;}

bool operator >(Pereg2& cl)
{
   if( h > cl.geth()) return true;
   return false;
}

Pereg2& operator = ( Pereg2 &M)
{
  if(&M == this) return *this;
  if(name) delete [] name;
  if(M.name)
  {
     name = new char [strlen(M.name)+1];
     strcpy(name,M.name);
  }
  else name = 0;
    this->h = M.h;
     return *this;
}
};

 class Pereg3
 {
 int a;
 public:
 char name[10];
 int get_a(){return a;}
 void set_a(int a) {this->a = a;}
 Pereg3(int a, char* nam)
 {
  strcpy(name,nam);
  this->a = a;
 }

Pereg3(){;}
```

```
Pereg3& operator +(Pereg3& cl)
{
 a = a + cl.a;
 return *this;
}
};

void main()
{
  Pereg per(10);
  Pereg2 per2(10,"20"), per3(30,"40");
  Pereg3 per4 (4,"444"), per5(5,"555"), per6;
  per.set_h(0);
  cout << "                              \n";
  cout << !per << char(10);
  cout <<(per3 > per2) << char(10);
  per2 = per3;
  cout << per2.geth() <<"   "<<per2.name  << char(10);

  per5 = per4;
  cout << per5.name << char(10);
  per4 = per4 + per5;
  cout << per4 .get_a()<< char(10);
}
```

Результат роботи програми:

```
1
1
30    40
444
8
```

5.4 Перевантаження операторів new і delete

Щоб забезпечити альтернативні варіанти управління пам'яттю, можна визначати власні варіанти операцій **new** і **new[]** для виділення динамічної пам'яті під об'єкт і масив об'єктів відповідно, а також операції **delete** і **delete []** для її звільнення.

Ці функції-операції повинні відповідати наступним правилам:

– їм не потрібно передавати параметр типу класу;

– першим параметром функцій **new** і **new[]** повинен передаватися розмір об'єкту типу **size_t** (це тип, який повертає оператор **sizeof**, він визначається в заголовному файлі <**stddef.h**>); при виклику він передається у функції неявним чином;

– вони повинні визначатися з типом повертаного значення **void***, навіть якщо **return** повертає вказівку на інші типи (найчастіше на клас);

– операція **delete** повинна мати тип повернення **void** і перший аргумент типу **void***;

– операції виділення і звільнення пам'яті є статичними елементами класу.

Поведінка перевантажених операторів повинна відповідати діям, виконуваними ними за умовчанням. Для оператору **new** це означає, що вона повинна повертати правильне значення, коректно обробляти запит на виділення пам'яті нульового розміру і породжувати виключення при неможливості виконання запиту (про виключення розповідається в розділі "Обробка виключних ситуацій").

Для оператору **delete** слід дотримуватись умови, що видалення нульової вказівки має бути безпечним, тому всередині оператору необхідна перевірка вказівки на нуль і відсутність яких-небудь дій у разі рівності.

Стандартні операції виділення і звільнення пам'яті можуть використовуватися в області дії класу разом з перевантаженими (за допомогою оператору доступу до зони видимості :: для об'єктів цього класу і безпосередньо – для будь-яких інших).

Перевантаження оператору виділення пам'яті застосовується для економії пам'яті, підвищення швидкодії програми або для розміщення даних в деякій конкретній області. Наприклад, хай описується клас, що містить вказівку на деякий об'єкт:

```
class Obj {...};
class pObj
{
 private:
 Obj *p;
};
```

При виділенні пам'яті під об'єкт типу **pObj** за допомогою стандартного оператору new

```
pObj *p = new pObj;
```

фактична кількість байтів перевищуватиме **sizeof(pObj)**, оскільки **new** зазвичай записує в початок виділеної області її розмір (для того, щоб правильно відпрацьовував оператор **delete**).

Для невеликих об'єктів ці витрати можуть виявитися дуже значними. Для економії пам'яті можна написати власну операцію **new** класу **pObj**, яка виділятиме великий блок пам'яті, а потім розміщувати в ньому вказівки на **Obj**. Для цього в

об'єкт **pObj** вводиться статичне поле **headOfFree**, в якому зберігається вказівка на перший вільний осередок блоку для розміщення чергового об'єкту.

Невживані осередки зв'язуються в список. Щоб не позичати пам'ять під поле зв'язку, використовується об'єднання (**union**), за допомогою якого один і той же осередок використовується або для розміщення вказівки на об'єкт, або для зв'язку з наступним вільним осередком:

```
class pObj
{
 public:
 static void * operator new(size_t size);
//...
 private:
union
{
 Obj *p; // Вказівка на об'єкт
 pObj *next; // Вказівка на наступний вільний осередок
};
 static const int BLOCK_SIZE; // Розмір блоку
// Заголовок списку вільних осередків
 static pObj *headOfFree;
};

void * pObj::operator new(size_t size)
{
// Перенаправити запити невірної кількості пам'яті
// стандартній операції new:
if (size != sizeof(pObj)) return ::operator new(size);
pObj*p = headOfFree; //Вказівка на перший вільний осередок
// Перемістити вказівку списку вільних осередків:
if (p) headOfFree = p->next;
// Якщо вільної пам'яті немає. виділяємо черговий блок:
else
{
 pObj *newblock = static_cast<pObj*>
 (::operator new(BLOCK_SIZE * sizeof(pObj)));
 // Всі осередки вільні, окрім першого (він буде
// зайнятий), зв'язуємо їх:

for(int i=1;i<BLOCK_SIZE;i++)
    newblock[i].next = &newblock[i + 1];
newblock[BLOCK_SIZE-1].next = 0;
```

```
// Встановлюємо початок списку вільних осередків:
headOfFree = &newblock[1];
p = newblock;
}
return p; // Повертаємо вказівку на виділену пам'ять
}
```

Перевантажений оператор **new** успадковується, тому вона викликається для похідних об'єктів. Якщо їх розмір не відповідає розміру базового (а так, швидше за все, і є), це може викликати проблеми. Щоб їх уникнути, на початку операції перевіряється відповідність розмірів. Якщо розмір об'єкту не дорівнює тому, для якого перевантажений оператор **new**, запит на виділення пам'яті передається стандартному оператору **new**.

У програмі, що використовує клас **pObj**, має бути присутньою ініціалізація його статичних полів:

```
pObj *pObj::headOfFree; // Встановлюється в 0 за умовчанням
const int pObj::BLOCK_SIZE = 1024;
```

Як видно з цього прикладу, окрім економії пам'яті досягається ще і висока швидкодія, адже в більшості випадків виділення пам'яті зводиться до декількох простих операторів.

Природно, що якщо оператор **new** перевантажений, те ж саме повинно бути виконано і для оператору **delete** (наприклад, в нашому випадку стандартний оператор **delete** не знайде на початку об'єкту вірної інформації про його розміри, що приведе до невизначеної поведінки програми).

У розглянутому прикладі оператор **delete** повинен додавати звільнений елемент пам'яті до списку вільних осередків:

```
void pObj::operator delete(void* ObjToDie, size_t size)
{
 if (ObjToDie == 0) return;
 if (size != sizeof(pObj))
{
 ::operator delete(ObjToDie);
 return;
}
 pObj *p = static_cast<p0bj*>(0bjToDie);
 p->next = headOfFree;
 headOfFree = p;
}
```

У операторі **delete** виконана перевірка відповідності розмірів об'єктів, аналогічна приведеною в операторі **new**.

5.5 Перевантаження оператору приведення типу

Можна визначити функції-оператор, які здійснюватимуть перетворення об'єкту класу до іншого типу. Формат:

operator ім'я нового типу ();

Тип значення, яке повертається і параметри вказувати не потрібно. Можна визначати віртуальні функції перетворення типу.

Приклад:

```
monstr::operator int() {return health;}
...
monstr Vasia;
cout << int(Vasia);
```

5.6 Перевантаження оператору виклику функції

Клас, в якому визначений оператор виклику функції, називається *функціональним.* Від такого класу не вимагається наявності інших полів і методів:

```
class if_greater
{
 public:
 int operator ( ) (int a, int b) const {return a>b;}
};
```

Використання такого класу має специфічний синтаксис. Розглянемо приклад:

```
if_greater x;
cout <<x(1, 5) << endl;                 // Результат - 0
cout << if_greater( )(5, 1) << endl; // Результат - 1
```

Оскільки в класі **if_greater** визначена операція виклику функції з двома параметрами, вираз x(1, 5) є допустимим (те ж саме можна записати у вигляді x.operator () (1, 5)). Як видно з прикладу, об'єкт функціонального класу використовується так, як якби він був функцією.

У другому операторові виведення вираз **if_greater()** використовується для виклику конструктору за умовчанням класу **if_greater**. Результатом виконання цього виразу є об'єкт класу **if_greater**. Далі, як і у попередньому випадку, для цього об'єкту викликається функція з двома аргументами, записаними в круглих дужках. Оператор () можна визначати тільки як метод класу. Можна визначити пере-

вантажені оператори виклику функції з різною кількістю аргументів. Функціональні об'єкти широко застосовуються в стандартній бібліотеці C++.

5.7 Перевантаження оператору індексування

Оператор індексування [ ] зазвичай перевантажується, коли тип класу представляє безліч значень, для яких індексування має сенс. Оператор індексування повинен повертати посилання на елемент, що міститься в множині. Покажемо це на прикладі класу **Vect**, призначеного для зберігання масиву цілих чисел і безпечної роботи з ним:

```
#include <iostream>
using namespace std;

class Vect
{
 public:
 explicit Vect(int n = 10);
 Vect(const int a[], int n); //ініціалізація масивом
 ~Vect() { delete [] p; }
 int& operator [] (int i);
 void Print();
 private:
 int* p;
 int size;
};

Vect::Vect (int n): size(n)
{
 p = new int[size];
}

Vect::Vect(const int a[], int n): size(n)
{
 p = new int[size];
 for (int i = 0; i < size; i++) p[i] = a[i];
}

// Перевантаження оператору індексування:
int& Vect::operator[] (int i)
{
if(i < 0 || i >= size)
{
```

```
 cout << "Невірний індекс (i = " << i << ")" << endl;
 cout << "Завершення програми" << endl;
 exit(0);
}
 return p[i];
}

void Vect::Print()
{
 for (int i = 0; i<size; i++)
 cout << p[i] << " ";
 cout << endl;
}

void main()
{
 int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
 Vect a(arr, 10);
 a.Print ();
 cout << a[5] << endl;
 cout << a[12] << endl;
}
```

Результат роботи програми:

```
1 2 3 4 5 6 7 8 9 10
6
Невірний індекс (i = 12)
Завершення програми
```

Перевантажений оператор індексування отримує цілий аргумент і перевіряє, чи лежить його значення в межах діапазону масиву. Якщо так, то повертається адреса елементу, що відповідає семантиці стандартної операції індексування. У даному прикладі конструктор з параметром за умовчанням оголошений як **explicit** для того, щоб він не був конструктором перетворення типу**,** що викликається неявно. Ключове слово **explicit** вказує на те, що цей конструктор викликатиметься тільки явним чином.

Оператор [] можна визначати тільки як метод класу.

Завдання

Завдання отримуються з розділу 3 "Спадкування класів" та доповнюються перевантаженими операторами.

РОЗДІЛ 6. ОБРОБКА ВИКЛЮЧНИХ СИТУАЦІЙ

6.1 Загальні відомості про виключні ситуації

Виключна ситуація, або *виключення* – це виникнення непередбаченої або аварійної події, яка може породжуватися некоректним використанням апаратури. Наприклад, це ділення на нуль або звернення за неіснуючою адресою пам'яті. Зазвичай ці події приводять до завершення програми з системним повідомленням про помилку. C++ дає програмістові можливість відновлювати програму і продовжувати її виконання.

Виключення C++ не підтримують обробку асинхронних подій, таких, як помилки устаткування або обробку переривань, наприклад, натиснення клавіш Ctrl+C. Механізм виключень призначений тільки для подій, які відбуваються в результаті роботи самої програми і вказуються явним чином. Виключення виникають тоді, коли деяка частина програми не змогла зробити те, що від неї було потрібно. При цьому інша частина програми може спробувати зробити що-небудь інше.

Виключення дозволяють логічно розділити обчислювальний процес на дві частини – виявлення аварійної ситуації та її обробка. Це важливо не тільки для кращої структуризації програми. Головною причиною є те, що функція, яка виявила помилку, може не знати, що робити для її виправлення, а код, який використовує цю функцію може знати, що робити, але не вміти визначити місце виникнення. Це особливо актуально при використанні бібліотечних функцій і програм, що складаються з багатьох модулів.

Інша перевага виключень полягає в тому, що для передачі інформації про помилку в функцію, яка визивається не потрібно застосовувати повертане значення, параметри або глобальні змінні, тому інтерфейс функцій не розширюється. Це особливо важливо, наприклад, для конструкторів, які по синтаксису не можуть повертати значення.

Місце, в якому може відбутися помилка, повинне входити в контрольований блок – складеного оператора, перед яким записано ключове слово **try**. Розглянемо, яким чином реалізується обробка виключних ситуацій:

1. Обробка виключення починається з появи помилки. Функція, в якій вона виникла, генерує виключення. Для цього використовується ключове слово **throw** з параметром, що визначає вид виключення. Параметр може бути константою, змінною або об'єктом і використовується для передачі інформації про виключення його обробникові.
2. Відшукується відповідний обробник виключення і йому передається управління.
3. Якщо обробник виключення не знайдений, викликається стандартна функція **terminate**, яка викликає функцію **abort**, що аварійно завершує поточний процес. Можна встановити власну функцію завершення процесу.

При вивченні функцій наголошувалося, що при виклику кожної функції в стеку створюється область пам'яті для зберігання локальних змінних і адреси повернення в функцію, яка визивається. Термін *стек викликів* позначає послідовність викликаних, але ще незавершених функцій. *Розкручуванням стека* називається процес звільнення пам'яті з-під локальних змінних і повернення управління функції, яка викликається. Коли функція завершується, відбувається природне розкручування стека. Той же самий механізм використовується і при обробці виключень. Тому після того, як виключення було зафіксоване, виконання не може бути продовжене з точки генерації виключення.

6.2 Синтаксис виключень

Ключове слово **try** слугує для позначення *контрольованого блоку* – код, в якому може генеруватися виключення. Блок береться у фігурні дужки:

```
try
{
…

}
```

Всі функції, які прямо або побічно викликаються з **try**-блока, також вважаються такими, що належать цьому блоку.

Генерація (породження) виключення відбувається по ключовому слову **throw**, яке вживається або з параметром, або без нього:

```
throw [ вираз ];
```

Тип виразу, що стоїть після **throw**, визначає тип породжуваного виключення. При генерації виключення виконання поточного блоку припиняється, і відбувається пошук відповідного обробника і передача йому управління. Як правило, виключення генерується не безпосередньо в **try**-блоці, а у функціях, прямо або побічно в нього вкладених. Не завжди виключення, що виникло у внутрішньому блоці, може бути відразу правильно оброблено. В цьому випадку використовуються вкладені контрольовані блоки, і виключення передається на вищий рівень за допомогою ключового слова **throw** без параметрів.

Обробники виключень починаються з ключового слова **catch**, за яким в дужках слідує тип оброблюваного виключення. Вони повинні розташовуватися безпосередньо за **try**-блоком. Можна записати один або декілька обробників відповідно до типів оброблюваних виключень. Синтаксис обробників нагадує визначення функції з одним параметром – типом виключення.

Існує три форми запису:

```
catch(тип ім'я) { ... /* тіло обробника */ }
catch(тип){ ... /* тіло обробника */ }
catch(...){ … /* тіло обробника */ }
```

Перша форма застосовується, коли ім'я параметра використовується в тілі обробника для виконання яких-небудь дій – наприклад, виведення інформації про виключення. Друга форма не припускає використання інформації про виключення, грає роль тільки його тип. Багатокрапка замість параметра означає, що обробник перехоплює всі виключення. Оскільки обробники є видимими в тому порядку, в якому вони записані, обробник третього типу слід поміщати після всіх останніх. Приклад:

```
catch(int i)
{
... // Обробка виключень типу int
}
catch(const char *)
{
... // Обробка виключень типу const char*
}
catch(Overflow)
{
... // Обробка виключень класу Overflow
}
catch(...)
{
... // Обробка всіх необслужених виключень
}
```

Після обробки виключення управління передається першому операторові, що знаходиться безпосередньо за обробниками виключень. Туди ж, минувши код всіх обробників, передається управління, якщо виключення в **try**-блоці не згенерувало.

6.3 Перехоплення виключень

Коли за допомогою **throw** генерується виключення, функції виконавчої бібліотеки C++ виконують наступні дії:

1) створюють копію параметра **throw** у вигляді статичного об'єкту, який існує до тих пір, поки виключення не буде оброблено;

2) у пошуках відповідного обробника розкручують стек, викликаючи деструктори локальних об'єктів, що виходять з області дії;

3) передають об'єкт і управління обробникові, що має параметр, сумісний з типом цього об'єкту.

При розкручуванні стека всі обробники на кожному рівні є видимими послідовно, від внутрішнього блоку до зовнішнього, поки не буде знайдений відповідний обробник.

Обробник вважається знайденим, якщо тип об'єкту, вказаного після **throw**:

– той же, що і вказаний в параметрі **catch** (параметр може бути записаний у формі **T**, **const T**, **T&** або **const T&**, де **T** – тип виключення);

– є похідним від вказаного в параметрі **catch** (якщо спадкування проводилося з ключем доступу **public**);

– є вказівкою, яка може бути перетворена по стандартних правилах перетворення вказівок до типу вказівки в параметрі **catch**.

Обробники похідних класів слід розміщувати до обробників базових, оскільки інакше їм ніколи не буде передано управління. Обробник вказівки типу **void** автоматично приховує вказівку будь-якого іншого типу, тому його також слід розміщувати після обробників вказівок конкретного типу.

Розглянемо приклад.

```
#include <fstream>
using namespace std;

class Hello
{
 // Клас, який інформує про своє створення та знищення
 public:
 Hello(){cout << "Hello!" << endl;}
 ~Hello(){cout << "Bye!" << endl;}
};

void f1()
{
// Відкриваємо файл
 ifstream ifs("\\INVALID\\FILE\\NAME");

 if (!ifs)
 {
  cout << "Генеруємо виключення" << endl;
  throw "Помилка при відкритті файлу";
 }
}
```

```
void f2()
{
 Hello h; // Створюємо локальний об'єкт
 f1();    // Викликаємо функцію, яка генерує виключення
}

void main()
{
try
{
 cout<< "Входимо в try-блок" << endl;
 f2();
 cout << "Виходимо з try-блока" << endl;
}

catch(int i)
{
 cout<<"Викликаний обробник int, виключення - " <<i<<endl;
 return;
}
catch(const char * p)
{
 cout<<"Викликаний обробник const char*, виключення - "
 << p << endl;
return;
}
catch(...)
{
 cout << "Викликаний обробник всіх виключень" << endl;
 return;
}

}
```

Результати виконання програми:

```
Входимо в try-блок
Hello!
Генеруємо виключення
Bye!
Викликаний обробник const char*, виключення - Помилка
при відкритті файлу
```

Зверніть увагу, що після породження виключення був викликаний деструктор локального об'єкту, хоча управління з функції **f1** було передане обробникові, що знаходиться у функції **main**. Повідомлення "Виходимо з **try**-блоку" не було виведено. Таким чином, механізм виключень дозволяє коректно знищувати об'єкти при виникненні помилкових ситуацій. Тому *виділення і звільнення ресурсів корисно оформляти у вигляді класів,* конструктор яких виділяє ресурс, а деструктор – звільняє. Як приклад можна привести клас для роботи з файлом. Конструктор класу відкриває файл, а деструктор – закриває. В цьому випадку є гарантія, що при виникненні помилки файл буде коректно закритий, і інформація не буде загублена.

Розглянемо приклад використання виключень при контролі введення даних. Введення має бути цілим числом в діапазоні 0-20:

```
#include <iostream>
using namespace std;
#include <conio.h>
#include <windows.h>

void GotoXY(int X, int Y)
{
 COORD coord = { Y, X };
 SetConsoleCursorPosition(
 GetStdHandle(STD_OUTPUT_HANDLE), coord);
}

bool IsNumeric(char *str, bool dr )
{
 int i,pos = 0;
 if (!strlen(str)||str[0] == '.'|| str[0] == ',')
    return false ;
for (i = 0; i < (int)strlen(str); i++ )
{
 if (str[i] == ',')str[i] = '.';
 if (i == 0 && (str[0] == '+' ||  str[0] == '-'))
 continue;
if ((str[i] >= '0' && str[i] <='9') || str[i]=='.')
continue;
 return false;
}

    for (i = 1; i < (int)strlen(str);i++)
        if (str[i]=='.')pos++;
```

```
    if ( (!dr && pos) || (pos > 1)) return false ;
    return true;
}

char* recode(char *txt)
{
static char buf[257];
strcpy(buf,txt);

for(unsigned int i=0;i<strlen(buf);i++)
 {
  if (buf[i]>='A'&&buf[i]<='п')buf[i]-=64;
  if (buf[i]>='p'&&buf[i]<='я')buf[i]-=16;
 }

return buf;
}

 void kontr (char* str) throw (const char*)
 {
  if (!IsNumeric(str, 0))throw
  "Введено не ціле число\n";
  int a = atoi(str);
  if ( a < 0 || a > 20) throw
  "Помилка – Діапазон 0 – 20\n";
 }

void main()
{
char k[100];
bool vozv;

while(1)
{
vozv = true;
GotoXY(0,0);
cout << "                                    " ;
GotoXY(0,0);
cout << recode("Введіть значення k =");
cin >> k;
```

```
try
{
 kontr(k);
}

catch (const char* d)
{
  GotoXY(1,0);
  cout << recode((char*)d);
  vozv = false;
}

if(vozv)break;
} // while

system("cls");
getch();
}
```

У даному прикладі функція **IsNumeric()** повертає **true** тільки при введенні цілого числа, в противному випадку – false. У функції **kontr()** генеруються текстові виключення при невиконанні контролю значень, що вводяться. Слід зазначити, що використання виключень при контролі введення не є кращим варіантом. Приклад наводиться для практики використання виключень.

Розглянемо приклад обробки виключень класу:

```
#include <iostream>
using namespace std ;

char* recode(char *txt)
{
 static char buf[257];
 strcpy(buf,txt);

 for(unsigned int i=0;i<strlen(buf);i++)
 {
   if (buf[i]>='A'&&buf[i]<='п')buf[i]-=64;
   if (buf[i]>='р'&&buf[i]<='я')buf[i]-=16;
 }

return buf;
}
```

```
class CTest
{
 public:
  CTest(){};
  ~CTest(){};
  char *Show()
{return recode("Виключення у класі CTest class");}
};

class CDtorDemo
{
 public:
  CDtorDemo();
  ~CDtorDemo();
};

CDtorDemo::CDtorDemo()
{
 cout << recode("Конструктор") << endl;
}

CDtorDemo::~CDtorDemo()
{
 cout << recode("Деструктор") << endl;
}

void MyFunc()
{
 CDtorDemo D;
 throw CTest();
}

void main()
{
  cout << "In main" << endl;
  try
   {
    MyFunc();
   }
 catch( CTest E )
    {
     cout << E.Show() << endl;
```

```
    }
}
```

Результати виконання програми:

```
In main
Конструктор
Деструктор
Виключення у класі CTest class
```

У даному прикладі у функції **MyFunc()** створюється об'єкт класу **CDtorDemo** (при цьому в конструкторі виводиться текст "Конструктор"), потім генерується виключення класу **CTest**. Слід зазначити, що обробка цього виключення виконується після того, як об'єкт класу **CDtorDemo** звільняється, тобто спочатку виводиться в деструкторі текст "Деструктор". Тільки після цього виводиться текст обробки виключення "Виключення у класі CTest Class".

Як уже згадувалося, виключення може бути як стандартного, так і визначеного користувачем типу. При цьому немає необхідності визначати цей тип глобально – достатньо, щоб він був відомий в точці породження виключення і в точці його обробки. Клас для представлення виключення можна описати усередині класу, при роботі з яким воно може виникати. Конструктор копіювання цього класу має бути оголошений як **public**, оскільки інакше буде неможливо створити копію об'єкту при генерації виключення (конструктор копіювання, що створюється за умовчанням, має специфікатор **public**).

6.4 Список виключень функції

У заголовку функції можна задати список виключень, які вона може прямо або побічно породжувати. Оскільки заголовок є інтерфейсом функції, вказівка в ньому списку виключень дає користувачам функції необхідну інформацію для її використання, а також гарантію, що при виникненні непередбаченого виключення ця ситуація буде виявлена.

Алгоритм обробки виключення представлений на рис. 6.1.

Типи виключень перераховуються в дужках через кому після ключового слова **throw**, розташованого за списком параметрів функції, наприклад:

```
void f1() throw (int, const char*){ /*Тіло функції*/ }
void f2() throw (Oops*){ /*Тіло функції*/ }
```

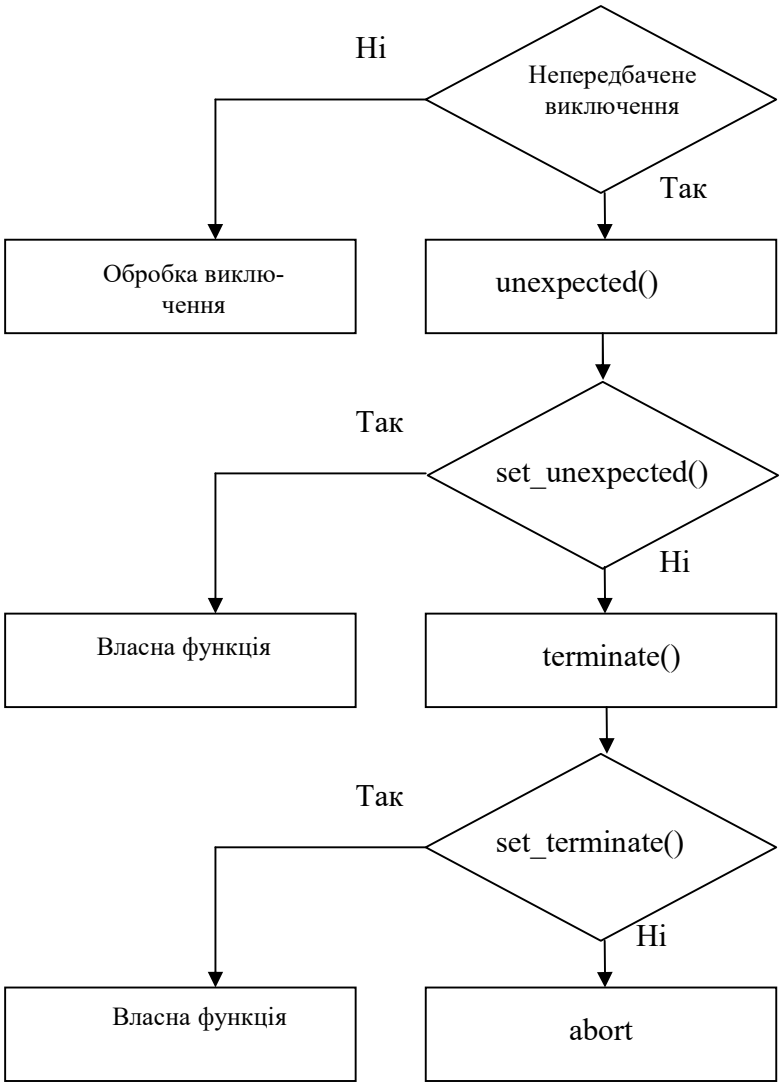

Рис. 6.1 Алгоритм обробки виключення

Функція **f1** повинна генерувати виключення тільки типів **int** і **const char***.

Функція **f2** повинна генерувати тільки виключення типу вказівки на клас **Oops** або похідних від нього класів.

Якщо ключове слово **throw** не вказане, функція може генерувати будь-яке виключення. Порожній список означає, що функція не повинна породжувати виключень:

