Кафедра комп'ютерної інженерії та електроніки

Назва дисципліни – Об`єктно-орієнтоване програмування.

Викладач: проф. Запухляк Руслан Ігорович

1. Запухляк Р.І. Об`єктно-орієнтоване програмування. 2018. Електронний ресурс. http://www.d-learn.pu.if.ua
2. Кравець П.О. Об’єктно-орієнтоване програмування. Львів: Видавництво Львівської політехніки, 2012. 624 с. Розділ 2, 4-10

Розділ 2
КЛАСИ

2.1. Оголошення класу

Клас (*class*) – це структурований тип даних, який інкапсулює (приховує, обмежує область досяжності) оголошення полів даних (атрибутів) та функцій для їх опрацювання. *Функції класу* інакше називаються *методами*. Дані, що мають класовий тип, називаються *об'єктами*, або *екземплярами* класу.

Клас використовується для позначення множини об'єктів, які мають однакову структуру, поведінку та відношення з об'єктами інших класів.

Структура (зміст) класу називається його *протоколом*.

Сукупність методів класу, через які відбувається доступ до елементів класу, називається *інтерфейсом*.

Елементи класу (дані та методи) захищені оголошенням класу:

```
class  назва_класу{
private:
// дані та методи;
protected:
// дані та методи;
public:
// дані та методи;
};
```

Клас C++ визначає три *рівні* захисту своїх елементів:

- `private` (закриті) – діє за замовчуванням;
- `protected` (захищені);
- `public` (відкриті).

Закриті елементи можуть використовуватися тільки у межах класу. Захищені елементи доступні у класі та у похідних від нього класах. Відкриті елементи досяжні у класі та поза класом у межах дії встановленого простору імен.

Клас надає зовнішнім функціям та класам можливість доступу до своїх елементів за допомогою методів, розміщених у відкритій частині.

У протоколі класу дозволяється довільний порядок розміщення та повторення секцій `private`, `protected`, `public`.

Розрізняють два стилі оформлення класів – Borland та Microsoft. У стилі Borland секції класу розміщують у послідовності `private` – `protected` – `public`. У стилі Microsoft секції класу розміщують у зворотному порядку: `public` – `protected` – `private`.

У C++ не дозволяється здійснювати ініціалізацію даних класу при їх оголошенні всередині класу (крім статичних констант). Для цього використовують метод спеціального призначення, який називається конструктором. Назва конструктора збігається з назвою класу. Конструктор не повертає значення, може мати список параметрів, може бути перевантаженим.

У класі за замовчуванням завжди існує конструктор без параметрів (void-конструктор), конструктор копіювання, деструктор та операторна функція присвоєння об'єктів.

Ініціалізація даних конструктором може здійснюватися у *списку ініціалізації* після символу двокрапка, або в тілі конструктора. Константні поля даних, дані з типом посилання, успадковані дані, дані з типом іншого класу (якщо перекритий конструктор за замовчуванням) ініціалізуються конструктором після символу ':' .

Конструктори автоматично викликаються при створенні об'єктів у сегментах коду, стека або у динамічній пам'яті.

Якщо об'єкт виходить із області досяжності коду програми, він автоматично знищується. Наприклад, перед завершенням функції знищуються всі її локальні об'єкти. Для цього викликається метод спеціального призначення – деструктор. Назва деструктора складається з символу ~ (тильда) та назви класу. Деструктор не має параметрів і не повертає значення. Деструктор не може бути перевантажений.

Крім членів класу, оголошення класу може містити прототипи дружніх (friend) функцій або класів. Методи класу та дружні до класу функції мають можливість доступу до елементів усіх частин класу. Рівні захисту діють тільки на членів класу і не діють на друзів класу.

Визначати методи та дружні функції можна як у класі, так і поза класом.

Методи класу та дружні функції, крім операторних, можуть мати параметри за замовчуванням, які задаються в оголошеному у класі прототипі або у заголовку при визначенні функції у класі або за межами класу.

Після оголошення у класі дозволяється довільний порядок звернення до методів (догори або донизу програми).

Елементи класу (окрім друзів, конструкторів, деструктора та операції присвоєння) допускають успадкування, яке полягає у використанні оголошень даних та методів базового класу у похідному класі.

Приклад оголошення та застосування класу:

```
class Base
{
 int x;       // елемент даних
              // (за замовчуванням private)
protected:
```

```
 int y;       // елемент даних
public:
 int z;       // елемент даних
// перевантажені конструктори
 Base(){x=y=z=0;}
 Base(int a){x=a; y=z=0;}
 Base(int a, int b):x(a),y(b){z=0;}
 Base(int a, int b, int c);
// деструктор
 ~Base(){}
// метод класу
  void print() {cout<<x<<' '<<y<<' '<<z<<endl;}
// дружня функція
friend void output(Base a);
 };

// визначення конструктора поза класом
Base::Base(int a, int b, int c)
{x = a; y = b; z = c;}

// визначення дружньої функції
void output(Base a)
{cout<<a.x<<' '<<a.y<<' '<<a.z<<endl;}

// головна функція
void main()
{
// оголошення об'єктів
 Base obj1;          //викликається конструктор без
                     // параметрів
 Base obj2(1);       //викликається конструктор з одним
                     // параметром
 Base obj3(1,2);     //викликається конструктор з двома
                     //параметрами
 Base obj4(1,2,3); //викликається конструктор з
                       // трьома параметрами
// виклик методу
 obj4.print();       // 1 2 3
// виклик дружньої функції
 output(obj4);       // 1 2 3
}
```

У протоколі класу дозволяється оголошувати типи даних за допомогою специфікатора `typedef`. У разі використання внутрішніх типів, досяжних поза класом, необхідно перед їх іменами використовувати назву класу, наприклад:

```
class Base
{
 int x,y,z;
```

```
public:
 typedef void * PVOID;
 Base(int a=0, int b=0, int c=0) {x=a; y=b; z=c;}
 ~Base(){}
};

void main()
{
 Base::PVOID q;
// ...
}
```

2.2. Досяжність елементів класу

Досяжність визначає можливість доступу до елементів `private`, `protected` або `public` частин класу із заданого місця програми. Виконаємо оголошення класів:

```
class Base {
       protected:
       int x;
       public:
       Base(int x1=0)
          { x=x1;}
       // ...
       };

class Derived: public Base
       {int y;
       public:
       Derived(int x1=0, int y1=0):Base(x1)
       {y=y1;}
       // ...
       };
Base a, *p=&a, &r=a;
```

На основі цього оголошення у табл. 2.1 наведено варіанти можливих звернень до елементів класу.

Введемо позначення: FN – Field Name, CN – Class Name, ON – Object Name. Справедливо відношення FN < CN < ON, яке означає таке. Якщо дозволяється доступ за іменем елемента, то дозволяється доступ за іменем класу та іменем об'єкта. Якщо дозволяється доступ за іменем класу, то дозволяється доступ за іменем об'єкта.

Варіанти доступу до елементів класу у межах класу, похідного класу та зовнішніх функцій і класів подано у табл. 2.2

Таблиця 2.1

Синтаксис звернень до елементів класу

Спосіб доступу	Приклад	Право доступу
прямий, за іменем елемента класу (Field Name)	`x`	• для функцій-членів класу (методів) – без обмеження прав доступу; • для методів похідних класів – з врахуванням режимів успадкування;

через об'єкт, вказівник або посилання на об'єкт (Object Name)	`a.x`
	`p->x`
	`(*p).x`
	`r.x`
		• для методів і друзів класу – без обмежень прав доступу;
		• для друзів похідних класів – з врахуванням режимів успадкування;
		• для методів інших класів та зовнішніх функцій – з врахуванням рівнів захисту

з використанням імені класу та операції доступу `::` (Class Name)	`Base::x`	1) так здійснюється доступ до статичних елементів класу: • для методів та друзів класу – без обмежень прав доступу; • для методів та друзів похідних класів – з врахуванням режимів успадкування; • для методів інших класів та зовнішніх функцій – з врахуванням рівнів захисту 2) так методи класу звертаються до перекритих елементів.

Таблиця 2.2

Дія рівнів захисту елементів класу

Клас Base		Клас `Base`		Клас `Derived`		Зовнішні функції та класи
		методи	друзі	методи	друзі	
`private:`	дані	FN	ON	–	–	–
	методи	FN	ON	–	–	–
`protected:`	дані	FN	ON	FN	ON	–
	методи	FN	ON	FN	ON	–
`public:`	дані	FN	ON	FN	ON	ON
	методи	FN	ON	FN	ON	ON

Прочерки означають відсутність доступу до закритої або захищеної частин класу `Base` з похідного від нього класу `Derived` та зовнішніх функцій і класів. Як видно з табл. 2.2, зовнішні функції та класи не мають доступу до `private`- та `protected`-частин класу.

Клас може надати можливість доступу до елементів private- та protected-частини із зовнішніх функцій. Для цього у його public-частині повинні існувати відповідні методи, або зовнішні функції чи класи повинні бути оголошені дружніми до класу Base, наприклад:

```
class  Base
{
  private:
      int x;
  protected:
      int y;
  public:
      int z;
   Base(int a=0,int b=0,int c=0):x(a),y(b),z(c){}
  ~Base(){}

  void print(void);
  friend void output(Base a);
};

void Base::print(){cout<<x<<' '<<y<<' '<<z<<endl;}

void output(Base a)
{  cout<<a.x<<' '<<a.y<<' '<<a.z<<endl;}

void main()
{Base obj(1,2,3);
//  cout<<obj.x<<endl;    // не доступно
//  cout<<obj.y<<endl;    // не доступно
  cout<< obj.z<<endl;    // 3

  obj.Print();           // 1 2 3
  Get(obj);              // 1 2 3
}
```

2.3. Дані класу

Дані (або атрибути) класу не можуть мати класи пам'яті auto, extern, register. Можуть бути:

- звичайними, без кваліфікаторів (int x;)
- статичними (static int x;)
- константними (const int x;)

Можуть мати допустимі для C++ типи, зокрема тип раніше визначеного класу (зовнішнього або в ієрархії успадкування), вказівник або посилання на раніше визначений або на свій клас.

Дані класу не можуть мати тип цього самого класу.

Можуть перекриватися в успадкованих класах. Для доступу до перекритих значень використовується ім'я базового класу та операція дозволу доступу, наприклад, `Base::z`.

2.3.1. Статичні дані

Статичні дані оголошуються за допомогою слова `static` і мають бути визначені поза класом на глобальному рівні оголошень (незалежно від їх розміщення у класі). Статичні дані зберігаються разом з глобальними, окремо від об'єкта класу. При визначенні статичних даних за межами протоколу класу слово `static` не використовується.

Можуть бути ініціалізовані. За замовчуванням ініціалізуються нульовими значеннями.

Статичні дані класу є спільними для всіх об'єктів цього класу та успадкованих від нього класів. За допомогою статичних полів можна передавати інформацію всім об'єктам цього класу. Статичні поля існують навіть тоді, коли не оголошений жоден об'єкт класу.

Звернення до статичних полів даних здійснюється з використанням імені класу та операції дозволу доступу, наприклад: `A::x`. Крім того, для звернення до статичних полів можна використовувати об'єкт, вказівник або посилання на об'єкт класу.

Нехай маємо оголошення класу `A`, який містить статичне поле даних `x` та нестатичне поле `y`:

```
class A {public:
// оголошення статичного елемента даних
     static int x;
            int y;
};

// визначення статичного елемента даних
int A::x=5;

// Оголошення об'єктів класу
A   a, b;
```

На рис. 2.1 зображено схематичну структуру двох об'єктів класу `A`, які перекриваються по статичному полю `x` і мають різні нестатичні поля `y`.

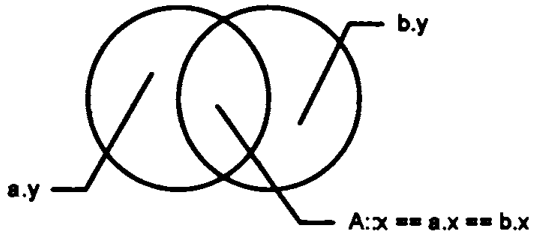

Рис. 2.1. Перетин об'єктів за статичними елементами даних

Статичні поля класу можуть бути ініціалізовані також конструктором у його тілі (всередині блоку {}), але не у списку ініціалізації (після символу ':'). Особливістю такого способу ініціалізації є те, що при кожному створенні нового об'єкта змінюватимуться статичні поля усіх об'єктів класу.

Можливість доступу до статичних полів із зовнішніх функцій визначається рівнями захисту частин класу `private`, `protected`, `public`. Наприклад:

```
class Base
{
private:
static int x;
protected:
static int y;
public:
static int z;
};

// Визначення статичних даних
// необхідне для усіх частин класу
int Base::x=1;
int Base::y=2;
int Base::z=3;

void main()
{
//cout<<Base::x<<endl; // не доступно
//cout<<Base::y<<endl; // не доступно
  cout<<Base::z<<endl; // доступно
}
```

2.3.2. Константні дані

Константні дані оголошуються за допомогою слова `const`. Ініціалізація константних даних здійснюється після двокрапки у списку ініціалізації конструктора, але не у тілі конструктора. Константні дані не можна змінити у програмі. Наприклад:

```
class A {
const float y;
public:
A(float y1) :  y(y1){}
};
```

Константні дані можуть бути оголошені як статичні (або навпаки). Константні статичні дані є спільними для усіх об'єктів класу і не можуть бути змінені у програмі. Ініціалізація таких даних здійснюється поза класом разом з їх визначенням на глобальному рівні з використанням слова const.

Крім того, константні статичні дані *цілого* типу можуть бути проініціалізовані безпосередньо при їх оголошенні у класі. Як і раніше, такі статичні дані потребують визначення поза класом (особливо, якщо константи необхідно зберігати як окремі об'єкти в пам'яті, визначати їх адресу тощо). Наприклад:

```
class A {
 public:
 static const int x=5;
 static const float y;
 };

const int A::x;
const float A::y=7;

void main()
{
cout<<A::x<<' '<<A::y<<endl;
}
```

Ініціалізація константного статичного масиву здійснюється поза класом у місці визначення статичного масиву, наприклад:

```
class A{
 static const int v[3];
 public:
 static void print()
 { for(int i=0; i<3; i++)
    cout << v[i] << ' ';
       cout << endl;
 }

 };

const int A::v[3]={1, 2, 3};

void main(){
A::print();
}
```

У класі можна визначити також символічні константи перелікового типу:

```
class A {
enum {DAY = 30, MONTH = 3, YEAR = 2012};
// ...
};
```

Такі константи можуть мати тільки цілий знаковий тип.

2.4. Функції протоколу класу

Функції, оголошені у протоколі класу, можуть бути:

- членами класу;
- друзями класу.

Функції-члени та друзі класу мають доступ до усіх частин класу.

Якщо тіло функції розміщене у протоколі класу і не містить операторів зміни напрямку обчислень (крім умовного), то за замовчуванням така функція вважається вбудованою, або `inline`-функцією. Якщо тіло функції розміщене за протоколом класу, то така функція теж може бути вбудованою, якщо для її оголошення використовується слово `inline`, наприклад:

```
class A { int x;
          public:
          A(int x1){x=x1;}
          int Get();
        };

inline int A::Get( ) {return x;}
```

Особливістю `inline`-функцій є те, що для їх виклику не формується команда `call`, а тіло функції повністю розміщується у точці виклику. `Inline`-функції не повинні містити структурованих операторів, крім умовного оператора `if-else`.

Функції-члени класу інакше називаються *методами класу*. Методи можуть мати параметри будь-яких стандартних або раніше визначених типів, а також тип свого класу, посилання або вказівник на свій клас. Результат обчислення повертається за допомогою оператора `return` або за допомогою параметрів-вказівників чи посилань. Метод може повертати значення одного із стандартних або визначених типів, зокрема об'єкт, посилання або вказівник на об'єкт свого класу.

2.4.1. Перевантаження методів класу

Методи класу допускають перевантаження. Перевантажені методи мають однакові імена, але повинні відрізнятися кількістю параметрів або їх типами. Найбільш часто перевантажуються конструктори об'єктів класу. Наприклад:

```
class String
  {
   char *str;
   public:

// перевантажені конструктори
     String();
     String(char);
     String(char *);
     String(String&);

// деструктор
     ~String();
  };
```

Методи класу перекривають однойменні зовнішні функції, або методи базових класів у разі їх успадкування. Для доступу до перекритих функцій використовується операція дозволу доступу, наприклад:

```
int func(int x){return x*x;}

class A
{
int x;
public:
A(int x1=0){x=x1;}
int func(int x){return ::func(x)*A::x;}
};

void main()
{
A a(5);
cout<<a.func(4)<<endl;   // 80
}
```

2.4.2. Види методів класу

Можливі види методів класу наведено у табл. 2.3. Символами '+' позначено допустимі комбінації оголошень. Так, деструктор може бути віртуальним, хоча не дозволяє успадкування. Операторні функції `new` та `delete` є статичними за замовчуванням, можуть мати оголошення `volatile` та дозволяють успадкування.

Таблиця 2.3

Види методів класів

Методи класу	конструктор	деструктор	звичайні	static	const	volatile	virtual	успадковані
конструктор	×	–	–	–	–	+	–	–
деструктор	–	×	–	–	–	–	+	–
звичайні	–	–	×	–	–	–	–	+
static	–	–	–	×	–	+	–	+
const	–	–	–	–	×	+	+	+
volatile	–	–	–	+	+	×	+	+
virtual	–	+	–	–	+	+	×	+
operator (крім new, delete, =)	–	–	–	–	+	+	+	+
operator new, delete	–	–	–	+	–	+	–	+
operator=	–	–	–	–	–	+	+	–

Методи класу стисло охарактеризовано у табл. 2.4.

Таблиця 2.4

Властивості методів класів

Види методів класу	Властивості
конструктор	• призначений для ініціалізації об'єктів класу • має назву, яка збігається з назвою класу • існує за замовчуванням (без параметрів) • може мати параметри за замовчуванням • явно не повертає значення • може перевантажуватися • йому неявно передається вказівник this
деструктор	• знищує об'єкти класу • не має параметрів і не повертає значення • існує за замовчуванням • не перевантажується • передається вказівник this
звичайні	• повертає типи, допустимі для даних-членів класу, а також тип свого класу, посилання або вказівник на свій клас • можуть мати параметри за замовчуванням • не можуть бути викликані з константних та volatile-об'єктів • їм передається вказівник this

Продовження табл. 2.4

Види методів класу	**Властивості**
`static`	• призначені для роботи зі статичними даними та статичними методами класу • можуть мати параметри за замовчуванням • можуть викликатися без об'єкта класу (з іменем класу) • можуть бути викликані з константних та `volatile`-об'єктів • їм не передається вказівник `this`
`const`	• не можуть змінювати будь-які поля класу • не можуть повертати неконстантний вказівник або неконстантне посилання на елемент класу • можуть мати параметри за замовчуванням • можуть викликатися з константних та неконстантних об'єктів • константні об'єкти можуть викликати тільки константні методи • їм передається вказівник `this`
`volatile`	• викликаються зі звичайних об'єктів • можуть мати параметри за замовчуванням • їм передається вказівник `this`
`virtual`	• підтримують механізм поліморфізму (пізнє зв'язування) • можуть мати параметри за замовчуванням • їм передається вказівник `this`
`operator` (крім `new`, `delete`, `=`)	• призначені для перевантаження операцій • можуть бути методами або друзями класу • не повинні мати параметрів за замовчуванням • викликаються неявно при використанні знаку операції, застосованого до об'єктів класу, або явно • операторним методам (не друзям) передається вказівник `this`
`operator new, delete`	• можуть бути тільки членами класу (а також операції `()`, `[]`, `->`) • є статичними за замовчуванням • не повинні мати параметрів за замовчуванням • їм не передається вказівник `this`
`operator=`	• не може бути константним • не повинен мати параметрів за замовчуванням • існує за замовчуванням • йому передається вказівник `this`

2.4.3. Виклики функцій протоколу класу

У протоколі класу оголошуються методи класу та дружні функції.

Методи викликаються у межах класу безпосередньо, за їх іменами. Поза класом вони викликаються за допомогою об'єкта класу, посилання або вказівника на клас. Наприклад:

```
class A
{
int x;
public:
A(int x1=0){x=x1;}
int Get(){return x;}
};

void main()
{
A a(5);
// виклик методу за допомогою об'єкта
cout<<a.Get()<<endl;      // 5

A & r = a;
// виклик методу за допомогою посилання
cout<<r.Get()<<endl;      // 5

A *p = &a;
// виклик методу за допомогою вказівника на об'єкт
cout<<p->Get()<<endl;     // 5
}
```

Статичні методи викликаються з використанням імені класу, хоча їх можна викликати також з об'єкта, посилання чи вказівника на об'єкт.

Дружні до класу функції викликаються безпосередньо, без застосування об'єкта, посилання або вказівника на об'єкт класу. Наприклад:

```
class A
{
int x;
public:
A(int x1=0){x=x1;}
friend int Get(A a){return a.x;}
};

void main()
{
A a(5);
// виклик дружньої функції
cout<<Get(a)<<endl;       // 5
}
```

Якщо метод повертає об'єкт, посилання або вказівник на об'єкт, то можливий ланцюжковий виклик методів. Наприклад, нехай необхідно розробити клас-калькулятор, який містить методи для виконання арифметичних операцій додавання, віднімання, множення та ділення дійсних чисел. Такий клас можна використати для обчислення виразу над об'єктами класу.

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

class calc{
float x;
public:
calc(float x=0);
calc plus(calc);
calc minus(calc);
calc mult(calc);
calc div(calc);
void print();
};

calc::calc(float x)
{ this->x=x;}

calc calc::plus(calc b)
{
 return x+b.x;
}

calc calc::minus(calc b)
{
 return x-b.x;
}

calc calc::mult(calc b)
{
 return x*b.x;
}

calc calc::div(calc b)
{
if(b.x!=0) return x/b.x;
else {cout<<"Ділення на нуль"<<endl; exit(-1);}
}

void calc::print()
{
 cout.width(-10);
```

```
 cout.precision(2);
 cout<<x<<endl;
}

void main()
{
 calc a(1), b(2), c(3), d;
 //обчислення виразу (a-b)*c/(a+b+c)
 d=a.minus(b).mult(c).div(a.plus(b).plus(c));
 d.print();

 //обчислення виразу (a-b*c)/(a+b+c)
 d=a.minus(b.mult(c)).div(a.plus(b).plus(c));
 d.print();
}
```

Ланцюжковий виклик методів, які повертають об'єкт або посилання на об'єкт, здійснюється за допомогою багатократного застосування операції "крапка" до результатів методів. Відповідно, якщо методи повертають вказівники на об'єкти класу, то замість крапки необхідно застосувати операцію "стрілка".

2.4.4. Вказівник this

Кожний нестатичний метод класу має доступ до об'єкта, з якого він викликаний, через вказівник `this`. За допомогою цього вказівника можна отримати доступ до всіх елементів класу (даних та методів):

```
class Base { int x;
       public:
       Base(int x1){x=x1;}
       void print()
 {// у всіх випадках забезпечується доступ до поля x
  cout<<x<<' ';
  cout<<(*this).x<<' ';
  cout<<this->x<<' ';
  cout<<Base::x<<endl;

 }
};

void main()
{
Base a(5);
a.print();          // 5   5   5   5
}
```

Вказівник `this` можна використати для доступу до даних, перекритих параметрами методу класу, наприклад:

```
class Base { int x,y;
             public:
              Base(int x, int y)
              {this->x=x;
              this->y=y;
              }
            };
```

Перекриті дані класу можна проініціалізувати також у списку ініціалізації конструктора, наприклад:

```
Base::Base(int x, int y) : x(x), y(y){}
```

Розіменований вказівник this використовується для повернення методом класу посилання на об'єкт свого класу, наприклад:

```
class Base { int x,y;
             public:
              Base& Copy(const Base& a)
              {x=a.x;
              y=a.y;
              return *this;
              }
           };
```

2.4.5. Константні та статичні методи класу

Статична функція оголошується за допомогою слова static, яке записується перед заголовком функції тільки у протоколі класу. Статична функція може бути викликана незалежно від існування об'єкта класу. Тоді вона викликається з іменем класу, наприклад Base::f1() з врахуванням рівнів захисту елементів класу.

Константна функція оголошується за допомогою слова const, яке записується після її заголовка. Якщо константна функція визначається за межами класу, то слово const повинно зберігатися в кінці заголовка функції.

Приклад програми використання статичних та константних методів.

```
class Base {
       static int x;
       const int y;
public:
       Base(int z) : y(z){}
       static int GetX();
       int GetY() const;
    };

int Base::x=1;
```

```
int Base::GetX()    // static не потрібно
{
return x;
}

int Base::GetY() const   // const потрібно
{
return y;
}

void main()
{
 Base a(2);
 cout<<Base::GetX()<<endl; //або cout<<a.GetX()()<<endl;

 cout<<a.GetY()<<endl;
}
```

Константний метод не може змінювати дані класу, наприклад, компіляція наступної програми призведе до помилки:

```
class A {
       int x;
 public:
 A(int x1=0) : x(x1){}
 int inc() const
  {
   return ++x; // Помилка
  }
 };

void main()
{
 A a(5);
 cout<<a.inc()<<endl;
}
```

Статичному методу не передається вказівник this, тому він не може звертатися до нестатичних елементів класу безпосередньо, за їх іменами. Однак, статичний метод може звертатися до нестатичних елементів класу через посередника – об'єкт, посилання або вказівник на об'єкт класу, наприклад:

```
class A{
 int x;
public:
 A(int x=0){this->x=x;}
 void print(){cout<<x<<endl;}
 static void output()
 {// cout<<x<<endl;
      // Помилка: звернення до нестатичного елемента даних
```

```
  // print();
      // Помилка: звернення до нестатичної функції
  A a(7);
// Звернення до нестатичних елементів через об'єкт
  cout<<a.x<<endl;
  a.print();
 }
};

void main()
{
A::output();
}
```

Статичний метод не може бути константним та навпаки.

Права доступу до елементів класу з врахуванням виду методів класу деталізовано у табл. 2.5. У таблиці використано позначення, введені у табл. 2.1 та табл. 2.2. Методи класу, друзі та зовнішні функції і класи реалізують доступ до звичайних, статичних та константних елементів класу так, як це зображено у відповідній клітинці таблиці. Як і раніше, справедливе відношення: FN < CN < ON. У таблиці показано доступ тільки до `public`-частини класу. Для визначення доступу до усього класу необхідно керуватися табл. 2.2.

Таблиця 2.5

Доступ до статичних та константних елементів класу

		Методи класу			**Друзі класу**	**Зовнішні функції та класи**
public:	**Види елементів класу**	**звичайні**	**static**	**const**		
Дані	**звичайні**	FN	ON	FN	ON	ON
	static	FN	FN	FN	CN	CN
	const	FN	ON	FN	ON	ON
Методи	**звичайні**	FN	ON	ON	ON	ON
	static	FN	FN	ON	CN	CN
	const	FN	ON	FN	ON	ON

2.5. Вказівники на елементи класу

2.5.1. Вказівники на дані класу

У програмі дозволяється оголошувати вказівники на дані класу. Вказівник на нестатичне поле оголошується та ініціалізується так:

```
тип  ім'я_класу::*ім'я_вказівника=&ім'я_класу::ім'я_поля;
```

Наприклад:

```
class Base{
        // ...
        public:
        int x;
        const int y;
        static int z;
        Base(int x1=0, int y1=0):y(y1)
              {x = x1;}
        };

int Base::z=3;

int Base::*p1=&Base::x;
```

Вказівник на нестатичне поле даних класу визначає зміщення відносно адреси об'єкта класу (рис. 2.2). Тому вказівник на елемент класу має використовуватися разом з об'єктом цього класу (вказівником або посиланням на об'єкт).

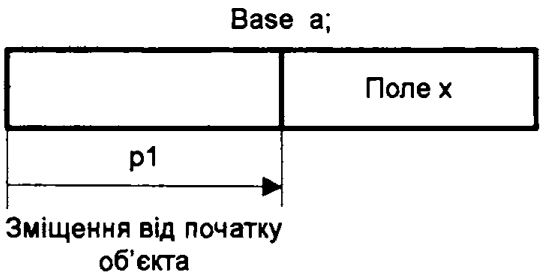

Рис. 2.2. Вказівник на нестатичний елемент даних

Для використання вказівника використовується операція .* (крапка із зірочкою), якщо звернення до поля даних здійснюється за допомогою об'єкта чи посилання на об'єкт, або операції ->* (стрілка із зірочкою), якщо звернення до поля даних здійснюється за допомогою вказівника на об'єкт. Наприклад:

```
Base a;
a.*p1=5;    // a.x=5
```

```
Base *q=new Base;
q->*p1=5;
```

Для оголошення вказівника на константний елемент необхідно використати слово const перед ідентифікатором базового типу:

```
const тип  ім'я_класу :: * ім'я_вказівника = &ім'я_класу ::
                           ім'я_поля;
```

Наприклад:

```
const int Base::*p2=&Base::y;
```

Як відомо, константні дані не можна змінити у програмі. Але зміна константного поля можлива за допомогою вказівника на нього з подальшим явним перетворенням до типу змінної, наприклад:

```
void main()
{
Base a(5,7);
// a.y=9;            // не можна змінити константу
a.*(int Base::*)p2=9;    // можна змінити константу за
                     // допомогою вказівника на поле класу
 cout<<a.y<<endl;  // 9
}
```

Практична необхідність зміни константного елемента даних є сумнівною. Варіант використання вказівника на константний елемент для перевантаження операції присвоєння розглянуто у п. 4.6.

Вказівник на статичне поле класу оголошується як звичайний вказівник на змінну програми:

```
тип  *ім'я_вказівника = &ім'я_класу :: ім'я_поля;
```

Наприклад:

```
int *p3=&Base::z;
```

Не дозволяється перетворення вказівника на нестатичний елемент класу до звичайного вказівника.

Можливість оголошення вказівників на поля класу в різних місцях програми регулюється правилами доступу до частин класу private, protected, public:

1) дані-члени класу можуть мати тип вказівника на елемент класу;
2) методи класу та їх параметри можуть мати тип вказівника на елемент класу;
3) зовнішня функція може оголосити вказівник на елементи public-частини класу;

4) на глобальному рівні можна оголосити вказівник на елементи public-частини класу;
5) функції-члени можуть оголосити вказівники на всі елементи класу, розміщені в private, protected або public-частинах класу;
6) друзі класу можуть оголосити вказівники на всі елементи класу, розміщені в private, protected або public-частинах класу.

2.5.2. Вказівники на методи класу

Крім вказівників на дані, можна оголошувати вказівники на методи класу. Вказівник на метод класу можна використати для безпосереднього виклику методу або як аргумент чи тип результату іншої функції.

Вказівник на нестатичний метод класу оголошується та ініціалізується так:

```
тип_функції  (ім'я_класу::*ім'я_вказівника)(типи
       параметрів) = &ім'я_класу::ім'я_методу;
```

Використання такого вказівника здійснюється за допомогою операції .* (крапка із зірочкою), якщо виклик методу здійснюється за допомогою об'єкта або посилання на об'єкт, або операції ->* (стрілка із зірочкою), якщо виклик методу здійснюється за допомогою вказівника на об'єкт. Наприклад:

```
class Base{
       int x;
       public:
       Base(int y){x=y;}
          int GetX(){return x;}
       };

int (Base::*p)()=&Base::GetX;

void main()
{
  Base a(5);
  cout<<(a.*p)()<<endl;  // або a.GetX();
  Base *q=&a;
  cout<<(q->*p)()<<endl;
}
```

Не дозволяється визначення вказівника на конструктор та на деструктор класу.

Вказівник на константний метод класу оголошується подібно до попереднього із застосуванням слова const:

```
тип_функції  (ім'я_класу::*ім'я_вказівника)(типи
       параметрів) const = &ім'я_класу::ім'я_методу;
```

Вказівник на статичний метод класу оголошується як звичайний вказівник на функцію:

```
тип_функції  (*ім'я_вказівника)(типи
       параметрів) = &ім'я_класу::ім'я_методу;
```

Можливість оголошення вказівника на метод класу у вибраному місці програми регулюється правилами доступу до частин класу `private`, `protected`, `public`.

Вказівники на дані та методи класу можна використовувати як типи аргументів функцій, що дає змогу передавати елементи класу через список фактичних-формальних параметрів.

Приклад передавання методу класу у зовнішню функцію через список аргументів:

```
#include <iostream>
class A
{
 int x;
public:
 A(int a=0){x=a;}
 int Get();
};

int A::Get()
{
return x;
}

void func(A a, int (A::*pfunc)())
{
cout<<(a.*pfunc)()<<endl;    // a.Get()
}

void main()
{ A a(5);
func(a,&A::Get);
}
```

2.6. Конструктори

2.6.1. Призначення та особливості конструкторів

Конструктор – це метод класу з тим самим іменем, що й клас. Особливості конструктора:

- призначений для виділення динамічної пам'яті та ініціалізації даних класу;
- якщо конструктор не оголошений, то компілятор згенерує конструктор за замовчуванням (без параметрів);
- для конструктора явно не вказується результуючий тип, конструктор не повертає значення;
- визначається всередині класу або поза протоколом класу;
- може перевантажуватися;
- може мати параметри за замовчуванням;
- не успадковується;
- не може мати оголошень `const`, `virtual`, `static`;
- може мати список ініціалізації, який задається у заголовку після символу `':'`. Так ініціалізуються константні поля (`const`), посилання на будь-який тип, поля з типом іншого класу, конструктори базових класів у разі успадкування;
- може бути розміщений в `private`, `protected` та `public`-частинах класу. При оголошенні об'єктів у різних місцях програми необхідно враховувати, чи буде доступний відповідний конструктор. Найчастіше конструктори розміщують у `public`-частині класу, що забезпечує можливість створення об'єктів за межами класу;
- автоматично викликається при створенні об'єкта у сегменті даних, стеку, динамічній пам'яті та при перетворенні типів, коли значення певного типу перетворюється в об'єкт класу у виразі, передачі аргументів у функцію, поверненні результату з функції;
- не викликається при оголошенні вказівників або посилань на клас;
- може бути конструктором копіювання, якщо має один параметр з типом `T&` або `const T&`;
- не може бути викликаний явно з використанням об'єкта класу (або вказівника чи посилання на клас);
- при успадкуванні раніше викликаються конструктори базових класів, а потім – похідних класів.

2.6.2. Види конструкторів

Можна виділити такі види конструкторів за їх призначенням:

- ініціалізації (звичайні);
- копіювання:
- перетворення типів.

Конструктори ініціалізації

Конструктори ініціалізації викликаються:

- при створенні нового об'єкта у сегменті даних, стеку, динамічній пам'яті та ініціалізації його полів (всіх або деяких з них);
- для створення локального об'єкта у виразі:

```
тип_класу(список параметрів конструктора).
```

Приклади створення об'єктів, коли викликається конструктор ініціалізації:

```
1) A a(1,2); // або        A a=A(1,2);
2) A *p=new A(1,2);
3) A& q=*new A(1,2);
```

Необхідно розрізняти ініціалізацію та присвоєння об'єктів. Ініціалізація здійснюється при створенні нового об'єкта, а присвоєння – для вже існуючого об'єкта. Наприклад:

```
A a = A(1, 2);      // ініціалізація
a = A(1, 2);        // присвоєння
A b = a;            // ініціалізація
b = a;              // присвоєння
```

Конструктор копіювання

Особливості конструктора копіювання:

- має один параметр з посиланням на тип класу: `A&` або `const A&`. Може мати більше одного параметра, які задаються за замовчуванням;
- не допускає перевантаження – у класі може бути тільки один конструктор копіювання;
- якщо не визначений у класі, то конструктор копіювання генерується автоматично і виконує порозрядне копіювання об'єктів;
- виконує копіювання усіх видів полів даних, зокрема константних та з типом посилання; константні дані та дані з типом посилання ініціалізуються у списку ініціалізації конструктора (після двокрапки);
- якщо поля класу мають тип вказівника або посилання на будь-який тип, то конструктор копіювання повинен бути визначений користувачем;

- якщо визначення конструктора копіювання розміщено у закритій або захищеній частинах класу, то неможливо створити копію об'єктів поза межами класу (крім друзів класу).

Приклад оголошення та визначення конструктора копіювання:

```
class A{ const int x;
         int& y;
         int z;
       public:
// визначення конструктора ініціалізації
        A(int x1=0, int y1=0, int z1=0):x(x1),y(y1)
         {z=z1;}
// оголошення конструктора копіювання
        A(const A&);
 };
// визначення конструктора копіювання
A::A(const A& a): x(a.x), y(a.y)
 {z=a.z;}
```

Конструктор копіювання викликається:

- при створенні нового об'єкта та його ініціалізації вже існуючим об'єктом цього ж класу; новий об'єкт може бути створений у сегменті даних, стеку, динамічній пам'яті, може бути глобальним або локальним;
- при передачі об'єктів через список параметрів функції "за значенням" (не через вказівники або посилання);
- при поверненні функціями локальних об'єктів класу "за значенням" (не через вказівники або посилання);

Приклади створення об'єктів, коли викликається конструктор копіювання:

```
    A a;
1)  A b(a); // або A b=a;
2)  A *p=new A(a);
3)  A& q=*new A(a);
```

Якщо в протоколі класу є поля з типом вказівника або посилання на будь-який тип, то для такого класу необхідно визначити власний конструктор, який здійснює глибоке копіювання об'єктів. При глибокому копіюванні створюються реальні копії усіх внутрішніх елементів (об'єктів) даних, а не тільки копії їх адрес. Необхідність цього демонструється таким прикладом:

```
class A
{
 int* p;
 public:
// звичайний конструктор
 A(int x){p=new int(x);}
// конструктор копіювання
```

```
 A(const A& a){p=new int(*a.p);}
// деструктор
~A(){delete p;}
// методи класу
 int Get(void){return *p;}
 void Set(int x){*p=x;}
};

void main()
{
 int x=5;
 A a(x);
 cout<<"*a.p=="<<a.Get()<<endl;
 A b(a);
 cout<<"*b.p=="<<b.Get()<<endl;
 a.Set(7);
 cout<<"*b.p=="<<b.Get()<<endl;
}
```

У цьому прикладі, завдяки визначенню конструктора копіювання, який закріплює за вказівником `p` нову область динамічної пам'яті, маємо, що `a.p!=b.p`, тобто вказівники `p` обох об'єктів адресують різні області оперативної пам'яті. У результаті зміна елемента, який адресується полем `a.p`, не вплине на значення елемента, який адресується полем `b.p`. Результатом роботи цієї програми буде:

```
*a.p==5
*b.p==5
*b.p==5
```

Відповідну схему адресування динамічної пам'яті (ДП) зображено на рис. 2.3.

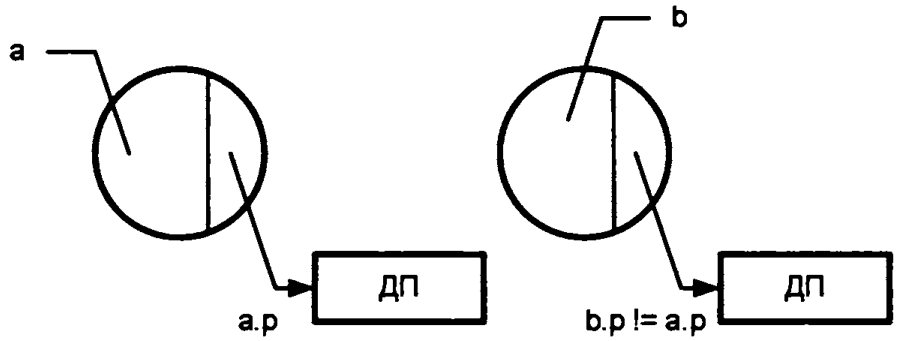

Рис. 2.3. Схема дії визначеного у класі конструктора копіювання

Якщо у наведеній програмі закоментувати конструктор копіювання, то діятиме конструктор копіювання за замовчуванням, який виконає неглибоке (порозрядне) копіювання об'єктів, наприклад, таке:

```
A::A(const A& a){p = a.p;}
```

У результаті вказівники a.p та b.p набудуть однакових значень і адресуватимуть одну й ту саму область пам'яті (рис. 2.4).

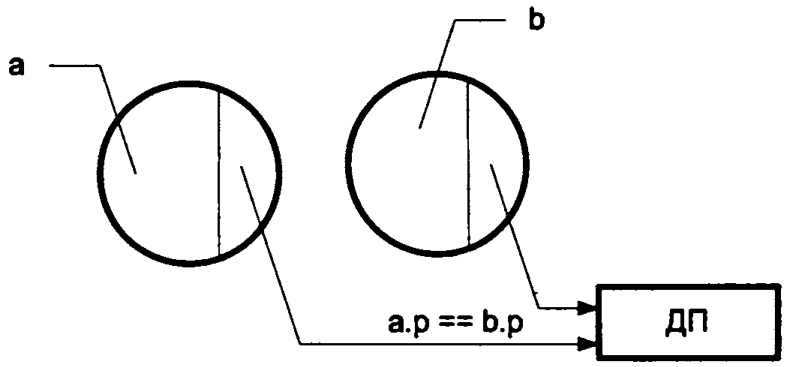

Рис. 2.4. Схема дії конструктора копіювання за замовчуванням

Ця область пам'яті буде спільною для всіх об'єктів класу, створених на основі одного й того самого базового об'єкта (подібно до статичних полів). У результаті роботи програми отримаємо, що виклик функції a.Set(7) для об'єкта *a* змінив значення елемента *b.p з 5 на 7. Результатом роботи програми, яка використовує конструктор копіювання за замовчуванням, буде:

```
*a.x==5
*b.x==5
*b.x==7
```

Якщо поле класу є посиланням на будь-який тип, то оголошення класу та його конструктор копіювання матимуть вигляд:

```
class A
{
 int& x;
 public:
// звичайний конструктор
 A(int y):x(*new int(y)){}
// конструктор копіювання
 A(const A& a):x(*new int(a.x)){}
// деструктор
 ~A(){delete &x;}
// методи класу
 int GetX(void){return x;}
 void SetX(int y){x=y;}
};
```

Відсутність визначеного у класі конструктора копіювання може призвести до помилок, які важко виявити. Наприклад:

```
class A{
 int *p;
```

```
public:
A(int x=0)
{ cout<<"A::A(int)"<<endl;
 p = new int(x); }

//==============================
A(A& a)
{ cout<<"A::A(A&)"<<endl;
  p = new int(*a.p);}
//==============================*/
~A()
{ cout<<"A::~A()"<<endl;
         delete p;}

void set(int x=0){*p=x;}

friend void print(A);
};

void print(A a)
{
 cout << *a.p << endl;
}

void main()
{
A a(5);
for(int i=0;i<3; i++)  print(a);
}
```

Якщо закоментувати конструктор копіювання, то багатократне звернення до функції `print()` призведе до неправильних результатів роботи програми. Функція `print()` отримує об'єкт класу `A` "за значенням". Для цього викликається конструктор копіювання "за замовчуванням", який виконає порозрядне копіювання полів об'єкта. У нашому прикладі – це копія вказівника `p`, за яким конструктор закріпив область динамічної пам'яті при створенні об'єкта у функції `main()`. Перед завершенням роботи функції `print()` викликається деструктор, який звільнить динамічну пам'ять, закріплену за вказівником `p`. Хоча в результаті цього значення вказівника `p` не зміниться, але область динамічної пам'яті більше за ним не буде закріплена. Наступне звернення до функції `print()` призведе до повторного звільнення вже звільненої динамічної пам'яті. У результаті отримаємо збій у роботі програми.

Помилку можна виправити, якщо ввести у клас конструктор копіювання, який закріплюватиме за вказівником `p` нову область динамічної пам'яті при кожному зверненні до функції `print()`, а деструктор звільнятиме її перед завершенням роботи цієї функції.

Інший спосіб усунення помилки за відсутності перевантаженого конструктора копіювання полягає у використанні типу посилання A& при оголошенні параметра функції print(). Для параметрів-посилань та вказівників з типом класу копія об'єкта не створюється і конструктор копіювання (і відповідний йому деструктор) не викликається.

Якщо полем класу є об'єкт іншого класу, то перед викликом конструктора копіювання зовнішнього об'єкта викликається конструктор для копіювання його внутрішнього об'єкта, наприклад:

```
class A {
 int x;
 public:
// конструктор ініціалізації
A(int x1=0){x=x1;
 cout<<"A(int)"<<endl;
 }
// конструктор копіювання
 A(const A& a){
 x=a.x;
 cout<<"A(const A&)"<<endl;}
 };

// контейнерний клас
class B{
 A z;  // внутрішній об'єкт
public:
// конструктор ініціалізації
B(const A& a1):z(a1){
 cout<<"B(const A&)"<<endl;
 }
// конструктор копіювання
//==============================
 B(const B& b):z(b.z){
 cout<<"B(const B&)"<<endl;}
//==============================*/
};

void main()
{
A a(5);
B b(a);
B c(b); // викликаються конструктори копіювання
}
```

Результатом виконання цієї програми є такі виклики конструкторів:

A(int) - створення об'єкта a(5)

A(const A&) - ініціалізація внутрішнього об'єкта z класу B

`B(const A&)` - створення об'єкта `b(a)`

`A(const A&)` - копіювання внутрішнього об'єкта `z` класу `B`

`B(const B&)` - копіювання об'єкта `c(b)`.

За відсутності визначеного у класі `B` конструктора копіювання `B::(const B&)` спочатку буде викликаний конструктор копіювання `A::(const A&)` класу `A` для копіювання внутрішнього об'єкта `B::z`, а потім – конструктор копіювання об'єкта класу `B` за замовчуванням.

Конструктори перетворення типів

Конструктор перетворення типів призначений для перетворення значення заданого типу до типу класу. Має один параметр з типом, який вимагає перетворення, або декілька параметрів, значення яких задаються за замовчуванням. Наприклад:

```
class A{
       int x;
 public:
 // конструктори перетворення типів
    A(int x){this->x=x;}
    A(double x) {this->x=int(x);}
};
```

Конструктор перетворення типів викликається у таких випадках:

- для ініціалізації об'єкта класу значенням, тип якого є відмінним від типу цього класу;
- у виразах виду: класовий тип ⊕ інший тип (за відсутності перевантаженої операції з точною відповідністю списку параметрів);
- для явного перетворення будь-якого типу до типу класу;
- в операції присвоєння: класовий тип = інший тип (за відсутності перевантаженої операції з точною відповідністю параметрів);
- для передавання параметрів у функцію, коли формальний параметр має класовий тип, а фактичний – інший тип;
- для повернення значення функцією, коли функція має класовий тип, а вираз оператора `return` має інший тип.

За допомогою конструктора допускається неявне та явне перетворення типів.

Приклади неявного перетворення типу:

```
1) A a; a=5; // за відсутності перевантаження operator=
2) A a(5);
3) A a=5;
4) A a, b(1);
   a=b+5; // необхідна перевантажена операція +
```

Приклади явного перетворення типів:

- у стилі мови С

```
A a;
a=(A)1;
```

- функціональний стиль мови С++

```
A a;
a=A(1);
```

Неявне перетворення типів можна заборонити, оголосивши явні конструктори перетворення типів зі словом `explicit`, наприклад:

```
class A{
int x;
public:
explicit A(int x){this->x=x;}
};
```

Тоді ініціалізації об'єктів, що вимагають неявного перетворення типів, наприклад, `A a(5)` та `A a=5`, будуть неможливими.

2.7. Деструктор класу

2.7.1. Оголошення та автоматичний виклик деструктора

Деструктор призначений для звільнення пам'яті, відведеної під поля об'єкта. Назва деструктора формується з ідентифікатора класу, перед яким записується символ ~ (тильда). Деструктор не має параметрів і не повертає значення. Деструктор не перевантажується і не успадковується. Існує за замовчуванням, але його потрібно визначити, якщо для полів об'єкта була виділена динамічна пам'ять.

Деструктор викликається автоматично, якщо об'єкт виходить із області досяжності програми. Пам'ять, відведена у стеку для локального об'єкта з класом пам'яті "автоматичний" (`auto`, діє за замовчуванням), буде вивільнено під час завершення роботи функції. Пам'ять статичного (`static`) локального об'єкта буде звільнено перед завершенням роботи функції `main()`. Якщо локальний об'єкт розміщено в області динамічної пам'яті, то при завершенні роботи функції деструктор не викликається. Слід зазначити, що всю не звільнену динамічну пам'ять буде автоматично звільнено після завершення роботи програми.

Варіанти виклику конструкторів та деструктора для локальних об'єктів демонструє така програма:

```
class A {
 int * p;
 public:
 A(int x=0){
       p=new int(x);
       cout<<"A(int)="<<*p<<endl;
       }
 ~A(void){
       cout<<"~A()="<<*p<<endl;
       if(p) delete p;
       }
 };

void func(void)
{
A a(5);
A* p = new A(7);
// ...
}

void main()
{
func();
}
```

У результаті виконання програми отримаємо таке виведення:

```
A(int)=5
A(int)=7
~A(int)=5
```

Конструктор викликається при створенні автоматичного локального об'єкта у стеку, проініціалізованого значенням 5 та об'єкта у динамічній пам'яті, проініціалізованого значенням 7. Під час завершення роботи функції func() викликається деструктор тільки для об'єкта у стеку, а об'єкт у динамічній пам'яті залишається неушкодженим.

Знищення об'єктів у динамічній пам'яті здійснюється за допомогою операції delete, наприклад:

```
void func(void)
{
A a(5);
A* p = new A(7);
//...
delete p;
}
```

Операція `delete` призведе до виклику деструктора. В результаті отримаємо таке виведення:

```
A(int)=5
A(int)=7
~A(int)=7
~A(int)=5
```

Якщо у функції `func()` оголосити статичний об'єкт, то він буде розміщений не у стеку, а в сегменті даних, наприклад:

```
void func(void)
{
static A a(5);
A* p = new A(7);
//...
}
```

При завершенні роботи функції `func()` деструктор для статичного об'єкта не викликається. Такий об'єкт буде знищений при завершенні роботи програми в цілому – у кінці роботи функції `main()`.

Наведемо приклад програми спільної роботи конструкторів та деструктора:

```
class A {
 int * p;
 public:
 A(int x=0){
       cout<<"A(int)"<<endl;
       p = new int(x);
       }
  A(A& a){
       cout<<"A(A&)"<<endl;
       p=new int(*a.p); // *p=*a.p;
       }
 ~A(void){
       cout<<"~A()"<<endl;
       if(p) delete p;
       }
 int Get(void){return *p;}
 void Set(int x){*p=x;}
 };

A func(
 A a          // 2) конструктор A(A&) для a
      )
{
int x=a.Get();
if(x<0) a.Set(-x);
return a;    // 3) конструктор A(A&)  для a
```

```
// 4) деструктор  ~A()  для a
}

void main()
{
A *q;           // конструктор не викликається
q=new A(-5);  // 1) конструктор A(int) для *q
A a=func(*q);
A& r=a;       // конструктор не викликається
cout<<r.Get()<<endl;
delete q;     // 5) деструктор  ~A() для *q
// 6) деструктор ~A() для a
}
```

Номерами помічено порядок виклику конструкторів та деструктора. Виведення цієї програми матиме вигляд:

```
A(int)
A(A&)
A(A&)
~A()
5
~A()
~A()
```

2.7.2. Явний виклик деструктора

Методи класу та зовнішні функції можуть викликати деструктор явно через об'єкт класу, наприклад:

```
class A {
 int *p
 public:
 A(int x){p=new int(x);}
 ~A(){delete p;}
 int Get(void){return *p;}
 };

void main()
{
A a(5), *b=new A(7);
cout<<a.Get()<<endl;
cout<<b->Get()<<endl;
b->~A();      // можна так, але краще delete b;
a.~A();       // можна так, але деструктор
              // викликається автоматично
}
```

2.8. Структури та об'єднання

2.8.1. Структури

Структури у C++ є різновидностями класів з відкритим (public) доступом до елементів за замовчуванням. Структура може містити явно оголошені секції private, protected та public. Окрім даних, структура може мати конструктори, деструктор та методи роботи з даними. Наприклад:

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

struct Student
{
char *name;
Student(char *s="")        // конструктор
 {name = new char[20];
  strcpy(name, s);
 }
~Student()                 // деструктор
 {delete []name;
 }
char * GetName()           // метод читання елемента даних
 {return name;
 }
};

void main()
{
Student x("Козак");
cout<<x.name<<endl;
cout<<x.GetName()<<endl;
}
```

У цьому прикладі ідентифікатор Student визначає тег структури. Тег є конкретизацією типу структури. Для оголошення змінної з типом структури достатньо вказати тільки її тег та ідентифікатор з можливими аргументами конструктора структури.

Якщо змінна оголошується разом зі структурою, то тег структури можна не вказувати, наприклад:

```
struct {
// ...
} s1, s2, *ptr;
```

У структурі не можна оголосити елемент даних з типом цієї самої структури, але можна оголосити вказівник або посилання на таку структуру.

Структури можуть бути вкладеними. У вкладеній структурі можна оголосити вказівник або посилання на охоплюючу структуру. У зовнішній структурі можна оголосити вказівник, посилання або об'єкт внутрішньої структури, наприклад:

```
struct tag1 {
 struct tag2{
       tag1& p1;
       tag2* p2;
// ...
 };
 tag2 p3;
};
```

За необхідності виконують випереджувальне оголошення структури, якщо вона розміщена нижче місця використання її тегу.

Кожен елемент даних структури розміщується у пам'яті послідовно, елемент за елементом. У зв'язку із можливими неявними перетвореннями типів та вирівнюванням структури на межі байта або слова довжина структури може відрізнятися від суми довжин її полів даних. Визначення розміру структури необхідно здійснювати за допомогою операції `sizeof`, наприклад:

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

struct rec {
 long x;
 short y;
 static int z;
// Конструктор
 rec (long x1=0, short y1=0){x=x1; y=y1;}
// Метод виведення
void output(){cout<<hex<<x<<' '<<y<<endl;}
static void print(){cout<<z<<endl;}
 } s1(0xAAAAAAAA,0xBBBB);

int rec::z=0xFFFFFFFF;

void main()
{
rec s2(0xCCCCCCCC,0xDDDD);

cout.setf(ios::showbase|ios::uppercase);

s1.output();        // 0XAAAAAAAA  0xBBBB
s2.output();        // 0XCCCCCCCC  0XDDDD
rec::print();       // 0XFFFFFFFF
```

```
cout<<sizeof(s1.x)<<' '  // 0X4
 <<sizeof(s1.y)<<' '     // 0X2
 <<sizeof(s1)<<' '       // 0X8
 <<sizeof(rec::z)<<endl; // 0X4
}
```

Розмір структури дорівнює 8 байтів, оскільки двобайтовий тип `short` автоматично перетворюється до чотирибайтового типу `int`. Статичні дані структури розміщуються окремо від інших елементів, що видно із виведення програми.

2.8.2. Об'єднання

Об'єднання відрізняється від структури тим, що для усіх полів даних виділяється одна і та сама область оперативної пам'яті, інакше, поля даних накладаються одне на інше. Розмір об'єднання визначається розміром його найдовшого поля.

Об'єднання є різновидом класу з `public`-доступом до елементів за замовчуванням. Об'єднання може містити явно оголошені секції `private`, `protected` та `public`. Об'єднання може мати конструктори, деструктор та методи для роботи з даними (окрім віртуальних методів). Об'єднання не є повноцінним класом, оскільки не дозволяється його успадкування.

Наприклад:

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

union mix {
 long x;
 short y;
 static int z;
// Конструктор
 mix (long x1=0, short y1=0){x=x1; y=y1;}
// Деструктор
 ~mix(){}
// Метод виведення
 void output(){cout<<hex<<x<<' '<<y<<endl;}
 static void print(){cout<<z<<endl;}
 } u1(0xAAAAAAAA, 0xBBBB);

int mix::z=0xFFFFFFFF;

void main()
{
mix u2(0xCCCCCCCC,0xDDDD);

cout.setf(ios::showbase|ios::uppercase);
```

```
u1.output();        // 0XAAAABBBB  0XBBBB
u2.output();        // 0XCCCCDDDD  0XDDDD
mix::print();       // 0XFFFFFFFF

cout<<sizeof(u1.x)<<' '  // 0X4
 <<sizeof(u1.y)<<' '     // 0X2
 <<sizeof(u1)<<' '       // 0X4
 <<sizeof(mix::z)<<endl; // 0X4
}
```

Статичним елементам об'єднання виділяється окрема від інших даних область оперативної пам'яті, оскільки вони визначаються поза класом на глобальному рівні.

В об'єднаннях не рекомендується визначати конструктори та деструктор. Поля іменованого об'єднання можна ініціалізувати у програмі за допомогою відповідного об'єкта або безпосередньо – для анонімних об'єднань.

При об'єднанні вказівників закріплені за ними області динамічної пам'яті не можуть бути вивільнені деструктором у зв'язку із втратою значень вказівників при їх накладанні.

Крім скалярних значень, можна об'єднувати структуровані дані, наприклад, об'єкти різних класів. При об'єднанні об'єктів у відповідних класах вимагається наявність оригінального конструктора за замовчуванням. Наприклад:

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

class A {
 long x;
public:
 //    A(long x1=0){x=x1;}     // помилка
 void set_x(long x1){x=x1;}
 long get_x(){return x;}
 };

class B {
 short y;
public:
 // B(short y1=0){y=y1;}       // помилка
 void set_y(short y1){y=y1;}
 short get_y(){return y;}
};

union mix {
 A a; B b;
// Конструктор об'єднання
 mix (long n=0, short m=0){a.set_x(n); b.set_y(m);}
```

```
 } u1(0xAAAAAAAA, 0xBBBB), u2;

void main()
{

u2.a.set_x(0xCCCCCCCC);
u2.b.set_y(0xDDDD);

cout.setf(ios::showbase|ios::uppercase);

cout<<hex<<u1.a.get_x()<<' '    // 0XAAAABBBB
    <<u1.b.get_y()              // 0XBBBB
    <<endl;
cout<<u2.a.get_x()<<' '         // 0XCCCCDDDD
    <<u2.b.get_y()              // 0XDDDD
    <<endl;
}
```

Визначення конструкторів у класах A та B перекривають дію оригінальних конструкторів за замовчуванням, що призводить до помилки.

У програмах не рекомендується об'єднувати об'єкти, оскільки при цьому втрачається інформація про відношення між класами.

2.9. Друзі класу

Друзі класу оголошуються за допомогою слова `friend`. Друзі не є членами класу, але наділені правами доступу до елементів усіх його частин через об'єкт, посилання або вказівник на об'єкт класу.

Друзями класу можуть бути:

- зовнішні функції;
- цілий зовнішній клас;
- методи інших класів.

Друзі підпорядковуються таким правилам:

- рівні захисту `private`, `protected` та `public` ніяк не діють на оголошення друзів класу;
- дружність не є взаємною властивістю, якщо клас A оголошує клас B своїм другом, то це не означає, що A є другом B;
- друзі не успадковуються, якщо клас A оголошує B своїм другом, то похідні від B класи не будуть дружніми до A;
- відношення дружби не є перехідним, якщо клас A оголошує B своїм другом, то похідні від A класи не будуть визнавати дружність класу B.

Дружні функції оголошуються у класі, а визначаються, як правило, поза класом, наприклад:

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

class A{
 public:
 friend void output();
 };

void output()
  {cout<<"Друг класу A"<<endl;}

void main()
{
 output();
}
```

При визначенні поза класом слово `friend` у заголовку функції повторно не записується. Назва класу перед іменем дружньої функції не вказується, оскільки така функція не є членом класу. Дружню функцію викликають безпосередньо, без використання імені об'єкта (або вказівника чи посилання на клас).

Сучасні компілятори дозволяють визначати дружні функції у межах класу, однак такі функції стають частково залежними від класу. Так, виклик дружньої функції без параметрів стає неможливим. Наприклад:

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

class A{
 public:
 friend void output()
  {cout<<"Дружня функція класу A"<<endl;}
};


void main()
{
 output(); // помилка
}
```

Щоб виправити цю помилку, необхідно визначити дружню функцію поза класом або з параметром, що має тип класу, наприклад:

```
class A{
 public:
```

```
 friend void output(A a)
  {cout<<"Дружня функція класу A"<<endl;}
};

void main()
{
 A a;
 output(a);
}
```

Клас не інкапсулює у собі визначення дружньої функції. Визначена у класі дружня функція може бути викликана з інших класів у межах дії встановленого простору імен, наприклад:

```
class A{
 int x;
public:
 A(int x1=0){x=x1;}
 friend void output(A a)
 {cout<<a.x<<endl;}
};

class B{
 A a;
public:
 B(A a1):a(a1){}
 void print(){output(a);}
};

void main()
{
A a(7);
output(a);
B b(a);
b.print();
}
```

У цьому прикладі визначена у класі A дружня функція output() викликається у методі print() класу B.

Визначена у класі A дружня функція може бути оголошена дружньою і до класу B без її повторної реалізації. Наприклад:

```
#include <iostream>

using namespace std;

class A{
 int x;
public:
```

```
 A(int x1=0){x=x1;}
 friend void output(A a)
 {cout<<a.x<<endl;}
};

class B{
 A a;
public:
 B(A a1):a(a1){}
 operator A() {return a;}
 friend void output(A a);
};

void main()
{
A a(7);
output(a); // 7

B b(9);
output(b); // 9
}
```

У цьому прикладі визначена у класі A функція output(A) оголошена дружньою до двох класів – A та B. Функція виводить на екран поле A::x класу A. Клас B є контейнером для об'єкта класу A. Для виведення об'єкта B::a класу B виконується перетворення об'єкта класу B в об'єкт класу A за допомогою операції перетворення типу operator A().

Заголовок дружньої функції (назва, список параметрів, тип результату) може збігатися із заголовком методу класу, наприклад:

```
class A{
       int x;
 public:
       A(int x1=0){x=x1;}
       int func(A& y){return x+y.x;}
       friend int func(A& y){return y.x;}
};

void main()
{
 A a(5), b(7);
 cout<<a.func(b)<<endl;  //  12
 cout<<func(b)<<endl;    //   7
}
```

Дружня функція може бути перевантажена (відрізняється списком параметрів). Якщо вона визначається всередині декількох класів, то перевантаження

поширюється на усі ці класи у межах дії заданого простору імен. Так, у класах A та B можна виконати перевантаження дружньої функції, визначеної у класі A:

```
class A{
public:
  void output()
 {cout<<"Метод класу A"<<endl;}
  friend  void output(A a)
 {cout<<"Дружня функція класу A"<<endl;}
  friend  void output(A a, int x)
 {cout<<"Перевантажена дружня функція класу A"<<endl;}
};

class B{
  friend  void output(A a);
  friend  void output(B b)
 {cout<<"Перевантажена дружня функція класу B"<<endl;}
};

void main()
{
A a;
a.output();
output(a);
output(a, 5);

B b;
output(b);
}
```

У цьому прикладі визначену у класі A функцію output(A) можна оголосити дружньою до класу B, але не можна виконати її повторне визначення (реалізацію) у класі B.

Дружня функція не може бути оголошена константною. Оголошення const використовується тільки для методів класу.

Дружня функція може мати клас пам'яті static, але не може бути статичною, подібно методам класу, наприклад:

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

class A{
friend  static void output();
};

static void output()
 {cout<<"Дружня функція класу A зі статичним \
       класом пам'яті"<<endl;
 }
```

```
void main()
{
 output();
}
```

Використане для дружньої функції `output()` оголошення `static` обмежує можливість використання цієї функції поза поточним файлом.

Відношення `friend` не є транзитивним, інакше – друзі друзів не є друзями класу. Розглянемо приклад, у якому клас `B` оголошено дружнім до класу `A`, а функція `main()` є дружньою до класу `B`:

```
class A {
 int x;
 public:
 A(int x=0){this->x=x;}
 friend class B;
 };

class B {
 int y;
 A  z;
 public:
 B(int y=0, A a=0):z(a){this->y=y;}
 int GetX(){return z.x;} // A::x доступне
 friend void main();
 };

void main()
{
A a(1);
B b(2,a);
//cout<<b.z.x<<endl;      // A::x не доступне
cout<<b.GetX()<<endl;     // 1
cout<<b.y<<endl;          // 2
}
```

Дружня до класу `B` функція `main()` не є другом класу `A`. Тому з функції `main()` не можна за допомогою `b.z.x` прочитати поле `A::x`. Замість цього необхідно викликати метод `B::GetX()`, який читає це поле завдяки дружбі класу `B` до класу `A`.

Для того, щоб мати доступ до поля даних `b.z.x` без використання методу `B::GetX()`, необхідно оголосити функцію `main()` дружньою до обох класів `A` та `B`.

Має значення порядок взаємного розміщення класів або зовнішніх функцій для можливості їх доступу до складових елементів класу, у якому вони оголошені як дружні.

Клас або зовнішня функція можуть бути оголошені дружніми до вищерозміщеного класу, як це показано у попередньому прикладі.

Для забезпечення доступу до елементів усіх частин класу через його об'єкти (або вказівники або посилання) зовнішня функція повинна бути оголошена дружньою тільки до вищерозміщеного класу або визначатися всередині класу.

Якщо зовнішня функція оголошена дружньою до нижчерозміщеного класу, то вона може оголошувати вказівники або посилання на цей клас, але не може використати їх для звернення до елементів класу.

Якщо у класі необхідно викликати нижчерозміщену зовнішню функцію або метод іншого класу, то необхідно виконати випереджувальне оголошення такої функції, наприклад:

```
class B;
int get(B& b);

class A{
 B& x;
 public:
 A(B& x1):x(x1){}
 friend void print(A a){cout<<get(a.x)<<endl;}
};

class B{
 int x;
public:
 B(int x=0){this->x=x;}
 friend int get(B& b)
 {return b.x;}

};

void main()
{
B b(5);
A a(b);
print(a);
}
```

Клас A можна оголосити дружнім до нижчерозміщеного класу B та мати доступ до усіх його елементів. Для використання імені класу B у вищерозміщеному класі A необхідно виконати випереджувальне оголошення класу B. Реалізації методів класу A, які надають доступ до класу B, повинні бути розміщені нижче класу B. Наприклад:

```
class B; // потрібно випереджувальне оголошення класу B
```

```
class A {
 int x;
 public:
 A(int x=0){this->x=x;}
 int Get(B b);     // реалізація методу A::Get(B)
// повинна бути розміщена поза класом B
 };

class B {
 int y;
 public:
 B(int y=0){this->y=y;}
 friend class A;
 };

// реалізація методу A::Get(B)
 int A::Get(B b){return b.y;}

void main()
{
A a(1);
B b(2);
cout<<a.Get(b)<<endl;
}
```

Метод класу A може бути оголошений дружнім тільки до нижчерозміщеного класу B. Якщо метод класу A використовує ім'я нижчерозміщеного класу B, то його реалізація повинна розміщуватися нижче класу B.

```
class B;
class A {
 //...
 public:
 int Get(B b);
};

class B {
 //...
 public:
 friend int A::Get(B);
 };
int A::Get(B b){/*...*/}
```

Дружні методи класів можуть бути звичайними, константними або статичними. Методам класу A (крім статичних), оголошеним дружніми до класу B, передається вказівник this на об'єкт власного класу A.

Тільки вище розміщуються template – класи, дружні до заданого класу.

Класи можуть бути взаємодружніми, наприклад:

```
#include <iostream>
```

```
using namespace std;
class B;

class A {
 int x;
 public:
 A(int x=0){this->x=x;}
 int Get(B b);
 friend class B;
 };

class B {
 int y;
 public:
 B(int y=0){this->y=y;}
 int Get(A a){return a.x;}
 friend class A;
 };

 int A::Get(B b){return b.y;}

void main()
{
A a(1);
B b(2);
cout<<a.Get(b)<<endl;     // 2
cout<<b.Get(a)<<endl;     // 1
}
```

2.10. Приклад програми

Розробити клас для роботи з рядками символів. Перевантажити конструктори для розміщення рядка в області динамічної пам'яті: конструктор без параметрів для формування порожнього рядка, конструктори з параметрами для утворення рядка із одного символу, утворення рядка з рядка, конструктор копіювання рядків. Визначити деструктор для вивільнення відведеної для рядка динамічної пам'яті. Перевантажити методи для конкатенації рядка з рядком та рядка із символом. Визначити методи для порівняння рядків, введення та виведення рядка. Перевантажити дружні функції для копіювання рядка у рядок та символу у рядок. Застосувати методи та дружні функції для конкатенації, копіювання і виведення рядків. Виконати сортування масиву рядків символів.

```
#include <iostream>
```

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

class String
  {
   char *str;      // Вказівник на рядок
   public:

// Перевантажені конструктори
     String();
     String(char);
     String(char *);
     String(String&);

// Деструктор
     ~String();

// Дружні функції копіювання рядка та символу у рядок
      friend String& Copy(String&, const String&);
      friend String& Copy(String&, const char&);

// Перевантажений метод конкатенації рядків і рядка з символом
      String Plus(String);
      String Plus(char);

// Методи порівняння рядків
      int Lt(String&);
      int Eq(String&);
      int Gt(String&);

// Методи введення і виведення рядка
      void Input();
      void Output();
  };

// Конструктор без параметрів виділяє динамічну пам'ять
// для порожнього рядка
String::String()
     {
      str=new char;
      *str=0;
     }

// Конструктор з параметром виділяє динамічну пам'ять
// для рядка, який складається з одного символу
     String::String(char c)
     {
      str=new char[2];
      str[0]=c;
```

```
      str[1]='\0';
     }

// Конструктор з параметром виділяє динамічну пам'ять
// для рядка символів
  String::String(char *s)
  {
    str=new char[strlen(s)+1];
    strcpy(str,s);
  }

// Конструктор копіювання рядка
      String::String(String& s)
       {
      str=new char[strlen(s.str)+1];
      strcpy(str, s.str);
       }

// Деструктор
     String::~String()
     { delete []str;}

// Функція копіювання рядка у рядок
  String& Copy(String& s1, const String& s2)
  { if(&s1!=&s2)
   {
    delete[] s1.str;
    s1.str=new char[strlen(s2.str)+1];
    strcpy(s1.str, s2.str);
   }
    return s1;
  }

// Функція копіювання символу у рядок
   String& Copy(String& s1, const char& c)
   {
  delete[] s1.str;
     s1.str=new char[2];
     s1.str[0]=c;
     s1.str[1]='\0';
     return s1;
   }

// Метод конкатенації рядків
   String String::Plus(String s2)
  { String s;
    delete s.str;
    s.str=new char[strlen(str)+strlen(s2.str)+1];
    strcpy(s.str, str);
```

```
    strcat(s.str, s2.str);
    return s;
  }

// Метод конкатенації рядка і символу
   String String::Plus(char c)
  { String s;
    delete s.str;
    s.str=new char[strlen(str)+2];
    strcpy(s.str, str);
    s.str[strlen(str)]=c;
    s.str[strlen(str)+1]='\0';
    return s;
  }

//  Метод Lt (Little) порівняння рядків
   int String::Lt(String& s2)
  {
   return strcmp(str,s2.str)<0;
  }

// Метод Gt (Great) порівняння рядків
   int String::Gt(String& s2)
  {
   return strcmp(str,s2.str)>0;
  }

// Метод Eq (Equivalence) порівняння рядків
   int String::Eq(String& s2)
  {
   return strcmp(str,s2.str)==0;
  }

// Метод введення рядка з клавіатури
   void String::Input()
  {
   cin>>str;
  }