```
void f() throw ()
{
 // Тіло функції, яка не породжує виключень
}
```

Виключення не входять в прототип функції. При перевизначенні в похідному класі віртуальної функції можна задавати список виключень, такий же або більш обмежений, ніж у відповідній функції базового класу.

Вказівка списку виключень ні до чого не зобов'язує – функція може прямо або побічно породити виключення, яке вона обіцяла не використовувати. Ця ситуація виявляється під час виконання програми і приводить до виклику стандартної функції **unexpected**, яка за умовчанням просто викликає функцію **terminate**. За допомогою функції **set_unexpected** можна встановити власну функцію, яка викликатиметься замість **terminate** і визначатиме дію програми при виникненні непередбаченої виключної ситуації.

Функція **terminate** за умовчанням викликає функцію **abort**, яка завершує виконання програми. За допомогою функції **set_terminate** можна встановити власну функцію, яка викликатиметься замість **abort** і визначатиме спосіб завершення програми. Функції **set_unexpected** і **set_terminate** описані в заголовному файлі <**exception**>.

6.5 Виключення в конструкторах та деструкторах

Мова C++ не дозволяє повертати значення з конструктора і деструктора. Механізм виключень дає можливість повідомити про помилку, що виникла в конструкторі або деструкторі об'єкту. Для цього створимо клас **Vector**, в якому обмежується кількість запрошуваної пам'яті:

```
class Vector
{
 public:
 class Size{}; // Клас виключення
 enum {max = 32000}; // Максимальна довжина вектору
 Vector(int n) // Конструктор
{
 if (n < 0 || n > max) throw Size(); ... }
...
}
```

При використанні класу **Vector** можна передбачити перехоплення виключень типу **Size**:

```
try
{
 Vector *p = new Vector(i);
 ...
}
```

```
catch(Vector::Size)
{
... // Обробка помилки розміру вектора
}
```

У обробнику може використовуватися стандартний набір основних способів видачі повідомлень про помилку і відновлення. Усередині класу, що визначає виключення, може зберігатися інформація про виключення, яка передається обробникові. Сенс цієї техніки полягає в тому, щоб забезпечити передачу інформації про помилку з точки її виявлення в місце, де для обробки помилки є достатньо можливостей.

Якщо в конструкторі об'єкту генерується виключення, автоматично викликаються деструктори для повністю створених в цьому блоці до теперішнього моменту об'єктів, а також для полів даних поточного об'єкту, що є об'єктами, і для його базових класів. Наприклад, якщо виключення виникло при створенні масиву об'єктів, деструктори будуть викликані тільки для успішно створених елементів.

Якщо об'єкт створюється в динамічній пам'яті за допомогою операції **new** і в конструкторі виникне виключення, пам'ять з-під об'єкту коректно звільняється.

6.6 Ієрархії виключень

Використання власних класів виключень переважніше застосування стандартних типів даних. За допомогою класів можна гнучкіше організувати передачу інформації про виключення, легко диференціювати обробку виключень, а крім того, з'являється можливість використовувати ієрархії класів.

Оскільки механізм управління виключеннями дозволяє створити обробник для базового класу, споріднені виключення часто можна представити у вигляді ієрархії. Проводячи виключення від загального базового класу, можна в обробнику перехоплювати посилання або вказівку на базовий клас, використовуючи поліморфізм. Наприклад, в математичній бібліотеці можна організувати класи таким чином:

```
class Matherr{}:
class Overflow: public Matherr{}; // Переповнення
class Underflow: public Matherr{}; // Зникнення порядку
class ZeroDivide: public Matherr{}; // Ділення на нуль
```

Для представлення помилок введення/виведення можуть використовуватися наступні класи:

```
class IOerr{}:
class Readerr: public IOerr{}; // Помилка читання
```

```
class Writerr: public IOerr{}; // Помилка запису
class Seekerr: public IOerr{}; // Помилка пошуку
```

Залежно від обставин можна використовувати або обробник виключень базового класу, який перехоплюватиме і похідні виключення, або власні обробники похідних класів.

Існує ряд *стандартних виключень,* які генеруються операціями або функціями C++. Всі вони є похідними від бібліотечного класу **exception**, описаного в заголовному файлі <**stdexcept**>. Наприклад, операція **new** при невдалому виділенні пам'яті генерує виключення типу **bad_alloc**.

Програміст може визначити власні виключення, похідні від стандартних [1].

Завдання

Завдання отримуються з розділу 3 "Спадкування класів" та доповнюються обробкою виключних ситуацій.

РОЗДІЛ 7. РЯДКИ

7.1 Рядки в стилі C++

C++ не містить стандартного типу даних "рядок". Замість цього він підтримує масиви символів, що завершуються нуль-символом. Бібліотека містить функції для роботи з такими масивами, успадковані від C і описані в заголовному файлі <**string.h**> (<**cstring**>). Вони дозволяють досягти високої ефективності, але дуже незручні і небезпечні у використанні, оскільки вихід за межі рядка не перевіряється. Для нового стандарту C++ розроблена стандартна бібліотека роботи з типом даних **string**, який позбавлений цих недоліків. Основні дії з рядками виконуються за допомогою операцій і методів, а довжина рядка змінюється динамічно відповідно до потреб. Для використання класу необхідно підключити до програми заголовний файл <**string**>. Порівняльний аналіз старого і нового стилю роботи з рядками розглянемо на прикладі:

```
#include <cstring>
#include <string>
#include <iostream>
using namespace std;

void main ()
{
// Рядки із завершальним нулем
 char c1[80], c2[80] , c3[80];

 string s1, s2, s3;
// Привласнення рядків
strcpy(c1, "old string one");
strcpy(c2, c1);

s1 = "new string one";
s2 = s1;
// Конкатенація рядків
strcpy(c3, c1);
strcpy(c3, c2);
s3 = s1 + s2;
// Порівняння рядків
if (strcmp(c2, c3) < 0 ) cout << c2;
else cout << c3;
if (s2 < s3) cout << s2;
else cout << s3;
}
```

Як видно з прикладу, виконання будь-яких дій з рядками старого стилю вимагає використання функцій і менш наочно. Крім того, необхідно перевіряти, чи достатньо місця в рядку-приймачі при копіюванні, тобто фактично код роботи з рядками старого стилю має бути ще довшим. Рядки типу **string** захищені від виходу інформації за їх межі і з ними можна працювати так само, як з будь-яким вбудованим типом даних, тобто за допомогою операцій. Розглянемо основні особливості і прийоми роботи з рядками.

7.1.1 Конструктори і операції привласнення

У класі **string** визначено декілька **конструкторів.** Нижче в спрощеному вигляді приведені заголовки, які найбільш часто використовуються:

```
string();
string(const char * );
string(const char *, int n);
string(string &);
```

Перший конструктор створює порожній об'єкт типу **string**. Другий створює об'єкт типу **string** на основі рядка старого стилю, третій створює об'єкт типу **string** і записує туди **n** символів з рядка, вказаного першим параметром. Останній конструктор є конструктором копіювання, який створює новий об'єкт як копію об'єкту, переданого йому як параметр.

У класі **string** визначено три операції присвоювання:

```
string& operator=(const string& str);
string& operator=(const char* s);
string& operator=(char c);
```

Як видно із заголовків, рядку можна привласнювати інший рядок типу **string**, рядок старого стилю або окремий символ, наприклад:

```
string s1;
string s2("Вася");
string s3(s2);
s1 = 'X';
s1 = "Вася";
s2 = s3;
```

7.1.2 Операції

Нижче приведені допустимі для об'єктів класу **string** операції:

Операції	Дія	Операції	Дія
=	присвоювання	>	більше
+	конкатенація	>=	більше або рівно
==	рівність	[ ]	індексація
!=	нерівність	<<	виведення
<	менше	>>	введення
<=	менше або рівно	+=	додавання

Синтаксис операцій та їх дія очевидні. Розміри рядків встановлюються автоматично так, щоб об'єкт міг містити присвоюване йому значення. Треба відзначити, що для рядків типу **string** не дотримується відповідність між адресою першого елементу рядка та ім'ям, як це було у разі рядків старого стилю, тобто **&s[0]** не рівне **s**. Окрім операції індексації, для доступу до елементу рядка визначена функція **at**:

```
string s("Вася");
cout << s.at(1); // Буде виведений символ а
```

Якщо індекс перевищує довжину рядка, породжується виключення **out_of_range**. Для обробки частин рядків (наприклад, пошуку підрядка, вставки в рядок, видалення символів) в класі **string** визначена безліч різноманітних методів (функцій).

7.2. Функції класу string

Функції класу **string** для зручності розгляду можна розбити на декілька категорій: привласнення і додавання частин рядків, перетворення рядків, пошук підрядків, порівняння і отримання характеристик рядків.

7.2.1 Привласнення і додавання частин рядків

Для **привласнення** частині одного рядка іншого слугує функція **assign**:

```
assign(const string& str);
assign(const string& str, size_type pos, size_type n);
assign(const char* s, size_type n);
```

Тут **size_type** – беззнаковий цілий тип.

Перша форма функції привласнює рядок **str** рядку, який викликає функцію, при цьому дія функції еквівалентна операції присвоювання:

```
string s1(Вася"), s2;
s2.assign(s1); // Рівносильно s2 = s1;
```

Друга форма привласнює рядку, який викликає функцію, частину рядка **str**, починаючи з позиції **pos**. Якщо **pos** більше довжини рядка, породжується виключення **out_of_range**. Рядку, який викликає функцію, привласнюється **n** символів, або, якщо **pos + n** більше, ніж довжина рядка **str**, всі символи до кінця рядка **str**.

Третя форма привласнює рядку,який викликає функцію, **n** символів рядка **s** старого типу.

Для **додавання** частині одного рядка до іншого слугує функція **append**:

```
append(const string& str );
append(const string& str, size_type pos, size_type n);
append(const char* s, size_type n);
```

Перша форма функції додає рядок **str** до кінця рядка, який викликає функцію, при цьому дія функції еквівалентна операції конкатенації (+).

Друга форма додає до рядка, який викликає функцію, частину рядка **str**, починаючи з позиції **pos**. Якщо **pos** більше довжини рядка, породжується виключення **out_of_range**. До рядка, який викликає функцію, додається **n** символів, або, якщо **pos + n** більше, ніж довжина рядка **str**, всі символи до кінця рядка **str**. Якщо довжина результату більше максимально допустимої довжини рядка, породжується виключення **length_error**.

Третя форма додає до рядка, який викликає функцію, **n** символів рядка **s** старого типу.

7.2.2 Перетворення рядків

Для **вставки** в один рядок частини іншого рядка слугує функція **insert**:

```
insert (size_type pos1, const string& str );
insert (size_type pos1, const string& str,
        size_type pos2, size_type n);
insert (size_type pos, const char* s, size_type n);
```

Перша форма функції вставляє рядок **str** в рядок, який викликає функцію, починаючи з позиції **pos1**. Іншими словами, рядок, який викликає функцію, замінщається рядком, який складається з перших **pos1** символів, за якими слідує рядок **str** цілком, а після нього розташовується решта символів рядка, який викликає фу-

нкцію. Якщо **pos1** більше довжини рядка, породжується виключення **out_of_range**. Якщо довжина результату більше максимально допустимої довжини рядка, породжується виключення **length_error**.

Друга форма функції вставляє в рядок, який викликає функцію, частину рядка **str**, починаючи з позиції **pos1**. Рядок, який викликає функцію, заміщається рядком, який складається з перших **pos1** символів, за якими слідують **n** елементів рядка **str**, починаючи з позиції **pos2**, а після них розташовується решта символів рядка, який викликає функцію. Якщо **n** більше довжини рядка **str**, копіюється весь залишок рядка **str**. Якщо **pos1** або **pos2** більше довжини відповідного рядка, породжується виключення **out_of_range**. Якщо довжина результату більше максимально допустимої довжини рядка, породжується виключення **length_error**.

Третя форма функції вставляє в рядок, який викликає функцію, **n** елементів рядка **s** старого типу, починаючи з позиції **pos**.

Для **видалення** частини рядка слугує функція **erase**:

```
erase(size_type pos = 0, size_type n = npos);
```

Вона видаляє із рядка **n** елементів, починаючи з позиції **pos**. Якщо **pos** не вказане, елементи видаляються з початку рядка. Якщо не вказане **n**, видаляється весь залишок рядка.

Очищення всього рядка можна виконати за допомогою функції **clear**:

```
void clear();
```

Для **заміни** частині рядка слугує функція **replace**:

```
replace(size_type pos1,size_type n1,const string& str);
replace(size_type pos1, size_type n1,
    const string& str, size_type pos2,size_type n2);
```

Тут **pos1** позиція рядка, починаючи з якого виконується заміна, **n1** – кількість елементів, що видаляються, **pos2** – позиція рядка **str**, починаючи з якої він вставляється в рядок, який викликає функцію, **n2** – кількість вставляємих елементів рядка **str**. Якщо **pos1** або **pos2** більше довжини відповідного рядка, породжується виключення **out_of_range**. Якщо довжина результату більше максимально допустимої довжини рядка, породжується виключення **length_error**.

Третя форма функції заміни дозволяє замінити **n1** символів рядка на **n2** символів рядка старого стилю **s**:

```
replace(size_type pos1, size_type n1,
        const char* s, size_type n2);
```

Для **обміну** вмісту двох рядків слугує функція **swap**:

```
swap(strings s);
```

Для **виділення частини рядка** слугує функція **substr**:

```
string substr(size_type pos=0, size_type n=npos) const;
```

Ця функція повертає рядок довжиною **n**, починаючи з позиції **pos**. Якщо **pos** більше довжини рядка, породжується виключення **out_of_range**. Якщо **n** більше довжини рядка, повертається весь залишок рядка.

Іноді потрібно перетворювати об'єкти типу **string в рядки старого стилю.** Для цього призначена функція **c_str**:

```
const char* c_str() const;
```

Вона повертає константну вказівку на рядок, що закінчується нуль-символом. Цей рядок не можна намагатися змінити. Вказівка, яка на неї посилається, може стати некоректною після будь-якої операції над рядком-джерелом. Аналогічно працює функція **data,** за тим виключенням, що не додає в кінець рядка нуль-символ:

```
const char* data ( ) const;
```

Функція **copy** копіює в масив **s n** елементів рядка, починаючи з позиції **pos**. Нуль-символ в результуючий масив не заноситься. Функція повергає кількість скопійованих елементів:

```
size_type copy(char*s,size_type n,size_type pos=0) const;
```

Приклад використання функцій зміни вмісту рядків:

```
#include <string>
#include <iostream>
using namespace std;

void main ()
{
 string s1("прекрасная королева"), s2("ле"),
 s3("корова");
 cout <<"s1= " << s1 << endl;
 cout <<"s2= " << s2 <<endl;
 cout <<"s3= " << s3 <<endl;
```

```
 // Застосування функції insert;
 cout <<"після insert:" << endl;
 cout << "s3= " << s3.insert(4, s2) << endl;
 cout << "s3= " << s3.insert(7, "к") << endl;
 // Застосування функції erase:
 s1.erase(0,3);
 cout << "після erase:" << endl;
 cout << "s1= " << s1.erase(12, 2) << endl;
 // Застосування функції replace;
 cout << "після replace:" << endl;
 cout << "s1= " << s1.replace(0, 3, s3, 4, 2) << endl;
}
```

Результат роботи програми:

```
s1= прекрасная королева
s2= ле
s3- корова
після insert:
s3= королева
s3= королевка
після erase:
s1= красная корова
після replace:
s1= лесная корова
```

7.2.3 Пошук підрядків

Для пошуку в класі **string** передбачена велика різноманітність функцій. Нижче приведені основні:

```
size_type find(const string&str,size_type pos=0) const;
```

Шукає саме ліве входження рядка **str** в рядок, який викликає функцію, починаючи з позиції **pos**, і повертає позицію рядка або **npos**, якщо рядок не знайдений.

```
size_type find(char c, size_type pos = 0) const;
```

Шукає саме ліве входження символу **c** рядок, який викликає функцію, починаючи з позиції **pos**, і повертає позицію символу або **npos**, якщо символ не знайдений.

```
size_type rfind(const string&str,size_type pos=npos) const;
```

Шукає саме праве входження рядка **str** в рядок, який викликає функцію, до позиції **pos**, і повертає позицію рядка або **npos**, якщо рядок не знайдений.

```
size_type rfind(char c, size_type pos = npos) const;
```

Шукає саме праве входження символу **c** в рядок, який викликає функцію, до позиції **pos**, і повертає позицію символу або **npos**, якщо символ не знайдений.

```
size_type find_first_of(const string& str,
                        size_type pos = 0) const;
```

Шукає саме ліве входження будь-якого символу рядка **str** в рядок, який викликає функцію, починаючи з позиції **pos**, і повертає позицію символу або **npos**, якщо входження не знайдене.

```
size_type find_first_of(char c, size_type pos=0) const;
```

Шукає саме ліве входження символу **c** в рядок, який викликає функцію, починаючи з позиції **pos**, і повертає позицію символу або **npos**, якщо входження не знайдене.

```
size_type find_last_of(const string& str,
                       size_type pos = npos) const;
```

Шукає саме праве входження будь-якого символу рядка **str** в рядок, який викликає функцію, починаючи з позиції **pos**, і повертає позицію символу або **npos**, якщо входження не знайдене.

```
size_type find_last_of(char c, size_type pos=npos) const;
```

Шукає саме праве входження символу **c** в рядок, який викликає функцію, починаючи з позиції **pos**, і повертає позицію символу або **npos**, якщо входження не знайдене.

```
size_type find_first_not_of(const string& str,
                            size_type pos = 0) const;
```

Шукає саму ліву позицію, починаючи з позиції **pos**, для якої жоден символ рядка **str** не збігається з символом рядка, який викликає функцію.

```
size_type find_first_not_of(char c, size_type pos=0) const;
```

Шукає саму ліву позицію, починаючи з позиції **pos**, для якої символ **c** не збігається з символом рядка, який викликає функцію.

```
size_type find_last_not_of(const string& str,
                           size_type pos = npos) const;
```

Шукає саму праву позицію до позиції **pos**, для якої жоден символ рядка **str** не збігається з символом рядка, який викликає функцію.

```
size_type find_last_not_of(char c,
                           size_type pos = npos) const;
```

Шукає саму праву позицію до позиції **pos**, для якої символ **c** не збігається з символом рядка, який викликає функцію.

Для кожної функції існує варіант, що дозволяє шукати в заданому рядку підрядки старого стилю.

Приклад застосування функцій пошуку:

```
#include <string>
#include <iostream>
using namespace std;

void main()
{
 string s1("лесная королева"), s2("ле");
 cout << "s1= " << s1 << endl;
 cout << "s2= " << s2 << endl;
 int i = s1.find(s2);
 int j = s1.rfind(s2);
 cout << "перше s2 в s1 " << i << endl;
 cout << "останнє s2 в s1 " << j << endl;
 cout << "перше 'o' в s1 " << s1.find('o') << endl;
 cout << "останнє 'o' в s1 " << s1.rfind ('o') << endl;
 cout<<"перше в s1 " << s1.find_first_of("ae") << endl;
 cout<<"останнє в s1 "<< s1.find_last_of("ae") << endl;
}
```

Результат роботи програми:

```
s1= лесная королева
s2= ле
перше s2 в s1 0
останнє .s2 в s1 11
перше 'o' в s1 8
останнє  'o' в s1 10
перше в s1 1
останнє в s1 14
```

7.2.3 Порівняння частин рядків

Для порівняння рядків застосовуються перевизначені операції відношення, а якщо потрібно порівнювати частини рядків, використовується функція **compare**:

```
int compare(const string& str) const;
int compare(size_type pos1, size_type n1,
            const string& str) const;
int compare(size_type pos1, size_type n1,
const string& str, size_type pos2, size_type n2) const;
```

Перша форма функції порівнює два рядки повністю і повертає значення, менше 0, якщо рядок, який викликає функцію, лексикографічно менший **str**, рівний нулю, якщо рядки однакові, і більше нуля – якщо рядок, який викликає функцію, більший. Ця форма є аналогом функції порівняння рядків **strstr** бібліотеки С.

Друга форма функції виконує аналогічні дії, але порівнює з рядком **str n1** символів рядка, який викликає функцію, починаючи з **pos1**. Третя форма функції порівнює **n1** символів рядка, який викликає функцію, починаючи з **pos1**, з підрядком рядка **str** довжиною **n2** символів, починаючи з **pos2**.

Аналогічні форми функцій існують і для порівняння рядків типу **string** з рядками старого стилю.

Приклад використання функції порівняння рядків:

```
#include <string>
#include <iostream>
using namespace std;
void main ()
{
 string s1("лесная королева"), s2("ле"), s3("корова");
 cout << "s1= " << s1 << endl;
 cout << "s2= " << s2 << endl;
 cout << "s3= " << s3 << endl;
 if (s2.compare(s3) > 0) cout << "s2 > s3 " << endl;
 if (s1.compare(7, 4, s3) < 0)
 cout << "s1[7-10] < s3 " << endl;
 if (s1.compare(7, 4, s3, 0, 4 ) == 0)
 cout << "s1[7-10] == s3[0-3] " << endl;
}
```

Результат роботи програми:

```
s1= лесная королева
```

```
s2= ле
s3= корова
s2 > s3
s1[7-10] < s3
s1[7-10] == s3[0-3]
```

7.2.4 Отримання характеристик рядків

У класі **string** визначено декілька функцій-членів, що дозволяють отримати довжину рядка і об'єм пам'яті, займаний об'єктом:

```
size_type size() const; // Кількість елементів рядка
size_type length() const; // Кількість елементів рядка
size_type max_size() const; //Максимальна довжина рядка
size_type capacity() const;//Об'єм пам'яті, займаний рядком
bool empty() const; //Істина, якщо рядок порожній
```

Завдання

Всі завдання необхідно реалізувати з використанням класу string.

Перелік варіантів завдань:

Варіант 1

У заданому тексті замінити слово А на слово В (довжини слів не співпадають).

Варіант 2

У заданому тексті визначити кількість слів.

Варіант 3

Маємо відомість, яка складається із 10 прізвищ. Знайти і надрукувати порядкові номера прізвищ, які задаються.

Варіант 4

У слові "лірика" після кожного складу вставити склад "ма".

Варіант 5

У тексті забрати зайві пропуски між словами, залишивши по одному.

Варіант 6

У тексті вставити між словами замість одного два пропуски.

Варіант 7

У заданому слові переставити літери в алфавітному порядку.

Варіант 8

У текстовій змінній "71D523CE8" поставити числа у порядку зменшення.

Варіант 9

Розділити заданий текст на рядки. У формі розділення рядків використаний символ %.

Варіант 10

У виразі "МІСЯЧНИЙ ШЛЯХ" переставити місцями слова.

Варіант 11

Дано слово "АЛГОРИТМ". Вивести його на друк у зворотному порядку.

Варіант 12

Скільки разів у тексті зустрічається задане слово? (Слова розділені пропусками).

Варіант 13

У слові "ЕЛЕКТРОНІКА" переставити літери у порядку зворотному алфавіту.

Варіант 14

Надрукувати найдовше слово із заданого тексту "Ціль статистики зосереджується в наданні фактів у найбільш стислій формі".

Варіант 15

У текстовій змінній "2C35IA4" представити числа у порядку зростання.

Варіант 16

Існує відомість, яка складається із 10 прізвищ. Вивести на друк цю відомість у алфавітному порядку.

Варіант 17

В аналізуючому слові всі голосні літери видалити та на їх місце поставити пропуск. Вивести на дисплей початкове слово та результат перетворень.

Варіант 18

У слові "ЛІТЕРАТУРА" кожну голосну літеру виділити символом ".

Варіант 19

У заданому слові змінити місцями першу і останню літери. Вивести на екран дисплея початкове слово і результат.

Варіант 20

У заданому тексті видалити частину тексту , яка взята в дужки. (Разом з дужками).

РОЗДІЛ 8. ШАБЛОНИ КЛАСІВ

8.1. Загальна характеристика динамічних структур даних

Будь-яка програма призначена для обробки даних, від способу організації яких залежать алгоритми роботи, тому вибір структур даних повинен робитись перед створенням алгоритмів. Найчастіше в програмах використовуються масиви, структури і їх поєднання, наприклад, масиви структур, полями яких є масиви та структури.

Пам'ять під дані виділяється або на етапі компіляції (в цьому випадку необхідний об'єм має бути відомий до початку виконання програми, тобто заданий у вигляді константи), або під час виконання програми за допомогою операції new. У обох випадках виділяється безперервна ділянка пам'яті.

Якщо до початку роботи з даними неможливо визначити, скільки пам'яті буде потрібно для їх зберігання, пам'ять виділяється в міру необхідності окремими блоками, пов'язаними один з одним за допомогою вказівок. Такий спосіб організації даних називається динамічними структурами даних, оскільки їх розмір змінюється під час виконання програми. З динамічних структур в програмах найчастіше використовуються стеки, черги, лінійні списки. Вони розрізняються способами зв'язку окремих елементів і допустимими операціями.

Динамічні структури широко застосовують і для ефективнішої роботи з даними, розмір яких відомий, особливо для вирішення задач сортування, оскільки впорядковування динамічних структур не вимагає перестановки елементів, а зводиться до зміни вказівок на ці елементи. Наприклад, якщо в процесі виконання програми потрібно багато разів упорядковувати великий масив даних, має сенс організувати його у вигляді лінійного списку.

Елемент будь-якої динамічної структури даних є структурою (struct), що містить принаймні два поля: для зберігання даних і для вказівки. Полів даних та вказівок може бути декілька. Опис простого елементу виглядає таким чином:

```
struct Node
{
Data d; // тип даних Data має бути визначений раніше
Node *p;
};
```

Розглянемо реалізацію основних операцій з динамічними структурами даних (стек, черга, лінійний список).

8.2. Стек

Стек реалізує принцип обслуговування LIFO (last in – first out, останнім прийшов, – першим пішов). Стек можна представити як стопку книг, які складаються

одна на одну. Так першою буде взята остання книга в стопці.

Нижче приведена програма, яка формує стек з п'яти цілих чисел (1, 2, 3, 4, 5) і виводить його на екран. Функція поміщення елементу в стек називається push, а вибірки – pop. Вказівка для роботи із стеком (top) завжди посилається на його вершину.

```
#include <iostream>
using namespace std;

struct Node
{
 int d;
 Node *p;
};

Node * first(int d);
void push(Node **top, int d);
int pop(Node **top);

void main()
{
 Node* top = first(1);
 for (int i = 2; i<6; i++) push(&top, i);
 while (top)
   cout << pop(&top) << " ";
}

// Початкове формування стеку
Node * first(int d)
{
 Node *pv = new Node;
 pv->d = d;
 pv->p = 0;
 return pv;
}

// Занесення в стек
void push(Node **top, int d)
{
 Node *pv = new Node;
 pv->d = d;
 pv->p = *top;
 *top = pv;
}
```

```
// Вибірка із стеку
int pop (Node **top)
{
 int temp = (*top)->d;
 Node *pv = *top;
 *top = (*top)->p;
 delete pv;
 return temp;
}
```

Результат роботи програми:

```
5 4 3 2 1
```

8.3. Черга

Черга реалізує принцип обслуговування FIFO (first in – first out, першим прийшов, – першим пішов). Приклад черги – черга людей в магазині. Нижче приведена програма, яка формує чергу з п'яти цілих чисел і виводить її на екран. Функція поміщення елементу в кінець черги називається add, а вибірки – del. Вказівка на початок черги називається pbeg, вказівка на кінець – pend.