// Метод виведення рядка на екран
   void String::Output()
  {
   cout<<str<<endl;
  }

// Функція сортування масиву рядків.
  void sort(String *a, int n)
  {
   String s;
```

```
   for(int i=0; i<n-1; i++)
    for(int j=i+1; j<n; j++)
     if(a[i].Gt(a[j])) {
                Copy(s, a[i]);
                Copy(a[i], a[j]);
                Copy(a[j], s);
               }
   }

// Головна функція
  void main()
  {
   String s,s1("Hello"), s2("world");

   Copy(s, s1.Plus(' ').Plus(s2));

   Copy(s, s.Plus("!!!"));
   s.Output();    // Hello world!!!

    if(s1.Lt(s2)) cout<<"s1<s2"<< endl;
    else cout<<"s1>=s2"<< endl;

   const int n=5;
   String a[n] = {"6543",
                  "7842",
                  "3472",
                  "5219",
                  "3243"};

   sort(a,n);

   for(int i=0;i<n;i++)
    a[i].Output();
  }
```

Вдале визначення і перевантаження методів класу дає змогу будувати семантично зрозумілі конструкції програми, наприклад, `s1.Plus(s2)` – виконати конкатенацію рядків `s1` та `s2`, `s1.Eq(s2)` – перевірити тотожність рядків та ін. Однак засобами мови C++ завдяки перевантаженню потрібних операцій (див. п. 4.10) можна будувати більш лаконічні та інтуїтивно зрозумілі вирази над об'єктами класу.

Розділ 4
ПЕРЕВАНТАЖЕННЯ ОПЕРАЦІЙ

4.1. Операторні функції

Операції C++ є контекстнозалежними і можуть бути перевантажені у класі. Перевантажені операції використовуються як макрокоманди над об'єктами класу.

Перевантаження допускається для таких операцій:

1) первинних:

()	[]	->

2) унарних:

!	~	+	-	++	--	&	*	new	delete

3) бінарних:

->*	*	/	%	+	-	<<	>>
<	<=	>	>=	==	!=	&	^
|	&&	||	=	*=	/=	%=	+=
-=	&=	^=	|=	<<=	>>=	,	

Не можуть бути перевантажені операції

.	.*	::	?:	sizeof	typeid	throw

dynamic_cast	static_cast	const_cast	reinterpret_cast

та директиви препроцесора

#	##

Перевантаження операцій здійснюється за допомогою *операторних функцій*. Оголошення операторних функцій:

```
Тип    operator    позначення_операції (
       список формальних параметрів )
```

```
{
Локальні оголошення;
оператори;
return вираз;
}
```

При перевантаженні не можна змінити пріоритети та асоціативність операцій. Не можна перевантажити операції, визначені для вбудованих типів даних. Не можна вводити нові позначення операцій.

Операторні функції можуть бути оголошені членами або друзями класу. Тільки членами класу оголошуються такі операції:

()	[]	->	new	delete	=

Усі інші операції, які допускають перевантаження, можуть бути оголошені як членами, так і друзями класу.

Операції `new` та `delete` за замовчуванням є статичними (`static`). Усі інші операції не можуть бути статичними.

Усі операції, крім =, допускають успадкування. Операція = не може бути константною (`const`). Якщо операція змінює значення полів класу, то відповідна операторна функція не може бути константною.

Операторні функції не можуть мати параметрів за замовчуванням.

4.2. Варіанти перевантаження операцій

Якщо операторна функція є членом класу, то її першим (неявним) параметром є вказівник `this`. Кількість параметрів операторної функції визначається її видом та способом оголошення – як члена або як друга класу. Варіанти перевантаження унарних і бінарних операцій подано у табл. 4.1.

Таблиця 4.1

Варіанти перевантаження операцій

Операції	Члени класу	Друзі класу
Унарні	Без параметрів	Один параметр з типом класу, посиланням або вказівником на об'єкт класу
Бінарні	Один параметр (другий)	Два параметри (перший параметр повинен мати тип класу, посилання або вказівника на об'єкт класу)

Примітка. операції `()` та `new` можуть мати більше двох параметрів.

4.2.1. Унарні операції – члени класу

Унарна операція-член класу перевантажується за допомогою нестатичної операторної функції без параметрів. Наприклад:

```
class A {
int  x;
public:
A(int y=0):x(y) { }
A&  operator-() const;
};

A&  A::operator-() const
 { return A(-x);}

void main()
{
Aa(1), b;
b=-a;
// ...
}
```

4.2.2. Унарні операції – друзі класу

Якщо унарна операція перевантажується як друг класу, то вона має один параметр з типом класу, посилання або вказівника на об'єкт класу. Наприклад:

```
class A {
 int x;
 public:
 A(int y=0):x(y) { }
 friend A operator-(const A&);
 };

A operator-(const A& a)
{ return A(-a.x);}

void main()
{
A a(1), b;
b = -a;
// ...
}
```

4.2.3. Бінарні операції – члени класу

Перевантаження бінарної операцій як члена класу є нестатичною операторною функцією з одним параметром (другим, перший є вказівником this). Наприклад:

```
class A {
 int   x;
 public:
 A(int y=0):x(y) { }
 A  operator-(const A&) const;
 };

A  A::operator-(const A&a) const
{ return A(x-a.x);}

void main()
{
A a(1), b(2), c;
c = b - a;
// ...
}
```

4.2.4. Бінарні операції – друзі класу

Бінарна операція, перевантажена як друг класу, є операторною функцією з двома параметрами (перший – змінна класу, посилання або вказівник на об'єкт класу). Наприклад:

```
class A {
 int x;
 public:
 A(int y=0):x(y) { }
 friend A operator-(const A&, const A&);
 };

A  operator-(const A& b, const A& a)
{ return A(b.x-a.x);}

void main()
{
A a(1), b(2),c;
c = b - a;
// ...
}
```

4.2.5. Виклики операторних функцій

Можливі два варіанти викликів операторних функцій – явний та неявний. Явний виклик операторних функцій здійснюється так само, як і звичайних дружніх функцій або методів класу. Наприклад, для розглянутих варіантів перевантаження маємо такі варіанти явних викликів:

```
b = a.operator-();        // унарний мінус - метод класу
b = operator-(a);         // унарний мінус - друг класу
c = b.operator-(a);       // бінарний мінус - метод класу
c = operator-(b,a);       // бінарний мінус - друг класу
```

Неявний виклик операторних функцій будується на основі застосування позначень відповідних операцій до об'єктів класу. Наприклад:

```
b = -a;       // унарний мінус - метод класу
b = -a;       // унарний мінус - друг класу
c = b - a;    // бінарний мінус - метод класу
c = b - a;    // бінарний мінус - друг класу
```

Очевидно, що другий варіант неявного застосування операторних функцій є більш прийнятним, особливо у складних виразах над об'єктами класу.

Для ілюстрації цього наведемо приклад програми-калькулятора, побудованої на перевантаженнях арифметичних операцій додавання, віднімання, множення та ділення дійсних чисел. Використаємо перевантажені операції для обчислення виразу над об'єктами класу. Для виведення результату обчислення на екран перевантажимо операцію виведення у потік.

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

class calc{
float x;
public:
calc(float x=0);
calc operator+(calc b);
calc operator-(calc b);
calc operator*(calc b);
calc operator/(calc b);
friend ostream& operator<<(ostream&, calc&);
};

calc::calc(float x)
{ this->x=x;}

calc calc::operator+(calc b)
{
```

```
 return x+b.x;
}

calc calc::operator-(calc b)
{
 return x-b.x;
}

calc calc::operator*(calc b)
{
 return x*b.x;
}

calc calc::operator/(calc b)
{
   if(b.x!=0) return x/b.x;
   else {cout<<"Ділення на нуль"<<endl; exit(-1);}
}

ostream& operator<<(ostream& os, calc& a)
{
 os.width(-10);
 os.precision(2);
 os<<a.x;
 return os;
}

void main()
{
 calc a(1), b(2), c(3), d;
 // обчислити вираз (a-b)*c/(a+b+c)
 d=(a-b)*c/(a+b+c);
 cout<<d<<endl;

 // обчислити вираз (a-b*c)/(a+b+c)
 d=(a-b*c)/(a+b+c);
 cout<<d<<endl;
}
```

Порівняйте текст цієї програми з прикладом програми п. 2.4.3, яка реалізує обчислення виразів на основі викликів відповідних методів класу.

4.2.6. Рекомендації щодо вибору варіанта перевантаження операцій

Ряд операцій допускає перевантаження у вигляді як методів, так і друзів класу. Може виникнути запитання: як краще оголошувати операторні функції? У зв'язку з цим можна сформулювати такі рекомендації:

- якщо операторним функціям не потрібен прямий доступ до внутрішніх даних класу, то вони оголошуються друзями класу, щоб не порушувати принципу інкапсуляції;
- операторні функції, які модифікують внутрішні дані класу, наприклад, =, +=, ++, повинні оголошуватися методами класу;
- якщо необхідне неявне перетворення типів аргументів операторної функції, то вона повинна оголошуватися дружньою до класу;
- як правило, унарні операції оголошуються членами класу, а бінарні – друзями класу, що підтримує комутативність деяких бінарних операцій (наприклад, `a + b == b + a`).

Для ілюстрації останнього розглянемо приклад класу, який перевантажує бінарну операцію `'+'` як члена класу:

```
class A {
 int x;
 int y;
public:
 A(int=0,int=0);
 A  operator+(A);
};
A::A(int x1, int y1):x(x1), y(y1){}

A  A::operator+(A a)
{return A(x+a.x, y+a.y);}

void main()
{ A a(1,2), c;
c = a + 7;   // добре, евівалентно a.operator+(7);
//c = 7 + a; // помилка, не визначено 7.operator+(a);
}
```

Операція `a+7` еквівалентна явному виклику операторної функції `a.operator+(7)`. Така операція є коректною за наявності конструктора перетворення типу `A(int)`, який перетворює тип `int` до типу класу `A`. У наведеному прикладі його роль виконує конструктор з параметрами за замовчуванням `A(int=0,int=0)`.

При заміні місцями операндів отримаємо: `7+a`. Оскільки при оголошенні операторної функції членом класу доступ до першого аргумента здійснюється за

допомогою вказівника this, то для перетворення аргумента 7 конструктор перетворення не викликається. Результатом цього є семантична помилка, для усунення якої необхідно операторну функцію operator+ оголосити другом класу:

```
class A {
// ...
public:
 A(int=0,int=0);
 friend A operator+(A,A);
};

A operator+(A a, A b)
{return A(a.x+b.x, a.y+b.y);}
```

Тепер операція 7+a еквівалентна явному виклику операторної функції operator+(7,b). Для перетворення першого аргумента до типу класу буде викликаний конструктор перетворення типу.

Слід зауважити, що неявне перетворення типів не здійснюється для неконстантних параметрів-посилань.

4.3. Особливості перевантаження первинних операцій

4.3.1. Операція-функтор ( )

Операція () перевантажується як член класу, може повертати один з визначених типів та може мати змінну кількість параметрів. Застосована до об'єкта класу, ця операція візуально зображає виклик методу класу. Наприклад:

```
#include <iostream>
class A
{
 int x, y;
 public:
 A(int=0, int=0);
A& operator() (int,int); // може мати інший тип
                  // та іншу кількість аргументів
 void print(void)
 {cout<<x<<' '<<y<<endl;}
};

A::A(int a1, int a2) : x(a1),y(a2){}

A& A::operator() (int x1,int y1)
{
```

```
x=x1;
y=y1;
return *this;
}

void main()
{
 A a(1,2), f;
 a.print();      // 1 2
 a=f(3,4); // викликається перевантажена операція ()
 a.print();      // 3 4
}
```

4.3.2. Операція-індексатор [ ]

Перевантажується як бінарна нестатична функція-член класу. Якщо повертає посилання, то дозволяє використання з обох сторін операції присвоєння. Можливість використання зліва від операції присвоєння пояснюється тим, що функції з типом посилання можна присвоїти значення, тип якого відповідає типу функції.

Для прикладу розглянемо перевантаження операції [] для контролю діапазону індекса одновимірного масиву цілих чисел:

```
#include <iostream>

using namespace std;

class A{
int *p;
int size;
public:
A(int n){p=new int[size=n];
 for(int i=0;i<size;i++) p[i]=0;
     }
~A(){delete []p;}

int& operator[](int i)
 { if(i<0 || i>=size)
     throw "Індекс поза діапазоном допустимих значень";
   return p[i];
 }
};

void main()
{
A obj(10); // оголошено один об'єкт
cout<<"Початок"<<endl;
```

```
try{
    for(int i=0; i<=20; i++)
    {obj[i]=i;          // викликається A::operator[]
     cout<<obj[i]<<endl; // викликається A::operator[]
    }
   }
catch(char * s)
{ cout<<s<<endl;}

cout<<"Кінець"<<endl;
}
```

Зверніть увагу, що операція `[]` застосовується до об'єкта класу. У функції `main()` оголошено один об'єкт `obj`, який містить масив з 10 цілих чисел. У циклі `for` позначення `obj[i]` означає застосування операції `[ ]` до об'єкта `obj`, а не звернення до `i`-го елементу масиву об'єктів.

Як тільки змінна `i` циклу `for` дорівнюватиме 10, операторна функція `A::operator[]` згенерує виняткову ситуацію і керування буде передано з цієї функції в `cath`-обробник. Програма виведе послідовні значення від 0 до 9.

На основі операції `[]` можна побудувати роботу з двовимірними (і багатовимірними) масивами даних, інкапсульованими в об'єкт класу. Наприклад:

```
#include <iostream>

using namespace std;

class matrix{
float **v;
int  row, col;
public:
matrix(int, int);
~matrix();

float* operator[](int);

};

matrix::matrix(int n, int m)
{
 row=n;
 col=m;
 v=new float*[row];
 for(int i=0; i<row; i++)
  v[i]=new float[col];

 for(int i=0; i<row; i++)
  for(int j=0; j<col; j++)
```

```
    v[i][j]=0;
}

matrix::~matrix()
{
for(int i=0; i<row; i++)
 delete []v[i];

 delete []v;
}

float* matrix::operator[](int i)
{
return v[i];
}

void main()
{
matrix obj(3,4);

cout<<"Введення матриці дійсних чисел"<<endl;
for(int i=0; i<3; i++)
   for(int j=0; j<4; j++)
     cin>>obj[i][j];

cout<<"Виведення матриці дійсних чисел"<<endl;
for(int i=0; i<3; i++)
  { for(int j=0; j<4; j++)
     cout<<obj[i][j]<<' ';
    cout<<endl;
  }
}
```

Як і в попередньому прикладі, операція `[]` застосована до об'єкта `obj`. Перевантажена операція `[]` повертає вказівник на початок `i`-го рядка матриці дійсних чисел. Другі квадратні дужки виразу `obj[i][j]` забезпечують звернення до `j`-го елемента `i`-го рядка, тобто `obj[i][j] == *(obj[i]+j)`.

4.3.3. Операція ->

Використовується для доступу до елементів структури або класу за допомогою вказівника на структуру або клас.

Особливості:

- перевантажується як унарна постфіксна операція, хоча вимагається наявність ідентифікатора після ->;
- повинна бути нестатичною функцією-членом класу;

- не повинна мати параметрів;
- повинна повертати вказівник на структурований тип (може повертати посилання чи об'єкт класу, до яких, своєю чергою, можна застосувати операцію ->):

```
class A{
public:
int x;
A (int x){this->x=x;}
A* A::operator->(){ return this;}
} a(5);
```

- оригінальне значення операції -> не губиться, а тільки затримується. Так, для попереднього оголошення вираз a->x інтерпретується як (a.operator->())->x.

Наступна програма, завдяки перевантаження операції ->, забезпечує доступ до даних класу A за допомогою об'єкта класу B. Необхідно звернути увагу на те, що перевантажена операція -> застосовується до об'єкта класу, на відміну від її стандартного використання для вказівника на структурований елемент даних.

```
#include <iostream>
using namespace std;
// цільовий клас
class A{
      int x;
      public:
      A(int);
      friend void main();
     };
// конструктор
      A::A(int x)
      {this->x=x;}
// клас-посередник
class B{
      A* p;
      public:
      B(int);
      ~B();
      A* operator->();
      A& operator*();
      A& operator[](int);
      A* operator&();
      };
// конструктор
      B::B(int x)
      {p=new A(x);}
```

```
// деструктор
      B::~B()
       {delete p;}
// перевантажена операція ->
      A* B::operator->()
      {
       cout<<"B::operator->()\n";
       if(p==NULL)
       throw "Помилка виділення динамічної пам'яті";
       return p;
      }
// перевантажена операція розіменування вказівника
     A& B::operator*()
     {return *p;}
// перевантажена операція квадратні дужки
     A& B::operator[](int i)
   {return p[i];}
// перевантажена операція визначення адреси
   A* B::operator&()
   {return p;}

// головна функція
void main()
{
 B b(5);
 cout<<"Початок"<<endl;

 try{
 cout<<"b->x = " << b->x << endl;
//  b->x == (b.operator->())->x
 cout<<"(*b).x = " << (*b).x << endl;
 cout<<"b[0].x = " << b[0].x << endl;
 cout<<"(&b)->x = " << (&b)->x << endl;
}

catch(char * s)
{
 cout<<s<<endl;
}

 cout<<"Кінець"<<endl;
}
```

Клас, який перевантажує операцію `operator->`**, називають** `smart`**-вказівником. Такий клас містить вказівник, посилання або об'єкт іншого класу. Коли операція** `operator->` **застосовується до** `smart`**-вказівника, то її виклик переадресовується об'єкту іншого класу. При перевантаженні операції** `operator->`

можуть виконуватися перевірки, необхідні для правильної організації роботи з динамічною пам'яттю.

Разом з перевантаженням операції operator->, як правило, перевантажуються пов'язані з нею операції розіменування вказівника (operator*), квадратні дужки (operator[]), визначення адреси (operator&) та арифметичні операції над вказівниками, наприклад, інкремента (operator++) та декремента (operator--).

За допомогою smart-вказівника можна мати доступ до даних, як це було показано у попередньому прикладі, та до методів, як це демонструє наступний приклад:

```
#include <iostream>

using namespace std;

class A{
      public:
         void func(){cout<<"A::func()"<<endl;}
       };

class B{
      A* q;
      public:
         B(A* p){q=p;}

         A* operator->(){return q;}
      };

void main()
{
A* p = new A;
B b(p);
b->func();    // A::func()
}
```

4.4. Операції інкремента та декремента

Ці операції застосовуються у префіксній (++x, --x) та постфіксній (x++, x--) формах.

Перевантажуються як унарні операції. Можуть бути членами або друзями класу. Для розрізнення префіксної та постфіксної форм у постфіксній формі використовується додатковий параметр типу int, який явно не задається.

Можливі заголовки операторних функцій інкремента, перевантажених у класі A, наведені у табл. 4.2.

Таблиця 4.2

Варіанти прототипів операторної функції інкремента

Операція ++	Префіксна форма	Постфіксна форма
Метод класу	`A& operator++();`	`A& operator++(int);`
Друг класу	`friend A& operator++ (A&);`	`friend A& operator++ (A&,int);`

Перевантаження операторної функції декремента здійснюється аналогічно.

Приклад програми, у якій використовуються перевантажені операції префіксного та постфіксного інкремента:

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

class A
{
 int x, y;
 public:
 A(int,int);
 A& operator++();
 friend A& operator++(A&,int);
 void print(void)
 {cout<<x<<' '<<y<<endl;}
};

A::A(int a1, int a2): x(a1), y(a2){}

A& A::operator++()
{
++x;
++y;
return *this;
}

A& operator++(A& a, int)
{
a.x++;
a.y++;
return a;
}

void main()
{
```

```
 A a(1,2);
 a.print();  // 1  2
 a++;
 a.print();  // 2  3
 ++a;
 a.print();  // 3  4
}
```

4.5. Операції new та delete

4.5.1. Прототипи операторних функцій

Операція `new` використовується для виділення, а операція `delete` – для звільнення динамічної пам'яті. Ці операції можуть бути перевантажені у контексті класу або без використання класів.

У контексті класу перевантажуються за допомогою методів, не можуть бути друзями класу. Відповідні операторні функції є статичними (`static`) за замовчуванням. У них не можна використовувати нестатичні елементи класу. Успадковуються, не можуть бути віртуальними.

Стандарно визначені операції `new` та `delete` мають такі прототипи:

- для виділення динамічної пам'яті для одного елемента визначеного типу:

```
void *operator new(size_t);
void  operator delete(void *);
```

- для виділення динамічної пам'яті для масивів елементів:

```
void * operator new[](size_t);
void   operator delete[](void *);
```

Формальний параметр функції `new` набуває значення розміру об'єкта у байтах, для якого виділяється область динамічної пам'яті. Це значення обчислюється автоматично і його не потрібно задавати як значення фактичного параметра. Тип `size_t` є перевизначеним цілочисловим типом. Операторна функція `new` повертає вказівник на виділену область динамічної пам'яті.

Параметром операторної функції `delete` є вказівник на область динамічної пам'яті, попередньо виділеної операцією `new`.

Хоча операторна функція `new` оголошена як така, що повертає вказівник на порожній тип, але реально повертає вказівник на тип, для якого виділяється пам'ять. Тому явного перетворення типу не вимагається. Наприклад:

```
class A{
// ...
};
```

```
A *p = new A;       // один об'єкт
A *q = new A[10];   // масив з 10 об'єктів
```

Операція звільнення пам'яті від масиву об'єктів `delete []q` не вимагає задання кількості елементів у квадратних дужках. Якщо значення у квадратних дужках і вказується, то воно ігнорується.

4.5.2. Приклад програми перевантаження new та delete

Розглянемо приклад перевантаження операцій `new` та `delete` у контексті класу. Крім виділення та звільнення динамічної пам'яті, операторні функції `new` та `delete` здійснюють виведення ідентифікаційного повідомлення.

У цьому прикладі насправді виконано перевизначення стандартних операторних функції `new` та `delete`, оскільки вони мають однакові (зі стандартними) прототипи.

```
#include <iostream>
#include <cstdlib>

using namespace std;

class A
{
 int x;
 public:
 A(int x=0){cout<<"A(int)="<<x<<endl;
            this->x=x;
          }
 void print(){cout<<x<<endl;}
~A(){cout<<"~A()="<<x<<endl;
 }

void* operator new(size_t);
// перевизначає стандартну операцію new
void  operator delete(void *);
// перевизначає стандартну операцію delete

void* operator new[](size_t);
// перевизначає стандартну операцію new[]
void operator delete[](void *);
// перевизначає стандартну операцію delete[]
};

// операція new

void* A::operator new(size_t size)
{
```

```
 cout<<"A::new(size_t size)"<<endl;
 return malloc(size);
}

// операція delete
void A::operator delete(void *p)
{
cout<<"A::delete(void *p)"<<endl;
free(p);
}

// операція new[]

void* A::operator new[](size_t size)
{
 cout<<"A::new[](size_t size)"<<endl;
 return malloc(size);
}

// операція delete[]

void A::operator delete[](void *p)
{
cout<<"A::delete[](void *p)"<<endl;
free(p);
}

void main()
{

A *p=new  A(1);

// A::new(size_t size)
// A(int)=1

p->print();        // 1

delete p;
// ~A()=1
// A::delete(void *p)

A *q=new A[4];
// для масивів потрібен конструктор без параметрів
// конструктори та деструктори викликаються для
// кожного елемента масиву

// A::new[](size_t size)
// A(int)=0
// A(int)=0
```

```
// A(int)=0
// A(int)=0

for(int i=0;i<4; i++) q[i]=A(i+1);
// A(int)=1  ~A()=1
// A(int)=2  ~A()=2
// A(int)=3  ~A()=3
// A(int)=4  ~A()=4

for(int i=0;i<4; i++) q[i].print();
// 1
// 2
// 3
// 4

delete [] q;
// ~A()=4
// ~A()=3
// ~A()=2
// ~A()=1
// A::delete[](void *p)
}
```

Коментарі функції `main()` показують виведення, викликані дією операцій `new` та `delete`. Операції `new` призводять до виклику конструкторів об'єктів, а операції `delete` – їх деструкторів.

4.5.3. Додатковий список параметрів операції new

Операції `new` та `new[]` можуть мати більше одного параметра. Тоді вони перевантажуються так:

```
class A{
  char* p;
public:
  // ...
  void * operator new(size_t size,
                      додатковий список параметрів);
  void * operator new[](size_t size,
                      додатковий список параметрів);
   };
```

Елементи додаткового списку параметрів можуть мати довільний тип.

При виклику перевантажених операцій `new` та `new[]` додатковий список аргументів задається у круглих дужках після слова `new`. Приклад використання операцій `new` та `new[]` у функції `main()`:

```
void main()
{
```

```
A  *p = new(додатковий список аргументів)
           A(аргументи конструктора);
A  *q = new(додатковий список аргументів)
           A[кількість об'єктів];
}
```

4.5.4. Розміщувана форма оператора new

Якщо додатковим параметром операції new є вказівник на раніше визначену змінну, то за допомогою такої операції можна розмістити об'єкт класу за відомою адресою. У цьому випадку динамічна пам'ять для об'єкта не виділяється, а операція new повертає адресу раніше виділеної області пам'яті.

Якщо це адреса раніше створеного об'єкта цього самого класу, то можна виконати його нову ініціалізацію повторним викликом конструктора. В іншому випадку розмір об'єкта не повинен перевищувати розмір раніше визначеної змінної.

Звільнення динамічної пам'яті від розміщуваного об'єкта здійснюється явним викликом деструктора, тому що така область явно не керується розподілювачем пам'яті.

Наприклад:

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

class A
{int x;
 public:
A(int x=0)
{ this->x=x; cout<<"A()="<<x<<endl; }
~A(){ cout<<"~A()="<<x<<endl; }
inline void* operator new(size_t t, void *p)
{ return p; }
};

void  main()
{
A a(5);

// розмістити динамічний об'єкт за адресою об'єкта a
A* ptr1 = new (&a) A(7);

// розмістити динамічний об'єкт за адресою масиву x
char x[sizeof( A )];
A* ptr2 =new( &x[0] ) A(9);

ptr2 -> ~A();
ptr1 -> ~A();
}
```

Результатом роботи програми є такі виведення:

```
A()=5
A()=7
A()=9
~A()=9
~A()=7
~A()=7
```

Останнє виведення пояснюється тим, що об'єкт `a(5)` за допомогою розміщуваної форми операції `new` отримав нове значення, що дорівнює `A(7)`.

4.6. Операція присвоєння

Особливості операції присвоєння:

- використовується для присвоєння одного об'єкта іншому існуючому об'єкту класу;
- існує за замовчуванням у кожному класі, який не містить константних елементів даних або даних з типом посилання;
- повинна бути розміщена у відкритій частині класу для можливості присвоєння об'єктів поза класом;
- не успадковується;
- не може бути константною;
- оголошується як нестатична функція-член класу:

```
class T{ int x;
     int y;
 public:
 T(int a=0, int b=0):x(a),y(b){}
 T&    operator=(const T&);
 };

T&     T::operator=(const T& t)      // або (T& t)
{
x=t.x;
y=t.y;
return *this;
}
```

Алгоритм перевантаження операції присвоєння подібний алгоритму конструктора копіювання об'єктів. Відмінність полягає в тому, що конструктор копіювання створює нову копію об'єкта, а в операції присвоєння використовуються раніше створені об'єкти.

Тип операторної функції, визначений як посилання на тип класу T, робить можливим ланцюжкове присвоєння об'єктів:

```
T a, b, c(1,2);
a = b = c;
```

У загальному випадку параметр операторної функції присвоєння може мати тип, відмінний від типу класу, для якого здійснюється перевантаження. Це робить можливим присвоєння об'єкту класу T значення некласового типу або значення з типом іншого класу. Цього самого можна досягти за допомогою конструктора перетворення типу, застосованого до правої частини операції присвоєння.

Якщо операторна функція не перевантажена, то за замовчуванням компілятор будує код порозрядного копіювання об'єктів.

Якщо полем класу B є об'єкт іншого класу A, то навіть за відсутності перевантаженої операції B::operator= присвоєння об'єктів класу B призведе до виклику операції A::operator=, наприклад:

```
class A{
 int x;
public:
 A(){cout<<"A::A()"<<endl;}
 A(int x1):x(x1){cout<<"A::A(int)"<<endl;}
 A(A& a):x(a.x){cout<<"A::A(A&)"<<endl;}
 A& operator=(A& a)
 {cout<<"A::operator=(A&)"<<endl;
  x=a.x;
  return *this;
 }
 void output() {cout<<x<<endl;}
};

class B
{
 A a;
 public:
 B(){cout<<"B::B()"<<endl;}
 B(A& a1):a(a1){cout<<"B::B(A&)"<<endl;}
 void output() {a.output();}
};

void main()
{
 A a(5);            // A::A(int)
 B b1;              // A::A()
                    // B::B()
 B b2(a);           // A::A(A&)
```

```
                    // B::B(A&)
 b1=b2;             // A::operator=(A&)
 b1.output();       // 5
}
```

Якщо клас містить поля, які є вказівниками або посиланнями на будь-який тип, то операція присвоєння повинна бути перевантажена. Це демонструється наступними прикладами. У першому прикладі поле класу є вказівником.

```
#include <iostream>

using namespace std;

class A
{
 int* p;
 public:
 A(int x=0){p=new int(x);}
 A& operator=(A&);
 int Get(void){return *p;}
 void Set(int x){*p=x;}
};

 A& A::operator=(A& a)
 {
  cout<<"A::operator=(A&)"<<endl;
 if(this!=&a)       // виключаємо самоприсвоєння об'єктів
 {delete p;
 p=new int(*a.p);
 }
 return *this;
 }

 void main()
{

 int x=5;
 A a(x);
 cout<<a.Get()<<endl;     // 5
 A b;
 b=a;
 cout<<b.Get()<<endl;     // 5
 a.Set(7);
 cout<<b.Get()<<endl;     // 5
}
```

Якщо операція присвоєння є перевантажена так, як це зроблено вище, то після присвоєння об'єктів поля-вказівники будуть адресувати різні області оперативної пам'яті (рис. 4.1).

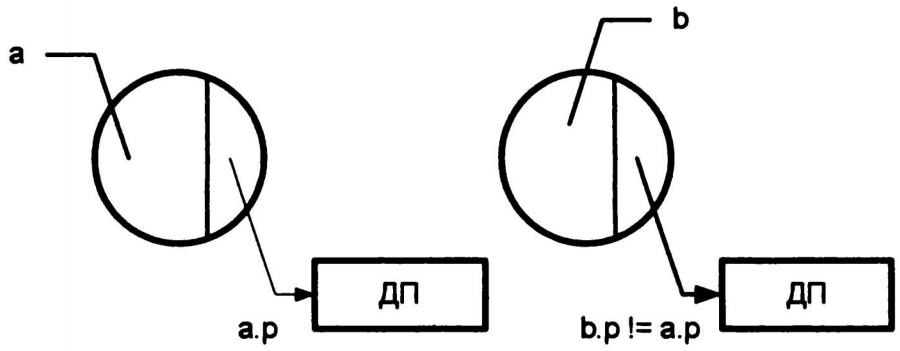

Рис. 4.1. Схема дії перевантаженої операції присвоєння

Якщо ж не перевантажувати операції присвоєння, то виконається порозрядне присвоєння об'єктів, і їх поля-вказівники адресуватимуть одну і ту саму область пам'яті (рис. 4.2).

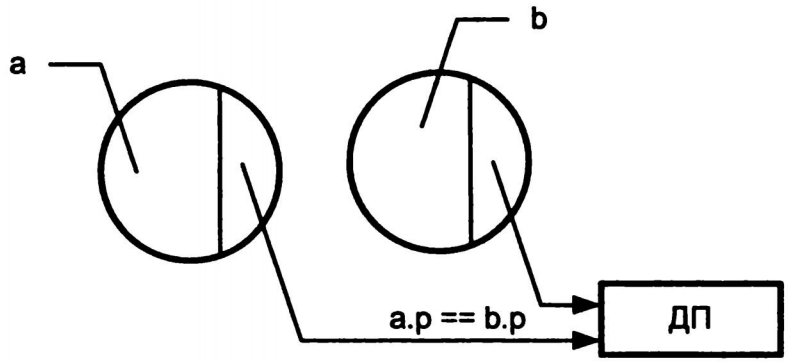

Рис. 4.2. Схема дії операції присвоєння за замовчуванням

За відсутності перевантаження операції присвоєння виведення програми матиме вигляд:

```
5
5
7
```

Якщо поле класу має тип посилання, то перевантаження операції присвоєння є обов'язковим і може бути виконане так:

```
class A
{
 int& r;
 public:
  A(int x=0):r(*new int(x)){}
 A& operator=(A&);
 int& Get(void){return r;}
 void Set(int x){r=x;}
};
```

```
A& A::operator=(A& a)
{
 cout<<"A::operator=(A&)"<<endl;
if(this!=&a)
{delete &r;
r=*new int(a.r);
}
return *this;
}

void main()
{

int x=5;
A a(x);
cout<<a.Get()<<endl;     // 5
A b;
b=a;
cout<<b.Get()<<endl;     // 5
a.Set(7);
cout<<b.Get()<<endl;     // 5
}
```

Якщо у попередній програмі закоментувати перевантаження операції присвоєння, то виникає помилка компіляції програми.

Властивості об'єктів з полями-посиланнями є аналогічними об'єктам з полями-вказівниками.

Якщо у класі визначено константний елемент даних, то компілятор не генерує операцію присвоєння за замовчуванням, і об'єкти класу не можуть бути присвоєні один одному. За необхідності присвоєння таких об'єктів можна виконати, перевантаживши операцію `operator=` за допомогою вказівника на константний елемент даних, наприклад:

```
class A
 {
  const int x;
  public:
 // конструктор
       A(int x1=0): x(x1){}
 // метод читання даних класу
  int Get(void){return x;}
// перевантажена операція присвоєння
A& operator=(A& a)
 {int A::*p=(int A::*)&A::x;
  this->*p=a.x;
  return *this;
 }
};
```

```
void main()
{
 int x=5;
 A a(x);
 cout<<"a.x=="<<a.Get()<<endl;      // 5
 A b(7);
 cout<<"b.x=="<<b.Get()<<endl;      // 7
 b=a;
 cout<<"b.x=="<<b.Get()<<endl;      // 5
}
```

4.7. Операція перетворення типу

Операторна функція перетворення типу призначена для перетворення об'єкта класу у значення іншого типу, зокрема у тип іншого класу. Оголошується як метод класу:

```
operator  інший_тип();
```

Допускає перевантаження. У протоколі класу може бути оголошено декілька операторних функцій перетворення типу. Функція перетворення типу не має явної специфікації типу результату та не має параметрів. Повинна повертати значення, яке має "інший тип". Може бути віртуальною, успадковується.

Операція перетворення типу застосовується у випадках, протилежних до варіантів застосування конструктора перетворення типу. Можливими є такі варіанти викликів:

- для ініціалізації змінної об'єктом класу, тип якого відрізняється від типу змінної (інший тип);
- у виразах виду: інший тип ⊕ класовий тип;
- для явного перетворення класового типу до будь-якого іншого типу;
- в операції присвоєння виду: інший тип = класовий тип;
- при передачі параметрів у функцію, коли формальний параметр має інший тип, а фактичний аргумент – класовий тип;
- повернення значення функцією, коли вираз оператора `return` має класовий тип, а функція має інший тип.

Приклад оголошення та використання операторної функції перетворення типу:

```
class A{
 int x;
 public:
 A(int x=0){this->x=x;}
```

```
 operator int() { return x;}
 operator long() { return (long)x;}
 operator unsigned() { return (unsigned)x;}
 operator char() { return (char)x;}
};

void main()
{
A a(5);
int x=a;      // неявне перетворення типу A у тип int
cout<<int(a)<<endl; // явне перетворення типу A у тип int
}
```

4.8. Перевантаження потокових операцій введення–виведення

Потокові операції введення–виведення перевантажуються як друзі класу, оскільки їх першим параметром повинен бути об'єкт потокового класу: `istream` – для введення, `ostream` – для виведення об'єктів класу. Наприклад:

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

class A {
 int x,y;
 public:
 A(int x=0,int y=0){
             this->x=x;
             this->y=y;
             }
 friend istream& operator>>(istream&, A&);
 friend ostream& operator<<(ostream&, A&);
 };

 istream& operator>>(istream& is, A& a)
 {
  is>>a.x>>a.y;
  return is;
 }

 ostream& operator<<(ostream& os, A& a)
 {
  os<<a.x<<' '<<a.y;
  return os;
 }
```

Перевантаження операцій потокового введення–виведення дає можливість цілісно вводити та виводити об'єкти оголошеного класу, наприклад:

```
void main()
{
A a;
cin>>a;              // введення об'єкта
cout<<a<<endl;       // виведення об'єкта
}
```

4.9. Послідовність виклику операторних функцій та конструкторів перетворення типу

Якщо у програмі існують перевантажені операторні функції та конструктори перетворення типів, то у виразах насамперед будуть викликатися ті операторні функції, які мають точну відповідність у типах аргументів. Якщо таких функцій не знайдено, то спочатку будуть викликані конструктори перетворення типів, а потім наявні для цих типів операторні функції. Наприклад:

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

class A{
 int x;
 public:
 A(int x) { cout<<"A::A(int)"<<endl;
     this->x=x;
   }

 A(const A& a):x(a.x){cout<<"A::A(A&)"<<endl;}

 A operator+(int x){ cout<<"A::operator+(int)"<<endl;
            x+=this->x;
            return A(x);
          }

 A operator+(A a){ cout<<"A::operator+(A)"<<endl;
          a.x+=this->x;
          return a;
        }

 A& operator=(A& b){ cout<<"A::operator="<<endl;
            this->x=b.x;
            return *this;
          }
```

```
 void get(void) {cout<<x<<endl;}
};

void main()
{
A a(1);
 A b=a+5;
 b.get();
}
```

Результатом роботи даної програми буде:

```
A::A(int)           // створення об'єкта a(1)
A::operator+(int)   // виконання операції a+5
A::A(int)           // створення тимчасового об'єкта A(x)
                    // у функції A::operator+(int)
6                   // результат додавання
```

Аналіз виведення показує, що для обчислення виразу `a+5` буде викликана перевантажена операторна функція `A::operator+(int)`, оскільки вона має точну відповідність типів аргументів до типів операндів. Необхідно звернути увагу на те, що у виразі

```
A b=a+5;
```

здійснюється ініціалізація об'єкта `b`, а не операція присвоєння.

Однак, якщо виконати ініціалізацію

```
A b(a+5);
```

то виведення буде таким:

```
A::A(int)           // створення об'єкта a(1)
A::operator+(int)   // виконання операції a+5
A::A(int)           // створення тимчасового об'єкта
                    // у функції A::operator+(int)
A::A(A&)            // конструктор копіювання при
                    // створенні об'єкта b
6                   // результат додавання
```

Якщо з тексту вилучити оголошення та визначення операторної функції `A::operator+(int)`, то виведення попередньої програми зміниться:

```
A::A(int)           //  створення об'єкта a(1)
A::A(int)           //  створення об'єкта a(5)
A::operator+(A)     //  виконання операції a+A(5)
A::A(A&)            //  повернення локального об'єкта
                    //  з функції A::operator+(A)
6                   //  результат додавання
```

Тобто, перед виконанням операції додавання буде викликаний конструктор `A::A(int)`, який перетворить ціле число `5` на об'єкт класу `A`, а потім буде виконано перевантажену операцію додавання `A::operator+(A)`.

4.10. Приклад програми

Написати програму на основі класу роботи з рядками символів. Перевантажити конструктори для формування рядка в області динамічної пам'яті. Визначити деструктор для вивільнення відведеної для рядка динамічної пам'яті. Перевантажити операції для присвоєння рядків та присвоєння рядку одного символа. Визначити дружні операторні функції для конкатенації рядка з рядком та рядка з символом, порівняння рядків, введення та виведення рядка. Застосувати методи та дружні функції для конкатенації, присвоєння і виведення рядків. Відсортувати масив рядків символів.

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

class String
  {
char *str;    // Вказівник на рядок

   public:

// Перевантажені конструктори
     String();
     String(char);
     String(char *);
     String(String&);

// Деструктор
     ~String();

// Перевантажена операція присвоєння рядків
// та присвоєння символу рядку
     String& operator=(const String&);
     String& operator=(const char&);

// Перевантажена операція конкатенації рядків і
// конкатенації рядка з символом
     friend String operator +  (String, String);
     friend String operator +  (String, char);
```

```
// Перевантажені операції порівняння рядків
     friend int    operator <  (String&, String&);
     friend int    operator == (String&, String&);
     friend int    operator >  (String&, String&);

// Перевантажені операції введення і виведення рядка
     friend istream& operator>>(istream&, String&);
     friend ostream& operator<<(ostream&, String&);
  };

// Конструктор без параметрів виділяє динамічну пам'ять
// для порожнього рядка
String::String()
     {
      str=new char;
      *str=0;
     }

// Конструктор з параметром виділяє динамічну пам'ять
// для рядка, який складається з одного символу
     String::String(char c)
     {
      str=new char[2];
      str[0]=c;
      str[1]='\0';
     }

// Конструктор з параметром виділяє динамічну пам'ять
// для рядка символів
  String::String(char *s)
  {
    str=new char[strlen(s)+1];
    strcpy(str,s);
  }

// Конструктор копіювання рядка
      String::String(String& s)
       {
 str=new char[strlen(s.str)+1];
 strcpy(str, s.str);
       }

// Деструктор
     String::~String()
     { delete []str;}

// Перевантажена операція присвоєння рядків
  String& String::operator=(const String& s2)
  { if(this!=&s2)
   {
```

```
    delete[] str;
    str=new char[strlen(s2.str)+1];
    strcpy(str,s2.str);
   }
    return *this;
  }

// Перевантажена операція присвоєння рядка одного символу
   String& String::operator=(const char& c)
   {
delete[] str;
     str=new char[2];
     str[0]=c;
     str[1]='\0';
     return *this;
   }

// Перевантажена операція конкатенації рядків
  String operator+(String s1, String s2)
  { String s;
    delete s.str;
    s.str=new char[strlen(s1.str)+strlen(s2.str)+1];
    strcpy(s.str,s1.str);
    strcat(s.str,s2.str);
    return s;
  }

// Перевантажена операція конкатенації рядка і символу
  String operator+(String s1, char c)
  { String s;
    delete s.str;
    s.str=new char[strlen(s1.str)+2];
    strcpy(s.str,s1.str);
    s.str[strlen(s1.str)]=c;
    s.str[strlen(s1.str)+1]='\0';
    return s;
  }

// Перевантажена операція < порівняння рядків
  int operator<(String& s1,String& s2)
  {
   return strcmp(s1.str,s2.str)<0;
  }

// Перевантажена операція > порівняння рядків
  int operator>(String& s1,String& s2)
  {
   return strcmp(s1.str,s2.str)>0;
  }
```

```
// Перевантажена операція == порівняння рядків
  int operator==(String& s1,String& s2)
  {
   return strcmp(s1.str,s2.str)==0;
  }

// Перевантажена операція введення рядка з клавіатури
  istream& operator>>(istream& is, String& s)
  {
   is>>s.str;
   return is;
  }

// Перевантажена операція виведення рядка на екран
  ostream& operator<<(ostream& os, String& s)
  {
   os<<s.str;
   return os;
  }

// Функція сортування масиву рядків.
// Викликаються перевантажені операції > та =
  void sort(String *a, int n)
  {
   String s;
   for(int i=0; i<n-1; i++)
    for(int j=i+1; j<n; j++)
     if(a[i]>a[j]) {
           s=a[i];
           a[i]=a[j];
           a[j]=s;
          }
   }

// Головна функція
void main()
{
String s, s1("Hello"), s2("world");
s=s1+' '+s2; // Викликаються перевантажені операції + та =
// або s = operator + (operator + (s1, ' '), s2);

s=s+"!!!"; // Викликаються перевантажені операції + та =

// викликається перевантажена операція << виведення у потік
cout<< s <<endl;   // Hello world!!!

// Викликається перевантажена операція порівняння <
if(s1<s2) cout<<"s1<s2"<< endl;
else cout<<"s1>=s2"<< endl;
```

```
    const int n=5;
    String a[n] = {"6543",
                   "7842",
                   "3472",
                   "5219",
                   "3243"};

    sort(a,n);

// викликається перевантажена операція << виведення у потік
    for(int i=0;i<n;i++)
     cout<<a[i]<<endl;
    }
```

Порівняйте текст цієї програми з програмою підрозділу 3.9, реалізованою на основі методів класу. Замість методів класу у вищенаведеній програмі використано знаки операцій, що робить можливим записувати вирази над об'єктами класу у компактній формі. Наприклад, вираз `s1+s2` означає конкатенацію двох рядків.

Розділ 5
ВИДИ КЛАСІВ

Об'єктно-орієнтована програма здебільшого будується на основі декількох класів. Тому важливо визначити правила взаємовідношення класів та їх розміщення у програмі.

Класи можуть бути:

- глобальні, оголошення яких здійснюється поза функціями;
- локальні, які розміщуються всередині функцій;
- дружні, яким надаються права доступу до усіх частин класу;
- не вкладені, коли у протоколі класу немає оголошення інших класів;
- вкладені, коли у протоколі класу є оголошення інших класів;
- контейнери та колекції, які містять об'єкти інших класів;
- ітератори, призначені для забезпечення доступу до елементів інших класів;
- успадковані, коли похідний клас використовує оголошення даних та методів базового класу;
- шаблонні, які є узагальненим визначенням множини класів;
- абстрактні, що містять нереалізовані віртуальні методи;
- інтерфейсні, призначені для забезпечення однакового доступу до різних версій об'єктів (компонентів) відкомпільованого класу, незалежно від їх внутрішньої реалізації.

5.1. Глобальні невкладені класи

Глобальними називаються класи, розміщені поза оголошеннями функцій. Допускається використання імен вищерозміщених класів для оголошень даних у нижчерозміщених класах.