```
#include <iostream>
using namespace std;

struct Node
{
int d;
Node *p;
};

Node * first(int d);
void add(Node **pend, int d);
int del(Node **pbeg);

void main()
{
 Node *pbeg = first(1);
 Node *pend = pbeg;
 for (int i = 2; i<6; i++) add(&pend, i);
 while (pbeg)
   cout << del(&pbeg) << " ";
}
```

```
// Початкове формування черги
Node* first(int d)
{
 Node *pv = new Node;
 pv->d = d;
 pv->p = 0;
 return pv;
}

// Додавання в кінець
void add(Node **pend, int d)
{
 Node *pv = new Node;
 pv->d = d;
 pv->p = 0;
 (*pend)->p = pv;
 *pend = pv;
}

// Вибірка
int del(Node **pbeg)
{
 int temp = (*pbeg)->d;
 Node *pv = *pbeg;
 *pbeg = (*pbeg)->p;
 delete pv;
 return temp;
}
```

Результат роботи програми:

```
12 3 4 5
```

8.4. Лінійний список

Найпростіший спосіб зв'язати безліч елементів – зробити так, щоб кожен елемент містив посилання на наступний. Такий список називається однонаправленим (однозв’язним). Якщо додати в кожен елемент друге посилання – на попередній елемент, вийде двонаправлений список (двозв’язний), якщо останній елемент зв'язати вказівкою з першим, вийде кільцевий список.

Кожен елемент списку містить ключ, що ідентифікує цей елемент. Ключ зазвичай буває або цілим числом, або рядком і є частиною поля даних. У якості ключа в процесі роботи із списком можуть виступати різні частини поля даних.

Наприклад, якщо створюється лінійний список із записів, які містять прізвище, рік народження, стаж роботи та стать, будь-яка частина запису може виступати як ключ: при впорядковувань списку за алфавітом ключем буде прізвище, а при пошуку, наприклад, ветеранів праці ключем буде стаж. Ключі різних елементів списку можуть збігатися.

Над списками можна виконувати наступні операції:

– початкове формування списку (створення першого елементу);
– додавання елементу в кінець списку;
– читання елементу із заданим ключем;
– вставка елементу в задане місце списку (до або після елементу із заданим ключем);
– видалення елементу із заданим ключем;
– впорядковування списку по ключу.

Розглянемо двонаправлений лінійний список. Для формування списку і роботи з ним потрібно мати принаймні одну вказівку – на початок списку. Зручно завести ще одну вказівку – на кінець списку. Для простоти допустимо, що список складається з цілих чисел, тобто опис елементу списку виглядає таким чином:

```
struct Node
{
int d;
Node *next;
Node *prev;
};
```

Нижче приведена програма, яка формує список з 5 чисел, додає число в список, видаляє число із списку і виводить список на екран. Вказівка на початок списку позначена pbeg, на кінець списку – pend, допоміжні вказівки – pv та pkey.

```
#include <iostream>
using namespace std;

struct Node
{
 int d;
 Node *next;
 Node *prev;
};

Node* first(int d);
void add(Node **pend, int d);
Node* find(Node * const pbeg, int i);
bool remove(Node **pbeg, Node **pend, int key);
Node* insert(Node * const pbeg, Node **pend,int key,int d);
```

```
void main()
{

// Формування першого елементу списку
Node *pbeg = first(1);
Node *pend = pbeg;
//Додавання в кінець списку чотирьох елементів 2, 3, 4 та 5
for (int i = 2; i<6; i++) add(&pend, i);
// Вставка елементу 200 після елементу 2
insert(pbeg, &pend, 2, 200);
// Видалення елементу 5
if(!remove (&pbeg, &pend, 5)) cout << "не знайдений";
Node *pv = pbeg;
while (pv)
{
// виведення списку на екран
cout << pv->d << " ";
pv = pv->next;
}
}

// Формування першого елементу
Node* first(int d)
{
 Node *pv = new Node;
 pv->d = d;
 pv->next = 0;
 pv->prev = 0;
 return pv;
}

// Додавання в кінець списку
void add(Node **pend, int d)
{
 Node *pv = new Node;
 pv->d = d;
 pv->next = 0;
 pv->prev = *pend;
 (*pend)->next = pv;
 *pend = pv;
}

// Пошук елементу по ключу
Node* find(Node * const pbeg, int d)
{
 Node *pv = pbeg;
```

```
 while (pv)
 {
  if(pv->d == d) break;
  pv = pv->next;
 }
 return pv;
}

// Видалення елементу
bool remove(Node **pbeg, Node **pend, int key)
{
 if(Node *pkey = find(*pbeg, key)) //1
 {
  if (pkey == *pbeg)
  { // 2
   *pbeg = (*pbeg)->next;
   (*pbeg)->prev =0;
  }
   else if (pkey == *pend)
   {// 3
    *pend = (*pend)->prev;
    (*pend)->next =0;
   }
  else
   { // 4
    (pkey->prev)->next = pkey->next;
    (pkey->next)->prev = pkey->prev;
   }
delete pkey;
return true; // 5
}
return false; // 6
}

// Вставка елементу
Node *insert (Node* const pbeg,Node **pend,int key,int d)
{
 if(Node *pkey = find(pbeg,key))
 {
  Node *pv = new Node;
  pv->d = d;
  // 1 - встановлення зв'язку нового вузла з наступним
  pv->next = pkey->next;
  // 2 - встановлення зв'язку нового вузла з попереднім
  pv->prev = pkey;
  // 3 - встановлення зв'язку попереднього вузла з новим
```

```
  pkey->next = pv;
  // 4 - встановлення зв'язку наступного вузла з новим
  if(pkey!=*pend) (pv->next)->prev = pv;
  // Оновлення вказівки на кінець списку
  // якщо вузол вставляється в кінець
 else *pend = pv;
  return pv;
 }
 return 0;
}
```

Результат роботи програми:

```
1 2 200 3 4
```

Всі параметри, не змінні всередині функцій, повинні передаватися з модифікатором const. Вказівки, які можуть змінитися (наприклад, при видаленні із списку останнього елементу вказівку на кінець списку потрібно скорегувати), передаються за адресою.

Розглянемо докладніше функцію видалення елементу із списку remove. Її параметрами є вказівки на початок і кінець списку і ключ елементу, який треба видалити. У рядку 1 виділяється пам'ять під локальну вказівку pkey, якій привласнюється результат виконання функції знаходження елементу по ключу find. Ця функція повертає вказівку на елемент у разі успішного пошуку та 0, якщо елементу з таким ключем в списку немає. Якщо pkey набуває ненульового значення, умова в операторі if стає істинною (елемент існує), і управління передається оператору 2, якщо немає – виконується повернення з функції із значенням false (оператор 6).

Видалення із списку відбувається по-різному залежно від того, знаходиться елемент на початку списку, в середині або в кінці. У операторі 2 перевіряється, чи знаходиться елемент, що видаляється, на початку списку – в цьому випадку слід скорегувати вказівку pbeg на початок списку так, щоб вона вказувала на наступний елемент в списку, адреса якого знаходиться в полі next першого елементу. Новий початковий елемент списку повинен мати в своєму полі вказівку на попередній елемент значення 0.

Якщо елемент, що видаляється, знаходиться в кінці списку (оператор 3), потрібно змістити вказівку pend кінця списку на попередній елемент, адресу якого можна отримати з поля prev останнього елементу. Крім того, потрібно обнулити для нового останнього елементу вказівку на наступний елемент. Якщо видалення походить з середини списку, то єдине, що треба зробити, – забезпечити двобічний зв'язок попереднього і наступного елементів. Після корегування вказівок пам'ять з-під елементу звільняється, і функція повертає значення true.

8.5. Шаблони функцій

За допомогою шаблонів можна створювати родові функції (**generic functions**). У родовій функції тип даних, з яким функція працює, задається у якості параметру. Це дозволяє одну й ту ж функцію використовувати з декількома різноманітними типами даних та без необхідності програмувати нову версію функції або класу для кожного конкретного типу даних. Таким чином шаблони дають можливість створювати багатократно використовуємі програми.

Родова функція визначає базовий набір операцій, які будуть застосовуватись до різних типів даних. Родова функція працює з тим же типом даних, який вона отримує у якості параметру. Як відомо, багато алгоритмів логічно однакові, незалежно від того, для обробки яких типів даних вони призначені. Наприклад, алгоритм швидкого сортування однаковий як для масивів цілих, так і для масивів дійсних чисел. Це той випадок, коли данні, що сортуються відрізняються тільки по типам. За допомогою створення родової функції можна незалежно від типу даних визначити сутність алгоритму. Після того як це зроблено, компілятор автоматично генерує правильний код для фактично використовуваного при виконанні функції типу даних. При створенні родової функції – створюється функція, яка може автоматично перевантажуватися сама.

Родова функція створюється за допомогою ключового слова **template**. Нижче представлено типову форму визначення функції-шаблону:

template <class Фтип> повертаєме_значення ім'я_функції (список параметрів)
{
//тіло функції
}

Тут замість ***Фтип*** вказується тип використовуваних функцією даних. Цей тип можна вказувати усередині визначення функції. Однак, він є фіктивним, його компілятор автоматично замінить реальним типом даних при створенні конкретної версії функції.

Наведемо приклад програми в якій створюється родова функція, яка міняє місцями значення двох змінних, що передаються їй в якості параметрів. Оскільки процес обміну двох значень не залежить від типу змінних – його реалізація ефективна за допомогою родової функції.

```
#include <iostream>
using namespace std;

//Функція-шаблон
template <class A> void swap1(A&x, A&y)
{
```

```
 A temp;
 temp = x;
 x = y;
 y = temp;
}

void main()
{
 int a, b;
 a = 5; b = 10;
 double x, y;
 x = 5.8; y = 10.27;

 cout<<"a = "<<a<<"  b = "<<b<<"\n";
 cout<<"x = "<<x<<"  y = "<<y<<"\n";

 swap1(a, b); //обмін цілих чисел
 swap1(x, y); //обмін дійсних чисел

 cout<<"a = "<<a<<"  b = "<<b<<"\n";
 cout<<"x = "<<x<<"  y = "<<y<<"\n";
}
```

Результат виконання програми наступний:

```
a = 5        b = 10
x = 5.8      y = 10.27
a = 10   b = 5
x = 10.27    y = 5.8
```

Ключове слово **template** використовується для визначення родової функції. Рядок:

```
template <class A> void swap1(A&x, A&y)
```

повідомляє компілятору дві речі: по-перше, створюється шаблон, та по-друге, починається визначення родової функції. Тут **A** – родовий тип даних. Після рядка з ключовим словом **template** функція **swap()** оголошується з типом даних **A** значень, що обмінюються. У функції **main()** функція **swap1()** викликається з двома різними типами даних: цілими та дійсними. Оскільки функція **swap1()** – родова функція, компілятор автоматично створює дві її версії: одну для обміну цілих значень, іншу для обміну дійсних значень.

За допомогою інструкції **template** можна визначити більше одного родового типу даних. Наприклад, у даній програмі створюється родова функція, в якій є два родові типи даних:

```
#include <iostream>
using namespace std;

//Функція-шаблон
template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
 cout<<"x = "<<x<<"  y = "<<y<<"\n";
}

void main()
{
 myfunc(10,"Hello");
 myfunc(5.15,13L);
}
```

Результат виконання програми наступний:

```
x = 10   y = Hello
x = 5.15     y = 13
```

У даному прикладі при генерації конкретних екземплярів функції **myfunc()**, типи даних **type1** та **type2** замінюються компілятором на типи даних **int** та **char*** або **double** та **long** відповідно.

8.6 Загальні відомості шаблонів класів

Шаблони класів надають можливість створювати класи, які параметризуються. Клас, який параметризується, створює сімейство споріднених класів, які можна застосовувати до будь-якого типу даних, передаваного як параметр. Найбільш широке застосування шаблони знаходять при створенні *контейнерних класів.* Контейнерним називається клас, який призначений для зберігання якимнебудь чином організованих даних і роботи з ними. Стандартна бібліотека C++ містить безліч контейнерних класів для організації структур даних різного вигляду (вони описані в розділі "Контейнерні класи").

Перевага використання шаблонів полягає в тому, що як тільки алгоритм роботи з даними визначений і відлагоджений, він може застосовуватися до будь-яких типів даних без переписування коду.

Оскільки списки часто застосовуються для організації даних, зручно описати список у вигляді класу, а оскільки може потрібно зберігати дані різних типів, цей клас має бути параметризованим.

Для засвоєння матеріалу доцільно розглянути версію класу, який не параметризується, "список".

Список складається з вузлів, зв'язаних між собою за допомогою вказівок. Кожен вузол зберігає ціле число, що є ключем списку. Опишемо допоміжний клас для представлення одного вузла списку:

```
class Node
 {
  public:
  int d;
  Node* next;
  Node* prev;
  Node (int dat = 0)
  {
   d = dat;
   next = prev =0;
  }
 };
```

Оскільки цей клас буде описаний всередині класу, що представляє список, поля для простоти доступу із зовнішнього класу зроблені доступними (**public**). Це дозволяє обійтися без функцій доступу і зміни полів. Назвемо клас списку **List**:

```
class List
{
 class Node
 {
 ...
 };
 Node*pbeg,*pend;//Вказівки на початок та кінець списку
 public:
 List() {pbeg = 0;pend=0;} //Конструктор
 ~List(); // Деструктор
 void add(int d); // Додавання вузла у кінець списку
 Node * find(int i); // Пошук вузла по ключу
 bool remove (int key); // Видалення вузла

 // Вставка вузла d після вузла с ключем key
 Node* insert(int key, int d);
 void print();// Друк списку у прямому напрямі
 void print_back();// Друк списку в оберненому напрямі
 };
```

Розглянемо реалізацію методів класу. Метод **add** виділяє пам'ять під новий об'єкт типу **Node** і приєднує його до списку, оновлюючи вказівки на його початок та кінець:

```
void List<Data>::add(Data d)
{
 Node* pv = new Node(d);//Виділення пам’яті під новий вузол
 if(pbeg == 0)pbeg = pend = pv; // Перший вузол списку
 else
{
 // Зв’язування нового вузла з попереднім
 pv->prev = pend;
 pend->next = pv;
 pend = pv; // Оновлення вказівки на кінець списку
 }
}
```

Метод **find()** виконує пошук вузла із заданим ключем і повертає вказівку на нього у разі успішного пошуку і 0 у разі відсутності такого вузла в списку:

```
Node* List::find(int d)
{
 Node *pv = pbeg;
 while(pv)
 {
  if(pv->d == d)break;
  pv = pv->next;
 }
 return pv;
}
```

Метод **insert()** вставляє в список вузол після вузла з ключем **key** і повертає вказівку на вставлений вузол. Якщо такого вузла в списку немає, вставка не виконується і повертається значення 0:

```
Node * List::insert(int key, int d)
{
 if(Node *pkey = find(key))
 { // Пошук вузла з ключем key
 //Виділення пам’яті під новий вузол та його ініціалізація:
 Node *pv = new Node(d);
 // Встановлення зв’язку нового вузла з наступним:
```

```
 pv->next = pkey->next;
 // Встановлення зв'язку нового вузла з попереднім:
 pv->prev = pkey;
 // Встановлення зв'язку попереднього вузла з новим:
 pkey->next = pv;
 // Встановлення зв'язку наступного вузла з новим:
 if (pkey != pend) (pv->next)->prev = pv;
 // Оновлення вказівки на кінець списку,
 // якщо вузол вставляється в кінець:
 else pend = pv;
 return pv;
 }
 return 0;
}
```

Метод **remove** видаляє вузол із заданим ключем із списку і повертає значення **true** у разі успішного видалення і **false**, якщо вузол з таким ключем в списку не знайдений:

```
bool List::remove(int key)
{
 if(Node *pkey = find(key))
 {
 if (pkey == pbeg)
 { // Видалення з початку списку
  pbeg = pbeg->next;
  pbeg->prev = 0;
}
else if (pkey == pend)
{ // Видалення з кінця списку
  pend = pend->prev;
  pend->next = 0;
}
else
{ // Видалення з середини списку
(pkey->prev)->next = pkey->next;
(pkey->next)->prev = pkey->prev;
}
 delete pkey;
 return true;
}
 return false;
}
```

Методи друку списку в прямому і зворотному напрямі поелементно переглядають список, переходячи по відповідних посиланнях:

```
void List ::print()
{
  Node* pv = pbeg;
  cout << char(10) << " list ";

    while(pv)
    {
     cout << pv->d << "  ";
     pv = pv->next;
    }
    cout << char(10);
}
void List::print_back()
{
  Node* pv = pend;
  cout << char(10) << " list back  ";

    while(pv)
    {
     cout << pv->d << "  ";
     pv = pv->prev;
    }
    cout << char(10);
}
```

Деструктор списку звільняє пам'ять з-під всіх його елементів:

```
List::~List()
{
 if(pbeg!=0)
 {
  Node* pv = pbeg;
  while(pv)
  {
   pv = pv->next;
   delete pbeg;
   pbeg = pv;
  }
 }
}
```

Приклад програми, яка використовує клас **List**:

```
void main()
{
 List L;
 L.add(5);
 L.add(4);
 L.insert(5,25);
 L.insert(25,100);
 L.print();
 L.print_back();
 L.remove(25);
 L.print();
 return;
}
```

Результат роботи програми:

```
list  5  25  100  4
list  back  4  100  25  5
list  5  100  4
```

8.7 Створення шаблонів-класів

Клас **List** призначений для зберігання цілих чисел. Щоб зберігати в ньому дані будь-якого типу, потрібно описати цей клас як шаблон і передати тип в якості параметру.

Синтаксис опису шаблону:

template <опис_параметрів_шаблону> визначення_класу:

Параметри шаблону перераховуються через кому. В якості параметрів можуть використовуватися типи, шаблони і змінні.

Типи можуть бути як стандартними, так і визначеними користувачем. Для їх опису використовується ключове слово **class**. Усередині шаблону параметр типу може застосовуватися в будь-якому місці, де допустимо використовувати специфікацію типу, наприклад:

```
template <class Data> class List
{
 class Node
{
   public:
```

```
    Data d;
    Node* next;
    Node* prev;
     Node (Data dat = 0)
   {
     d = dat;
     next = 0;
     prev = 0;
   }
};
```

Клас **Data** можна розглядати як формальний параметр, на місце якого при компіляції буде підставлений конкретний тип даних. Для будь-яких параметрів шаблону можуть бути задані значення за умовчанням, наприклад:

```
template<class T> class myarray {/*..*/};
...
template<class K,class V,tempiate<class T>class C=myarray>
class Map
{
 C<K> key;
 C<V> value;
 ...
};
```

Область дії параметра шаблону – від точки опису до кінця шаблону, тому параметр можна використовувати при описі наступних за ним, наприклад:

```
template<class T, T* p, class U = T> class X{ /*...*/};
```

Методи шаблону класу автоматично стають шаблонами функцій. Якщо метод описується поза шаблоном, його заголовок повинен мати наступні елементи:

```
template <опис_параметрів_шаблону>
поверт_тип ім'я_класу <параметри_шаблону>::
ім'я функції (список_параметрів функції)
```

Опис параметрів шаблону в заголовку функції повинен відповідати шаблону класу, при цьому імена параметрів можуть не збігатися. Простіше розглянути синтаксис опису методів шаблону на прикладі:

```
template <class Data> void List<Data>::print()
{ / * тіло функції */ }
```

Тут <**class Data**> – опис параметра шаблону, **void** – тип значення, яке повертається функцією. **List** – ім'я класу, <**Data**> – параметр шаблону, **print** – ім'я функції без параметрів.

У разі декількох параметрів порядок їх проходження в описі параметрів і параметрах шаблону має бути один і той же, наприклад:

```
template<class T1, class T2> struct A
{
 void f1 ();
};

template<class T2, class T1> void A<T2, T1 >::f1(){...}
```

Нижче перераховані правила опису шаблонів.

1. Локальні класи не можуть містити шаблони як свої елементи.
2. Шаблони методів не можуть бути віртуальними.
3. Шаблони класів можуть містити статичні елементи, дружні функції і класи.
4. Шаблони можуть бути похідними як від шаблонів, так і від звичайних класів, а також бути базовими і для шаблонів, і для звичайних класів.
5. Усередині шаблону не можна визначати **friend**-шаблони.

Як приклад шаблону розглянемо повний опис класу двохзв'язного списку **List** з параметрами.

```
#include <iostream>
using namespace std;

template <class Data> class List
{
 public:
 class Node
{
 public:
 Data d;
 Node* next;
 Node* prev;
 Node (Data dat = 0)
 {
  d = dat; next = 0; prev = 0;
 }
};
 Node* pbeg,*pend;
```

```
public:
 List(){pbeg = 0;pend=0;}
 ~List();
 void add(Data d);
 Node* find(Data i);
 bool remove (Data key);
 Node* insert(Data key, Data d);
 void print();
 void print_back();
};

template <class Data>
List<Data>::~List()
{
 if(pbeg!=0)
 {
  Node* pv = pbeg;
  while(pv)
  {
   pv = pv->next;
   delete pbeg;
   pbeg = pv;
  }
 }
}

template <class Data>
void List<Data>::print()
{
 Node* pv = pbeg;
 cout << char(10) << " list ";
 while(pv)
{
 cout << pv->d << "  ";
 pv = pv->next;
}
cout << char(10);
}

template <class Data>
void List<Data>::print_back()
{
```

```
 Node* pv = pend;
 cout << char(10) << " list back  ";
 while(pv)
{
 cout << pv->d << "  ";
 pv = pv->prev;
}
cout << char(10);
}

template <class Data>
void List<Data>::add(Data d)
{
 Node* pv = new Node(d);
 if(pbeg == 0)pbeg = pend = pv;
 else
{
 pv->prev = pend;
 pend->next = pv;
 pend = pv;
}
}

template <class Data>
typename List<Data>:: Node*
List<Data>::insert(Data key, Data d)
{
 if(Node*pkey = find(key))
{
 Node* pv = new Node(d);
 pv->next = pkey->next;
 pv->prev = pkey;
 pkey->next = pv;
 if(pkey!=pend)(pv->next)->prev = pv;
 else pend = pv;
 return pv;
}
 return 0;
}
```

```
template <class Data>
typename List<Data>::Node*
List<Data>::find(Data d)
{
 Node *pv = pbeg;
 while(pv)
 {
  if(pv->d == d)break;
  pv = pv->next;
 }
 return pv;
}

template <class Data>
bool List<Data>::remove(Data key)
{
 if(Node*pkey = find(key))
{
 if (pkey == pbeg)
 {
  pbeg = pbeg->next;
  pbeg->prev = 0;
 }
else if (pkey == pend)
{
 pend = pend->prev;
 pbeg->next = 0;
}
else
{
 (pkey->prev)->next = pkey->next;
 (pkey->next)->prev = pkey->prev;
}
 delete pkey;
 return true;
}
 return false;
}

void main()
{
 List<int> List_int;
```

```
 List_int.add(5);
 List_int.add(4);
 List_int.insert(5,25);
 List_int.insert(25,100);
 List_int.print();
 List_int.print_back();
 List_int.remove(25);
 List_int.print();
}
```

Результат роботи програми:

```
list  5  25  100  4
list  back  4  100  25  5
list  5  100  4
```

У функції **main()** створюється за допомогою шаблону об'єкт **List_int** (цей процес називається інстанціюванням). У загальному вигляді при описі об'єкту після імені шаблону в кутових дужках перераховуються його аргументи:

```
ім'я_шаблону <аргументи>
ім'я_об'єкта [(параметри_конструктору)];
```

При використанні параметрів шаблону за умовчанням список аргументів може виявитися порожнім, при цьому кутові дужки опускати не можна:

```
tempiate<class T = char> class String;
String <>* p;
```

Для спрощення використання шаблонів класів можна застосувати перейменування типів за допомогою **typedef**:

```
typedef List <double> Ldb1;
Ldb1 List_double;
```

Ми розглянули приклад коли клас **Node** знаходиться усередині класу **List**. Якщо ці класи представлені роздільно, тоді опис класу двохзв'язного списку матиме наступний вигляд:

```
#include <iostream>
using namespace std;

template <class Data> class Node
{
 public:
```

```
 Data d;
 Node* next;
 Node* prev;
 Node (Data dat = 0)
 {
  d = dat;
  next = 0;
  prev = 0;
 }
};

template <class Data> class List
{
 Node<Data>* pbeg,*pend;
 public:
 List(){pbeg = 0;pend=0;}
 ~List();
 void add(Data d);
 Node<Data>* find(Data i);
 bool remove (Data key);
 Node<Data>* insert(Data key, Data d);
 void print();
 void print_back();
};

template <class Data>
List<Data>::~List()
{
 if(pbeg!=0)
{
 Node<Data>* pv = pbeg;
 while(pv)
 {
  pv = pv->next;
  delete pbeg;
  pbeg = pv;
 }
}
}
```

```
template <class Data>
void List<Data>::print()
{
 Node<Data>* pv = pbeg;
 cout << char(10) << " list ";

 while(pv)
 {
  cout << pv->d << "  ";
  pv = pv->next;
 }
 cout << char(10);
}

template <class Data>
void List<Data>::print_back()
{
 Node<Data>* pv = pend;
 cout << char(10) << " list back  ";
 while(pv)
 {
  cout << pv->d << "  ";
  pv = pv->prev;
 }
 cout << char(10);
}

template <class Data>
void List<Data>::add(Data d)
{
 Node<Data>* pv = new Node<Data>(d);
 if(pbeg == 0)pbeg = pend = pv;
 else
 {
  pv->prev = pend;
  pend->next = pv;
  pend = pv;
 }
}
```

```
template <class Data>
Node<Data>* List<Data>::insert(Data key,Data d)
{
 if(Node<Data>*pkey = find(key))
 {
  Node<Data>* pv = new Node<Data>(d);
  pv->next = pkey->next;
  pv->prev = pkey;
  pkey->next = pv;
  if(pkey!=pend)(pv->next)->prev = pv;
  else pend = pv;
  return pv;
 }
 return 0;
}

template <class Data>
Node<Data>* List<Data>::find(Data d)
{
 Node<Data> *pv = pbeg;
 while(pv)
 {
  if(pv->d == d)break;
  pv = pv->next;
 }
 return pv;
}

template <class Data>
bool List<Data>::remove(Data key)
{
 if(Node<Data>*pkey = find(key))
 {
  if (pkey == pbeg)
 {
  pbeg = pbeg->next;
  pbeg->prev = 0;
 }
 else if (pkey == pend)
 {
  pend = pend->prev;
  pbeg->next = 0;
```

```
 }
 else
 {
  (pkey->prev)->next = pkey->next;
  (pkey->next)->prev = pkey->prev;
 }
 delete pkey;
 return true;
 }
 return false;
}
```

При визначенні синтаксису шаблону було сказано, що в нього, окрім типів і шаблонів, можуть передаватися *змінні.* Вони можуть бути цілого або перерахованого типу, а також вказівками або посиланнями на об'єкт або функцію. У тілі шаблону вони можуть застосовуватися в будь-якому місці, де допустимо використовувати константний вираз. Як приклад створимо шаблон класу, що містить блок пам'яті певної довжини і типу:

```
template <class Type, int kol>
class Block
{
 public:
 Block(){p = new Type [kol];}
 ~Block(){delete [] p;}
 operator Type * ();
 protected:
 Type * p;
};
template <class Type, int kol>
Block <Type, kol>:: operator Type *()
{
 return p;
}
```

В даному випадку опис об'єкту має вигляд:

```
Block <char, 128> buf;
Block <monstr, 100> stado;
```

Після створення та відлагодження шаблони класів зручно поміщати в заголовні файли.

8.8 Спеціалізація шаблонів класів

Кожна версія класу або функції, що створюється за шаблоном, містить однаковий базовий код; змінюється тільки те, що пов'язане з параметрами шаблону. При цьому ефективність роботи версій, що створюються для різних типів даних, може сильно розрізнятися.

Якщо для якого-небудь типу даних існує ефективніший код, можна або передбачити для цього типу спеціальну реалізацію окремих методів, або повністю перевизначити (спеціалізувати) шаблон класу.

Для *спеціалізації методу* потрібно визначити варіант його коду, вказавши в заголовку конкретний тип даних. Наприклад, якщо заголовок узагальненого методу **print** шаблону **List** має вигляд

```
template <class Data> void List <Data>::print();
```

спеціалізований метод для виведення списку символів виглядатиме таким чином:

```
void List <char>::print()
{
... // Тіло спеціалізованого варіанту метода print
}
```

Якщо в програмі створити екземпляр шаблону **List** типу **char**, відповідний варіант методу буде викликаний автоматично.

При *спеціалізації цілого класу* після опису узагальненого варіанту класу поміщається повний опис спеціалізованого класу, при цьому потрібно заново визначити всі його методи. Допустимо, потрібно спеціалізувати шаблон **Block**, описаний вище, для зберігання 100 цілих величин:

```
class Block <int, 100>
{
public:
 Block(){p = new int [100];}
 ~Block(){delete [ ] p;}
 operator int *( );
 protected:
 int * p;
};

Block <int, 100>:: operator int *()
{
 return p;
}
```

При визначенні екземплярів шаблону **Block** з параметрами **int** і 100 буде задіяний спеціалізований варіант.

8.9 Переваги та недоліки шаблонів

Шаблони є могутнім і ефективним засобом поводження з різними типами даних, який можна назвати параметричним поліморфізмом, а також забезпечують безпечне використання типів, на відміну від макросів препроцесора. Проте слід мати на увазі, що програма, що використовує шаблони, містить повний код для кожного породженого типу, що може збільшити розмір виконуваного файлу. Крім того, з деякими типами даних шаблони можуть працювати менш ефективно, чим з іншими. В цьому випадку має сенс використовувати спеціалізацію шаблону.

Стандартна бібліотека C++ надає великий набір шаблонів для різних способів організації зберігання і обробки даних (див. розділ "Контейнерні класи").

Завдання

Перелік варіантів завдань:

Варіант 1

Скласти програму, яка містить динамічну інформацію про наявність автобусів в автобусному парку.

Відомості про кожен автобус містять:

• номер автобуса;

• прізвище і ініціали водія;

• номер маршруту.

Програма повинна забезпечувати:

• початкове формування даних про всі автобуси в парку у вигляді списку;

• при виїзді кожного автобуса з парку вводиться номер автобуса, і програма видаляє дані про цей автобус із списку автобусів, що знаходяться в парку, і записує ці дані в список автобусів, що знаходяться на маршруті;

• при в'їзді кожного автобуса до парку вводиться номер автобуса, і програма видаляє дані про цей автобус із списку автобусів, що знаходяться на маршруті, і записує ці дані в список автобусів, що знаходяться в парку;

• за запитом видаються відомості про автобуси, що знаходяться в парку, або про автобуси, що знаходяться на маршруті.

Варіант 2

Скласти програму, яка містить поточну інформацію про книги в бібліотеці.

Відомості про книги містять:

• номер УДК;

• прізвище і ініціали автора;

• назва;

• рік видання;
• кількість екземплярів даної книги в бібліотеці.
Програма повинна забезпечувати:
• початкове формування даних про всі книги в бібліотеці у вигляді двійкового дерева;
• додавання даних про книги, що знов надходять в бібліотеку;
• видалення даних про списувані книги;
• за запитом видаються відомості про наявність книг в бібліотеці, впорядковані по роках видання.

Варіант 3

Скласти програму, яка містить поточну інформацію про заявки на авіаквитки.
Кожна заявка містить:
• пункт призначення;
• номер рейсу;
• прізвище і ініціали пасажира;
• бажану дату вильоту.
Програма повинна забезпечувати:
• зберігання всіх заявок у вигляді списку;
• додавання заявок в список;
• видалення заявок;
• виведення заявок по заданому номеру рейсу та даті вильоту;
• виведення всіх заявок.

Варіант 4

Скласти програму, яка містить поточну інформацію про заявки на авіаквитки.
Кожна заявка містить:
• пункт призначення;
• номер рейсу;
• прізвище та ініціали пасажира;
• бажану дату вильоту.
Програма повинна забезпечувати:
• зберігання всіх заявок у вигляді двійкового дерева;
• додавання та видалення заявок;
• по заданому номеру рейсу та даті вильоту виведення заявок з їх подальшим видаленням;
• виведення всіх заявок.

Варіант 5

Скласти програму, яка містить поточну інформацію про книги в бібліотеці.
Відомості про книги містять:
• номер УДК;
• прізвище та ініціали автора;
• назва;

• рік видання;

• кількість екземплярів даної книги в бібліотеці.

Програма повинна забезпечувати:

• початкове формування даних про всі книги в бібліотеці у вигляді списку;

• при взятті кожної книги вводиться номер УДК, і програма зменшує значення кількості книг на одиницю або видає повідомлення про те, що необхідної книги в бібліотеці немає, або необхідна книга знаходиться на руках;

• при поверненні кожної книги вводиться номер УДК, і програма збільшує значення кількості книг на одиницю;

• за запитом видаються відомості про наявність книг в бібліотеці.

Варіант 6

Скласти програму, яка містить динамічну інформацію про наявність автобусів в автобусному парку.

Відомості про кожен автобус містять:

• номер автобуса;

• прізвище і ініціали водія;

• номер маршруту;

• ознака того, де знаходиться автобус - на маршруті або в парку.

Програма повинна забезпечувати:

• початкове формування даних про всі автобуси у вигляді списку;

• при виїзді кожного автобуса з парку вводиться номер автобуса, і програма встановлює значення ознаки "автобус на маршруті";

• при в'їзді кожного автобуса до парку вводиться номер автобуса, і програма встановлює значення ознаки "автобус в парку";

• за запитом видаються відомості про автобуси, що знаходяться в парку, або про автобуси, що знаходяться на маршруті.

Варіант 7

Скласти програму, яка відшукує прохід по лабіринту.

Лабіринт представляється у вигляді матриці, що складається з квадратів. Кожен квадрат або відкритий, або закритий. Вхід в закритий квадрат заборонений. Якщо квадрат відкритий, то вхід в нього можливий з боку, але не з кута. Кожен квадрат визначається його координатами в матриці.

Програма знаходить прохід через лабіринт, рухаючись від заданого входу. Після відшукання проходу програма виводить знайдений шлях у вигляді координат квадратів. Для зберігання шляху використовувати стек.

Варіант 8

Гаражна стоянка має одну смугу стоянки, причому в'їзд і виїзд знаходяться в одному кінці смуги. Якщо власник автомашини приходить забрати свій автомобіль, який не є найближчим до виходу, то всі автомашини, що загороджують проїзд, видаляються, машина даного власника виводиться із стоянки, а інші машини повертаються на стоянку в початковому порядку.

Написати програму, яка моделює процес прибуття і від'їзду машин. Прибуття або від'їзд автомашини задається командним рядком, який містить ознаку прибуття або від'їзду і номер машини. Програма повинна виводити повідомлення при прибутті або виїзді будь-якої машини. При виїзді автомашини зі стоянки повідомлення повинне містити число разів, яке машина видалялася із стоянки для забезпечення виїзду інших автомобілів.

Варіант 9

Скласти програму, яка моделює заповнення гнучкого магнітного диска.

Загальний об'єм пам'яті на диску 360 Кбайт. Файли мають довільну довжину від 18 байт до 32 Кбайт. В процесі роботи файли або записуються на диск, або видаляються з нього. На початку роботи файли записуються підряд один за одним. Після видалення файлу на диску утворюється вільна ділянка пам'яті, і знов записуваний файл або розміщується на вільній ділянці, або, якщо файл не вміщається у вільну ділянку, розміщується після останнього записаного файлу. У разі, коли файл перевершує довжину найбільшої вільної ділянки, видається аварійне повідомлення. Вимога на запис або видалення файлу задається в командному рядку, який містить ім'я файлу, його довжину в байтах, ознака запису або видалення. Програма повинна видавати по запиту зведення про зайняті і вільні ділянки пам'яті на диску.

Вказівка: слід створити список зайнятих ділянок і список вільних ділянок пам'яті на диску.

Варіант 10

У файловій системі каталог файлів організований як лінійний список. Для кожного файлу в каталозі містяться наступні відомості:

• ім'я файлу;
• дата створення;
• кількість звернень до файлу.

Скласти програму, яка забезпечує:

• початкове формування каталогу файлів;
• виведення каталогу файлів;
• видалення файлів, дата створення яких менше заданої;
• вибірку файлу з найбільшою кількістю звернень.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

Варіант 11

Предметний покажчик організований як лінійний список.

Кожна компонента покажчика містить слово і номери сторінок, на яких це слово зустрічається. Кількість номерів сторінок, що відносяться до одного слова, від одного до десяти.

Скласти програму, яка забезпечує:

• початкове формування предметного покажчика;

• виведення предметного покажчика;

• виведення номерів сторінок для заданого слова.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

Варіант 12

Текст допомоги для деякої програми організований як лінійний список.

Кожна компонента тексту допомоги містить термін (слово) і текст, що містить пояснення до цього терміну. Кількість рядків тексту, що відносяться до одного терміну, від однієї до п'яти.

Скласти програму, яка забезпечує:

• початкове формування тексту допомоги;

• виведення тексту допомоги;

• виведення пояснюючого тексту для заданого терміну.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

Варіант 13

Картотека в бюро обміну квартир організована як лінійний список.

Відомості про кожну квартиру містять:

• кількість кімнат;

• поверх;

• площа;

• адреса.

Скласти програму, яка забезпечує:

• початкове формування картотеки;

• введення заявки на обмін;

• пошук в картотеці відповідного варіанту: при рівності кількості кімнат та поверху і відмінності площ в межах 10% виводиться відповідна картка і видаляється із списку, в іншому випадку заявка, що поступила, включається в список;

• виведення всього списку.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

Варіант 14

Англо-російський словник побудований як двійкове дерево.

Кожна компонента містить англійське слово, відповідне йому російське слово і лічильник кількості звернень до даної компоненті. Первинне дерево формується згідно англійському алфавіту. В процесі експлуатації словника при кожному зверненні до компоненті в лічильник звернень додається одиниця.

Скласти програму, яка:

• забезпечує початкове введення словника з конкретними значеннями лічильників звернень;

• формує нове представлення словника у вигляді двійкового дерева по наступному алгоритму: а) у старому словнику шукається компонента з найбільшим значенням лічильника звернень; б) знайдена компонента заноситься в новий словник та видаляється із старого; в) перехід до п. а) до вичерпання початкового словника;

• проводить виведення початкового і нового словників.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

Варіант 15

Анкета для опиту населення містить дві групи питань.

Перша група містить відомості про респондента:

• вік;

• стать;

• освіта (початкова, середня, вища).

Друга група містить власне питання анкети, відповідь на які або ТАК, або НІ.

Скласти програму, яка:

• забезпечує початкове введення анкет і формує з них лінійний список;

• на основі аналізу анкет видає відповіді на наступні питання: а) скільки чоловіків старше 40 років, що мають вищу освіту, відповіли ТАК на питання анкети; б) скільки жінок молодше 30 років, що мають середню освіту, відповіли НІ на питання анкети; в) скільки чоловіків молодше 25 років, що мають початкову освіту, відповіли ТАК на питання анкети;

• проводить виведення всіх анкет і відповідей на питання.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

Варіант 16

Скласти програму, яка містить поточну інформацію про книги в бібліотеці.

Відомості про книги містять:

• номер УДК;

• прізвище і ініціали автора;

• назва;

• рік видання;

• кількість екземплярів даної книги в бібліотеці.

Програма повинна забезпечувати:

• початкове формування даних про всі книги в бібліотеці у вигляді списку;

• додавання даних про книги, що знов надходять в бібліотеку;

• видалення даних про списувані книги;

• за запитом видаються відомості про наявність книг в бібліотеці, впорядковані по роках видання.

Варіант 17

На міжміській телефонній станції картотека абонентів, яка містить відомості про телефони і їх власників, організована як лінійний список.

Скласти програму, яка:

• забезпечує початкове формування картотеки у вигляді лінійного списку;

• проводить виведення всієї картотеки;

• вводить номер телефону і час розмови;

• виводить сповіщення на оплату телефонної розмови.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

Варіант 18

На міжміській телефонній станції картотека абонентів, яка містить відомості про телефони і їх власників, організована як двійкове дерево.

Скласти програму, яка:

• забезпечує початкове формування картотеки у вигляді двійкового дерева;

• проводить виведення всієї картотеки;

• вводить номер телефону і час розмови;

• виводить сповіщення на оплату телефонної розмови.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

Варіант 19

Автоматизована інформаційна система на залізничному вокзалі містить відомості про відправлення поїздів дальнього проходження. Для кожного поїзда вказується:

• номер поїзда;

• станція призначення;

• час відправлення.

Дані в інформаційній системі організовані у вигляді лінійного списку.

Скласти програму, яка:

• забезпечує первинне введення даних в інформаційну систему і формування лінійного списку;

• проводить виведення всього списку;

• вводить номер поїзда і виводить всі дані про цей поїзд;

• вводить назву станції призначення і виводить дані про всі поїзди, які слідують до цієї станції.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

Варіант 20

Автоматизована інформаційна система на залізничному вокзалі містить відомості про відправлення поїздів дальнього проходження.

Для кожного поїзда вказується:

• номер поїзда;

• станція призначення;

• час відправлення.

Дані в інформаційній системі організовані у вигляді двійкового дерева.

Скласти програму, яка:

• забезпечує первинне введення даних в інформаційну систему і формування двійкового дерева;

• проводить виведення всього дерева;

• вводить номер поїзда і виводить всі дані про цей поїзд;

• вводить назву станції призначення і виводить дані про всі поїзди, які слідують до цієї станції.

Програма повинна забезпечувати діалог за допомогою меню і контроль помилок при введенні.

РОЗДІЛ 9. МОДУЛЬНІ ПРОГРАМИ
(проектування об’єктно-орієнтованого програмування)

9.1 Короткі відомості

Метою вивчення даного розділу є ознайомлення з технікою організації багатомодульних програм. Розглядається розділення класу на визначення і реалізацію, зокрема особливості організації багатомодульних програм з використанням шаблонів. Вивчаються деякі прийоми підвищення незалежності модулів – делегування. Розглядаються простори імен.

Великі програми потрібно розбивати на частини. Окрема частина великої програми називається *модулем.* У C++ відсутні конструкції для позначення модуля. У новому стандарті визначено поняття одиниці трансляції. *Одиниця трансляції* – це окремий файл з початковим текстом на C++, який виходить після обробки препроцесором. Розбиття програми на окремі модулі вимагає подальшої збірки. У C++ збірка виконується засобами інтегрованого середовища. У інтегрованому середовищі створюється *проект* у складі якого перераховуються всі модулі.

Збірка здійснюється одним з двох способів.

1. Об'єднуються початкові тексти.
2. Програма збирається з об'єктних модулів; *об'єктний модуль* – це результат компіляції однієї одиниці трансляції початкового тексту.

У C++ об'єднання початкових текстів робиться за допомогою препроцесора. Недолік – компіляція програми може займати багато часу. Другий спосіб був створений для того, щоб уникнути компіляції всієї програми цілком. Такий спосіб називається *роздільною* трансляцією. Процес збірки повної програми з об'єктних модулів називається *компоновкою,* і виконує цю роботу програма-компонувальник, яка входить до складу системи програмування. Ця програма часто називається "лінкером" (від англійського слова **linker**), що, власне, і має на увазі "складальника". При компоновці в програму збираються не тільки розроблені модулі, але і *стандартні.* Стандартні модулі не транслюються разом з програмою – вони поставляються разом з системою у вигляді об'єктних модулів і об'єктних бібліотек.

Розділення програм на модулі вимагає узгодження визначень і оголошень в різних одиницях трансляції. У всіх одиницях трансляції мають бути узгоджені оголошення і визначення класів, функцій, змінних і констант, перерахувань, шаблонів і просторів імен.

9.2 Збірка вихідних текстів

Розроблений клас зберігається в окремому файлі **ім’я_файлу.cpp**. Наприклад, реалізацію двостороннього списку можна назвати ім'ям **List.cpp**. Інший модуль-файл з ім'ям **main.cpp** може містити код програми-клієнта, що використовує стек. При

створенні проекту засобами інтегрованого середовища файл **main.cpp** має бути *єдиним* в проекті.

Файл **List.cpp** повинен включати наступну конструкцію:

```
#ifndef _LIST    //  "вартовий" визначений?
#define _LIST    //  визначення "вартового"
class List
{   //визначення класу
};
#endif /*_LIST */   // кінець #ifndef
Перший рядок може бути записаний по-іншому,
#if !defined(_STACK)// "вартовий" визначений?
```

що еквівалентно попередньому варіанту. Така послідовність команд препроцесора називається "*вартовим" включення* і призначена для запобігання повторному включенню вмісту файлу в програму.

Приклад вмісту файлу **main.cpp**:

```
#include "List.cpp" // файл з реалізацією стеку
#include <iostream> // системний заголовок
using namespace std; // стандартний простір імен

void main()
{
 List L;
 L.add(5);
 L.add(4);
 L.insert(5,25);
 L.insert(25,100);
 L.print();
 L.print_back();
 L.remove(25);
 L.print();
}
```

Об'єднання модулів в єдину програму здійснюється препроцесором по директиві включення:

```
#include им'я_файлу
```

Ім'я файлу задається або у вигляді "файл", або у вигляді <файл>. У першому варіанті передбачається, що модуль-файл повинен знаходитися в поточному каталозі – там же, де знаходиться файл **main.cpp**. Можна помістити файл, що підключається,

в будь-який каталог, але тоді потрібно указувати його повне ім'я, наприклад

```
"c:\\Piter\\ch1\\List.cpp"
```

або

```
"c:/Chab/ch1/List.cpp"
```

Друга форма <файл> застосовується для включення стандартних бібліотечних файлів. Файл шукається препроцесором в стандартному каталозі інтегрованого середовища \include. У цей каталог можна поміщати і власні файли-модулі. Ім'я файлу потрібно указувати абсолютно точно. Пропуски усередині кутових дужок або усередині лапок – значущі, тому без необхідності додаткові пропуски писати не можна.

Імена стандартних файлів-заголовків рекомендується писати без розширення .h. Щоб попередити можливі конфлікти імен, заголовки бібліотек **C** слід починати з символу "**c**". У складі C++ є 32 системних файлів-заголовків:

<algorithm> <iomanip><list> <queue> <streambuf>
<bitset> <ios> <locale> <set> <string>
<complex> <iosfwd> <map> <sstream> <typeinfo>
<deque> <iostream><memory> <stack> <utility>
<exception> <istream><new> <stdexcept> <valarray>
<fstream> <iterator> <numeric><strstream> <vector>
<functional> <limits> <ostream>

та 18 стандартних файлів-заголовків, успадкованих від C:

<cassert> <siso646><csetjmp><cstdio> <ctime>
<cctype> <climits><csignal> <cstdlib> <cwchar>
<cerrno> <clocale><cstdarg><cstring> <cwctype>
<cfloat> <cmath> <cstddef>

9.3 Відділення інтерфейсу від реалізації

Зазвичай *інтерфейс класу* (який є його визначенням) відділяється від його реалізації. Інтерфейс класу і визначення статичних полів поміщаються у файл-заголовок ім’я_файлу.h (заголовок, "шапка" – від англійського header); "вартового" включення потрібно задати в заголовку. Реалізація методів класу виноситься в окремий файл ім’я_файлу.cpp. До інших модулів, які використовують даний клас, підключається тільки заголовок; заголовок підключається і до файлу з реалізацією методів.

```
/*TClass.h, файл з інтерфейсом класу */
// необхідні системні заголовки
#ifndef _STRAG   // "вартовий" визначений?
```

```
#define _STRAG   // визначення "вартового"

class TClass
{ // поля та прототипи методів
};
#endif /*_STRAG */                // кінець ifndef
// визначення статичних полів
/* TClass.cpp, файл с реалізацією класу */
// необхідні системні заголовки
#include "TClass.h"
// реалізація методів зовнішнім способом
```

У проект інтегрованого середовища включаються файл реалізації і файл з програмою-клієнтом. Файл-заголовок до складу проекту зазвичай не входить.

Розділення класу на інтерфейс і реалізацію не заважає спадкуванню. Клас-спадкоємець теж ділиться на інтерфейс-заголовок і реалізацію; заголовок базового класу підключається до заголовка класу-спадкоємця.

```
/* Base.h, файл з інтерфейсом базового класу */
// необхідні системні заголовки
#ifndef _STRAGBASE    // "вартовий" визначений?
#define _STRAGBASE    // визначення "вартового"

class Base
{ // поля та прототипи методів
};
#endif /* _STRAGBASE */           // кінець ifndef
// визначення статичних полів
/* Base.cpp. файл з реалізацією базового класу */
// необхідні системні заголовки
#include "Base.h"
// реалізація методів зовнішнім способом
/* Derive.h, файл з інтерфейсом класу-спадкоємця */
// необхідні системні заголовки
#include "Base.h"// для спадкування!
#ifndef _STRAGDERIVE  // "вартовий" визначений?
#define _STRAGDERIVE  // визначення "вартового"
class Derive: public Base
{ // поля та прототипи методів
};
#endif /* STRAGDERIVE */      // кінець ifndef
// визначення статичних полів
/* Derive.cpp. файл з реалізацією класу-спадкоємця */
```

```
// необхідні системні заголовки
#include "Derive.h"
// реалізація методів зовнішнім способом
```

У проект включаються файли реалізації базового класу, похідного і код програми-клієнта.

9.4 Шаблони та модульність. Простір імен

Шаблони не можна ділити на інтерфейс та реалізацію і транслювати окремо. Шаблон є тільки заготівкою для побудови коду: поки шаблон не інстанційований ("конкретизований конкретним типом"), об'єктний код з нього не транслюється – складальникові нічого робити. Модулі, що використовують шаблон повинні підключити файл з повним визначенням шаблону за допомогою #include. Такий спосіб організації коду з шаблонами називається моделлю включення.

Другий варіант організації коду з шаблонами – модель явного інстанціювання. Цей спосіб працює і для шаблонів функцій, і для шаблонів класів. В цьому випадку шаблони інстанціюються (тобто реалізуються, конкретизуються) за допомогою директиви явного інстанціювання.

Тут клас-шаблон, як і звичайний клас, ділиться на інтерфейс і реалізацію, і в проект включаються наступні файли:

– файл з визначенням інтерфейсу класу-шаблону;
– файл з реалізацією класу-шаблону;
– файл з директивами явного інстанціювання;
– файл з програмою-клієнтом.

Ділення програми на файли-модулі – це фізичне розділення програми на частини. Щоб розділити велику програму на логічно зв'язані частини, до C++ передбачені простори імен. У новому стандарті визначений стандартний простір імен **std**.

Простір імен може мати ім'я. Оголошення простору імен – це призначення імені для області, в якій будуть видні компоненти простору імен. Синтаксис іменованого оголошення виглядає так:

```
namespace ім'я
{ // оголошення і визначення
}
```

Ідентифікатор **namespace** є зарезервованим словом. Простір імен може містити оголошення і визначення змінних, функцій, класів, типів, шаблонів і так далі. Ці імена вважаються членами даного простору імен. Визначення мають бути в однині.

У простір імен можна включати і заголовні файли, наприклад:

```
namespace SPACE { #include "List.h" }
```

Доступ до елементів простору імен (у тій же одиниці трансляції, але поза ним, або в іншій одиниці трансляції) виконується за допомогою операції дозволу видимості (::). Ідентифікатор простору імен слугує кваліфікатором для імені компоненту:

<простір імен>::<ім'я компоненту>

Наприклад, імена із стандартного простору імен можна писати так:

```
std::cin
std::cerr
```

Щоб не супроводжувати імена кваліфікаторами, можна використовувати using-об'яви

```
using <простір імен>::<ім'я компоненту>
```

Слово **using**, так само як і **namespace**, є зарезервованим словом. Надалі оголошені імена можна використовувати в програмі без кваліфікатора. Для стандартних імен з простору **std** це виглядає так:

```
using std::cout;
using std::endl;
```

Можна оголосити доступним відразу весь простір імен за допомогою **using**-директиви

```
using namespace <простір імен>;
```

Надалі всі імена з вказаного простору імен можна писати без префікса. Ця директива діє до кінця одиниці трансляції. На інші файли її вплив не розповсюджується. Для використання будь-якого імені із стандартного простору імен **std** потрібно задати директиву

```
using namespace std;
```

Цю директиву можна писати всякий раз, коли у вас в програмі зустрічається включення стандартного заголовка із списку, вказаного в новому стандарті.

Будь-яку форму **using** можна використовувати усередині деякого простору імен з метою включення в нього імен з іншого простору. Не використовуйте глобальну директиву **using** в заголовних файлах!

У визначеннях членів імена з того ж простору імен дозволено використовувати без вказівки кваліфікатор – аналогічно тому, як в методах класу дозволено вживати будь-які імена з того ж класу без префікса.

Стандарт C++ дозволяє оголошувати синоніми імен (псевдоніми), наприклад:

```
namespace MFC = Microsoft_Foundation_Class;
```

Така декларація використовується для перевизначення довгих імен, придуманих розробниками бібліотек.

Простори імен можуть бути вкладеними, наприклад:

```
namespace External {// зовнішній простір імен
double a;
namespace Internal {// вкладений простір імен
void F() { a=7.2;}// працює External ::a
int a;
void G() { a++; }     // працює Internal ::a;
}
}
```

Для звернення до імен з вкладеного простору слід вказувати подвійний префікс-кваліфікатор, наприклад:

```
External :: Internal ::F();
```

Псевдоніми можна призначати і для вкладених просторів імен, наприклад:

```
namespace Borland_Builder {   // зовнішнє
/* члени зовнішнього namespace */
namespace Visual_Component_Library {   // вкладене
/* члени внутрішнього namespace */
}
}
// Псевдоніми namespace
// Зовнішній простір
namespace BB = Borland_Builder;
// Внутрішній простір задається з кваліфікатором
namespace VCL=Borland_Builder::Visual_Component_Library;
```

9.5 Фізичне розділення простору імен

При фізичному розділенні програми на модулі простір імен розбивається на частини. Але якщо в різних одиницях трансляції оголошений один і той же простір імен, він "склеюється" в єдине ціле. Саме таким чином визначений стандартний простір імен **std**.

Ця властивість "склеювання" простору імен добре підходить для організації бібліотеки. Аналогічно класам, початковий код розділяється на дві частини: інтерфейс та реалізацію. Інтерфейсна частина поміщається в окремий заголовний файл, а реалізація – в інший файл. У клієнтському коді підключається тільки заголовний файл.

Приклад:

```
/* vector.h */
namespace vector
{
const double pi = 3.141615926;
class vector {/*...*/};
vector operator+(const vector &vl, const vector &v2);
double scalar_product(const vector&vl, const vector&v2);
double length(const vector &v);
}

/* vector.cpp */
#include "vector.h"
namespace vector
{ // реалізація методів класу vector
vector operator+(const vector &vl, const vector &v2)
{/*...*/}
double scalar_product (const vector&v1,const vector &v2)
{/*...*/}

double length(const vector &v)
{/*...*/}
}
/* user.cpp. клієнтський код */
#include "vector.h"
// ...
void F(vector::vector &v)
{ // ...
double d = vector::length(v);
//...
}
```

Така організація простору імен аналогічна розділенню класу на інтерфейс та реалізацію.

9.6 Міжмодульні змінні та функції

Кожне ім'я, що позначає об'єкт, має деяку зону видимості (дії), в якій це ім'я оголошене і може використовуватися. У стандарті C++ визначені наступні зони видимості: оператор, прототип, блок, функція, клас, файл, простір імен. Імена в одній зоні видимості не мають бути однакові, але в різних областях вони можуть і збігатися. Імена, оголошені поза всіма зонами видимості, входять в глобальний простір імен. Так, всі імена Windows API входять в глобальний простір імен. Для звернення до глобального імені використовується операція дозволу контексту ::, оскільки за

умовчанням (без вказівки кваліфікаторів) завжди вибирається ім'я з "найменшою" зоною видимості.

Відповідно до правила одного визначення (one-definition rule – ODR), одиниця трансляції не повинна містити більш за одне визначення будь-якої змінної, функції, класу, перерахування і шаблону. Оголошень може бути декілька: оголошення додають деяке ім'я в дану зону видимості і використовуються для узгодження типів. Змінна, визначена у файлі поза всіма класами і функціями, називається глобальною. У іншому модулі-файлі вона має бути оголошена з ключовим словом **extern**.

```
// Модуль з визначенням змінної
int Global = 1;
// Модуль з оголошенням змінної
extern int Global;
```

Визначення функції включає тіло, а оголошенням є прототип. Визначення задається в одному модулі, а в інших можна використовувати тільки прототип, наприклад:

```
// файл з оголошенням функції
void f(void)
{ cout << "f()" << endl; }
// будь-який інший файл з прототипом функції
void f(void);
```

Слово **extern** писати не потрібно, хоча і не забороняється. Прототипи

```
void f(void);
extern void f(void);
```

є еквівалентними.

9.7 Ініціалізація глобальних об'єктів

Глобальні і статичні змінні компілятор розміщує в статичній пам'яті, і час життя таких змінних збігається з часом виконання програми. Статичні змінні ініціалізуються неявно (за умовчанням) до початку виконання функції **main()**. Ця ініціалізація називається статичною; явна ініціалізація, що задається програмістом, називається динамічною. Вбудовані типи за умовчанням ініціалізуються нулями. Для глобальних об'єктів невбудованих типів викликається конструктор за умовчанням (без аргументів). Якщо в класі його немає, виникає помилка трансляції. Конструктор ініціалізації застосовується для явної ініціалізації.

В рамках одного модуля порядок ініціалізації змінних вбудованих типів визначається порядком оголошення. Конструктори для створення і ініціалізації глобального об'єкту теж викликаються в порядку оголошень об'єктів. Деструктори викликаються перед завершенням програми в зворотному порядку.

Черговість статичної ініціалізації глобальних об'єктів, розміщених в різних одиницях трансляції, стандартом не визначена.

Змінні і функції, оголошені у файлі з атрибутом **static**, схильні до внутрішнього скріплення і є локальними в модулі, де визначені. Константи володіють властивістю внутрішнього скріплення за умовчанням. У різних файлах можна оголошувати глобальні константи з однаковими іменами. Щоб зробити константу, оголошену в одному файлі, видимою в іншому, потрібно використовувати слово **extern**.

```
// модуль з визначенням глобальної константи
extern const int a = 2;
// модуль з оголошенням тієї ж константи
extern const int a;
```

Функції, оголошені як **inline**, за умовчанням зв'язуються внутрішнім чином: функцію, яка визначена в одному модулі, не видно в іншому модулі. Так само, як і визначення класу, визначення **inline**-функції може бути включене в програму кілька разів – по одному разу на модуль.

Можна зробити **inline**-функцію глобальною, як і константу, використовуючи у визначенні **extern**:

```
// модуль з визначенням глобальної inline-функції
extern inline void f(void)
{ cout << a << "f()\n"; }
```

У іншому модулі досить вказати прототип:

```
// модуль з оголошенням зовнішньої inline-функції
void f(void);
```

До прототипу можна додати **extern** та **inline**:

```
extern inline void f(void);
```

Для локалізації імені у файлі замість атрибуту **static** в C++ дозволяється задавати анонімні (неіменовані) простори імен:

```
namespace    // анонімний простір імен
{ // члени локального простору імен
}
```

Анонімні простори імен є локальними просторами для одиниці трансляції. Для кожного анонімного простору компілятор генерує унікальне внутрішнє ім'я, тому такі простори не "склеюються".

Завдання

Завдання отримуються з розділу 3 "Спадкування класів". Кожен клас повинен бути виконаний в окремому модулі програми.

РОЗДІЛ 10. КОНТЕЙНЕРНІ КЛАСИ

10.1 Загальні відомості

Контейнерні класи – це класи, призначені для зберігання даних, організованих певним чином. Прикладами контейнерів можуть служити масиви, лінійні списки або стеки. Для кожного типу контейнера визначені методи для роботи з його елементами, незалежні від конкретного типу даних, які зберігаються в контейнері, тому один і той же вид контейнера можна використовувати для зберігання даних різних типів. Ця можливість реалізована за допомогою шаблонів класів, тому частина бібліотеки C++, в яку входять контейнерні класи, а також алгоритми та ітератори, про які буде розказано в наступних розділах, називають *стандартною бібліотекою шаблонів* (STL – Standard Template Library).

Використання контейнерів дозволяє значно підвищити надійність програм, їх переносимість і універсальність, а також зменшити терміни їх розробки.

Незважаючи на ці переваги, універсальність і безпека використання контейнерних класів не можуть не відбиватися на швидкодії програми. Зниження швидкодії залежно від реалізації компілятора може бути дуже значним.

STL містить контейнери, що реалізовують основні структури даних, які використовуються при написанні програм, – *вектори, двосторонні черги, списки та їх різновиди, словники і множини.* Контейнери можна розділити на два типи: послідовні і асоціативні.

Послідовні контейнери забезпечують зберігання кінцевої кількості однотипних величин у вигляді безперервної послідовності. До них відносяться вектори (**vector**), двосторонні черги (**deque**) і списки (**list**), а також так звані адаптери, тобто варіанти контейнерів – стеки (**stack**), черги (**queue**) і черги з пріоритетами (**priority_queue**). Кожен вид контейнера забезпечує свій набір дій над даними. Вибір виду контейнера залежить від того, що потрібно робити з даними в програмі. Наприклад, при необхідності часто вставляти та видаляти елементи з середини послідовності слід використовувати список, а якщо включення елементів виконується головним чином в кінець або початок – двосторонню чергу.

Асоціативні контейнери забезпечують швидкий доступ до даних по ключу. Ці контейнери побудовані на основі збалансованих дерев. Існує п'ять типів асоціативних контейнерів: словники (**map**), словники з дублікатами (**multimap**), множини (**set**), множини з дублікатами (**multiset**) та бітові множини (**bitset**).

Практично в будь-якому контейнерному класі визначені поля перерахованих нижче типів:

Ітератор є аналогом вказівки на елемент. Він використовується для проглядання контейнера в прямому або зворотному напрямі. Все, що потрібне від ітератора – вміти посилатися на елемент контейнера і реалізовувати операцію переходу до його наступного елементу.

Поле	Пояснення
value_type	Тип елементу контейнера
size_type	Тип індексів, лічильників елементів тощо
iterator	Ітератор
const_iterator	Константний ітератор
reverse_iterator	Обернений ітератор
const_reverse_iterator	Константний обернений ітератор
reference	Посилання на елемент
const_reference	Константне посилання на елемент
key_type	Тип ключа (для асоціативних контейнерів)
key_compare	Тип критерію порівняння (для асоціативних контейнерів)

Константні ітератори використовуються тоді, коли значення відповідних елементів контейнера не змінюються. За допомогою ітераторів можна проглядати контейнери, не піклуючись про фактичні типи даних, які використовуються для доступу до елементів. Для цього в кожному контейнері визначено декілька методів, перерахованих в наступній таблиці.

Метод	Пояснення
iterator begin(), const_iterator begin() const	Вказують па перший елемент
iterator end(), const_iterator end() const	Вказують па елемент, наступний за останнім
Reverse_iterator rbegin(), const_reverseiterator rbeginO const	Вказують на перший елемент в зворотній послідовності
reverseiterator rend(), const_reverse_iterator rend() const	Вказують на елемент, наступний за останнім, в зворотній послідовності

У кожному контейнері ці типи і методи визначаються способом, залежним від їх реалізації. У всіх контейнерах визначені методи, що дозволяють отримати відомості про розмір контейнерів:

Метод	Пояснення
size()	Кількість елементів
max_size()	Максимальний розмір контейнера (порядка мільярду елементів)
Empty()	Булевська функція, яка показує, чи пустий контейнер

Інші поля і методи контейнерів ми розглянемо в міру необхідності. STL визначається в 13 заголовних файлах:

algorithm	**deque**	**functional**	**iterator**	**list**	**map**
memory	**numeric**	**queue set**	**stack**	**utility**	**vector**

10.2 Послідовні контейнери

Вектори (**vector**), двосторонні черги (**deque**) і списки (**list**) підтримують різні набори операцій, серед яких є співпадаючі операції. Вони можуть бути реалізовані з різною ефективністю:

Операція	Метод	vector	deque	list
Вставка в початок	push_front	-	+	+
Видалення з початку	pop_front	-	+	+
Вставка в конец	push_back	+	+	+
Видалення з кінця	pop_back	+	+	+
Вставка в довільне місце	Insert	**(+)**	**(+)**	+
Видалення з довільного місця	Erase	**(+)**	**(+)**	+
Довільний доступ до елементу	[ ], at	+	+	**-**

Знак (+) означає, що відповідна операція реалізується за постійний час, не залежний від кількості елементів **n** в контейнері. Знак (+) означає, що відповідна операція реалізується за час, пропорційний **n**. Для малих **n** час операцій, позначених (+), може перевищувати час операцій, позначених (+), але для великої кількості елементів останні можуть виявитися дуже дорогими. Як видно з таблиці, такими операціями є вставка і видалення довільних елементів черги і вектора, оскільки при цьому всі наступні елементи потрібно переписувати на нові місця.

Отже, вектор – це структура, що ефективно реалізовує довільний доступ до елементів, додавання в кінець та видалення з кінця. Двостороння черга ефективно реалізує довільний доступ до елементів, додавання в обидва кінці і видалення з обох кінців. Список ефективно реалізує вставку та видалення елементів в довільне

місце, але не має довільного доступу до своїх елементів.

Приклад роботи з вектором. У файлі знаходиться довільна кількість цілих чисел. Програма зчитує їх у вектор і виводить на екран в тому ж порядку.

```
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

void main()
{
 ifstream in ("text.txt");
 vector<int> v;
 int x;
 while (!in.eof(), in >> x)
        v.push_back(x);
 for (vector<int>::iterator i=v.begin(); i!=v.end(); ++i)
     cout << *i << "  ";
}
```

Оскільки файл містить цілі числа, використовується відповідна спеціалізація шаблону **vector** – **vector<int>**. Для створення вектора **v** використовується конструктор за умовчанням. Організовується цикл до кінця файлу, в якому з нього зчитується чергове ціле число. За допомогою методу **push_back** воно заноситься у вектор, розмір якого збільшується автоматично. Для проходу по всьому вектору вводиться змінна **i** як ітератор відповідного типу тобто тут оголошується змінна **i** типу "ітератор". За допомогою цього ітератора здійснюється доступ до всіх по порядку елементам контейнера, починаючи з першого. Метод **begin()** повертає вказівку на перший елемент, метод **end()** – на елемент, наступний за останнім. Порівнювати поточне значення з граничним слід саме за допомогою операції "**!=**", оскільки операції "**<**" або "**<=**" можуть бути для даного типу не визначені. Операція інкременту (++i) реалізована так, щоб після неї ітератор вказує на наступний елемент контейнера в порядку обходу. Доступ до елементу вектора виконується за допомогою операції розадресації, як для звичайних вказівок. У даному прикладі замість вектора можна було використовувати будь-який послідовний контейнер шляхом простої заміни слова **vector** на **deque** або **list**. При цьому змінилося б внутрішнє представлення даних і набір доступних операцій. Приведені операції в прикладі використовуються для будь-якого послідовного контейнера. Проте якщо замість циклу **for** вставити фрагмент

```
for (int i = 0; i<v.size(); i++) cout << v[i] << " ";
```

у якому використана операція доступу по індексу [ ], програма не працюватиме для контейнера типу **list,** оскільки в ньому ця операція не визначена.

10.2.1 Вектори (vector)

Для створення вектору можна скористатися наступними конструкторами (приведений спрощений запис):

```
explicit vector(); //1
explicit vector(size_type n,const T&value=T()); //2
template <class InputIter> // 3
vector(InputIter first, InputIter last);
vector(const vector<T>& x); //4
```

Ключове слово **explicit** використовується для того, щоб при створенні об'єкту заборонити виконувати неявно перетворення при наданні значення іншого типу. Конструктор 1 є конструктором за умовчанням. Конструктор 2 створює вектор довжиною **n** і заповнює його однаковими елементами-копіями **value**. Оскільки зміна розміру вектора займає певний час, при його створенні задавати початковий розмір дуже корисно. При цьому для вбудованих типів виконується ініціалізація кожного елементу значенням **value.** Якщо воно не вказане, елементи глобальних векторів ініціалізуються нулем. Якщо тип елементу вектора визначений користувачем, початкове значення формується за допомогою конструктору за умовчанням для даного типу. На місці другого параметра можна написати виклик конструктору з параметрами, створивши таким чином вектор елементів з необхідними властивостями.

ПРИМІТКА. Елементи будь-якого контейнера є копіями об'єктів, що вставляються в нього. Тому для них мають бути визначені конструктор копіювання і операція привласнення.

Конструктор 3 створює вектор шляхом копіювання вказаного за допомогою ітераторів діапазону елементів. Тип ітераторів має бути "для читання". Конструктор 4 є конструктором копіювання.

Приклади конструкторів:

```
// Створюється вектор з 10 рівних одиниці елементів
vector <int> v1 (10, 1);
// Создается вектор, равный вектору v1
vector <int> v2 (v1);
// Створюється вектор з 2 елементів, рівних першим двом елементам vl
vector <int> v3 (v1.begin(), v1.begin() + 2);
// Створюється вектор з 10 об'єктів класу monstr
// (працює конструктор за умовчанням);
```

```
vector <monstr> m1 (10);
// Створюється вектор з 5 об'єктів класу monstr із заданим ім'ям
// (працює конструктор з параметром char*);
vector <monstr> m2 (5, monstr("Вася"));
```

У шаблоні **vector** визначені операція привласнення і функція копіювання:

```
vector<T>& operator=(const vector<T>& x);
void assign(size_type n, const T& value);
template <class InputIter>;
void assign(InputIter first, InputIter last);
```

Тут через **T** позначений тип елементів вектора. Вектори можна привласнювати один одному точно так, як і стандартні типи даних або рядка. Після привласнення розмір вектора стає рівним новому значенню, всі старі елементи видаляються. Функція **assign** в першій формі аналогічна по дії конструктору 2, але застосовується до існуючого об'єкту. Функція **assign** в другій формі призначена для привласнення елементам вектора значень з діапазону, який визначається ітераторами **first** і **last**, аналогічно конструктору 3, наприклад:

```
vector <int> v1, v2;
// Першим 10 елементам вектора v1 привласнюється значення 1;
v1.assign (10, 1);
```

Доступ до елементів вектора здійснюється за допомогою наступних операторі та методів:

```
reference operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference at(size_type n);
reference front();
const_reference front() const;
reference back();
const_reference back() const;
```

Оператор [] здійснює доступ до елементу вектора по індексу без перевірки його виходу за межу вектора. Функція **at** виконує таку перевірку і породжує виключення **out_of_range** у разі виходу за межу вектора. Природно, що функція **at** працює повільніше, ніж оператор [], тому у випадках, коли діапазон визначений явно, переважно користуватися оператором []:

```
for (int i = 0; i<v.size(); i++) cout <<v[i]<< "  ";
```

В іншому випадку використовується функція **at** з обробкою виключення:

```
try
{
 //...
 v.at(i) = v.at(...);
}
catch (out_of_range) { ... }
```

Операції доступу повертають значення посилання на елемент (**reference**) або константного посилання (**const_reference**) залежно від того, чи застосовуються вони до константного об'єкту чи ні. Методи **front()** і **back()** повертають посилання відповідно на перший і останній елементи вектора (це не те ж саме, що **begin()** – вказівка па перший елемент і **end()** – вказівка на елемент, наступний за останнім. Приклад:

```
vector <int> v(5, 10);
v.front() = 100; v.back() = 100;
cout<<v[0]<<" "<<v[v.size() - 1];//Виведення: 100 100
```

Функція **capacity** визначає розмір оперативної пам'яті, яку займає вектор:

```
size_type capacity() const;
```

Пам'ять під вектор виділяється динамічно, але не під один елемент в кожен момент часу (це було б марнотратним витрачанням ресурсів), а відразу під групу елементів, наприклад, 256 або 1024. Перерозподіл пам'яті відбувається тільки при перевищенні цієї кількості елементів, при цьому об'єм виділеного простору подвоюється. Після перерозподілу будь-які ітератори, що посилаються на елементи вектора, стають недійсними, оскільки вектор може бути переміщений в іншу ділянку пам'яті, і не можна чекати, що пов'язані з ним посилання будуть оновлені автоматично. Існує також функція виділення пам'яті **reserve,** яка дозволяє задати, скільки пам'яті вимагається для зберігання вектора:

```
void reserve(size_type n);
```

Приклад застосування функції:

```
vector <int> v;
v.reserve(1000);//Виділення пам'яті під 1000 елементів
```

Після виконання цієї функції значення функції **capacity** буде рівне щонайменше **n**. Функцію **reserve** корисно застосовувати тоді, коли розмір вектору відомий заздалегідь.

Для зміни розмірів вектора застосовується функція **resize**:

```
void resize(size_type sz, T c = T());
```

Ця функція збільшує або зменшує розмір вектора залежно від того, більше значення **sz**, яке задається, чим значення **size()**, або менше. Другий параметр задає значення, яке привласнюється всім новим елементам вектора. Вони поміщуються в кінець вектора. Якщо новий розмір менший, ніж значення **size()**, з кінця вектора видаляється (**size() – sz**) елементів.

Визначені наступні методи для зміни об'єктів класу **vector**:

```
void push_back(const T& value);
void pop_back();
iterator insert(iterator position, const T& value);
void insert(iterator position,size_type n,const T&value);
template <class InputIter>
void insert(iterator position, InputIter first,
            InputIter last);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
void swap();
void clear(); // Очистка вектора
```

Функція **pushback** додає елемент в кінець вектора, функція **popbacк** – видаляє елемент з кінця вектора. Функція **insert** призначена для вставки елементу у вектор. Перша форма функції вставляє елемент **value** в позицію, задану першим параметром (ітератором), і повертає ітератор, що посилається на вставлений елемент. Друга форма функції вставляє у вектор **n** однакових елементів. Третя форма функції дозволяє вставити декілька елементів, які можуть бути задані будь-яким діапазоном елементів відповідного типу, наприклад:

```
#include <iostream>
#include <vector>
using namespace std;

void main()
{
 vector <int> v(2), v1(3,9);
 vector<int>::iterator t;
 int m[3] = {3, 4, 5};
```

```
 v.insert(v.begin(), m, m + 3);
 v1.insert(v1.begin() + 1, v.begin(), v.begin() + 2);
 cout << " v:  ";
 for(t=v.begin();t != v.end();t++)cout<<*t <<"  ";
 cout <<"\nv1:  ";;
 for(t=v1.begin();t != v1.end();t++)cout<<*t <<"  ";
}
```

Результат роботи програми:

```
v:   3  4  5  0  0
v1: 9  3  4  9  9
```

Вставка у вектор займає час, пропорційний кількості зміщених на нові позиції елементів. При цьому, якщо новий розмір вектора перевищує об'єм зайнятої пам'яті, відбувається перерозподіл пам'яті. Це – плата за легкість доступу по індексу. Якщо при вставці перерозподілу не відбувається, всі ітератори зберігають свої значення. Інакше вони стають недійсними.

Функція **erase** призначена для видалення одного елементу вектора (перша форма функції) або діапазону, заданого за допомогою ітераторів (друга форма):

```
vector <int> v;
for (int i = 1; i<6; i++) v.push_back(i);
// Вміст v: 1 2 3 4 5
v.erase(v.begin()); // Вміст v: 2 3 4 5
v.erase(v.begin(), v.begin() + 2); // Вміст v: 4 5
```

Кожен виклик функції **erase** так само, як і у разі вставки, займає час, пропорційний кількості зміщених на нові позиції елементів. Всі ітератори і посилання "правіше" місця видалення стають недійсними.

Функція **swap()** призначена для обміну елементів двох векторів одного типу, але не обов'язково одного розміру:

```
vector <int> v1, v2;
v1.swap(v2); // Еквівалентно v2.swap(v1)
```

Для векторів визначені операції порівняння ==, !=, <, <=, >, >=. Два вектори вважаються рівними, якщо рівні їх розміри і всі відповідні пари елементів. Один вектор менше іншого, якщо перший з елементів одного вектора, не рівний відповідному елементу іншого, менше.

Приклад з векторами, що містять цілі числа, рядки і структури:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct st
{
 int a;
 char mas[10];
};

void main()
{
 string mas[] = {"111","222","333","444","555"};
 string raz = "-------------------------\n";
 int i;
 for(i=0;i<5;i++)cout<<mas[i]<<" ";
 cout << endl;
 cout << raz;
 vector <int> vec1;
 vector <int> vec11;
 vector <char*> vec2;
 vector <st> vec3;
 vector<int>::iterator t1;
 vector<int>::iterator t11;
 vector<char*>::iterator t2;
 vector<st>::iterator t3;
 for(i = 1; i < 11;i++)
  {vec1.push_back (i);vec11.push_back (i);}
 t1 = vec1.begin();
 t11 = vec11.begin();
 vec1.insert(t1,t11+1,t11 + 5);
 for (t1 = vec1.begin();t1 != vec1.end();t1++)
      cout <<*t1<<" " ;
 cout << endl<< raz;
 vec1.reserve(100);
 cout<< vec1.size()<< "\n";
 vec1.resize(10,10);
 cout<<vec1.size() << "\n"<< raz;
 for (t1 = vec1.begin();t1 != vec1.end();t1++)
      cout <<*t1<<" " ;
 cout <<endl << raz ;
```

```
 t2 = vec2. begin();
 vec2.insert(t2,"ttt");
 for (i = 0; i < 3; i++)
      vec2.push_back((char*)mas[i].c_str ());
 cout<<vec2.front()<<" " <<vec2.back()<<endl<< raz ;
 for (i = 0,t2 = vec2.begin();t2 != vec2.end();t2++)
 {
  cout <<*t2<<" "<<vec2[i]<<"  " << vec2.at(i)<<"\n";
  i++;
 }
 cout << raz;
 vec2.erase(vec2.begin());
 for (t2 = vec2.begin();t2 != vec2.end();t2++)
 cout <<*t2<<"\n";
 cout << raz;
 st nn;
 nn.a = 10;
 strcpy (nn.mas,"aaa");
 vec3.push_back(nn);
 nn.a = 20;
 strcpy (nn.mas,"bbb");
 vec3.push_back(nn);
 i = 0;
 for (t3 = vec3.begin();t3 != vec3.end();t3++)
 {
  cout<<vec3[i].a <<"  "<<vec3.at(i).mas
      <<"  "<<(*t3).mas <<"\n";
  i++;
 }
}
```

Результат роботи програми:

```
111  222  333  444  555
-------------------------------------------
2  3  4  5  1  2  3  4  5  6  7  8  9  10
-------------------------------------------
14
10
-------------------------------------------
   2  3  4  5  1  2  3  4  5  6
-------------------------------------------
```

```
ttt   333
-----------------------------------------------
ttt    ttt    ttt
111 111 111
222 222 222
333 333 333
-----------------------------------------------
111
222
333
-----------------------------------------------
10  aaa  aaa
20  bbb  bbb
```

В даному прикладі приводяться описані вище методи. Він приведений для закріплення матеріалу роботи з вектором.

10.2.2. Двосторонні черги (deque)

Двостороння черга – це послідовний контейнер, який, разом з вектором, підтримує довільний доступ до елементів і забезпечує вставку і видалення з обох кінців черги за постійний час. Ті ж операції з елементами усередині черги займають час, пропорційний кількості переміщуваних елементів. Розподіл пам'яті виконується автоматично. Розглянемо схему організації черги (рис. 10.1). Для того, щоб забезпечити довільний доступ до елементів за постійний час, черга розбита на блоки, доступ до кожного з яких здійснюється через вказівку. На рисунку замальовані області відповідають зайнятим елементам черги. Якщо при додаванні в початок або в кінець блок виявляється заповненим, виділяється пам'ять під черговий блок (наприклад, після заповнення блоку 4 буде виділена пам'ять під блок 5, а після заповнення блоку 2 – під блок 1). При заповненні крайнього з блоків відбувається перерозподіл пам'яті під масив вказівок так, щоб використовувалися тільки середні елементи. Це не займає багато часу. Таким чином, доступ до елементів черги здійснюється за постійний час, хоча він і дещо більше, ніж для вектора.

Для створення двосторонньої черги можна скористатися наступними конструкторами (приведений спрощений запис), аналогічними конструкторам вектора:

```
explicit deque(); //1
explicit deque(size_type n, const T& value = T());//2
template <class InputIter> //3
deque(InputIter first , InputIter last);
deque(const vector <T>& x); //4
```

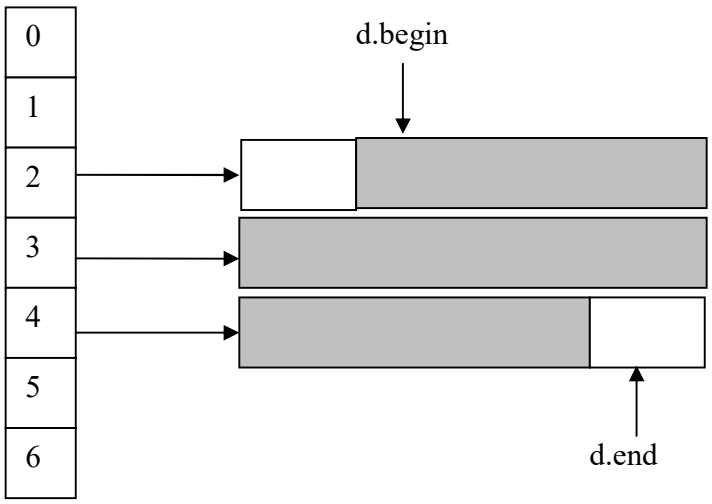

Рис. 10.1. Організація двосторонньої черги

Конструктор 1 є конструктором за умовчанням. Конструктор 2 створює чергу довжиною **n** і заповнює її однаковими елементами-копіями **value**. Конструктор 3 створює чергу шляхом копіювання вказаного за допомогою ітераторів діапазону елементів. Тип ітераторів має бути "для читання". Конструктор 4 є конструктором копіювання.

Приклади конструкторів:

```
// Створюється черга з 10 рівних одиниці елементів
deque <int> d2 (10, 1);
// Створюється черга, рівна черзі v1
deque <int> d4 ( v1 );
// Створюється черга з двох елементів, рівних першим двом
// елементам вектора v1 з попереднього розділу
deque <int> d3 (v1.begin(), v1.begin() + 2);
// Створюється черга з 10 об'єктів класу monstr
// (працює конструктор за умовчанням)
deque <monstr> m1 (10);
// Створюється черга з 5 об'єктів класу monstr із заданим ім'ям
// (працює конструктор з параметром char*)
deque <monstr> m2 (5, monstr("Вася в очереди"));
```

У шаблоні **deque** визначені операція привласнення, функція копіювання, ітератори, операції порівняння, операції і функції доступу до елементів і зміни об'єктів, аналогічні відповідним операціям і функціям вектора. Вставка і видалення так само, як і для вектора, виконуються за пропорційний кількості елементів час.