```
class A{
       int x;
 public:
       A(int x){this->x=x;}
       int Get(){return x;}
      };

class B{
       int y;
 public:
       B(int y){this->y=y;}
```

```
        int Get(){return y;}
        int Get(A a){return a.Get();}
    };

void main(){
A a(1);
B b(2);
cout<<a.Get()<<endl;            //1
cout<<b.Get()<<endl;            //2
cout<<b.Get(a)<<endl;           //1
}
```

Клас B може оголосити об'єкт, вказівник або посилання на вищерозміщений клас A та має доступ до елементів його public-частини (через об'єкт вище розміщеного класу). Для доступу до private або protected частин класу A з нижчерозміщеного класу B необхідно оголосити клас B дружнім до класу A.

Метод B::Get(A) демонструє принцип раннього зв'язування методів класу з його об'єктами. Звернення a.Get() у цьому методі забезпечує виклик методу A::Get() класу A, а не однойменного методу B::Get() класу B.

Вищерозміщений клас A може використати ім'я нижчерозміщеного класу за наявності випереджаючого оголошення класу B. Для цього перед класом A необхідно записати:

```
class B;
```

Тоді клас A може оголосити дані з типом вказівника або посилання на клас B.

Оголошення класу B дружнім до вищерозміщеного класу A дає можливість доступу з класу B до усіх частин класу A через об'єкт, розіменований вказівник чи посилання на клас A:

```
class A{
 friend class B;
        // ...
       };
```

Тоді, при збереженні структури попередньої програми, метод B::Get(A) матиме вигляд:

```
int B::Get(A a){return a.x;}
```

Оголошення класу B дружнім до класу A додатково відіграє роль випереджаючого оголошення. Це дозволяє використовувати ім'я нижчерозміщеного класу B у вищерозміщеному класі A. Детальніше друзі класу розглянуті у п. 2.9.

5.2. Контейнерні класи

5.2.1. Загальні відомості

Контейнерним називається клас, який містить дані з типом іншого класу. Забезпечує один зі способів повторного використання коду програми.

Контейнерні класи будуються на основі аналізу внутрішньої структури складного об'єкта або системи та виявленні їх складових елементів або підсистем. Під час аналізу виділяються функціонально повні або самодостатні елементи (підсистеми), які дають можливість побудувати класи з можливістю їх повторного, багатократного використання для побудови інших систем. Між контейнерними об'єктами діє відношення "містить": контейнерний клас містить об'єкти-представники інших класів. Контейнерний клас називають зовнішнім, а інкапсульовані ним об'єкти – внутрішніми.

Для прикладу розглянемо клас `point`, який описує координати точки (`x`, `y`) на площині. У відкритій частині клас містить конструктор ініціалізації `point(float, float)`, конструктор копіювання `point(const point&)` та метод `void output()` для виведення координат точки на екран:

```
class point{
// координати точки
float x, y;
public:
// конструктор ініціалізації
point(float x1, float y1)
{x=x1; y=y1;}
// конструктор копіювання
point(const point& p){x=p.x; y=p.y;}
// метод виведення
void output()
{cout<<x<<' '<<y<<endl;}
};
```

Клас `circle` містить дані, необхідні для побудови кола на площині. Він є контейнером для об'єкта класу `point`, який використовується для задання координат центра кола. Крім того, клас `circle` містить поле `radius` для визначення радіуса кола. У відкритій частині класу `circle` розміщено конструктор ініціалізації даних `circle(point&, float)` та метод їх виведення на екран:

```
// контейнерний клас
class circle{
// інкапсульований об'єкт з координатами центру кола
point center;
// радіус кола
```

```
float radius;
public:
// конструктор ініціалізації
circle(point& c, float r): center(c)
 {radius=r;}
// метод виведення
void output( )
 {center.output( ); cout<<radius<<endl; }
};
```

Якщо у класі внутрішнього об'єкта перекрито конструктор за замовчуванням (void-конструктор), то ініціалізація такого об'єкта повинна здійснюватися у списку ініціалізації конструктора контейнерного класу (після двокрапки). В іншому випадку ініціалізацію інкапсульованого об'єкта можна виконати у тілі конструктора контейнерного класу (у фігурних дужках).

Функція `main()` оголошує об'єкти класів `point`, `circle` та викликає метод `output()` для виведення вмісту об'єкта контейнерного класу:

```
void main()
{
// об'єкт класу point
point p(2, 5);
// об'єкт класу circle
circle c(p, 10);
// виклик методу контейнерного класу
c.output() ;
}
```

Контейнери переважно будуються на основі шаблонних класів (див. п. 10.2). Контейнер, на відміну від колекції, не допускає явного задання числа елементів і не підтримує розгалуженої структури. Найпростіші види контейнерів (статичні й динамічні масиви) вбудовані безпосередньо у мову C++. Крім того, стандартна бібліотека STL містить реалізації таких контейнерів, як вектор (`vector`), список (`list`), черга (`deque`), асоціативний масив (`map`), множина (`set`) та інших. Контейнер може містити колекцію об'єктів.

Колекція – програмний об'єкт, який складається з набору значень одного або різних типів і дає змогу звертатися до цих значень.

Колекція дозволяє записувати у себе та зчитувати значення елементів. Призначення колекції – слугувати сховищем об'єктів та надавати доступ до них. Як правило, колекції використовуються для зберігання груп однотипних об'єктів, що підлягають стереотипній обробці. Колекції відрізняються від контейнерів тим, що допускають розгалужену структуру та дозволяють додаткам визначати точну кількість елементів у колекції.

У контейнерах часто зустрічається реалізація алгоритмів роботи з інкапсульованими об'єктами.

Алгоритм – це функція для виконання операцій над об'єктами контейнера, наприклад, для їх сортування або пошуку. Кожен алгоритм для доступу до елементів контейнера використовує ітератори. Задача ітератора полягає у забезпеченні доступу до елементів даних класу для їх читання або встановлення. Ітератори забезпечують інтерфейс між контейнерами та алгоритмами.

5.2.2. Композиція та агрегування об'єктів

Розрізняють два основні способи побудови контейнерних класів, що ґрунтуються на *композиції* (безпосередньому включенні) та *агрегуванні* об'єктів.

У разі *композиції* зовнішній об'єкт містить внутрішні об'єкти. При знищенні зовнішнього об'єкта знищуються усі його внутрішні об'єкти.

Важливою є організація використання внутрішнього об'єкта зовнішнім об'єктом. У разі включення зовнішній об'єкт реалізує власні методи-обгортки для методів внутрішнього об'єкта. Такі методи-обгортки можуть розширювати функціональність внутрішнього об'єкта. Наприклад:

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

class A{
int x;
public:
A(int x=0){this->x=x;}
~A(){cout<<"~A()"<<endl;}
void print(){cout<<x<<endl;}
};

class B{
A a;
public:
B(A& x):a(x){}
~B(){cout<<"~B()"<<endl;}
void print(){a.print();}
};

void main()
{
A a(5);
B b(a);

b.print();  // 5

// викликаються ~B(), ~A() для об'єкта b
// викликається ~A() для об'єкта a
}
```

Клас B містить об'єкт класу A. Функція main() створює об'єкти обох класів. Разом зі знищенням об'єкта класу B знищується його внутрішній об'єкт класу A, що видно із виведення програми.

У разі *агрегування* зовнішній об'єкт містить лише вказівники або посилання на внутрішні об'єкти. При знищенні зовнішнього об'єкта агреговані ним об'єкти не знищуються.

Механізм агрегування ґрунтується на тому, що зовнішній об'єкт забезпечує безпосередні виклики методів внутрішнього об'єкта, наприклад, повертаючи вказівник на внутрішній об'єкт за допомогою перевантаженої операції operator->. Наприклад:

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

class A{
int x;
public:
A(int x=0){this->x=x;}
~A(){cout<<"~A()"<<endl;}
void print(){cout<<x<<endl;}
};

class B{
A* p;
public:
B(A* p){this->p=p;}
~B(){cout<<"~B()"<<endl;}
A* operator->(){return p;}
};

void main()
{
A *pa=new A(5);
B& pb=*new B(pa);

pb->print();        // 5

delete &pb;         // ~B()

if(pa) pa->print(); // 5
}
```

Функція main() створює об'єкти класів A та B в області динамічної пам'яті. Після знищення об'єкта класу B об'єкт класу A залишається неушкодженим, що видно із виведення програми.

У реальних програмних системах для роботи з включеними та агрегованими об'єктами використовуються інтерфейси – чисті абстрактні базові класи з набором віртуальних методів для забезпечення доступу до елементів класу.

5.2.3. Копіювання контейнерів

Під час роботи з контейнерами часто виникає необхідність у їх копіюванні. Розрізняють *глибоке* та *неглибоке* копіювання контейнерів.

При *глибокому* копіюванні здійснюють копіювання усіх внутрішніх об'єктів контейнера, навіть тоді, коли вони задаються своїми вказівниками. Наприклад:

```
// клас внутрішнього об'єкта
class A{
 int x; // поле даних
public:
// конструктор ініціалізації
 A(int x1){cout<<"A(int)"<<endl; x=x1;}
// конструктор копіювання
 A(const A& a){cout<<"A(const A&)"<<endl; x=a.x;}
// деструктор
 ~A(){cout<<"~A()"<<endl;}
// метод виведення елемента даних
 void output(){cout<<x<<endl;}
};
// контейнерний клас
class B{
 A *p; // вказівник на внутрішній об'єкт
public:
// конструктор ініціалізації
 B(A* p1){cout<<"B(A*)"<<endl; p=new A(*p1);}
// конструктор копіювання
 B(const B& b){cout<<"B(const B&)"<<endl; p=new A(*b.p);}
// деструктор
 ~B(){cout<<"~B()"<<endl; delete p;}
// метод виведення даних
 void output(){p->output();}
};

void main()
{
 A a(5);           // A(int)
 B b1(&a);         // B(A*), A(const A&)
 b1.output();      // 5
 B b2(b1);         // B(const B&), A(const A&)
 b2.output();      // 5
// ~B(), ~A() для об'єкта b2
```

```
// ~B(), ~A() для об'єкта b1
// ~A()для об'єкта a
}
```

Коментарями у функції `main()` позначено виведення, пов'язані з викликами конструкторів та деструкторів об'єктів.

Конструктори контейнерного класу виділяють динамічну пам'ять для внутрішнього об'єкта та копіюють його.

При глибокому копіюванні об'єкта контейнерного класу `B` викликається конструктор копіювання його внутрішнього об'єкта класу `A`. Результатом цього є повне копіювання контейнерного об'єкта.

При *неглибокому* копіюванні копіюють тільки значення вказівників на інкапсульовані в контейнер об'єкти. Наприклад:

```
// клас внутрішнього об'єкта
class A{
 int x; // поле даних
public:
// конструктор ініціалізації
 A(int x1){cout<<"A(int)"<<endl; x=x1;}
// конструктор копіювання
 A(const A& a){cout<<"A(const A&)"<<endl; x=a.x;}
// деструктор
 ~A(){cout<<"~A()"<<endl;}
// метод виведення поля даних
 void output(){cout<<x<<endl;}
};
// контейнерний клас
class B{
 A *p; // вказівник на внутрішній об'єкт
public:
// конструктор ініціалізації
 B(A* p1){cout<<"B(A*)"<<endl; p=p1;}
// конструктор копіювання
 B(const B& b){cout<<"B(const B&)"<<endl; p=b.p;}
// деструктор
 ~B(){cout<<"~B()"<<endl; delete p;}
// метод виведення даних
 void output(){p->output();}
};

void main()
{
 A a(5);          // A(int)
 B b1(&a);        // B(A*)
 b1.output();     // 5
 B b2(b1);        // B(const B&)
 b2.output();     // 5
```

```
// ~B(), ~A() для об'єкта b2
// ~B(), ~A() для об'єкта b1
// ~A()для об'єкта a
}
```

Конструктори контейного класу `B` здійснюють порозрядне копіювання полів даних, зокрема вказівників на внутрішні об'єкти. Динамічна пам'ять для внутрішнього об'єкта класу `A` не виділяється, його копія не створюється, що видно із виведення програми.

5.3. Ітератори

Ітератор – це об'єкт класу, призначений для перерахування, доповнення, видалення та виконання інших операцій над об'єктом іншого, можливо, контейнерного класу.

Ітератор є абстракцією вказівника. Ітератор є об'єктом, що може посилатися на інші об'єкти, інкапсульовані в контейнері. Основні функції ітератора – забезпечення доступу до об'єкта, на який він посилається (розіменування), і перехід від одного елемента контейнера до іншого (ітерація). Для вбудованих контейнерів (наприклад, масивів) у ролі ітераторів використовуються звичайні вказівники. У загальному випадку ітератори реалізуються у вигляді класів з набором перевантажених операторів.

Ітератори реалізують операції, подібні до операцій над вказівниками: розіменування, інкремент або декремент, додавання зміщення, віднімання ітераторів, порівняння. Ітератори дають змогу працювати з об'єктами в режимі "тільки читання", або "тільки запис", або "читання і запис".

Незалежно від фактичної організації контейнера (вектор, список, дерево) його елементи можна розглядати як послідовності, задані ітераторами початку та кінця. Як правило, метод контейнерного класу `begin()` повертає ітератор першого елемента послідовності, метод `end()` – останнього елемента. Крім звичайних ітераторів, контейнери оперують зворотними ітераторами, які переглядають послідовність елементів у контейнері у зворотному порядку. Це дає змогу застосовувати алгоритми як у прямій, так і у зворотній послідовності елементів. Наприклад, за допомогою функції `find()` можна шукати елементи як з початку, так і з кінця контейнера.

Ітератор може бути реалізований як зовнішній клас до контейнерного класу або бути вкладеним у контейнерний клас.

Приклад програми використання зовнішнього ітератора для роботи з контейнерним класом:

```
#include <iostream>

using namespace std;

#define T int

// Ітераторний клас
class iter
  {
protected:
// елемент даних
  T* i;
public:
// конструктор ініціалізації
  explicit iter(T* i1 = 0) : i(i1) { }
// конструктор копіювання
  iter(const iter& x) : i(x.i) { }
// перевантажені операції
  iter& operator=(const iter& x)
       { i = x.i; return *this; }
  T& operator*( ) const { return *i; }
  iter& operator++( ) { ++i; return *this; }
  iter& operator--( ) { --i; return *this; }
  iter& operator++(int) { i++; return *this; }
  iter& operator--(int) { i--; return *this; }
  iter operator+(int n) { return iter(i+n); }
  iter operator-(int n) { return iter(i-n); }
  iter& operator+=(int n) { i += n; return *this; }
  iter& operator-=(int n) { i -= n; return *this; }
  bool operator==(const iter& x) const
       { return i == x.i; }
  bool operator!=(const iter& x) const
       { return i != x.i; }
  bool operator<(const iter& x) const
       { return i < x.i; }
  bool operator>(const iter& x) const
       { return i > x.i; }
  bool operator<=(const iter& x) const
       { return i <= x.i; }
  bool operator>=(const iter& x) const
       { return i >= x.i; }
  friend int operator-(const iter& x, const iter& y)
       {return x.i - y.i;}
};

// Контейнерний клас
class A
{
private:
```

```
 T* p;        // вказівник на масив елементів даних
 int size;    // кількість елементів

public:
// конструктор ініціалізації
 A(int n)
 {p=new T[size=n];
  for(iter i=begin(); i!=end(); ++i) *i = T(i-begin());
 }
// конструктор копіювання
 A(const A& a)
 {p=new T[size=a.size];
  for(iter i=begin(); i!=end(); ++i) *i=a.p[i-begin()];
 }
// деструктор
~A(){delete []p;}
// перевантажена операція присвоєння
 A& operator=(const A& a)
 {
  if(this!=&a)
  {
   delete []p;
   p=new T[size=a.size];
   for(iter i=begin(); i!=end(); ++i) *i=a.p[i-begin()];
  }
 return *this;
 }
// метод визначення значеннння ітератора на початок масиву
 iter begin() {return iter(p);}
// метод визначення значеннння ітератора на кінець масиву
 iter end() {return iter(p+size);}

// Тестування ітераторів
    void  post_increment_test();
    void  post_decrement_test();
    void  offset_addition_test(const int);
    void  offset_subtraction_test(const int);
    void  comparison_test(iter, iter);

};


// Постфіксний інкремент ітератора
void A::post_increment_test()
{
   cout << "\nПостфіксний інкремент ітератора" << endl;

    for ( iter i = begin() ; i != end() ; i++)
    {
```

```
        cout << *i << ' ';
    }
cout<<endl;
}

// Постфіксний декремент ітератора
void A::post_decrement_test()
{
   cout << "\nПостфіксний декремент ітератора" << endl;

    for ( iter i = end() ; i != begin() ; )
    {
        i--;
        cout << *i << ' ';
    }
cout<<endl;
}

// Додавання зміщення до ітератора
void A::offset_addition_test(int const k)
{
    cout << "\nДодавання зміщення до ітератора" << endl;

    for ( iter i = begin() ; i != end() ; i += k )
    {
        cout << *i << ' ';
    }
cout<<endl;
}

// Віднімання зміщення від ітератора
void A::offset_subtraction_test(int const k)
{
    cout << "\nВіднімання зміщення від ітератора" << endl;

    for ( iter i = end()-1 ; i>=begin() ; i-=k)
    {
      cout << *i << ' ';
    }
cout << endl;
}

// Порівняння ітераторів
void A::comparison_test(iter i, iter j)
{
 cout << "\nПорівняння ітераторів" << endl;
 T t = *i - *j;
 if(t < 0) cout << *i << '<' << *j << endl;
 if(t == 0) cout << *i << '==' << *j << endl;
```

```
 if(t > 0) cout << *i << '>' << *j << endl;
}

// Головна функція
int main( )
{
    A a(10);
    a.post_increment_test();
    a.post_decrement_test( );
    a.offset_addition_test(2);
    a.offset_subtraction_test(2);
    a.comparison_test(a.begin( )+1,a.begin( )+3);

 return 0;
}
```

Для вправляння побудуйте ітераторний клас для роботи з контейнерами, що містять список або бінарне дерево елементів. Інкрементоване значення ітератора повинно вказувати на наступний елемент списку або дерева. Адреса наступного елемента списку зберігається у структурі його поточного елемента. Для переходу до наступного елемента дерева необхідно реалізувати алгоритм його обходу.

5.4. Локальні класи

Локальні класи оголошуються всередині функцій. Локальні класи мають такі особливості:

- не можуть використовувати `auto`-змінних цих функцій;
- можуть використовувати `static`-змінні, оголошені в цих функціях;
- не можуть мати `static`-елементів, які є членами класу;
- визначення (операторні частини) методів локального класу повинні бути розміщені всередині класу.

Приклад локального класу:

```
void func(int x)
{
class Local  // локальний клас
{
int y;
public:
Local (int z){y=z;}
void print() {cout<<y<<endl;}
};           // кінець оголошення локального класу
Local a(x);  // об'єкт локального класу
```

```
a.print();    // виклик методу локального класу
}

void main()
{func(5);}
```

Глобальні або локальні класи можуть бути вкладеними.

5.5. Вкладені класи

Один клас можна оголосити всередині іншого класу. Такий клас називається вкладеним. Область досяжності вкладеного класу обмежується охоплюючим класом. Глибина вкладеності класів не обмежується.

Вкладені класи можуть бути розміщені в будь-якій частині охоплюючого класу. Всередині вкладених класів можна оголошувати дані з типом посилання або вказівника на охоплюючий клас. Після оголошення вкладеного класу в охоплюючому класі можна оголосити об'єкт, посилання або вказівник на вкладений клас.

Приклад вкладених класів:

```
class A
{
int x;

public:
A(int x1){x=x1;}
void print()
  {
  cout<<"A::print()"<<endl;
  cout<<x<<endl;
  }

class B
{
A& a;
int y;
public:
B(A& a1, int y1):a(a1){y=y1;}
void print()
  {
  cout<<"A::B::print()"<<endl;
  a.print();
  cout<<y<<endl;
  }
```

```
class C
{
B& b;
int z;

public:
C(B& b1, int z1):b(b1){z=z1;}

void print()
  {
  cout<<"A::B::C::print()"<<endl;
  b.print();
  cout<<z<<endl;
  }

};
};
};

void main()
{
  A a(1);
  a.print();       // 1
  A::B b(a,2);
  b.print();       // 1 2
  A::B::C c(b,3);
  c.print();       // 1 2 3
 }
```

При зовнішньому визначенні методів та при створенні об'єктів вкладених класів необхідно вказати повний шлях вкладеності, який складається з імен класів, об'єднаних операцією `::` .

Вкладений клас інкапсулює власні дані та методи, але має повний доступ до елементів охоплюючого класу (за допомогою об'єкта, посилання або вказівника на об'єкт). Наприклад, метод виведення класу `A::B::C` можна реалізувати так:

```
void A::B::C::print()
  {
   cout<<b.a.x<<' '<<b.y<<' '<<z<<endl;
  }
```

Можливість використання вкладених класів, розміщених у `private` або `protected` частинах охоплюючого класу, регулюється загальними правилами доступу.

Внутрішній клас можна оголосити дружнім до зовнішнього класу і, навпаки, зовнішній – дружнім до внутрішнього.

Для вправляння побудуйте ітераторний клас, вкладений у контейнерний клас, що містить список об'єктів в області динамічної пам'яті.

5.6. Приклад програми

Розробити клас Storage для збереження інформації про вироби на складі підприємства. Клас Storage є контейнером для масиву об'єктів класу Product. Клас Product містить дані про дату надходження виробів на склад, код, назву, кількість виробів та вартість одного виробу. Клас Product є контейнером для класу Date. Розробити та використати клас-ітератор для звернення до елементів контейнерного класу Storage. Знайти сумарну вартість виробів заданого коду як суму добутків кількості виробів на ціну одного виробу.

```
#include <iostream>
#include <string>

using namespace std;

// клас дати
class Date{
 int day;    // день
 int month;  // місяць
 int year;   // рік
public:

// конструктор
 Date(int day1, int month1, int year1)
 {day = day1;
  month = month1;
  year = year1;
 }

// перевантажена операція введення з потоку
 friend istream& operator>>(istream& is, Date& d)
 {
   is>>d.day>>d.month>>d.year;
   return is;
 }

// перевантажена операція виведення у потік
 friend ostream& operator<<(ostream& os, Date& d)
 {
   os<<d.day<<' '<<d.month<<' '<<d.year<<endl;
   return os;
 }
};

// клас з інформацією про продукцію підприємства
class Product{
```

```
 Date d;            // дата
 string code;       // код виробу
 string name;       // назва виробу
 int count;         // кількість
 double price;      // ціна одного виробу
public:
// конструктор
 Product(Date d1=Date(1, 3, 2012),
         string code1="", string name1="",
         int count1=0, double price1=0):d(d1)
 {code = code1;
  name = name1;
  count = count1;
  price = price1;
 }

// метод читання дати
 Date& getDate() {return d;}

// метод читання коду
 string& getCode(){return code;}

// метод читання назви
 string& getname(){return name;}

// метод читання кількості
 int& getCount(){return count;}

// метод читання ціни
 double& getPrice(){return price;}

// перевантажена операція введення з потоку
 friend istream& operator>>(istream& is, Product& t)
 {
   is>>t.d>>t.code>>t.name>>t.count>>t.price;
   return is;
 }

// перевантажена операція виведення у потік
 friend ostream& operator<<(ostream& os, Product& t)
 {
   os<<t.d<<' '<<t.code<<' '<<t.name<<' '
<<t.count<<' '<<t.price<<endl;
   return os;
 }
};

#define T  Product
```

```
// Ітераторний клас
class iter
  {
protected:
  T* i;       // вказівник на елемент даних
public:

// конструктор
  iter(T* i1) : i(i1) { }

// конструктор копіювання
  iter(const iter& x) : i(x.i) { }

// перевантажена операція присвоєння ітераторів
  iter& operator=(const iter& x)
 { i = x.i; return *this; }

// перевантажена операція розіменування ітератора
  T& operator*( ) const { return *i; }

// перевантажена операція префіксного інкременту ітератора
  iter& operator++( ) { ++i; return *this; }

// перевантажена операція "не дорівнює"
// для порівняння ітераторів
  bool operator!=(const iter& x) const
{ return i != x.i; }

// перевантажена операція віднімання ітераторів
  friend int operator-(const iter& x, const iter& y)
{return x.i - y.i;}
};

// клас складу виробів
class Storage{
 T *p;        // вказівник на масив виробів
 int size;    // кількість елементів масиву
public:

// конструктор
 Storage(int n=1)
 {p=new T[size=n];
  for(iter i=begin(); i!=end(); ++i) *i = T();
 }

// конструктор копіювання
 Storage(const Storage& a)
 {p=new T[size=a.size];
  for(iter i=begin(); i!=end(); ++i) *i=a.p[i-begin()];
 }
```

```
// деструктор
 ~Storage() {delete []p;}

// перевантажена операція присвоєння об'єктів
 Storage& operator=(const Storage& a)
 {
  if(this!=&a)
  {
   delete []p;
   p=new T[size=a.size];
   for(iter i=begin(); i!=end(); ++i) *i=a.p[i-begin()];
  }
 return *this;
 }

// перевантажена операція введення з потоку
 friend istream& operator>>(istream& is, Storage& s)
 {
   for(iter i=s.begin(); i!=s.end(); ++i) is >> *i;
   return is;
 }

// перевантажена операція виведення у потік
 friend ostream& operator<<(ostream& os, Storage& s)
 {
   for(iter i=s.begin(); i!=s.end(); ++i)
os << *i << ' ';
       os<<endl;
   return os;
 }

// метод обчислення вартості виробів заданого коду
 double cost(string code){
   double v=0;
   for(iter i=begin(); i!=end(); ++i)
         if((*i).getCode() == code)
               v+=(*i).getCount()*(*i).getPrice();
   return v;
 }

// метод визначення значення ітератора перед масивом
 iter begin() {return iter(p);}

// метод визначення значення ітератора поза масивом
 iter end() {return iter(p+size);}
};

// головна функція
void main()
{
```

```
int n;
cout<<"Введіть кількість виробів: ";
cin>>n;
Storage s(n);
string code;
cout<<"Введіть дані про продукцію на складі:"<<endl;
cout<<"день місяць рік код назва кількість ціна"<<endl;
cin>>s;
cout<<"Введіть код виробу: ";
cin>>code;

cout<<"Вартість виробів з кодом "<<code<<
         " дорівнює "<<s.cost(code)<<endl;
}
```

Розділ 6
ОДИНАРНЕ УСПАДКУВАННЯ КЛАСІВ

6.1. Загальні правила успадкування класів

6.1.1. Суть успадкування класів

Успадкування класів та структур є одним із способів повторного використання коду програми. Об'єднання не допускають успадкування.

Успадкування полягає у використанні *оголошень* елементів базового класу у структурі іншого класу, який називається похідним від базового. У результаті успадкування об'єкт похідного класу набуває властивостей базового класу.

Похідний клас може користуватися полями даних та методами, успадкованими від базового класу без їх повторного оголошення. Крім того, похідний клас може оголосити власні поля даних та методи. Оголошення елементів похідного класу перекривають однойменні оголошення елементів базового класу.

Поля даних базового класу не копіюються у похідний клас. На машинному рівні для об'єкта похідного класу буде виділено окрему від базового класу область пам'яті під успадковані поля.

Конструктори, деструктори, операторна функція присвоєння та друзі класу не успадковуються.

Успадкування ґрунтується на родо-видовій подібності об'єктів. Виявляються спільні характеристики та властивості групи об'єктів, на основі *узагальнення* яких утворюється базовий клас (або суперклас). Похідні класи будуються на основі імпорту оголошень елементів базових класів та їх доповнення індивідуальними особливостями об'єктів.

Між об'єктами-представниками базового та похідного класу діє відношення "є": похідний клас є нащадком базового класу, а базовий клас є його предком. Похідний клас узагальнює властивості базового класу. В інформаційному плані нащадок є багатшим від предка. Механізм успадкування класів призводить до значного скорочення вихідних кодів програми і дає можливість будувати великі, відкриті для вдосконалення програмні системи.

6.1.2. Оголошення успадкування класів

Для успадкування базового класу необхідно після імені похідного класу через символ ' : ' вказати режим успадкування та ім'я базового класу:

```
class Base
{...};

class Derived: режим_успадкування  Base
{...};
```

Класи допускають множинне успадкування. Тоді список базових класів задається через кому після символу ':'.

Режим успадкування визначає спосіб успадкування елементів базового класу. Можливі такі значення режимів успадкування: private, protected, public. Режим успадкування визначає, в які частини похідного класу потрапляють елементи базового класу.

При private-успадкуванні елементи всіх частин базового класу потрапляють у private-частину похідного класу.

При protected-успадкуванні private-частина базового класу потрапляє у private-частину похідного класу, а protected- та public-частини базового класу потрапляють у protected-частину похідного класу.

При public-успадкуванні private, protected та public-частини базового класу потрапляють в однойменні частини похідного класу.

Для будь-якого варіанта успадковуються всі елементи базового класу у похідний клас, але private-частина базового класу безпосередньо недоступна у похідному класі. Схеми успадкування класів наведено у табл. 6.1.

Таблиця 6.1

Дія режимів успадкування класів

Режим успадкування	**Рівні захисту класу Base**	**Рівні захисту класу Derived**
private	private	private*
	protected	private
	public	private
protected	private	private*
	protected	protected
	public	protected
public	private	private*
	protected	protected
	public	public

У табл. 6.1 символ '*' позначає private-частину базового класу, безпосередньо не доступну у похідному класі.

За замовчуванням для класів діє режим успадкування private, а для структур – режим public.

Для ініціалізації успадкованих полів викликається конструктор базового класу у списку ініціалізації конструктора похідного класу (після символу ':').

6.1.3. Приклад успадкування класів

Нехай клас `circle` успадковує базовий клас `point`, який у захищеній частині містить координати точки на площині `x` та `y`. У відкритій частині класу `point` розміщено конструктор `point(float, float)` для ініціалізації полів `x`, `y`, конструктор копіювання `point(const point&)` та метод `output()` для виведення даних на екран:

```
// базовий клас
class point{
protected:
// координати точки
float x, y;
public:
// конструктор ініціалізації
point(float x1, float y1)
{x=x1; y=y1;}
// конструктор копіювання
point(const point& p){x=p.x; y=p.y;}
// метод виведення
void output()
{cout<<x<<' '<<y<<endl;}
};
```

Клас `circle` успадковує клас `point` у режимі `public`. Крім того, у класі `circle` оголошено власне поле даних `radius` та метод `output()`. У результаті в класі `circle` буде визначено три поля даних `x`, `y`, `radius` та два методи `output()`.

Конструктори класу `circle` ініціалізують поля даних. Успадковані дані ініціалізуються викликом конструктора базового класу `point` у списку ініціалізації конструктора похідного класу `circle`, наприклад:

```
// похідний клас
class circle: public point
{
// радіус кола
float radius;
public:
// конструктор ініціалізації
circle(float x, float y, float r):
       point(x, y) // виклик конструктора базового класу
       {radius=r;}
// перевантажений конструктор ініціалізації
```

```
circle(point& c1, float r):
       point(c1)   // виклик конструктора копіювання
                   // базового класу
       {radius=r;}
// конструктор копіювання
 circle(const circle& c):
       point(c.x, c.y)   // виклик конструктора
                         // базового класу
       {radius = c.radius;}
// метод виведення
void output( )
 {point::output( ); cout<<radius<<endl; }
};
```

Метод `output()` класу `circle` перекриває метод `output()`, успадкований з класу `point`. За правилами об'єктно-орієнтованого програмування звертатися до успадкованих полів у похідному класі необхідно за допомогою успадкованих методів базового класу. Тоді для виклику перекритого методу необхідно використати назву базового класу та операцію дозволу доступу `point::output()`.

У функції `main()` створюються об'єкти базового і похідного класу та викликається метод виведення даних об'єктів класу `circle`.

```
void main()
{
// об'єкт похідного класу
circle c1(2, 5, 10);
// виклик методу похідного класу
c1.output();

// об'єкт базового класу
point p(2, 5);
// об'єкт похідного класу
circle c2(p, 10);
// виклик методу похідного класу
c2.output();
}
```

Порівняйте організацію програми на основі успадкування класу з відповідним прикладом програми з п. 5.2.1, побудованої на основі контейнерного класу.

6.1.4. Зміна режимів успадкування

У похідному класі існує можливість зміни застосованого режиму успадкування елементів базового класу. Успадковані елементи можна перевести тільки на той самий рівень, на якому вони були розміщені в базовому класі.

Для цього в області дії цього рівня у похідному класі необхідно виконати оголошення:

```
ім'я_базового_класу::ім'я_елемента;
```

Після такого оголошення елемент (поле даних або метод), успадкований від базового класу, переводиться з рівня, що визначався режимом успадкування, на рівень, в межах дії якого виконано таке оголошення.

Якщо у протоколі базового класу було оголошено ряд перевантажених методів, то всі вони матимуть змінений рівень захисту. При зміні рівнів захисту перевантажених методів необхідно, щоб всі вони були розміщені в області дії тільки одного з рівнів захисту базового класу.

Наприклад:

```
class A
{
private:
int x;
void f1(){cout<<"A::f1()"<<endl;}

protected:
int y;
void f2(){cout<<"A::f2()"<<endl;}

public:

int z;
void f3(){cout<<"A::f3()"<<endl;}

A(int x=0,int y=0, int z=0)
{this->x=x; this->y=y; this->z=z;}

void print()
{ cout<<"A::print()"<<endl;
cout<<x<<' '<<y<<' '<<z<<endl;
}
};

class B: private A
{
protected:
int v;
A::y; // будуть protected, а не private
A::f2;
public:
A::z;  // будуть public, а не private
A::f3;
B(int x=0,int y=0,int z=0,int v=0):A(x,y,z)
```

```
 {this->v=v;}

void print()
{ cout<<"B::print()"<<endl;
  cout<<y<<' '<<z<<' '<<v<<endl;
}
};

void main()
{
B b(1,2,3,4);
b.print(); // 2  3  4
// b.A::print(); // не досяжно при private-успадкуванні
}
```

6.1.5. Перекриття елементів класу

Елементи похідного класу перекривають дію однойменних елементів базового класу. Якщо у будь-якій частині похідного класу містяться елементи, імена яких збігаються з успадкованими елементами базового класу, то відбувається перекриття таких успадкованих елементів.

Поля даних похідного класу перекривають однойменні дані базового класу. Для доступу до перекритих полів даних базового класу у похідному класі використовується конструкція:

```
ім'я_базового_класу::ім'я_поля;
```

Методи похідного класу перекривають однойменні методи базового класу. Перекриття методів реалізується через їх перевизначення (`override`) або перевантаження (`overload`).

Перевизначені функції мають однакове ім'я, однакову кількість та типи параметрів, однаковий тип результату. Перевизначення допускається між успадкованими класами, але не допускається у межах одного класу.

Для доступу до перекритих методів використовується ім'я класу, де визначений метод, та операція дозволу доступу:

```
ім'я_базового_класу::ім'я_методу(фактичні аргументи);
```

Перевантажені функції мають однакове ім'я, можуть мати однаковий або різний тип результату, але обов'язково відрізняються кількістю або типами своїх параметрів. Перевантаження допускається в межах одного класу та між успадкованими класами. Вибір конкретного перевантаженого методу визначається списком фактичних аргументів.

Приклад перевизначення та перевантаження методів успадкованих класів.

```
class A
{
protected:
int x;
public:
A(int x=0){this->x=x;}
void set(int x)
 {
 cout<<"A::set(int)"<<endl;
 this->x=x;
 }
};

class B:public A
{
int y;
public:
B(int x=0, int y=0):A(x){this->y=y;}
void set(int y)
      {
      cout<<"B::set(int)"<<endl;
      this->y=y;
      }
void set(int x, int y)
      {
      cout<<"B::set(int,int)"<<endl;
      this->x=x; this->y=y;
      }
void print()
  {
  cout<<x<<' '<<y<<endl;
  }
};

void main()
{
  B b(1,2);       b.print();  // 1 2
  b.A::set(3);    b.print();  // 3 2
  b.set(4);       b.print();  // 3 4
  b.set(5,6);     b.print();  // 5 6
}
```

У наведеному прикладі програми метод `set(int)` класу `B` перевизначає успадкований від класу `A` метод `set(int)`. Крім того, метод `set(int,int)` класу `B` перевантажує метод `set(int)` цього самого класу.

Для звернення до перекритого методу класу `A` із функції `main()` за допомогою об'єкта `b` використовується конструкція `b.A::set(3)`. Звернення `b.set(4)` забезпечує виклик методу класу `B`, а `b.set(5,6)` – перевантаженого методу класу `B`.

У загальному випадку метод похідного класу перекриває всі однойменні перевантажені методи базового класу, наприклад:

```
class A{
 public:
 void f(){}
 void f(int){}
       };

class B:public A
 {
 public:
 void f(int,int){};
 };

void main()
{
B b;
//b.f();      // помилка
//b.f(1);     // помилка
b.A::f();
b.A::f(1);
b.f(2,3);
}
```

6.1.6. Пов'язування методів з даними класу

Дані та методи одного класу є статично пов'язані між собою. Такі зв'язки зберігаються також у разі успадкування класів. Статичне пов'язування методів з даними класу може бути проілюстровано таким прикладом:

```
class A {
 protected:
 int x;
 public:
 A(int x){this->x=x;}
 int Get(){return x;};
 };

class B: public A
 {
 int x;
 public:
 B(int x,int y):A(x){this->x=y;}
 };

void main()
{
```

```
B b(1,2);
cout<<b.Get()<<endl;      // 1
}
```

У цій програмі клас B перекриває поле даних x та успадковує метод Get() класу A для читання поля x. У результаті клас B матиме два однойменні поля A::x та B::x, а також метод A::Get(). Метод A::Get() є жорстко "прив'язаним" до поля A::x класу A. При виклику цього методу за допомогою об'єкта класу B він виводить значення поля A::x, а не B::x.

6.2. Особливість успадкування закритої частини базового класу

Хоча private-частина базового класу успадковується похідним класом, але її елементи безпосередньо не доступні для використання у похідному класі. За необхідності такий доступ може бути реалізований одним із способів:

1) через методи базового класу, розміщені в protected або public-частинах базового класу;
2) оголошенням похідного класу другом до базового класу.

Обидва ці варіанти можуть бути дозволені лише базовим класом.

6.2.1. Доступ до private-частини базового класу за допомогою успадкованого public-методу

Нехай у закритій частині класу A оголошено елемент даних x, а у відкритій частині – конструктор та метод доступу GetX(). Похідний клас B успадковує поле x та метод GetX() у режимі public. Наприклад:

```
class A{
 int x;
 public:
 A(int x){this->x=x;}
 int GetX(void){return x;}
       };

class B:public A
       {
 int y;
 public:
 B(int x, int y):A(x)
  {this->y=y;}
 void print(void)
```

```
  {cout<<GetX()<<' '<<y<<endl;}
       };

void main()
{
B b(1,2);
  b.print(); // 1 2
}
```

Виведення на екран закритого поля x забезпечується методом `print()` класу B, який для цього викликає відкритий метод `GetX()`, успадкований від класу A.

6.2.2. Доступ до private-частини базового класу за допомогою дружнього похідного класу

Якщо клас-нащадок є дружнім до базового класу, то він має доступ до елементів усіх частин класу за їх іменами, а не за допомогою об'єкта, як це забезпечується для дружніх зовнішніх класів та функцій. Наприклад:

```
class A{
 int x;
 public:
 A(int x){this->x=x;}
 friend class B;
       };

class B:public A
       {
 int y;
 public:
 B(int x, int y):A(x)
  {this->y=y;}
 void print(void)
  {cout<<x<<' '<<y<<endl;}
       };

void main()
{
B b(1,2);
  b.print(); // 1 2
}
```

Метод `B::print()` звертається до успадкованого з `private`-частини базового класу A поля x безпосередньо, використовуючи відповідне ім'я елемента даних.

6.3. Порядок виклику конструкторів та деструкторів при успадкуванні класів

Основна задача конструктора полягає в ініціалізації полів-даних об'єктів класу. За необхідності у конструкторі можна виділяти динамічну пам'ять для даних об'єкта. Якщо клас містить оголошення віртуальних функцій, то конструктор додатково виконує ініціалізацію вказівника на таблицю віртуальних методів.

При успадкуванні класів діє такий порядок виклику конструкторів:

1) викликається конструктор базового класу; якщо явний виклик конструктора базового класу не здійснюється у списку ініціалізації конструктора похідного класу (після двокрапки), то викликається конструктор базового класу за замовчуванням (без параметрів);
2) перед викликом конструктора контейнерного класу (базового або похідного) викликаються конструктори його полів-об'єктів в порядку їх запису у протоколі відповідного класу; якщо ініціалізація об'єктів не здійснюється у списку ініціалізації конструктора контейнерного класу, то викликаються конструктори за замовчуванням;
3) викликається конструктор похідного класу.

Деструктори викликаються при виході об'єктів із області досяжності програми або при знищенні об'єктів у динамічній пам'яті. Деструктори успадкованих класів викликаються у зворотному порядку – від похідного до базового класу. Наприклад:

```
class A{
protected:
int x;
 public:
 A(int x=0){cout<<"A(int)"<<endl; this->x=x;}
 ~A(){cout<<"~A()"<<endl;}
             };

class B: public A
    {
protected:
int y;
 public:
B(int x=0, int y=0):A(x)
{cout<<"B(int,int)"<<endl;
 this->y=y;}
 ~B(){cout<<"~B()"<<endl;}
 };

class C: public B
    {
```

```
protected:
 int z;
 public:
C(int x=0, int y=0, int z=0):B(x,y)
{cout<<"C(int,int,int)"<<endl; this->z=z;}
 ~C(){cout<<"~C()"<<endl;}
 };

void main()
{
C c(1,2,3);
}
```

Створення одного об'єкта похідного класу C призведе до такого виведення:

```
A(int)
B(int,int)
C(int,int,int)
~C()
~B()
~A()
```

6.4. Успадкування статичних елементів класу

6.4.1. Успадкування статичних даних

За відсутності перекриття об'єкти класів-нащадків мають спільні статичні поля даних, успадковані від предків. Нехай клас A має статичне поле x, а клас B є похідним від A. Наприклад:

```
class A {
 public:
 static int x;
 };

int A::x=5;

class B: public A
 {};

void main()
{
A a;
B b;
B::x=7;
```

```
cout<<A.x<<' '<<a.x<<' '<< b.A::x <<endl;
// 7   7   7
cout<<B::x<<' '<<b.x <<endl;
// 7   7
}
```

Об'єкти `a` класу `A` та `b` класу `B` мають спільне статичне поле `x`, що демонструється виведеннями програми та на рис. 6.1.

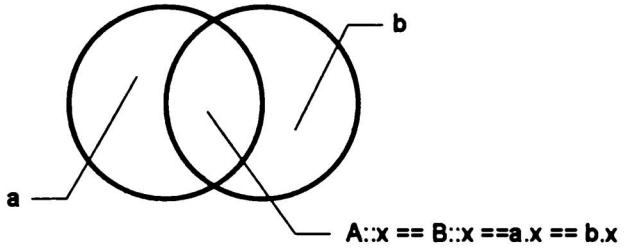

Рис. 6.1. Перетин об'єктів по статичних полях при успадкуванні класів

Якщо у похідному класі визначити однойменне статичне поле, то результат роботи попередньої програми зміниться.

```
class B: public A
 {
 public:
 static int x;
 };
int B::x;
```

Тепер об'єкт класу `B` матиме спільне з класом `A` статичне поле `A::x` та індивідуальне статичне поле `B::x`, що демонструє рис. 6.2.

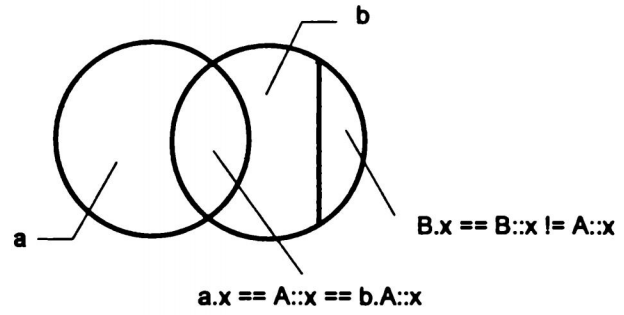

Рис. 6.2. Перекриття статичних полів при успадкуванні класів

За допомогою статичних полів даних можна передавати інформацію об'єктам в ієрархії успадкування класів.

6.4.2. Успадкування статичних методів

Статичні методи призначені для роботи зі статичними даними класу. Успадкування статичних методів відбувається за загальними правилами і визначається режимами успадкування.

За відсутності перекриття статичні методи базового класу будуть доступними для виклику у похідному класі.

```
class A {
 protected:
 static int x;
 public:
 static int Get(){return x;}
 };

int A::x=1;

class B: public A
 {
 static int x;
 public:
 };

int B::x=2;

void main()
{
A a;
B b;
cout<<A::Get()<<' '<<a.Get()<<' '<<b.A::Get()<<endl;
// 1   1   1
cout<<B::Get()<<' '<<b.Get()<<endl;
// 1   1
}
```

Якщо статичний метод похідного класу перекриває статичний метод базового класу, то у похідному класі буде два різні статичні методи.

```
class A {
 protected:
 static int x;
 public:
 static int Get(){return x;}
 };

int A::x=1;

class B: public A
 {
```

```
 static int x;
 public:
 static int Get(){return x;}
 };

int B::x=2;

void main()
{ A a;
B b;
cout<<A::Get()<<' '<<a.Get()<<' '<<b.A::Get()<<endl;
// 1   1   1
cout<<B::Get()<<' '<<b.Get()<<endl;
// 2   2
}
```

6.5. Успадкування константних елементів класу

6.5.1. Успадкування константних даних

Константні елементи класу успадковуються як звичайні елементи з тією відмінністю, що ініціалізація константних елементів класу здійснюється у списку ініціалізації конструктора. Допускається перекриття константних полів. Тоді у похідному класі будуть константні поля, успадковані від базового класу, та власні константні поля, що демонструє такий приклад:

```
class A {
 public:
 const int x;
 A(int x1):x(x1){}
 };

class B: public A
 {
    public:
 const int x;
 B(int x, int y):A(x), x(y){}
 };

void main()
{
A a(1);
cout<<a.x<<endl;                  // 1
B b(2,3);
cout<<b.A::x<<' '<<b.x<<endl;  // 2  3
}
```

6.5.2. Успадкування константних методів

Константні методи не можуть змінювати значення даних-членів класу. Їх успадкування підлягає загальним правилам. Якщо похідний клас перекриває константні методи базового класу, то у похідному класі існують методи, успадковані від базового класу, та власні однойменні методи. Перекриті методи викликаються за допомогою імені базового класу. Наприклад:

```
class A {
 protected:
 int x;
 public:
 A(int x){this->x=x;}
 int Get() const {return x;};
 };

class B: public A
 {
 int y;
 public:
 B(int x,int y):A(x){this->y=y;}
 int Get()const {return y;};
 };

void main()
{
B b(1,2);
cout<<b.A::Get()<<endl;  // 1
cout<<b.Get()<<endl;     // 2
}
```

6.6. Присвоєння об'єктів при успадкуванні класів

Об'єкту класу-предка можна присвоїти об'єкт одного з класів-нащадків. Це саме справедливе також для вказівників і посилань. Наприклад:

```
class A {
 protected:
 int x;
 public:
 A(int x){this->x=x;}
 int GetX(){return x;};
 };

class B: public A
 {
```

```
 int y;
 public:
 B(int x,int y):A(x){this->y=y;}
 int GetY(){return y;};
 };

void main()
{
A a(1);
B b(2,3);
cout<<a.GetX()<<endl;     // 1
cout<<b.GetX()<<endl;     // 2
cout<<b.GetY()<<endl;     // 3

a=b;
cout<<a.GetX()<<endl;     // 2

A* pa=&b;
cout<<pa->GetX()<<endl; // 2

A& ra=b;
cout<<ra.GetX()<<endl;    // 2
}
```

Присвоєння об'єкта (посилання, вказівника) похідного класу об'єкта (посиланню, вказівнику) базового класу можливе лише для public-успадкованих класів. Наступний приклад демонструє виникнення помилки присвоєння об'єктів, пов'язаної з `private`- (або `protected`-) успадкуванням.