Якщо ці операції виконуються над внутрішніми елементами черги, всі значення ітераторів та посилань на елементи черги стають недійсними. Після операцій додавання в будь-який з кінців всі значення ітераторів стають недійсними, а значення посилань на елементи черги зберігаються. Після операції вибірки з будь-якого кінця стають недійсними тільки значення ітераторів і посилань, пов'язаних з цими елементами. Окрім перерахованих, визначені функції додавання та вибірки з початку черги:

```
void push_front(const T& value);
void pop_front();
```

При вибірці елемент видаляється з черги. Для черги не визначені функції місткість і резерв, але є функції перевизначення та розмір.

10.2.3 Списки (list)

Список не надає довільного доступу до своїх елементів, зате вставка і видалення виконуються за постійний час. Клас **list** реалізований в **STL** у вигляді двохзв’язного списку, кожен вузол якого містить посилання на подальший і попередній елементи. Тому операції інкремента і декремента для ітераторів списку виконуються за постійний час, а пересування на **n** вузлів вимагає часу, пропорційного **n**. Після виконання операцій вставки та видалення значення всіх ітераторів і посилань залишаються дійсними. Список підтримує конструктори, операцію привласнення, функцію копіювання, операції порівняння та ітератори, аналогічні векторам і чергам.

Доступ до елементів списків обмежується наступними методами:

```
reference front();
const_reference front() const;
reference back();
const_reference back() const;
```

Для занесення в початок і кінець списку визначені методи, аналогічні відповідним методам черги:

```
void push_front(const T& value);
void pop_front();
void push_back(const T& value);
void pop_back();
```

Крім того, діє решта всіх методів для зміни об'єктів **list**, аналогічні векторам і чергам:

```
iterator insert(iterator position, const T& value);
void insert(iterator position,size_type n,const T&value);
template <class InputIter>
void insert(iterator position, Inputlter first,
            InputIter last);
iterator erase(iterator position);
iterator erase(iterator first , iterator last);
void swap();
void clear();
```

Для списку не визначена функція **capacity**, оскільки пам'ять під елементи відводиться в міру необхідності. Можна змінити розмір списку, видаливши або додавши елементи в кінець списку (аналогічно двосторонній черзі):

```
void resize(size_type sz, T c = T());
```

Окрім перерахованих, для списків визначено декілька специфічних методів. Зчеплення списків (**splice**) призначене для переміщення елементів з одного списку в іншій без перерозподілу пам'яті, тільки за рахунок зміни вказівок:

```
void splice(iterator position, list<T>& x):
void splice(iterator position, list<T>& x, iterator i );
void splice(iterator position, list<T>& x,
            iterator first , iterator last);
```

Обидва списки повинні містити елементи одного типу. Перша форма функції вставляє в список перед елементом, позиція якого вказана першим параметром, всі елементи списку, вказаного другим параметром, наприклад:

```
list <int> L1, L2;
...
// Формування списків
L1.splice(L1.begin() + 4, L2);
```

Другий список залишається порожнім. Не можна вставити список в самого себе. Друга форма функції переносить елемент, позицію якого визначає третій параметр, із списку **x** в список, який викликає функцію. Допускається переносити елемент в межах одного списку. Третя форма функції аналогічним чином переносить із списку в список декілька елементів. Їх діапазон задається третім і четвертим параметрами функції. Якщо для одного і того ж списку перший параметр знаходиться в діапазоні між третім і четвертим, результат не визначений. Приклад:

```
#include <iostream>
#include <list>
using namespace std;

void main()
{
 list<int> L1;
 list<int>::iterator i, j, k;
 int m;
 for (m = 0; m<5; m++)L1.push_back(m + 1);
 for (m = 12; m<14; m++)L1.push_back(m);
 cout << "Початковий список:";
 for (i = L1.begin(); i != L1.end();++i)
 cout << *i << "  ";
 cout << endl;
 i = L1.begin(); i++;
 k = L1.end();
 j = --k; k++; j -- ;
 L1.splice( i, L1, j , k);
 cout << "Список після зчеплення:";
 for ( i = L1.begin(); i != L1.end(); ++i)
 cout << *i << "  ";
}
```

Результат роботи програми:

```
Початковий список: 1 2 3 4 5 12 13
Список після зчеплення: 1 12 13 2 3 4 5
```

Для ітераторів списків не визначені операції складання і віднімання, тобто не можна написати **j = k – 1**, тому довелося скористатися допустимими для ітераторів списків операціями інкремента і декремента. У загальному випадку для пошуку елементу в списку використовується функція **find()**.

Для видалення елементу по його значенню застосовується функція **remove**:

```
void remove(const T& value);
```

Якщо елементів із значенням **value** в списку декілька, всі вони будуть видалені. Можна видалити із списку елементи, що задовольняють деякій умові.
Для цього використовується функція **remove_if()**:

```
template <class Predicate>
void remove_if(Predicate pred);
```

Параметром є клас-предикат, який задає умову, що накладається на елемент списку. Предикати будуть розглянуті нижче.

Для впорядковування елементів списку використовується метод **sort**:

```
void sort();
template <class Compare> void sort(Compare comp);
```

У першому випадку список сортується за збільшенням елементів (відповідно до визначення операції < для елементів), в другому – відповідно до функціонального об'єкту Compare. Функціональний об'єкт має значення **true**, якщо два передаваних йому значення повинні при сортуванні залишитися в попередньому порядку, та **false** – навпаки. Порядок проходження елементів, що мають однакові значення, зберігається. Час сортування пропорційний $N*log_2 n$, де N – кількість елементів в списку.

Метод **unique()** залишає в списку тільки перший елемент з кожної серії тих, що йдуть підряд однакових елементів. Перша форма методу має наступний формат:

```
void unique();
```

Друга форма методу **unique()** використовує як параметр бінарний предикат, що дозволяє задати власний критерій видалення елементів списку. Предикат має значення **true**, якщо критерій дотриманий, і **false** – в іншому випадку. Аргументи предиката мають тип елементів списку:

```
template <class BinaryPredicate>
void unique(BinaryPredicate binary_pred);
```

Для злиття списків призначений метод **merge()**:

```
void merge(list<T>& x);
template <class Compare>
void merge(list<T>& x, Compare comp);
```

Обидва списки мають бути впорядковані (у першому випадку відповідно до визначення операції < для елементів, в другому – відповідно до функціонального об'єкту Compare). Результат – впорядкований список. Якщо елементи в списку, який викликає функцію та в списку-параметрі збігаються, першими розташовуватимуться елементи із списку, який викликає функцію.

Метод **reverse()** призначений для зміни порядку проходження елементів списку на зворотній:

```
void reverse();
```

Приклад.

```
#include <iostream>
#include <list>
using namespace std;

void show (const char *str, const list<int> &L)
{
 cout << str <<":"<< endl;
 for (list<int>::const_iterator i = L.begin();
 i != L.end(); ++i)
 cout << *i <<" ";
 cout << endl;
}

void main()
{
 list<int> L;
 list<int>::iterator i;
 int x;
 int mas[12] = {56,34,54,0,76,23,51,11,51,11,76,88};

 for(int k = 0; k < 12; k++)
 L.push_back(mas[k]);
 show("Початковий список", L);
 L.push_front(1);
 i = L.begin(); L.insert(++i, 2);
 show("Після вставки 1 і 2 в початок", L);
 i = L.end(); L.insert(--i, 100);
 show("Після вставки 100 перед останнім", L);
 i = L.begin();
 x = *i;
 L.pop_front();
 cout <<"Видалили з початку " << x << endl;
 i = L.end();
 x = *--i; L.pop_back();
 cout <<"Видалили з кінця " << x << endl;
 show("Список після видалення", L);
 L.remove(76);
 show("Після видалення елементів із значенням 76",L);
```

```
 L.sort();
 show("Після сортування", L);
 L.unique();
 show("Після unique", L);
 list<int> LI (L);
 L.reverse();
 show("Після reverse", L);
}
```

Результат роботи програми:

```
Початковий список:
56 34 54 0 76 23 51 11 51 11 76 88
Після вставки 1 ? 2 в початок:
1 2 56 34 54 0 76 23 51 11 51 11 76 88
Після вставки 100 перед останнім:
1 2 56 34 54 0 76 23 51 11 51 11 76 100 88
Видалили з початку 1
Видалили з кінця 88
Список після видалення:
2 56 34 54 0 76 23 51 11 51 11 76 100
Після видалення елементів із значенням 76:
2 56 34 54 0 23 51 11 51 11 100
Після сортування:
0 2 11 11 23 34 51 51 54 56 100
Після unique:
0 2 11 23 34 51 54 56 100
Після reverse:
100 56 54 51 34 23 11 2 0
```

10.2.4 Стеки (stack)

У стеку допускаються тільки ті операції, що змінюють його розмір – додавання елементу у вершину стека і вибірка з вершини. Стек можна реалізувати на основі будь-якого: вектора, двосторонньої черги або списку. Таким чином, стек є не новим типом контейнера, а варіантом який має місце, тому він називається адаптером контейнера. У **STL** стек визначений за умовчанням на базі двосторонньої черги:

```
template <class T, class Container = deque<T> >
class stack
{
```

```
 protected:
 Container c;
 public:
 explicit stack(const Containers = Container());
 bool empty () const {return c.empty();}
 size_type size() const {return c.size();}
 value_type& top() {return c.back();}
 const value_type& top() const {return c.back();}
 void push(const value_type& x) {c.push_back(x);}
 void pop () {c.pop_back();}
};
```

З приведеного опису видно, що метод занесення в стек **push()** відповідає методу занесення в кінець **push_back()**, метод вибірки із стека **pop()** – методу вибірки з кінця **pop_back()**, крім того, доданий метод **top()** для отримання або зміни значення елементу на вершині стека. Конструктору класу **stack** передається в якості параметру посилання на базовий контейнер, який копіюється в захищене поле даних **c**.

Приклад використання стека:

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

void main()
{
 int x;
 int mas[12] = {56,34,54,0,76,23,51,11,51,11,76,88};
 stack <int, vector<int> > s;

 for(int k = 0; k < 12; k++)
 s.push(mas[k]);
 while (! s.empty())
 {
  x = s.top();
  cout << x <<" ";
  s.pop();
 }
}
```

Результат роботи програми

```
88 76 11 51 11 51 23 76 0 54 34 56
```

10.2.5 Черги (queue)

Для черги допускаються дві операції, що змінюють її розмір, – додавання елементу в кінець і вибірка з початку. Черга є адаптером, який можна реалізувати на основі двосторонньої черги або списку (вектор не підходить, оскільки в ньому немає операції вибірки з початку). У **STL** черга визначена за умовчанням на базі двосторонньої черги:

```
template <class T, class Container = deque<T> >
class queue
{
 protected:
 Container c:
 public:
 explicit queue(const Containers = Container());
 bool empty ()const (return c.empty();}
 size_type size() const {return c.size();}
 value_type& front() {return c.front();}
 const value_type& front () const (return c.front();}
 value_type& back () (return c.back();}
 const value_type& back () const (return c.back();}
 void push(const value_type& x) (c.push_back(x);}
 void pop() {c.pop_front();}
};
```

Методи **front()** і back() використовуються для набуття значень елементів, що знаходяться відповідно на початку і в кінці черги (при цьому елементи залишаються в черзі).

Приклад роботи з чергою (програма вводить дані в чергу і виконує вибірку з неї, поки черга не закінчиться):

```
#include <iostream>
#include <list>
#include <queue>
using namespace std;

void main()
{
```

```
 queue <int, list<int> > q;
 int x;
 int mas[12] = {56,34,54,0,76,23,51,11,51,11,76,88};
 for(int k = 0; k < 12; k++)
 q.push(mas[k]);
 cout <<"q.front():" << q.front()<< " ";
 cout <<"q.back(): " << q.back() << endl;
 while (1)
 {
  q.pop();
  if (q.empty())break;
  cout <<"q.front():" << q.front() <<" ";
  cout <<"q.back(): "<< q.back()<< endl;
 }
}
```

Результат роботи програми:

```
q.front(): 56 q.back(): 88
q.front(): 34 q.back(): 88
q.front(): 54 q.back(): 88
q.front(): 0  q.back(): 88
q.front(): 76 q.back(): 88
q.front(): 23 q.back(): 88
q.front(): 51 q.back(): 88
q.front(): 11 q.back(): 88
q.front(): 51 q.back(): 88
q.front(): 11 q.back(): 88
q.front(): 76 q.back(): 88
q.front(): 88 q.back(): 88
```

Як вже наголошувалося у послідовних контейнерів багато загального. Нижче наводиться приклад, в якому для заповнення контейнерів даними використовується один і той же метод **puch_back()**, виведення даних з контейнера виконується за допомогою ітератора. Проте виведення даних з контейнера для вектора і черги можливе за допомогою довільного доступу до елементу, а для списку це неприпустимо.

Приведемо приклад.

```
#include <iostream>
#include <vector>
#include <deque>
```

```
#include <list>
using namespace std;

void main()
{
 int mas[] = {1,2,3,4,5,6,7,8,9};
 int i;
 vector <int> vec;
 vector<int>::iterator t;
 deque<int>::iterator d;
 deque <int> dec;
 list<int>::iterator l;
 list<int> list;
 // ------------ vector ---------------------
 cout << endl <<"  vector " <<endl;
 for (i = 0; i < 9; i++)vec.push_back(mas[i]);
   for (i = 0,t = vec.begin();t != vec.end();t++,i++)
        cout <<"  "<<*t <<"/"<< vec[i];
 cout <<"\n";
 t = vec.begin();
 for (i = 0;i < vec.size();i++)
      cout <<"  "<<vec[i]<<"/" << *(t++);
 t = (vec.begin() + 1);
 *t = 100;
 cout << endl << "  "<<vec[0]<<"   " << *t ;
 vector<int>::iterator tt;
 tt = t;
 tt= t + 2;
 cout <<"\n  "<<*(tt)<<"  "<<"\n";
 // ------------ deque ---------------------
 cout << endl <<"  deque " <<endl;
 for (i = 0; i < 9; i++)
      dec.push_back(mas[i]);
 for (i = 0,d = dec.begin();d != dec.end();d++,i++)
      cout <<"  "<<dec[i]<<" "<<*d;
 // ------------ list ---------------------
 cout << endl <<"  list " <<endl;
 for (i = 0; i < 9; i++)
 list.push_back(mas[i]);
 for (l = list.begin();l != list.end();l++)
      cout <<"  "<<*l ;
}
```

Результат роботи програми:

```
vector
1/1  2/2  3/3  4/4  5/5  6/6  7/7  8/8  9/9
1/1  2/2  3/3  4/4  5/5  6/6  7/7  8/8  9/9
1      100
4
deque
1 1  2 2  3 3  4 4  5 5  6 6  7 7  8 8  9 9
list
1  2  3  4  5  6  7  8  9
```

10.2.6 Черги з пріоритетами (priority_queue)

У черзі з пріоритетами кожному елементу відповідає пріоритет, що визначає порядок вибірки з черги. За умовчанням він визначається за допомогою операції <; таким чином, з черги кожного разу вибирається максимальний елемент.

Для реалізації черги з пріоритетами підходить контейнер, що допускає довільний доступ до елементів, тобто, наприклад, вектор або двостороння черга. Тип контейнера передається другим параметром шаблону (перший, як завжди, тип елементів). Третім параметром вказується функція або функціональний об'єкт, за допомогою яких виконується визначення пріоритету:

```
template <class T, class Container = vector<T>,
class Compare =less<typename Container::value_type> >
class priority_queue
{
 protected:
 Container c;
 Compare comp;
 public:
 explicit priority_queue(const Compare& x= Compare(),
 const Container& = Container());
 template <class InputIter>
 priority_queue(lnputlter first, Inputlter last,
 const Compare&x=Compare(),const Container&=Container());
 bool empty () const {return c.empty();}
 size_typesize() const {return c.size();}
 const value_type& top() const {return c.front();}
 void push(const value_type& x);
 void pop();
};
```

Для елементів з рівними пріоритетами черга з пріоритетами є простою чергою. Як і для стеків, основними методами є **push()**, **pop()** і **top()**.

Приклад:

```
#include <iostream>
#include <queue>
using namespace std;

void main()
{
  priority_queue <int, vector<int>, less<int> > P;
  int x;
  P.push(13); P.push(51); P.push(200); P.push(17);
  while (!P.empty())
  {
   x = P.top(); cout<<"Вибраний елемент:" << x<<endl;
   P.pop();
  }
}
```

Результат роботи програми:

```
Вибраний елемент: 200
Вибраний елемент: 51
Вибраний елемент: 17
Вибраний елемент: 13
```

В даному прикладі третім параметром шаблону є шаблон, визначений у заголовному файлі <**functional**>. Він задає операцію порівняння на "менше". Можна задати стандартні шаблони **greater**<**тип**>, **geater_equal**<**тип**>, **less_equa1**<**тип**>. Якщо потрібно визначити інший порядок вибірки з черги, вводиться власний функціональний об'єкт. У приведеному нижче прикладі вибірка виконується по найменшій сумі цифр в числі:

```
#include <iostream>
#include <queue>
using namespace std;

class CompareSum
{
 public:
 bool operator()(int x, int y)
```

```
 {
  int sx = 0, sy = 0;
  while (x){sx += x % 10; x /= 10;}
  while (y){sy += y % 10; y /= 10;}
  return sx > sy ;
 }
};

void main()
{
 priority_queue <int, vector<int>, CompareSum > P;
 int x;
 P.push(13); P.push(51); P.push(200); P.push(17);
 while (!P.empty())
 {
  x = P.top(); cout <<"Вибраний елемент:"<<x << endl;
  P.pop();
 }
}
```

Результат роботи програми:

```
Вибраний елемент: 200
Вибраний елемент: 13
Вибраний елемент: 51
Вибраний елемент: 17
```

10.3 Асоціативні контейнери

10.3.1 Загальні відомості про асоціативні контейнери

Як вже вказувалося, асоціативні контейнери забезпечують швидкий доступ до даних за рахунок того, що вони, як правило, побудовані на основі збалансованих дерев пошуку (стандартом регламентується тільки інтерфейс контейнерів, а не їх реалізація).

Існує п'ять типів асоціативних контейнерів: словники (**map**), словники з дублікатами (**multimap**), множини (**set**), множини з дублікатами (**multiset**) і бітові множини (**bitset**). Словники часто називають також асоціативними масивами або відображеннями.

Словник побудований на основі пар значень, перше з яких є ключем для ідентифікації елементу, а друге – власне елемент. Можна сказати, що ключ асоційований з елементом, звідки і відбулася назва цих контейнерів. Наприклад, в анг-

ло-російському словнику ключем є англійське слово, а елементом – російське. Звичайний масив теж можна розглядати як словник, ключем в якому слугує номер елементу. У словниках, описаних в **STL**, в якості ключа може використовуватися значення довільного типу. Асоціативні контейнери описані в заголовних файлах <**map**> і <**set**>. Для зберігання пари "ключ-елемент" використовується шаблон **pair**, описаний в заголовному файлі <**utility**>:

```
template <class T1, class T2> struct pair
{
 typedef T1 first_type;
 typedef T2 second_type;
 T1 first;
 T2 second;
 pair();
 pair(const T1& x, const T2& y);
 template <class U,class V> pair(const pair<U, V> &p)
};
```

Шаблон **pair** має два параметри, що є типами елементів пари. Перший елемент має ім'я **first**, другий, – **second**. Визначено два конструктори: один повинен набувати два значення для ініціалізації елементів, другий (конструктор копіювання), – посилання на іншу пару. Конструктора за умовчанням у пари немає, тобто при створенні об'єкту йому потрібно привласнити значення явним чином. Для пари визначені перевірка на рівність і операція порівняння на менше (решта всіх операцій відношення генерується в **STL** автоматично на основі цих двох операцій). Пара **p1** менше пари **p2**, якщо **`p1.first < p2.first`** або `p1.first == p2.first && p1.second < p2.second`.

Для надання значення парі можна використовувати функцію **make_pair**:

```
template <class T1, class T2>
pair<T1, T2> make_pair(const T1& x, const T2& y);
```

Приклад формування пар:

```
#include <iostream>
#include <utility>
using namespace std;

void main()
{
 pair <int, double> p1(10,12.3), p2(p1);
 p2 = make_pair(20, 12.3);
```

```
 cout <<"pi: " << p1.first <<" " << p1.second <<endl;
 cout <<"p2: " << p2.first << " "<< p2.second <<endl;
 p2.first -= 10;
 if (p1 == p2) cout << "p1 == p2\n";
 p1.second -= 1;
 if (p2 > p1) cout  << "p2 > pl\n";
}
```

Результат роботи програми:

```
p1: 10 12.3
p2: 20 12.3
pi == p2
p2 > pi
```

Заголовний файл <**utility**> при використанні <**map**> або <**set**> підключається автоматично.

10.3.2 Словники (map)

У словнику (**map**), на відміну від словника з дублікатами (**multimap**), всі ключі мають бути унікальні. Елементи в словнику зберігаються у відсортованому вигляді, тому для ключів має бути визначене відношення "менше". Шаблон словника містить три параметри: тип ключа, тип елементу і тип функціонального об'єкту, що визначає відношення "менше":

```
template <class Key,class T,class Compare=less<Key> >
class map
{
 public:
 typedef pair <const Key, T> value_type;
 explicit map(const Compare& comp = Compare());
 template <class InputIter>
 map (InputIter first , InputIter last,
      const Compare& comp = Compare());
 map(const map <Key, T, Compare>& x);
...
}
```

Як видно з приведеного опису, тип елементів словника **value_type** визначається як пара елементів типу **Key** і **T**. Перший конструктор створює порожній словник, використовуючи вказаний функціональний об'єкт. Другий конструктор

створює словник і записує в нього елементи, які визначаються діапазоном вказаних ітераторів. Час роботи цього конструктора пропорційний кількості записуваних елементів, якщо вони впорядковані, і квадрату кількості елементів, якщо ні. Третій конструктор є конструктором копіювання.

Як і для всіх контейнерів, для словника визначені деструктори, операція привласнення і операції відношення. Для доступу до елементів по ключу визначений оператор [ ]:

```
T& operator[](const Key & x);
```

За допомогою цього оператору можна не тільки повертати значення елементів, але і оголошувати в словник нові. Як приклад приведемо телефонну книгу, ключем в якій слугує прізвище, а елементом – номер телефону:

```
#include <iostream>
#include <fstream>
#include <string>
#include <map>
using namespace std;

typedef map<string, long, less <string> > map_s1; //1

void main()
{
 int j;
 map_s1::iterator i;
 map_s1 m1;
 string str[] = {"Petya K", "Ivanova N.M.",
                 "Vovochka", "Vasia"};
 long nom[] = {741596 , 3856745, 230546, 745697};
 for(j = 0; j < 4;j++)
 {
  m1[str[j]] = nom[j]; //Занесення в словник
  cout << str[j] <<" "<< nom[j] << endl;
 }
 m1["Petya P."] = 295678; //Доповнення словника
 cout <<"ml:" << endl; // Виведення словника
 for (i = m1.begin(); i !=m1.end(); i++)
 cout << (*i).first << " " <<(*i).second << endl;
 i = m1.begin(); i++; //Виведення другого елементу
 cout << "2 elem: ";
 cout << (*i).first << " " << (*i).second << endl;
```

```
//Виведення елементу по ключу
 cout << "Vasia: "<< m1["Vasia"]<< endl;
}
```

Для ітераторів словника допустимі операції інкремента і декремента, але не операції + і –. Нижче приведений результат роботи програми (зверніть увагу, що словник виводиться у впорядкованому вигляді):

```
Petya K 741596
Ivanova N.M. 3856745
Vovochka 230546
Vasia 745697
ml:
Ivanova N.M. 3856745
Petya K 741596
Petya P. 295678
Vasia 745697
Vovochka 230546
2 elem: Petya K 741596
Vasia: 745697
```

Для пошуку елементів в словнику визначені наступні функції:

```
iterator        find(const key_type& x);
const_iterator  find(const key_type& x) const;
iterator        lower_bound(const key_type& x);
const iterator  lower_bound(const key_type& x) const;
iterator        upper_bound(const key_type& x);
const_iterator  upper_bound(const key_type& x) const;
size_type       count(const key_type& x) const;
```

Функція **find** повертає ітератор на знайдений елемент у разі успішного пошуку або **end()** в іншому випадку. Функція **upper_bound** повертає ітератор на перший елемент, ключ якого не менше **x**, або **end()**, якщо такого елементу немає (якщо елемент з ключем **x** є в словнику, буде повернений ітератор на нього). Функція **lower_bound** повертає ітератор на перший елемент, ключ якого більше **x**, або **end()**, якщо такого елементу немає. Додамо в приведений вище приклад оператори

```
getline(cin, str);
if (m1.find(str) != m1.end())
    cout << m1 [str];
else
 {
```

```
    cout << (*m1.upper_bound(str)).first << " " ;
    cout << (*m1.lower_bound(str)).first << " " ;
}
```

Якщо ввести з клавіатури прізвище, яке є в словнику, буде виведений відповідний номер телефону, а інакше – двічі підряд перше з прізвищ, яке за абеткою слідує за введеним, наприклад:

```
Petya M. // Підкресленням позначено введення користувача
Petya P. Petya P.
```

Функція **count()** повертає кількість елементів, ключ яких рівний **x** (таких елементів може бути 0 або 1).

Для вставки та видалення елементів визначені функції:

```
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator position,const value_type &x);
template <class InputIter>
void  insert(InputIter first, InputIter last);
void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);
void clear();
```

Перша форма функції використовується для вставки в словник пари "ключ-значення". Функція повертає пару, що складається з ітератора, вказуючого на вставлене значення, і булевої ознаки результату операції: **true**, якщо запису з таким ключем в словнику не було (тільки в цьому випадку відбувається додавання), і **false** в іншому випадку (ітератор вказує на існуючий запис). Час роботи функції пропорційний логарифму кількості елементів в словнику. Таким чином, скорегувати існуючий запис, використовуючи функцію вставки, не можна. Це робиться за допомогою операції доступу по індексу. Нижче приведено декілька прикладів вставки в словник:

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

typedef map <string,long,less <string> > map_s1; //1
```

```
void main()
{
 int j, num;
 string str;
 map_s1 m1;
 map_s1::iterator i;
 string st[] = {"Petya K", "Ivanova N.M.",
                "Vovochka", "Vasia"};
 long nom[] = {741596 , 3856745, 230546, 745697};
 for(j = 0; j < 4;j++)
 m1[st[j]] = nom[j]; // Занесення в словник
 map_s1 m2; // Створення порожнього словника
 m2.insert(map_s1::value_type("Lena", 3157725));
 str = "Anna";
 num = 5536590;
 m2.insert(make_pair(str, num));
 num = 5530000;
 // Спроба вставки існуючого запису
 m2.insert(make_pair(str,num));
 i = m1.begin();

 // Вставка в m2 першого елементу словника m1
 m2.insert(*i);
 m2["Lena"] = 2222222; // Корегування елементу

 // Виведення словника
 for (i = m2.begin(); i != m2.end(); i++)
 cout << (*i).first << " " << (*i).second <<  endl;
}
```

Результат роботи програми:

```
Anna 5536590
Ivanova N.M. 3856745
Lena 2222222
```

Друга форма функції **insert** застосовується для прискорення процесу вставки. З цією метою їй передається першим параметром позиція словника, починаючи з якої потрібно здійснювати пошук місця вставки. Вставка виконується тільки у разі відсутності значення **x** в словнику. Функція повертає ітератор на елемент словника з ключем, що міститься в **x**. Наприклад, якщо відомо, що елементи поміщатимуться в словник в порядку зростання, можна передавати першим параметром у

функцію вставки позицію попереднього елементу (в цьому випадку час вставки є константою):

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

typedef map <string,long, less <string> > map_s1;
typedef pair <string, long > pair_s1;

void main()
{
 pair_s1 p[3] = { pair_s1("Anna", 123123),
 pair_s1("Maria", 234234),
 pair_s1("Teresa", 345345)};
 map_s1 m1;
 map_s1::iterator i = m1.begin();
 for (int k = 0; k<3; k++)
 i = m1.insert(i, p[k]); // sic!
 for (i = m1.begin(); i != m1.end(); i++)
 cout << (*i).first <<" " << (*i).second << endl;
}
```

Результат роботи програми:

```
Anna 123123
Maria 234234
Teresa 345345
```

Третя форма функції **insert** використовується для вставки групи елементів, яка визначається діапазоном ітераторів. Функції видалення елементів і очищення словника аналогічні однойменним функціям інших контейнерів: перша форма функції **erase** видаляє елемент словника з позиції, заданої ітератором, друга – по заданому ключу, а третя видаляє діапазон елементів. Операції вставки в словник не приводять до псування пов'язаних з ними ітераторів і посилань, а операції видалення роблять недійсними тільки ітератори і посилання, пов'язані з елементами, які видаляються.

Приклад вставки, видалення, пошуку і впорядкування елементів елементів в словнику:

```
#include <iostream>
#include <string>
```

```
#include <map>
using namespace std;

// Предикат less
typedef map <string ,int, less<string> > mp; //1

// Предикат greater
// typedef map <string,int, greater<string> > mp; //2
typedef pair <string, int > pair_sl;

void main()
{
 mp ob;
 ob["5"] = 100;
 ob["4"] = 200;
 ob["3"] = 300;
 ob["1"] = 400;
 ob["2"] = 500;
 mp::iterator it;
 string aa,pd = "--------------------------------\n";
 for( it = ob.begin(); it != ob.end(); it++)
      cout<<(*it).first <<"    "<<(*it).second<<endl;
 cout << pd;
 it = ob.begin(); it++; // Виведення другого елементу
 cout << (*it).first <<"    "<<  (*it).second<<endl;
 cout << pd ;
 int num = 888;
 aa = "222";
 ob.insert(make_pair(aa,num));
 // Спроба вставки існуючого запису
 num = 999;
 ob.insert(make_pair(aa,num));it++;
 cout <<(*it).first <<" "<<  (*it).second << endl;
 cout << pd ;
 for( it = ob.begin(); it != ob.end(); it++)
      cout<<(*it).first <<"    "<<(*it).second<<endl;
 cout << pd ;
 //Пошук значення по ключу.
 aa = "3";
 it = ob.find(aa);
 cout <<ob[aa] <<"  "<<(*it).second;
 cout<<endl <<pd;
```

```
 it = ob.begin();
 ob.insert(*it);
 for( it = ob.begin(); it != ob.end(); it++)
      cout<<(*it).first <<"  "<<(*it).second<<endl;
 cout<<pd;
 pair_sl p[3] =
 {
  pair_sl("Anna", 123123),
  pair_sl("Maria", 234234),
  pair_sl("Teresa", 345345)
 };

 mp ob1;
 it =  ob1.begin();
 for (int k = 0; k<3; k++)
      it = ob1.insert(it, p[k]);
 ob1.erase("Maria");
 for( it = ob1.begin(); it != ob1.end(); it++)
      cout<<(*it).first <<"  "<<(*it).second<<endl;
}
```

Результат роботи програми:

```
1   400
2   500
3   300
4   200
5   100
--------------------
2   500
--------------------
222 888
--------------------
1   400
2   500
222 888
3   300
4   200
5   100
--------------------
300 300
--------------------
```

```
1    400
2    500
222  888
3    300
4    200
5    100
---------------------
Anna     123123
Teresa   345345
```

У даному прикладі третім параметром в шаблоні словника передається шаблон функціонального об'єкту, який називається предикатом. Докладніша інформація про предикати буде представлена в наступному розділі. У нашому випадку використовується предикат **less**<**string**>, забезпечуючий впорядковане по зростанню виведення словника за ключем типу **string**. У тексті програми закоментований предикат **greater**<**string**>. При його використанні дані словника виводяться в зворотному порядку. Слід звернути увагу на вставку пари з числом 999. Вона не виконана, оскільки в словнику вже є пара з ключем "222".

Розглянемо приклад роботи із словником, що включає ключ у вигляді цілих чисел і значення у вигляді структури. Тут показано формування словника, виведення його та вставку елементів.

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

struct str
{
 int a;
 string dd;
};

typedef map <int ,str> mp;

void main()
{
 mp ob;
 str struc;
 struc.a = 100;
 struc.dd = "aaa";
 int num;
```

```
 for(int i=1; i<5;i++)
 {
  struc.a = 100*i;
  struc.dd +=  (char)(i+65);
  ob[i] = struc;
 }
 mp::iterator it;
 for( it = ob.begin(); it != ob.end(); it++)
      cout <<(*it).first<<"    "<<((*it).second).a <<
           "    "<<((*it).second).dd << endl;
 it = ob.begin(); it++; // Вывод второго элемента
 cout << "2 - element: ";
 cout<<(*it).first <<"    "<<((*it).second).dd<<endl;
 str vs;
 vs.a = 88;
 vs.dd = "TTT";
 ob.insert(make_pair(200,vs));
 for( it = ob.begin(); it != ob.end(); it++)
      cout <<(*it).first <<"    "<<((*it).second).a
           << "    "<<((*it).second).dd << endl;
}
```

Результат роботи програми:

```
1    100    aaaB
2    200    aaaBC
3    300    aaaBCD
4    400    aaaBCDE
2 - element: 2    aaaBC
1    100    aaaB
2    200    aaaBC
3    300    aaaBCD
4    400    aaaBCDE
200    88    TTT
```

10.3.3 Множини (set)

Множина – це асоціативний контейнер, що містить тільки значення ключів, тобто тип **value_type** відповідає типу **Key**. Значення ключів мають бути унікальні. Шаблон множини має два параметри: тип ключа і тип функціонального об'єкту, що визначає відношення "менше":

```
template <class Key, class Compare = less<Key> >
class set
{
 public:
 typedef Key key_type;
 typedef Key value_type;
 explicit set(const Compare& comp = Compare());
 template <class InputIter>
 set(InputIter first, InputIter last,
 const Compare& comp = Compare());
 set(const set<Key, Compare>& x);
 pair<iterator,bool> insert(const value_type& x);
 iterator insert(iterator position,const value_type&x);
 template <class InputIter>
 void  insert(InputIter first , InputIter last);
 void erase(iterator position);
 size_type erase(const key_type& x);
 void erase(iterator first , iterator last);
 void swap(set<Key, Compare>&);
 void clear();
 iterator find(const key_type& x) const;
 size_type count(const key_type& x) const;
 iterator lower_bound(const key_type& x) const;
 iterator  upper_bound(const key_type& x) const;
 pair<iterator.iterator> equal_range
                          (const key_type& x) const;
...
}
```

З опису, приведеного з скороченнями, видно, що інтерфейс множини аналогічний інтерфейсу словника. Нижче приведений простий приклад, в якому створюється множина цілих чисел:

```
#include <iostream>
#include <set>
using namespace std;

typedef set<int,less<int> > seti;
seti::iterator i;

void main()
{
```

```
 int a[4] = {4,2,1,2};
 // Створюється порожня множина
 seti s1;
 // Множина створюється копіюванням масиву
 seti s2(a, a + 4);
 seti s3(s2);// Працює конструктор копіювання
 s2.insert(10);// Вставка елементів
 s2.insert(6);
 for ( i = s2.begin(); i != s2.end();i++)
 cout << *i << " " ;
 cout << endl;
 // Змінна для зберігання результату equal_range;
 pair <seti::iterator, seti::iterator > p;
 p = s2.equal_range(2);
 cout << *(p.first) <<" "<< *(p.second) <<  endl;
 p = s2.equal_range(5);
 cout << *(p.first) <<" "<<  *(p.second) <<  endl;
}
```

Результат роботи програми:

```
1 2 4 6 10
2 4
6 6
```

Як і для словника, елементи в множині зберігаються відсортованими. Елементи, що повторюються, в множину не заносяться.

Приведена нижче програма формує для заданого текстового файлу вказівку, тобто впорядкований за абеткою список слів, що зустрічаються в ньому, для кожного з яких показані номери рядків, що містять це слово. Якщо слово зустрічається в рядку більше одного разу, номер рядка виводиться один раз.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <set>
#include <map>
using namespace std;

typedef set<int, less<int> > set_i;
typedef map<string, set_i, less<string> > map_ss;
```

```
bool wordread (ifstream &in,string &word, int &num)
{
 char ch;
 // Пропуск до першої букви:
 for (;;)
 {
  in.get(ch);
  if (in.fail()) return false;
  if (isalpha(ch) || ch == '_') break;
  if (ch == '\n') num++;
 }
 word = "";
 // Пошук кінця слова:
 do
 {
  word += tolower(ch);
  in.get(ch);
 }
 while (!in.fail() && (isalpha(ch) || ch == '_'));

 if (in.fail()) return false;
 in.putback(ch); // Якщо символ – '\n'
 return true;
}

void main()
{
 map_ss m;
 map_ss::iterator im;
 set_i::iterator is, isbegin, isend;
 string word;
 int num = 1;
 ifstream in ("1.txt");
 if (!in)
 {
  cout << "Cannot open input file.\n";
  exit(1);
 }
 while (wordread(in, word, num))
 {
  im = m.find(word);
  if (im == m.end())
```

```
  im = m.insert(map_ss::value_type(word,
                set_i())).first;
  (*im).second.insert(num);
 }
 for (im = m.begin(); im != m.end(); im++)
 {
  cout << setiosflags(ios::left ) <<
       setw(15) << (*im).first.c_str();
  isbegin = (*im). second.begin();
  isend = (*im).second.end();
   for (is = isbegin; is != isend; is++)
        cout << " " << *is;
        cout << endl;
 }
}
```

Допустимо, вхідний файл **1.txt** містить наступний текст:

```
#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
#include <set>
#include <map>
const const
```

В цьому випадку програма виведе на екран:

```
const         7
fstream       1
include       1  2  3  4  5  6
iomanip       3
iostream      2
map           6
set           5
string        4
```

Розглянемо роботу програми докладніше. Функція **wordread** зчитує чергове слово з вхідного файлу, вважаючи, що в слово можуть входити алфавітно-цифрові символи і знак підкреслення, а також формує номер поточного рядка. Створювана вказівка зберігається в словнику, що складається з пар "рядок – множина". Рядок є ключем, в ньому міститься окреме слово з вхідного файлу. Безліч цілих чисел збе-

рігає номери рядків, в яких зустрічається дане слово. І множина, і словник без нашої участі забезпечують швидкий пошук і впорядковане зберігання елементів без дублікатів. Для кожного слова за допомогою функції **find** перевіряється, чи міститься воно в словнику. Якщо слова немає (функція повернула **end()**), в словник за допомогою функції **insert** додається пара, яка складається з цього слова і порожньої множини (виклик конструктора **set_i()).** Функція **insert** повертає пару, першим елементом якої (**first**) є ітератор на вставлений елемент. Він привласнюється змінній **im**. В наступному операторові відбувається додавання за допомогою функції **insert** нового елементу в множину, яка є другим (**second**) елементом словника. Далі виконується виведення словника. Для кожного слова в циклі виводяться елементи множини номерів рядків.

Завдання

Виконати завдання за допомогою контейнерів **vector** (вектор), **deque** (черга), **list** (список).

Перелік варіантів завдань:

Варіант 1

Є вектор цілих чисел, який складається з n елементів. Визначити добуток додатних парних чисел вектору. Якщо таких елементів немає, вивести повідомлення: "Додатних парних елементів у векторі немає".

Варіант 2

Є черга цілих чисел, яка складається з n елементів. Визначити добуток від'ємних чисел черги, що стоять на парних позиціях. Якщо таких елементів немає, вивести повідомлення: "Від'ємних елементів в черзі немає".

Варіант 3

Є список цілих чисел, який складається з n елементів. Визначити суму і кількість непарних чисел списку. Якщо таких елементів немає, вивести повідомлення: "Непарних елементів в списку немає".

Варіант 4

Є вектор цілих чисел, який складається з n елементів. Визначити в ньому суму і кількість чисел, кратних 5. Якщо таких елементів немає, вивести повідомлення: "Елементів кратних 5 у векторі немає".

Варіант 5

Є черга цілих чисел, яка складається з n елементів. Визначити у ній кількість чисел, кратних 2, і чисел, не кратних 3. Якщо таких елементів немає, вивести повідомлення: "Чисел, кратних 2 і чисел не кратних 3 в черзі немає".

Варіант 6

Є список цілих чисел, який складається з n елементів. Визначити кількість і добуток елементів списку, які знаходяться в діапазоні [0; 7]. Якщо таких елементів немає, вивести повідомлення: "Елементів з діапазону [0; 7] в списку немає".

Варіант 7

Є вектор цілих чисел, який складається з n елементів. У ньому визначити середнє арифметичне значення чисел, кратних трьом. Якщо таких елементів немає, вивести повідомлення: "Елементів кратних 3 у векторі немає".

Варіант 8

Є черга цілих чисел, яка складається з n елементів. Визначити середнє арифметичне значення елементів черги, які містяться в діапазоні [-3; 5]. Якщо таких елементів немає, вивести повідомлення: "Елементів, що задовольняють вимозі, в черзі немає".

Варіант 9

Є список цілих чисел, який складається з n елементів. У ньому визначити мінімальний елемент та його порядковий номер.

Варіант 10

Є вектор цілих чисел, який складається з n елементів. Визначити добуток непарних елементів вектору, що стоять на парних позиціях. Якщо таких елементів немає, вивести повідомлення: "Непарних елементів, що стоять на парних позиціях у векторі немає".

Варіант 11

Є черга цілих чисел, яка складається з n елементів. Визначити середнє арифметичне значення елементів черги, кратних восьми. Якщо таких елементів немає, вивести повідомлення: "Елементів кратних 8 в черзі немає".

Варіант 12

Є список цілих чисел, який складається з n елементів. У ньому серед додатних елементів визначити максимальний елемент.

Варіант 13

Є вектор цілих чисел, який складається з n елементів. Визначити добуток додатних елементів вектору та їх кількість. За відсутності додатних чисел вивести повідомлення "Додатних чисел у векторі немає".

Варіант 14

Є черга цілих чисел, яка складається з n елементів. Обчислити кількість елементів черги, кратних семи. За відсутності таких елементів вивести повідомлення "Елементів кратних 7 немає".

Варіант 15

Є список цілих чисел, який складається з n елементів. Визначити середнє арифметичне значення мінімального і максимального елементів списку.

Варіант 16

Є вектор цілих чисел, який складається з n елементів. Визначити суму і добуток додатних чисел, що стоять на парних позиціях.

Варіант 17

Є черга цілих чисел, яка складається з n елементів. Визначити середнє арифметичне значення додатних елементів і середнє арифметичне значення від’ємних елементів.

Варіант 18

Є список цілих чисел, який складається з n елементів. Визначити окремо кількість від’ємних чисел, кількість додатних чисел і чисел, рівних нулю.

Варіант 19

Є вектор цілих чисел, який складається з n елементів. Визначити суму додатних парних чисел вектору. У разі відсутності додатних парних чисел вивести повідомлення "Додатних парних чисел у векторі немає".

Варіант 20

Є черга цілих чисел, яка складається з n елементів. Визначити середнє арифметичне значення чисел черги, що стоять на парних позиціях, і середнє арифметичне значення чисел, що стоять на непарних позиціях.

РОЗДІЛ 11. АЛГОРИТМИ

11.1 Ітератори

Ітератор – це узагальнення поняття вказівки для роботи з різними структурами даних стандартним способом. Для того, щоб можна було реалізувати алгоритми, які коректно і ефективно працюють з даними різної структури, стандарт визначає не тільки інтерфейс, але й вимоги до часу доступу за допомогою ітераторів. Оскільки ітератор є узагальненням поняття "вказівка", семантика у них однакова, і всі функції, що приймають в якості параметрів ітератори, можуть також працювати із звичайними вказівками. У стандартній бібліотеці ітератори використовуються для роботи з контейнерними класами, потоками і буферами потоків.

У ітераторах використовуються поняття "Поточний вказуваний елемент" і "вказати на наступний елемент". Доступ до поточного елементу послідовності виконується аналогічно вказівкам за допомогою операцій "*" та "->". Перехід до наступного елементу – за допомогою операції інкремента ++. Для всіх ітераторів визначені також привласнення, перевірка на рівність і нерівність.

Дані можуть бути організовані різним чином – наприклад, у вигляді масиву, списку, вектора або дерева. Для кожного виду послідовності потрібний свій тип ітератора, що підтримує різні набори операцій. Відповідно до набору забезпечуваних операцій *ітератори діляться на п'ять категорій,* описаних в таблиці 11.1.

Нехай **i** та **j** – ітератори одного виду, **x** – змінна того ж типу, що і елемент послідовності, **n** – ціла величина. Тоді допустимі вирази:

i++ ; ++i; i = j; i!=j

Таблиця 11.1 Категорії ітераторів

Категорія ітератора	Операції	Контейнери
вхідний (input)	x = *i	всі
вихідний (output)	*i = x	всі
прямий (forward)	x = * i , * i = x	всі
двонаправлений (bidirectional)	x = *i , *i = x, -- i , i--	всі
довільного доступу (random access)	x = *i , *i = x, -- i , i--, i + n, i - n, i += n, i -= n, i < j , i > j , i <= j , i >= j	всі, окрім list

Ітераторні класи і функції описані в заголовному файлі <**iterator**>. При використанні стандартних контейнерів цей файл підключається автоматично. Ітератори можуть бути константними. Константні ітератори використовуються тоді, коли змінювати значення відповідних елементів контейнера немає необхідності.

Перераховані ітератори мають свої різновиди, так звані *адаптери ітераторів.* Розглянемо наступні адаптери: зворотні ітератори, ітератори вставки та потокові ітератори.

Зворотний ітератор **reverse_iterator,** що є різновидом двонаправлених ітераторів та ітераторів довільного доступу, дозволяє продивлятися послідовність у зворотному напрямі.

Наприклад, щоб продивитись вектор в зворотному порядку, можна скористатися наступним циклом:

```
vector <int> v;
for (vector <int> reverse_iterator i = v.rbegin();
     i != v.rend; ++i)
cout << *i << "   ";
```

Якщо контейнер оголошений як **const** (наприклад, в списку передаваних функції параметрів), то потрібно використовувати ітератор з префіксом **const _const_reverse_iterator.**

Ітератори вставки призначені для додавання нових елементів в початок, кінець або довільне місце контейнера. У стандартній бібліотеці визначено три шаблони класів ітераторів вставки, побудованих на основі вихідних ітераторів: **back_insert_iterator, front_insert iterator** і **insert iterator**. У цих ітераторах, а вони як і всі ітератори, є класами, визначено три функції: **back_inserter()** – вставляє елементи в кінець контейнера, **front_inserter()** – в початок, а **inserter()** – перед елементом, на який посилається її аргумент-ітератор. Приклад їх використання приведений нижче при описі алгоритмів.

Потокові ітератори введені для того, щоб стандартні алгоритми, які розглядаються при описі алгоритмів могли безпосередньо використовувати потоки введення/виведення. Потоки представляються у вигляді послідовностей. Визначено два шаблони класів потокових ітераторів: ітератор вхідного потоку **istreamiterator** і ітератор вихідного потоку **ostreamiterator**.

11.2 Функціональні об'єкти

Функціональним об'єктом називається клас, в якому визначена операція виклику функції (див. розділ 5.6). Ці об'єкти використовуються як параметри стандартних алгоритмів для завдання призначених для користувача критеріїв порівняння об'єктів або способів їх обробки. У тих алгоритмах, де в якості параметру можна використовувати функціональний об'єкт, можна використовувати і вказівку на функцію. При цьому застосування функціонального об'єкту може виявитися ефективнішим, оскільки операцію () можна визначити як вбудовану.

Стандартна бібліотека надає безліч функціональних об'єктів, необхідних для її ефективного використання. Вони описані в заголовному файлі <**functional**>.

У стандартній бібліотеці визначені шаблони функціональних об'єктів для операцій порівняння і логічних операцій, визначених в мові C++. Вони повертають значення типу **bool**, тобто є **предикатами**.

Таблиця 11.2. Предикати стандартної бібліотеки

Ім’я	Тип	Результат
equal_to	бінарний	x == y
not_equal_to	бінарний	x != y
greater	бінарний	x > y
less	бінарний	x < y
greater_equal	бінарний	x>= y
less_equal	бінарний	x<= y
log1cal_ancl	бінарний	x&& y
log1cal_or	бінарний	x || y
log1cal_not	унарний	!x

Нижче приведений шаблон об'єкту equal_to (решта об'єктів описана аналогічним чином):

```
template <class T> struct equal_to:
binary_function<T, T, bool>
{
 bool operator()(const T& x, const T& y) const
 {
  return x == y;
 }
};
```

Програміст може описати власні предикати для визначення критеріїв порівняння об'єктів. Це необхідно, коли контейнер складається з елементів призначеного для користувача типу.

За допомогою бінарних предикатів можна порівнювати два різні об'єкти. Часто потрібно порівняти об'єкт не з іншим об'єктом, а з константою. Щоб використовувати для цього той же самий предикат, потрібно зв'язати один з двох його аргументів з константою. Для цього використовуються зв’язувачі **bind2nd** і **bindlst**, що дозволяють пов'язати з конкретним значенням відповідно другий і перший аргумент бінарної функції. Зв’язувачі реалізовані в стандартній бібліотеці як шаблони функцій, що приймають першим параметром функціональний об'єкт **f** з двома аргументами, а другим – прив'язуване значення **value**. Результатом виклику функ-

ції є функціональний об'єкт, створений з вхідного об'єкту **f** шляхом "підстановки" **value** в його перший або другий аргумент.

Припустимо, потрібно обчислити кількість елементів цілочисельного масиву, які менше 40:

```
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

void main()
{
 int m[8] = {45, 65, 36, 25, 674, 2, 13, 35};
 cout<<count_if(m, m + 8, bind2nd(less<int>(), 40));
}
```

У цій програмі для підрахунку кількості елементів застосовується алгоритм стандартної бібліотеки **count_if**. Для його використання потрібно підключити заголовний файл <**functional**>. Першими двома параметрами **count_if** мають бути ітератори, які визначають початок і кінець оброблюваної послідовності. В якості ітераторів масиву використовуються вказівки на його елементи. Третій параметр функції **count_if** має бути бінарною функцією або функціональним об'єктом. Функція **bind2nd** перетворює умову порівняння x < y в умову x < 40.

Для того, щоб застосовувати зв’язувачі до звичайних вказівок на функції, потрібні спеціальні перетворювачі, або адаптери. Стандартна бібліотека визначає два функціональні об'єкти – вказівку на унарну функцію **pointer_to_unary_function** та вказівку на бінарну функцію **pointer_to_binary_function**, а також дві функції-адаптери **ptr_fun** з одним і двома аргументами, які перетворять переданий їм в якості параметру вказівку на функцію у функціональний об'єкт.

Приклад застосування адаптера функції:

```
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;
struct A{int x, y;};
bool lss(A a1,A a2){return a1.x < a2.x;}

void main()
{
 A ma[5] = {{2, 4},{3,1},{2,2}, {1,2}, {1,2}};
```

```
 A elem = {3,0};
 cout<<count_if(ma, ma+5,bind2nd(ptr_fun(lss),elem));
}
```

Функція **lss** задає правило порівняння елементів структури **A**. Алгоритм стандартної бібліотеки **count_if** обчислює кількість елементів структури **ma**, що задовольняють умові, заданій третім параметром. Цим параметром є функціональний об'єкт, створений зв'язувачем **bind2nd** з функціонального об'єкту, отриманого з функції lss за допомогою адаптера **ptr_fun**, і змінної, яка підставляється на місце другого параметра функції. В результаті буде обчислено кількість елементів структур **A**, поле **x** яких менше 3 (результат – 4).

11.3 Алгоритми

Алгоритми STL призначені для роботи з контейнерами і іншими послідовностями. Кожен алгоритм реалізований у вигляді шаблону або набору шаблонів функції, тому може працювати з різними видами послідовностей і даними різноманітних типів. Для налаштування алгоритму на конкретні вимоги користувача застосовуються функціональні об'єкти. Використання стандартних алгоритмів, як і інших засобів стандартної бібліотеки, позбавляє програміста від написання, відлагодження і документування циклів обробки послідовностей, що зменшує кількість помилок в програмі, знижує час її розробки і робить її більш читаною і компактною. Оголошення стандартних алгоритмів знаходяться в заголовному файлі <**algorithm**>, стандартних функціональних об'єктів – у файлі <**functional**>.

Всі алгоритми STL можна розділити на чотири категорії:

– немодифікуючі операції з послідовностями;
– модифікуючі операції з послідовностями;
– алгоритми, пов'язані з сортуванням;
– алгоритми роботи з множинами і пірамідами;

Крім того, бібліотека містить узагальнені чисельні алгоритми, оголошення яких знаходяться у файлі <**numeric**>. В якості параметрів алгоритму передаються ітератори, що визначають початок і кінець оброблюваної послідовності. Вид ітераторів визначає типи контейнерів, для яких може використовуватися даний алгоритм. Наприклад, алгоритм сортування (**sort**) вимагає для своєї роботи ітератори довільного доступу, тому він не працюватиме з контейнером **list**. Алгоритми не виконують перевірку виходу за межі послідовності.

При описі параметрів шаблонів алгоритмів використовуються наступні скорочення:

In – ітератор для читання;
Out – ітератор для запису;
For – прямий ітератор;
Bi – двонаправлений ітератор;

Ran – ітератор довільного доступу;
Pred – унарний предикат (умова);
Binpred – бінарний предикат;
Comp – функція порівняння;
Op – унарна операція;
BinOp – бінарна операція.

11.3.1 Немодифікуючі операції з послідовностями

Алгоритми цієї категорії продивляються послідовність, не змінюючи її значень.

Таблиця 11.3. Немодифікуючі операції з послідовностями

Алгоритм	Виконувана функція
adjacent_find	Знаходження пари сусідніх значень
count	Підрахунок кількості входжень значення в послідовність
count_if	Підрахунок кількості виконань умови в послідовності
equal	Попарна рівність елементів двох послідовностей
find	Знаходження першого входження значення в послідовність
find_end	Знаходження останнього входження однієї послідовності в іншу
find_first_of	Знаходження першого значення з однієї послідовності в іншій
find_if	Знаходження першої відповідності умові в послідовності
for_each	Виклик функції для кожного елементу послідовності
mismatch	Знаходження першого неспівпадаючого елементу в двох послідовностях
search	Знаходження першого входження однієї послідовності в іншу
search_h	Знаходження n-го входження однієї послідовності в іншу

Розглянемо приклади використання деяких алгоритмів.
Шаблон алгоритму **adjacent_find**:

```
tempiate<class For>For adjacent_find(For first,For last);
tempiate<class For, class BinPred>
For adjacent_find(For first, For last, BinPred pred);
```

Перша форма алгоритму знаходить в послідовному контейнері пару сусідніх однакових значень і повертає ітератор на перше з них або кінець послідовності (ітератор на елемент, наступний за останнім). Друга форма знаходить сусідні елементи, що задовольняють умові, заданій предикатом **pred** у вигляді функції або функціонального об'єкту.

Програма знаходження самої лівої пари однакових елементів цілочисельного масиву і пару елементів структури, у яких рівна сума полів):

```
#include <iostream>
#include <algorithm>
using namespace std;

struct A{ int x, y;};
bool f(A &a1, A& a2)
{
 return a1.x + a1.y == a2.x + a2.y;
}

void main ()
{
 int m[8] = {1, 60, 60, 25, 25, 2, 13, 35};
 cout << *(adjacent_find(m, m + 8)); //Виведення: 60
 A ma[5] = {{2,4}, {3,1}, {2,2}, {1,2}, {1,2}};
 //Виведення: 3
 cout<<"\n"<<(*adjacent_find(ma, ma+5, f)).x << endl;
};
```

Алгоритм **count_if** розглянутий вище в цьому розділі.

Алгоритми **find** та **find_if** виконують відповідно пошук заданого значення і пошук значення, відповідного заданому предикату **pred**. Приклад:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

class In
{
 public:
 bool operator()(int x) {return x > 10 && x < 50;}
};
```

```
void main()
{
 int mas[] = {1, 2, 33, 4, 5, 66, 7, 8, 9};
 vector <int> vec;
 vector<int>::iterator f;
 for (int i = 0; i < 9; i++)
 {
  vec.push_back(mas[i]);
  cout << vec[i] << "  ";
 }
 cout <<"\n";
 // Пошук елементу, рівного 66:
 cout << *find(vec.begin(), vec.end(), 66) << endl;
// Пошук елементу, що задовольняє умові 10<x<50:
 f = find_if(vec.begin(), vec.end(), In());
 if(f == vec.end())
    cout<<"Немає елементу в послідовності";
 else cout << *f;
}
```

Результат роботи програми:

```
1 2 33 4 5 66 7 8 9
66
33
```

Алгоритм **for_each** викликається для перегляду елементів контейнера. У нього можна передавати вказівку на функцію, яку потрібно викликати для кожного елементу контейнера, який треба переглянути:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void show(int a){ cout << a << " ";}
void main()
{
 int m[4] = {3, 5, 9, 6};
 int mas[] = {1,2,33,4,5,66,7,8,9};
 vector <int> vec;
 for (int i = 0; i < 9; i++)
 vec.push_back(mas[i]);
```

```
 for_each(mas, mas+9, show);
 cout << "\n";
 for_each(m, m + 4, show);
}
```

Результат роботи програми:

```
1  2  33  4  5  66  7  8  9
3  5  9  6
```

11.3.2 Модифікуючі операції з послідовностями

Алгоритми цієї категорії тим або іншим чином змінюють послідовність, з якою вони працюють. Вони використовуються для копіювання, видалення, заміни і зміни порядку проходження елементів послідовності.

Таблиця 11.4. Немодифікуючі операції з послідовностями

Алгоритм	Виконувана функція
copy	Копіювання послідовності, починаючи з першого елементу
copy_backward	Копіювання послідовності, починаючи з останнього елементу
fill	Заміна всіх елементів заданим значенням
fill_n	Заміна перших n елементів заданим значенням
generate	Заміна всіх елементів результатом операції
generate_n	Заміна перших n елементів результатом операції
iter_swap	Обмін місцями двох елементів, заданих ітераторами
random_shuffle	Переміщення елементів відповідно до випадкового рівномірного розподілу
remove	Переміщення елементів із заданим значенням
remove_copy	Копіювання послідовності з переміщенням елементів із заданим значенням
remove_copy_if	Копіювання послідовності з переміщенням елементів при виконанні предиката
remove_if	Переміщення елементів при виконанні предиката
replace	Заміна елементів із заданим значенням
replace_copy	Копіювання послідовності із заміною елементів із заданим значенням
replace_copy_if	Копіювання послідовності із заміною елементів при виконанні предиката
replace_if	Заміна елементів при виконанні предиката
reverse	Зміна порядку елементів на зворотний

Продовження таблиці 11.4

Алгоритм	Виконувана функція
reverse_copy	Копіювання послідовності в зворотному порядку
rotate	Циклічне переміщення елементів послідовності
rotate_copy	Циклічне копіювання елементів
swap	Обмін місцями двох елементів
swap_ranges	Обмін місцями елементів двох послідовностей
transform	Виконання заданої операції над кожним елементом послідовності
unique	Видалення рівних сусідніх елементів
unique_copy	Копіювання послідовності з видаленням рівних сусідніх елементів

Розглянемо приклади використання деяких алгоритмів.
Приклад використання алгоритмів **copy**, **copy_backward**

```
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

void main()
{
 int b[4], a[5] = {1, 2, 3, 4, 5}, i;
 copy (a + 1, a + 5, b);
 for (i = 0; i < 4; i++) cout << b[i]; // 2 3 4 5
 cout << endl;
 copy (a + 1, a + 5, a);
 for (i = 0; i < 5; i++) cout <<a[i];  // 2 3 4 5 5
 cout << endl;
 copy_backward (b, b + 3,b + 4);
 for (i = 0; i < 4; i++) cout << b[i]; // 2 2 3 4
 cout << endl;
}
```

Алгоритм **copy** можна застосовувати також для введення і виведення послідовності. Для цього третім параметром задається потоковий ітератор:

```
#include <iostream>
#include <algorithm>
using namespace std;
```

```
void main()
{
 const int N = 5;
 int a[N] = {1, 2, 3, 4, 5};
 copy (a, a + N, ostream_iterator<int>(cout," " ) );
 cout << endl;
}
```

Результат роботи програми:

```
1  2  3  4  5
```

У наступному прикладі розглядається копіювання вектору (**vector**) у список (**list**), використовуючи ітератори **back_inserter()**, **front_inserter()**, **inserter()**.

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

void main()
{
 int a[5] = {1,2,3,4,5};
 vector <int> w(a, a+5);
 list <int> L;
 list<int>::iterator ff = L.begin();

 copy (w.begin(),w.end(), back_inserter(L));
 copy (w.begin(),w.end(), front_inserter(L));
 copy (w.begin(),w.end(), inserter(L,L.begin()));
 cout <<"insert\n";

 for (ff = L.begin();ff != L.end();ff++)
     cout <<*ff <<" " ;
}
```

Результат роботи програми:

```
insert
1 2 3 4 5 5 4 3 2 1 1 2 3 4 5
```

Наведемо приклад використання алгоритмів **fill, flll_n**:

```
#include <iostream>
#include <algorithm>
```

```
using namespace std;

void main()
{
 int a[5], i;
 fill (a, a + 5,1);
 for (i = 0; i < 5; i++) cout << a[i]; // 1 1 1 1 1
 cout <<endl;
 fill_n(a + 2, 2, 0);
 for (i = 0; i < 5; i++) cout << a[i]; // 1 1 0 0 1
 cout << endl;
}
```

Наведемо приклад використання алгоритму **generate**:

```
#include <iostream>
#include <algorithm>
using namespace std;
int f ()
{
 static int i = 1;
 return (++i) * 3;
}

void main()
{
 int a[5],i;
 generate(a,a + 5,f);

 // 6 9 12 15 18
 for (i = 0; i < 5; i++)
      cout << a[i] << " ";
}
```

У даному прикладі елементи заповнюються за допомогою функції **f()**.

Алгоритм **random_shuffle.** Цей алгоритм виконує переміщення (перетасовування) елементів відповідно до випадкового рівномірного розподілу. Третім параметром алгоритму можна задати генератор випадкових чисел. Це дозволяє отримувати різні результати при кожному запуску програми. Генератор може бути функцією або функціональним об'єктом, одержуючим аргумент **n** типу **int** та що повертає ціле число в діапазоні від 0 до **n**.

```
#include <iostream>
#include <algorithm>
#include <time.h>
using namespace std;

struct random_gen
{
 random_gen(){srand((unsigned int)time(NULL));}
 int operator()(int n){return rand() % n;}
};

void main()
{
 int a[5] = {1,2,3,4,5};
 random_shuffle(a, a + 5,random_gen());

// 5 3 4 1 2
 for (int i = 0; i < 5; i++) cout << a[i] << " ";
 cout << endl;
}
```

Алгоритми **remove, remove_if.** Приклад застосування **remove** для видалення елементів вектора, значення яких дорівнюють 2:

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

void main()
{
 vector<int> a;
 int i;
 for (i = 0; i < 5; i++) a.push_back(i);
 for (i = 0; i < 5; i++) a.push_back(i);
 for (i = 0; i < a.size(); i++) cout<<a[i];
 cout<<endl;
 vector<int>::iterator k,p;
 p = remove(a.begin(), a.end(),2);
 for (i = 0; i < a.size(); i++) cout << a [i];
 cout << endl;
 for (k = a.begin(); k != p; k++) cout << *k;
}
```

Результат роботи програми:

```
0123401234
0134013434
01340134
```

Приклад застосування **remove_if** спільно з методом **erase** для видалення елементів вектора, значення яких лежать в межах від 10 до 50:

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool In (int x) {return x > 10 && x < 50;}

void main()
{
 vector<int> a;
 int i;
 for (i = 1; i<10; i++) a.push_back(i*10);
 for (i = 0; i <a.size(); i++) cout << a[i] << " ";
 cout << endl;
 vector<int>::iterator new_end =
              remove_if(a.begin(), a.end(), In);
 a.erase(new_end, a.end());
 for (i = 0; i< a.size(); i++) cout << a[i] << " ";
 cout << endl;
}
```

Результат роботи програми:

```
10 20 30 40 50 60 70 80 90
10 50 60 70 80 90
```

Алгоритми **replace, replace_if, replace_copy, replace_copy_if.**

Розглянемо приклад, в якому виконується копіювання вектора **a** в новий вектор **b** (використовується ітератор вставки) із заміною всіх елементів, які належать діапазону від 10 до 50, на значення 33:

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
```

```
bool In (int x) {return x > 10 && x < 50;}

void main()
{
 vector<int> a, v;
 vector<int>::iterator i;
 for (int k = 1; k < 10; k++)a.push_back(k * 10);
 for (i = a.begin(); i != a.end(); i++)cout<<*i<<" ";
 cout << endl;
 replace_copy_if(a.begin(), a.end(),
                 inserter(v, v.begin()), In,33);
 for (i = v.begin(); i != v.end(); i++)cout<<*i<<" ";
 cout << endl;
}
```

Результат роботи програми:

```
10 20 30 40 50 60 70 80 90
10 33 33 33 50 60 70 80 90
```

Алгоритм **transform.** У приведеному нижче прикладі перший виклик **transform** виконує перетворення масиву **a** по формулі: $a_i = a_i - b_i$, другий виклик змінює знак у елементів масиву **b** за допомогою стандартного функціонального об'єкту **negate**.

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

struct preobr: binary_function <double,double,double>
{
 double operator()(double x, double y) const
 {
  return x * x - y * y;
 }
};

void main()
{
 const int m = 5;
 double a[m] = {5, 3, 2, 3, 1},
 b[m] = { 1 ,10, -3, 2, -4};
```

```
 transform(a, a + m, b, a, preobr());
 transform(b, b + m, b, negate<double>());
 int i;
 for (i = 0; i<m; i++) cout << a[i] << " ";
 cout << endl;
 for (i = 0; i<m; i++) cout << b[i] << " ";
 cout<< endl;
}
```

Результат роботи програми:

```
24 -91 -5 5 -15
-1 -10 3-2 4
```

11.3.3 Алгоритми, пов'язані з сортуванням

Алгоритми цієї категорії упорядковують послідовності, виконують пошук елементів, злиття послідовностей, пошук мінімуму і максимуму, лексикографічне порівняння, перестановки тощо.

Таблиця 11.5. Алгоритми, пов'язані з сортуванням

Алгоритм	Виконувана функція
binary_search	Пошук заданого значення
equal_range	Знаходження послідовності елементів із заданим значенням
lnplace_merge	Злиття відсортованих послідовностей одного діапазону
lexicographical_compare	Лексикографічно перша з двох послідовностей
lower_bound	Знаходження першого входження заданого значення
max	Більше з двох значень
max_element	Найбільше значення в послідовності
merge	Злиття відсортованих послідовностей
min	Менше з двох значень
min_element	Найменше значення в послідовності
next_permutat1on	Наступна перестановка в лексикографічному порядку
nth_element	Поміщення n-го елементу на задане місце
partial_sort	Часткове сортування
partial_sort_copy	Часткове сортування з копіюванням
partition	Переміщення вперед елементів, що задовольняють умові

Продовження таблиці 11.5

Алгоритм	Виконувана функція
prev_permutation	Попередня перестановка в лексикографічному порядку
sort	Сортування
stable_partition	Переміщення вперед елементів, що задовольняють умові, із збереженням їх відносного порядку
stable_sort	Сортування, що зберігає порядок для однакових елементів
upper_bound	Знаходження першого елементу, більшого, ніж задане значення

Розглянемо приклади використання деяких алгоритмів. Для кожного з алгоритмів існує дві форми: одна використовує операцію "<", а інша – функцію порівняння, задану користувачем. Багатьом алгоритмам потрібні ітератори довільного доступу.

Алгоритм **lexicographical_compare**. Він виконує поелементне порівняння двох послідовностей або з використанням операції "<", або за допомогою заданої функції **comp**. Повертається **true**, якщо перша послідовність лексикографічно менше другої (тобто черговий елемент першої послідовності опинився менше відповідного елементу другої), і **false** в іншому випадку. Якщо довжини послідовностей не збігаються, елементи, яких бракує вважаються меншими відповідних елементів іншої послідовності.

Приклад:

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;


void main()
{
 const int m = 5;
 double a[m] = {5, 3, 2, 3, 1},b[m]={5, 3, 2, 3, 2},
 c[m] = {5, 3, 1, 3, 10};
 cout << lexicographical_compare(a,a+m,b,b+m);  //1
 cout << lexicographical_compare(a,a+m,c,c+m);  //0
 cout << lexicographical_compare(a,a+m,b,b+m,
         greater<int>());  //0
}
```

Алгоритм **merge** виконує злиття відсортованих послідовностей.
Приклад:

```
#include <iostream>
#include <algorithm>
using namespace std;

void main()
{
 const int m = 5;
 double a[m] = {3, 4, 8, 17, 20},
 b[m] = {5, 6, 8, 10, 35}, c[m * 2];
 int i;
 merge(a, a + m, b, b + m, c);

 // 3 4 5 6 8 8 10 17 20 35
 for (i = 0; i < m * 2; i++)cout << c [ i ] << " ";
 cout << endl;
}
```

Алгоритми **next_permutation, prev_permutation.**
Елементи будь-якої послідовності можна розташувати різними способами.
Для послідовності довжини **n** таких перестановок існує **n!** (1*2*...*n).
Алгоритм **next_permutation** проводить чергову перестановку в лексикографічному порядку, а алгоритм **prev_permutation** – попередню. Алгоритми повертають булеве значення **true**, якщо наступна перестановка існує, і **false** в іншому випадку.

```
template<class Bi>
bool next__permutation(Bi first , Bi last);
template<class Bi, class Compare>
bool next_permutation(Bi first, Bi last, Compare comp);
template<class Bi>
bool prev_permutation(Bi first, Bi last);
template<class Bi, class Compare>
bool prev_permutation(Bi first, Bi last, Compare comp);
```

Приклад:

```
#include <iostream>
#include <algorithm>
using namespace std;
```

```
void main()
{
 const int m = 3;
 int a[m]={1, 4, 2}, b[m];
 int i;
 copy(a, a + m, b);
 cout << " next_permutation(a, a + m);" << endl;
 while (next_permutation(a, a + m))
 {
  for (i = 0; i < m; i++) cout << a [ i ] << " ";
  cout << endl;
 }
 cout << " prev_permutation(b, b + m);" << endl;
 while (prev_permutation(b, b + m))
 {
  for (i = 0; i < m; i++) cout << b [ i ] << " ";
  cout << endl;
 }
}
```

Результат роботи програми:

```
 next_permutation(a, a + m):
2 1 4
2 4 1
4 1 2
4 2 1
 prev_permutation(b, b + m):
1 2 4
```

Алгоритми **partial_sort, partial_sort_copy.**

Алгоритм **partial_sort** також виконує часткове сортування послідовності.

Після виконання алгоритму елементи від **first** до **middle** розташовуватимуться в такому ж порядку, як після повного сортування.

Алгоритм **partial_sort_copy** виконує ті ж дії з копією послідовності.

```
tempiate<class Ran>
void partial_sort(Ran first, Ran middle, Ran last);
template<class Ran, class Compare>
void partial_sort(Ran first, Ran middle,
                  Ran last, Compare comp);
template<class In, class Ran>
Ran partial_sort_copy(In first, In last,
```

```
Ran result_first, Ran result_last);
template<class In, class Ran, class Compare>
Ran partial_sort_copy(In first, In last,
Ran result_first, Ran result_last, Compare comp);
```

Часткове сортування економить час в тих випадках, коли нас цікавлять тільки декілька найбільших або найменших значень, наприклад, "гаряча десятка".

```
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;

void main()
{
 const int m = 8;
 int a[m] = {3, 1, 2, 34, 8, 7, 20, 2};
 int i;
 partial_sort(a, a + 5, a + m, greater<int>());
 for (i = 0 ; i < m; i++) cout << a[i] << " ";
 cout << endl; // 34 20 8 7 3 1 2 2
}
```

11.3.4 Узагальнені чисельні алгоритми

Для використання чисельних алгоритмів потрібно підключити заголовний файл <**numeric**>.

Таблиця 11.6. Узагальнені чисельні алгоритми

Алгоритм	Виконувана функція
accumulate	Накопичення
inner_product	Скалярний добуток
partial_sum	Розрахунок суми з накопиченням
adjacent_difference	Обчислення різниці між суміжними елементами

Алгоритм **accumulate**.

Перша форма алгоритму **accumulate** використовується для накопичення суми елементів послідовності, заданої ітераторами **first** і **last**. Початкове значення суми (звичайно це 0) задається третім параметром. Тип цього параметра визначає тип результату (функція повертає обчислену суму):

```
template <class In, class T>
T accumulate(In first, In last, T init);
```

Друга форма алгоритму **accumulate** дозволяє виконувати над третім параметром і черговим елементом послідовності задану операцію:

```
template t<class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp binary_op);
```

У приведеному далі прикладі обчислюється сума, добуток і сума квадратів елементів масиву:

```
#include <iostream>
#include <numeric>
#include <functional>
using namespace std;

int sumkv( int s, int x){ return s + x * x;}

void main()
{
 const int m = 5;
 int a[m] = {3, 2, 5, 1, 6},sum = 0,mul = 1,sum2 = 0;
 cout<<accumulate(a, a + m, sum) << endl;
 cout<<accumulate(a,a+m,mul,multiplies<int>())<<endl;
 cout<<accumulate(a, a + m, sum2, sumkv) << endl;
}
```

Результат виконання програми:

```
17
180
75
```

Наступний приклад на базі списку (**list**) демонструє роботу алгоритмів **max_element()**, **accumulate()**, а також застосування алгоритму **accumulate()** для конкатенації рядків.

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
#include <numeric>
using namespace std;
```

```
void main()
{
 int a[5] = {5,1,7,41,2};
 list <int> L(a,a+5);

 list<int>::iterator ff = L.begin();

 for (ff = L.begin();ff != L.end();ff++)
      cout <<*ff <<" " ;

 L.sort();
 cout << "\n\nSort - >";
 for (ff = L.begin();ff != L.end();ff++)
      cout<<*ff <<" " ;

cout<<"\nmax =   " <<*max_element(L.begin(),L.end());
cout<<"\nSumma = " <<accumulate(L.begin(),L.end(),0);

vector<char*> rgs;
rgs.push_back("aaa");
rgs.push_back(" bbb");
rgs.push_back(" ccc");
cout<<"\nConc -> "<<accumulate(rgs.begin(),rgs.end(),
                    string(" nnn ")) << endl;
}
```

Результат виконання програми:

```
5 1 7 41 2

Sort - >1 2 5 7 41
max =   41
Summa = 56
Conc ->   nnn aaa bbb ccc
```

Алгоритм **inner_product**.

Перша форма алгоритму **inner_product** використовується для обчислення скалярного добутку двох послідовностей (скалярним добутком послідовностей **a** і **b** є вираз $\sum a_i b_i$). Початкове значення добутку задається четвертим параметром. Тип цього параметра визначає тип результату (функція повертає обчислений добуток):

```
template t<class In1, class In2, class T>
T inner_product(Inl first1, Inl last1,
                In2 first2, T init);
```

Друга форма алгоритму **inner_product** використовується для виконання над двома послідовностями дій, заданих за допомогою двох функцій або функціональних об'єктів. Перший використовується замість операції додавання, другої, – замість множення:

```
template t<class In1, class In2, class T,
class BinOp1, class BinOp2>
T inner_product(In1 first1, In1 last1, In2 first2,
T init, BinOp1 binary_op1, BinOp2 binary_op2);
```

Наступний виклик обчислює добуток сум відповідних елементів послідовностей **a** і **b**:

```
cout << inner__product(a, a + m, b, mu1,
        multiplies<int>(), plus<int>());
```

Алгоритм **partial_sum**.

Алгоритм **partial_sum** формує послідовності з часткових сум елементів. Наприклад, для послідовності чисел 3 1 2 3 5 результатом буде 3 4 6 9 14, тобто кожен елемент результату дорівнює попередньому елементу, збільшеному на поточний елемент початкової послідовності.

Замість суми четвертим параметром можна задати іншу операцію. Результат можна помістити і в початкову послідовність, для цього третім параметром вказується ітератор на її початок. Функція повертає ітератор на елемент, наступний за останнім результуючої послідовності.

```
template t<class In, class Out>
Out partial_sum(In first, In last, Out result);
template t<class In, class Out, class BinOp>
Out partial_sum(In first, In last, Out result,
                BinOp binary_op);
```

Алгоритм **adjacent_difference**.

Алгоритм **adjacent_difference** виконує обчислення різниці між суміжними елементами, тобто $d_i = a_i - a_{i-1}$. Замість різниці четвертим параметром можна задати іншу операцію. Функція повертає ітератор на елемент, наступний за останнім результуючої послідовності.

```
template t<class In, class Out>
Out adjacent_difference(In first,In last,Out result);
template t<class In, class Out, class BinOp>
Out adjacent_difference(In first,In last,Out result,
                        BinOp binary_op);
```

Цей алгоритм є зворотним попередньому, тобто виклик для однієї і тієї ж послідовності спочатку одного, а потім іншого алгоритму приведе до початкової послідовності.

Завдання

Виконати завдання за допомогою використання контейнерів та алгоритмів.

Перелік варіантів завдань:

Варіант 1

Визначити добуток додатних парних чисел масиву B(n). Якщо таких елементів немає, вивести на екран дисплея повідомлення: "Додатних парних елементів в масиві немає".

Варіант 2

Визначити добуток від'ємних чисел масиву A(n), що стоять на парних позиціях. Якщо таких елементів немає, вивести на екран дисплея повідомлення: "Від'ємних елементів в масиві немає".

Варіант 3

Визначити суму і кількість непарних чисел масиву M(n). Якщо таких елементів немає, вивести на екран дисплея повідомлення: "Непарних елементів в масиві немає".

Варіант 4

У масиві чисел K(n) визначити суму і кількість чисел, кратних 5. Якщо таких елементів немає, вивести на екран дисплея повідомлення: "Елементів кратних 5 в масиві немає".

Варіант 5

У масиві чисел X(n) визначити кількість чисел, кратних 2, і чисел, не кратних 3. Якщо таких елементів немає, вивести на екран дисплея повідомлення: "Чисел, кратних 2 і чисел не кратних 3 в масиві немає".

Варіант 6

Є масив чисел C(n), визначити кількість і добуток елементів масиву, які знаходяться в діапазоні $0 \le C(i) \le 7$. Якщо таких елементів немає, вивести на екран дисплея повідомлення: "Елементів з діапазону [0; 7] в масиві немає".

Варіант 7

У заданому масиві A(n) визначити середнє арифметичне значення чисел, кратних трьом. Якщо таких елементів немає, вивести на екран дисплея повідомлення: "Елементів кратних 3 в масиві немає".

Варіант 8

Визначити середнє арифметичне значення елементів масиву F(n), які задовольняють вимозі $-3 \le F(i) \le 5$. Якщо таких елементів немає, вивести на екран дисплея повідомлення: "Елементів, що задовольняють вимозі, в масиві немає".

Варіант 9

У числовому масиві A(n) визначити мінімальний елемент масиву і його порядковий номер. Результат вивести на екран дисплея.

Варіант 10

Визначити мінімальний елемент числового масиву K(n) і кількість елементів, рівних мінімальному елементу. Вивести на екран дисплея результат.

Варіант 11

Визначити добуток непарних елементів масиву P(n), що стоять на парних позиціях. Якщо таких елементів немає, на екран дисплея вивести повідомлення: "Непарних елементів, що стоять на парних позиціях в масиві немає".

Варіант 12

Визначити середнє арифметичне значення елементів масиву B(n), кратних восьми. Якщо таких елементів немає, на екран дисплея вивести повідомлення: "Елементів кратних 8 в масиві немає".

Варіант 13

У числовому масиві Z(n) серед додатних елементів визначити максимальний елемент масиву. Вивести на екран дисплея результат.

Варіант 14

Визначити кількість від'ємних елементів в масиві A(n) і на їх місце записати нулі. Якщо таких елементів немає, на екран дисплея вивести повідомлення: "Від'ємних елементів в масиві немає".

Варіант 15

Визначити добуток додатних елементів масиву і їх кількість. За відсутності додатних чисел вивести на екран дисплея "Додатних чисел в масиві немає".

Варіант 16

Обчислити кількість елементів цілочисельного масиву V(n), кратних семи. За відсутності таких елементів вивести на екран дисплея "Елементів, кратних 7, немає".

Варіант 17

Для числового масиву Z(n) визначити середнє арифметичне значення мінімального і максимального елементів. Результат вивести на екран дисплея.

Варіант 18

Для заданого масиву A(n) обчислити суму і кількість елементів, що задовольняють умові $2 \le A(i) \le 10$. Результат вивести на екран дисплея.

Варіант 19

Визначити суму і добуток додатних чисел, що стоять на парних позиціях в масиві B(n). Результат вивести на екран дисплея.

Варіант 20

У масиві C(n) визначити середнє арифметичне значення додатних елементів і середнє арифметичне значення від'ємних елементів. Результат вивести на екран дисплея.

СПИСОК ЛІТЕРАТУРИ

1. Стивен Поттс, Тиноти С. Монк. Borland C++. (Минск, Попурри, 1996г. – 741 с.
2. Р. Уинер. Язык Турбо С. – М.: Мир, 1991г. - 384 с.
3. Секунов Н.Ю. Самоучитель Visual C++ 6. – СПб.: БХВ – Санкт-Петербург, 1999. – 960 с.
4. Шилдт Г. Самоучитель C++, 3-е издание. – СПб.: BHV – Санкт-Петербург, 1998. – 688 с.
5. Фейсон Т. Объектно-ориентированное программирование на Borland C++ 4.5. – К.: "Диалектика", 1996. – 544 с.
6. Страуструп Б. Язык программирования C++. – СПб.: БИНОМ, 1999. – 991 с.
7. Страуструп Б. Дизайн и эволюция языка C++. – М.: ДМК, 2000. – 448 с.
8. Мейерс С. Эффективное использование C++. – М.: ДМК, 2000. – 240 с.
9. Лаптев В.В., Морозов А.В., Бокова А.В. C++. Объектно-ориентированное программирование. Задачи и упражнения. – СПб.: Питер, 2007. – 288 с.
10. Павловская Т.А. C/C++. Программирование на языке высокого уровня. – СПб.: Питер, 2003. – 461 с.