```
class A{
        int x;
   public:
        A(int x=0){this->x=x;}
        int Get(void){return x;}
     };

class B: private A // або protected
     {
      int y;
   public:
      B(int x, int y):A(x){this->y=y;}
      int Get(void){return y;}
     };

void main()
{
   A* p=new B(5,7);       // Помилка
   cout<<p->Get()<<endl;
}
```

У результаті компіляції програми отримаємо повідомлення про те, що неможливо перетворити вказівник `B*` до вказівника `A*`.

Зворотне присвоєння об'єкта похідного клаєу об'єкту базового класу можливе за наявності конструктора за замовчуванням `A::A()` та конструктора перетворення типу `B::B(A)`, наприклад такого:

```
B::B(A a){y=a.GetX();}
```

Вказівник (або посилання) на об'єкт похідного класу можна присвоїти вказівнику (посиланню) з типом базового класу за допомогою відповідного перетворення типів:

```
B* pb=(B*)&a;
B& rb=(B&)a;
```

Таке перетворення не завжди коректне. Для перетворення типів при успадкуванні класів необхідно використовувати операції приведення типів (див. п. 12.3) замість їх перетворення у стилі C/C++. Доцільність і правильність використання зворотних перетворень та присвоєнь об'єктів при успадкуванні класів визначає програміст.

Присвоєння об'єктів успадкованих класів можна змоделювати за допомогою їх об'єднання. При об'єднанні об'єктів їм виділяється одна і та сама область оперативної пам'яті, тобто вони накладаються один на одного. Має значення порядок ініціалізації об'єктів в об'єднанні. Для правильного перетворення спочатку потрібно проініціалізувати об'єкт класу предка, а потім – класу нащадка.

При накладанні об'єктів не викликається операція присвоєння і не враховується режим успадкування класів.

Наступний приклад демонструє об'єднання об'єктів при успадкуванні класів.

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

class A {
protected:
 long x;
public:
 void set_x(long x1){x=x1;}
 void output(){cout<<hex<<x<<endl;}
 };

class B: public A
{
 short y;
public:
```

```
 void set_y(long x1, short y1){x=x1; y=y1;}
 void output(){cout<<hex<<x<<" "<<y<<endl;}
};

union mix {
 A a;
 B b;
 // Конструктор об'єднання
 mix (long m=0, long n=0, short k=0)
 {a.set_x(m), b.set_y(n, k);}
 } u(0xAAAAAAAA, 0xBBBBBBB, 0xCCCC);

void main()
{
cout.setf(ios::showbase|ios::uppercase);

u.a.output();       // 0XBBBBBBBB
u.b.output();       // 0XBBBBBBBB  0XCCCC
}
```

При об'єднанні об'єктів у базовому та похідному класах не повинні перекриватися оригінальні конструктори за замовчуванням.

Присвоєння об'єктів за допомогою їх об'єднання слід розглядати як теоретичну можливість, а не як практичну доцільність.

6.7. Приклад програми

У C++ дозволяється успадкування коду класів, а не даних об'єктів класів. Для ілюстрації цього розглянемо такий приклад успадкування класів.

Нехай клас `Son` *успадковує з класу-предка ім'я батька* `Father::Name`. *Клас* `Son` *оголошує власне поле* `Son::Name`. *У результаті в похідному класі Son існуватимуть два поля імен* `Son::Name` *та* `Father::Name` *(ім'я та по батькові).*

```
#include <iostream>
#include <string.h>

using namespace std;

class Father{
protected:
char *Name;
public:
Father(char *N)
```

```
{Name=new char [20];
 strcpy(Name,N);
}
~Father()
{delete []Name;}
void Output()
 {cout<<Name<<endl;}
};

class Son: public Father
{
protected:
char *Name;
public:
Son(char *N1, char* N2="Unknown"):Father(N2)
{Name=new char [20];
 strcpy(Name,N1);
}
~Son()
{delete []Name;}
void Output()
 {cout<<Name<<" is Son of "<<Father::Name<<endl;}
};

void main()
{
 Father a("Pyotr");
 Son b("Ivan");
 b.Output();        // Ivan is Son of Unknown
 Son c("Vasyl", "Stephan");
 c.Output();        // Vasyl is Son of Stephan
}
```

У головній функції визначимо та проініціалізуємо об'єкти класів `Father` та `Son`. Успадкування класу `Father` класом `Son` не означає успадкування об'єкта `a` об'єктами `b` або `c`. Наприклад, якщо об'єкт базового класу `Father` визначає ім'я батька "`Pyotr`", а об'єкт похідного класу `Son` визначає ім'я сина "`Ivan`", то це не означає, що `Pyotr` є батьком `Ivan`.

Розділ 7
ПОЛІМОРФІЗМ ВІРТУАЛЬНИХ МЕТОДІВ

7.1. Види поліморфізму

Поліморфізм в об'єктно-орієнтованому проектуванні – це властивість програмної системи використовувати об'єкти з однаковим інтерфейсом без інформації про їх тип та внутрішню структуру. Інакше, поліморфізм – це використання під одним іменем різних функцій, призначених для опрацювання даних різних типів. У C++ є такі види поліморфізму:

- *статичний* – перевантаження і перевизначення функцій та операцій; шаблони функцій та класів;
- *динамічний* – перевизначення віртуальних функцій.

Статичний поліморфізм забезпечується під час компіляції програми, а динамічний – під час її виконання.

Поліморфізм *перевантаження функцій* (`overload`) полягає у можливості вибору компілятором різних реалізацій однієї і тієї самої функції залежно від кількості або типів її аргументів. Наприклад:

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

class A
{
public:
void f()
{cout<<"A::f()"<<endl;}
void f(int x)
{cout<<"A::f(int)"<<endl;}
void f(double x)
{cout<<"A::f(double)"<<endl;}
void f(int, int)
{cout<<"A::f(int, int)"<<endl;}
void f(int, int, int)
{cout<<"A::f(int, int, int)"<<endl;}
};

void main()
{
A a;
a.f();              // A::f()
a.f(1);             // A::f(int)
a.f(3.14);          // A::f(double)
```

```
a.f(2, 3);          // A::f(int, int)
a.f(4, 5, 6);       // A::f(int, int, int)
}
```

У цій програмі здійснюються виклики перевантажених методів `A::f()` класу `A`. Перевантажені методи мають однакове ім'я, але можуть відрізнятися їх внутрішньою реалізацією.

Перевантаження операцій ґрунтується на використанні операторних функцій (`operator`). Наступний приклад програми здійснює перевантаження бінарної операції + для додавання числових полів об'єктів та операції << для виведення складу об'єктів на екран.

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

class A
{
protected:
int x;
public:
A(int x=0){this->x=x;}
A operator+(A& a) {return A(x+a.x);}
friend ostream& operator<<(ostream& os, A& a)
{os<<a.x<<endl;
 return os;}
};

class B:public A
{
protected:
int y;
public:
B(int x=0, int y=0):A(x){this->y=y;}
B operator+(B& b)
 {return B(x+b.x, y+b.y);}
friend ostream& operator<<(ostream& os, B& b)
{os<<b.x<<' '<<b.y<<endl;
 return os;}
};

void main()
{
A a1(1), a2(2), a3;
a3 = a1+a2;
cout<<a3;     // 3

B b1(3, 4), b2(5, 6), b3;
b3 = b1+b2;
cout<<b3;     // 8  10
}
```

Функція `main()` демонструє поліморфне застосування перевантажених операцій. Залежно від типів операндів одні і ті самі операції `+` та `<<` будуть викликані з класу `A` або з класу `B`.

Поліморфізм шаблонних функцій та класів полягає в тому, що для них додатково оголошуються формальні параметри типів, а конкретні фактичні параметри типів задаються при визначенні екземплярів класів (при оголошенні об'єктів). Так, з одного загального оголошення під час компіляції програми можна отримати різні інстанції функції або класу. Шаблонні функції та класи розглядаються у розділі 10.

У контексті успадкування класів поліморфізм полягає у *перевизначенні* (`override`) нащадком *методів* базового класу. Тоді для роботи з об'єктами похідних класів використовується інтерфейс їх базового класу, наприклад:

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

class A
{
public:
void output()const
{cout<<"A::output()"<<endl;}
};

class B:public A
{
public:
void output()const
{cout<<"B::output()"<<endl;}
};

class C:public A
{
public:
void output()const
{cout<<"C::output()"<<endl;}
};

void main()
{
C *p1 = new C;
p1->output();       // C::output()

A *p2 = p1;         // p1 == p2
p2->output();       // A::output()

B *p3 = new B;
p3->output();       // B::output()
```

```
A *p4 = p3;          // p3 == p4
p4->output();        // A::output()
}
```

У базовому класі A визначено метод output(), який перевизначається у похідних класах B та C. У функції main() вказівник p2 на клас A проініціалізовано вказівником p1 на клас C. Така ініціалізація допускається без явного перетворення типу, оскільки клас C є похідним від класу A. Проініціалізовані вказівники використовуються для виклику методу output(). Незважаючи на те, що вказівники p1 та p2 набувають однакових значень, за допомогою p1 буде викликано метод C::output(), а за допомогою p2 – метод A::output(). Аналогічні дії відбудуться з вказівниками p3 та p4 на класи B та A. Те, який метод буде викликано за допомогою того чи іншого вказівника, визначатиметься під час компіляції програми (раннє зв'язування).

Розглянуті вище види поліморфізму є *статичними*. На відміну від них, при *динамічному* поліморфізмі вибір того або іншого методу здійснюється під час виконання програми застосуванням механізму пізнього зв'язування. Динамічний поліморфізм реалізується на основі віртуальних методів. Наприклад:

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

class A
{
public:
virtual void output()const
{cout<<"A::output()"<<endl;}
};

class B:public A
{
public:
virtual void output()const
{cout<<"B::output()"<<endl;}
};

class C:public A
{
public:
virtual void output()const
{cout<<"C::output()"<<endl;}
};

void main()
{
C *p1 = new C;
p1->output();        // C::output()
```

```
A *p2 = p1;          // p1 == p2
p2->output();        // C::output()

B *p3 = new B;
p3->output();        // B::output()

A *p4 = p3;          // p3 == p4
p4->output();        // B::output()
}
```

Тепер за допомогою вказівників `p2` та `p4` будуть викликані віртуальні методи `output()` з похідних класів `C` та `B` відповідно.

7.2. Віртуальні методи класу

7.2.1. Раннє та пізнє зв'язування

Термін "зв'язування" використовується для визначення особливостей виклику методів класів за допомогою їх об'єктів. Розрізняють раннє та пізнє зв'язування.

За *раннього зв'язування* поєднання методів з об'єктами класу здійснюється на етапі компіляції програми. Це означає, що за допомогою об'єкта класу, посилання або вказівника на клас можна викликати тільки методи цього самого класу (залежно від прав доступу). Якщо об'єкт, посилання або вказівник на базовий клас проініціалізовані об'єктом (або його адресою – для вказівника) похідного класу, то за їх допомогою не можна забезпечити виклик методів похідного класу.

Пізнє зв'язування реалізується за допомогою віртуальних методів (віртуальних функцій). *Віртуальний метод* – це сукупність перевизначених функцій-членів (з однаковим іменем, списком параметрів та типом результату) в ієрархії успадкування класів, оголошених зі словом `virtual`. При пізньому зв'язуванні поєднання віртуальних методів з об'єктами здійснюється під час виконання програми. Це означає, що за допомогою вказівника або посилання на базовий клас, проініціалізованих відповідно адресою або іменем об'єкта похідного класу, можна викликати віртуальні методи похідного класу.

Пізнє зв'язування забезпечує динамічний поліморфізм, коли для роботи з похідним класом використовується інтерфейс (набір віртуальних методів) базового класу. Реалізувати один і той самий інтерфейс у базовому і похідному класах можна по-різному. Тим самим здійснюється розширення або спеціалізація програмної системи. Динамічний поліморфізм ґрунтується на перевизначенні

віртуальних методів, які можуть бути звичайними (неіндексованими) або динамічними (індексованими). Пізнє зв'язування не підтримується, якщо віртуальні методи викликаються з конструкторів або деструктора класу.

Сукупність класів, у яких оголошується, визначається та перевизначається віртуальний метод, називається *поліморфічним кластером.* Поліморфічний кластер реалізується тільки для *public-успадкувань класів.* В одній ієрархії успадкування класів може бути визначено декілька різних поліморфічних кластерів.

Особливості віртуальних методів:

1) віртуальними можуть бути тільки функції-члени класу, друзі не можуть бути віртуальними;
2) шаблонні методи (з оголошенням `template`) не можуть бути віртуальними;
3) віртуальні методи можуть бути оголошені у класі або структурі, об'єднання не може мати віртуальних методів;
4) слово `virtual` достатньо записати тільки у першому оголошенні віртуальних методів;
5) віртуальні методи не можуть бути статичними (`static`);
6) віртуальний метод повинен бути визначений у класі або оголошений чистим;
7) для забезпечення пізнього зв'язування віртуальні методи повинні викликатися за допомогою вказівників або посилань;
8) у межах поліморфічного кластера віртуальний метод не повинен змінювати свій тип, кількість або типи параметрів. Якщо віртуальний метод не перевизначається у похідному класі, то діє віртуальний метод попереднього в ієрархії успадкування класу;
9) якщо на деякому рівні успадкування класів оголошується метод, ім'я якого збігається з попередньо оголошеним віртуальним методом, але метод має змінений список параметрів, то за наявності в оголошенні слова `virtual` утворюється новий поліморфічний кластер. Якщо слово `virtual` не використовується, то для цього класу віртуальний метод не перевизначається (діє визначення з попереднього класу). Новий поліморфічний кластер не утворюється, а попередньо визначений кластер не переривається. Якщо оголошення методу відрізняється тільки його типом порівняно з однойменним віртуальним методом, то виникає конфлікт і виводиться повідомлення про помилку (крім коваріантних методів);
10) у класі допускається перевантаження віртуальних методів. Кожен з таких методів започатковує власний поліморфічний кластер;
11) віртуальний метод може бути оголошений дружнім (`friend`) до іншого класу;

12) пізнє зв'язування реалізується в межах поліморфічного кластера за допомогою вказівників або посилань. Вказівник на клас-предок можна проініціалізувати адресою об'єкта-нащадка. Посилання на клас-предок можна проініціаліувати об'єктом класу-нащадка.

Поліморфізм – це властивість вказівника (або посилання) на базовий клас, проініціалізованого адресою (або іменем – для посилання) об'єкта похідного класу, викликати віртуальні методи похідного класу у межах дії поліморфічного кластера.

7.2.2. Приклад поліморфізму віртуальних методів

Нехай клас `B` успадковує клас `A`. У класі `A` визначений, а у класі `B` перевизначений віртуальний метод `f()`, який виводить повідомлення про належність до класу.

Якщо віртуальний метод викликається з об'єкта класу, то діє раннє зв'язування. Це означає, що тип об'єкта визначатиме клас, з якого буде викликаний віртуальний метод.

Якщо для виклику віртуального методу використовуються вказівник (або посилання), проініціалізований адресою (або іменем – для посилання) об'єкта похідного класу, то діє пізнє зв'язування. У результаті буде викликаний віртуальний метод з похідного класу.

Наприклад:

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

class A
{
  public:
  virtual  void f();
};

void A::f(void)
{
cout<<"A::f()"<<endl;
}

class B:public A   // всі успадкування в поліморфічному
                   // кластері повинні мати режим public
{
  public:
  virtual void f();
};
```

```
void B::f(void)
{
cout<<"B::f()"<<endl;
}

void main()
{
A a;
a.f(); // раннє зв'язування, викликається A::f()

B b;
b.f(); // раннє зв'язування, викликається B::f()

a = b;
a.f(); // раннє зв'язування, викликається A::f()

A *p = &b;
p->f();// пізнє зв'язування, викликається B::f()

A &x = b;
x.f(); // пізнє зв'язування, викликається B::f()
}
```

У межах поліморфічного кластера віртуальні методи мають однакову адресу. Для перевірки цього рекомендується внести таке доповнення до попередньої програми:

```
void (A::*pf1)(void)=&A::f;
void (B::*pf2)(void)=&B::f;
printf("%Fp %Fp\n",pf1,pf2);
```

Слід зазначити, що поліморфізм стосується лише віртуальних методів класу. Застосування вказівника (або посилання) на базовий клас, проініціалізованого адресою (або іменем) об'єкта похідного класу, для виклику невіртуального методу похідного класу або для звернення до даних похідного класу призведе до помилок. Наприклад:

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

class A{
public:
  int x;
  A(int x=0) {this->x=x;}
  virtual void print()
   {cout<<x<<endl;}
  };

class B:public A{
```

```
public:
 int y;
 B(int x=0, int y=0):A(x) {this->y=y;}
 virtual void print()
  {cout<<x<<' '<<y<<endl;}
 void output()
       {cout<<x<<' '<<y<<endl;}
 };

void main()
{
A *p = new B(1, 2);
// виклик віртуального методу
p->print();  // викликається віртуальний метод B::print()

// виклик невіртуального методу класу B
//p->output();     // помилка: output не є членом класу A

// звернення до даних
cout<<p->x<<endl;
// cout<<p->y<<endl; // помилка: y не належить до класу A
delete p;
}
```

7.2.3. Виключення поліморфізму віртуальних методів

Для виключення дії механізму пізнього зв'язування при виклику віртуального методу необхідно вказати назву класу, до якого він належить:

```
вказівник -> назва_класу::назва_методу(аргументи);
посилання.назва_класу::назва_методу(аргументи);
```

Тут вказівник і посилання мають оголошений тип базового класу та попередньо проініціалізовані відповідно адресою (для вказівника) та іменем (для посилання) об'єкта похідного класу.

Наприклад:

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

class A
{
  public:
  virtual  void f()
  {cout<<"A::f()"<<endl;}
};

class B:public A
{
```

```
  public:
  virtual void f()
  {cout<<"B::f()"<<endl;}
};

void main()
{

A *p = new B;
p->A::f();
}
```

У результаті викликається метод `A::f()` базового класу, а не метод `B::f()` похідного класу.

7.2.4. Коваріантні віртуальні методи

Віртуальний метод не повинен змінювати назву, кількість та типи параметрів і тип результату у межах свого поліморфічного кластера.

Зміна типу результату можлива лише для *коваріантних* віртуальних методів. Методи є коваріантними, якщо їх типи є узгодженими з типами класів в іншій ієрархії успадкувань. Наприклад:

```
#include <iostream>

using namespace std;

class B1
{
public:
 virtual void f1(){cout<<"B1::f1()"<<endl;}
};

class D1:public B1
{
public:
 virtual void f1(){cout<<"D1::f1()"<<endl; }
};

class B2
{
public:
 virtual B1* f2(){cout<<"B2::f2()"<<endl; return new B1;}
};

class D2:public B2
{
```

```
public:
 virtual D1* f2(){cout<<"D2::f2()"<<endl; return new D1;}
};

void main()
{

B2* p = new B2;
p->f2()->f1();      // B2::f2()
                    // B1::f1()
B2* q = new D2;
q->f2()->f1();      // D2::f2()
                    // D1::f1()
delete p;
delete q;
}
```

У цій програмі оголошено дві ієрархії успадкувань класів: `B1<-D1` та `B2<-D2`. Віртуальний метод базового класу `B2` повертає вказівник на базовий клас `B1`, а віртуальний метод похідного класу `D2` – на похідний клас `D1`. Ці вказівники використано для викликів віртуальних методів.

Віртуальні методи також є коваріантними, якщо їхні типи є вказівниками або посиланнями на класи у межах дії поліморфічного кластера. Наприклад:

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

class A
{
protected:
  int x;
public:
  A(int x=0){this->x=x;}
  virtual A& inc()
  {++x;
  return *this;}
  virtual void output()
  {cout<<x<<endl;}
};

class B:public A
{ int y;
public:
  B(int x=0, int y=0):A(x){this->y=y;}
  virtual B& inc()
  {++x;
   ++y;
  return *this;}
```

```
  virtual void output()
  {cout<<x<<' '<<y<<endl;}
};

void main()
{
A& a = *new B(2, 7);
a.inc();
a.output();  // 3  8
}
```

Віртуальний метод `inc()` відрізняється типом результату в ієрархії успадкування класів. У класі `A` він має тип посилання на `A`, а у класі `B` – тип посилання на `B`. Ці типи є узгодженими в ієрархії успадкування класів (у межах поліморфічного кластера). Незважаючи на це, механізм дії пізнього зв'язування не порушується, що видно із виведення програми.

7.2.5. Віртуальні операторні методи

Операторні методи класів можуть бути віртуальними. Для ілюстрації цього модифікуємо попередню програму, замінивши віртуальний метод `inc()` операторним методом префіксного інкременту. Наприклад:

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

class A
{
protected:
  int x;
public:
   A(int x=0){this->x=x;}
  virtual A& operator++()
  { ++x;
   return *this;}
  virtual void output()
  {cout<<x<<endl;}
};

class B:public A
{ int y;
public:
 B(int x=0, int y=0):A(x){this->y=y;}
  virtual B& operator++()
  {++x;
   ++y;
  return *this;}
```

```
  virtual void output()
  {cout<<x<<' '<<y<<endl;}
};

void main()
{
A& a = *new B(2, 7);
++a;
a.output();         // 3  8
}
```

У функції `main()` посилання на базовий клас `A` проініціалізовано об'єктом похідного класу `B`. Операція префіксного інкременту, застосована до такого посилання, призведе до виклику віртуального операторного методу `B::operator++()` з похідного класу, що підтверджується виведенням програми.

У цьому прикладі віртуальні методи мають коваріантні типи результатів. Оскільки посилання на базовий клас може бути проініціалізоване об'єктом похідного класу, то результат роботи програми не зміниться, якщо у класі `B` операторний метод матиме тип результату, тотожний типу результату операторного методу з класу `A`.

7.2.6. Вказівники на віртуальні методи

Поліморфізм може бути реалізований також за допомогою вказівників на віртуальні методи. У наступній програмі віртуальний метод `Get()` визначає поліморфний кластер у межах успадкувань класів `A<-B` (клас `A` є предком класу `B`). Вказівник на базовий клас `A` проініціалізовано адресою об'єкта похідного класу `B` і використано для виклику віртуального методу `Get()` за допомогою вказівника `pf` на цей метод.

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

class A{
  protected:
  int x;
  public:
  A(int x){this->x=x;}
  virtual void Get(){cout<<x<<endl;}
};

class B:public A
{
 int y;
 public:
 B(int x, int y):A(x){this->y=y;}
```

```
 virtual void Get(){cout<<x<<' '<<y<<endl;}
};

// вказівник на віртуальний метод
void (A::*pf)()=&A::Get;

void main()
{
 A *p=new B(1,2);
 (p->*pf)();        // 1  2
 delete p;
}
```

Звернення до віртуального методу `Get()` за допомогою вказівника `pf` еквівалентне зверненню `p->Get()` .

7.2.7. Реалізація поліморфізму за допомогою об'єднання

Поліморфізм віртуальних методів можна реалізувати об'єднанням посилань або вказівників на базовий і на похідний класи. У разі об'єднання посилань або вказівників у відповідних класах можуть бути визначені конструктори. На відміну від об'єднання об'єктів, не вимагається наявності оригінального конструктора за замовчуванням. Для забезпечення пізнього зв'язування має значення порядок розміщення посилань (або вказівників) в об'єднанні. Спочатку необхідно розмістити посилання на базовий клас, а потім – на похідний клас. Наприклад:

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

class A {
protected:
 long x;
public:
 A(long x1=0){x=x1;}
 virtual void output(){cout<<x<<endl;}
 };

class B: public A
{
 short y;
public:
 B(long x1=0, short y1=0){x=x1; y=y1;}
 void output(){cout<<x<<" "<<y<<endl;}
};
```

```
union mix {
 A& a;
 B& b;
 // Конструктор об'єднання
 mix (A& a1, B& b1):a(a1), b(b1) {}
 } ;

void main()
{
A a(5);
B b(6, 7);
mix u(a, b);
cout<<typeid(u.a).name()<<endl; // class B
u.a.output();      // 6  7
}
```

Результатом об'єднання посилань є те, що елемент `u.a` вказуватиме на об'єкт похідного класу `B`. Його застосування призведе до виклику віртуального методу `B::output()` з похідного класу `B`, що видно із виведення програми.

Якщо відмінити оголошення `virtual`, то відбудеться раннє зв'язування методів `output()` з об'єктами класів. За допомогою посилання `u.a` буде викликаний метод `A::output()` з базового класу `A`.

Незважаючи на таку властивість об'єднання, не слід використовувати його для роботи з віртуальними методами, оскільки при цьому не враховується інформація про відношення узагальнення класів, що може призвести до зменшення надійності або до аварійних станів роботи програми.

7.3. Динамічні віртуальні методи

7.3.1. Оголошення динамічних віртуальних методів

Динамічні віртуальні методи – це підклас віртуальних методів, який відрізняється способом виклику на етапі виконання. Оголошуються за допомогою індексу динамічного методу, наприклад:

```
virtual void f(void)=[100];
```

Індекс задається у квадратних дужках константою цілого типу. Він повинен бути унікальним серед індексів інших динамічних методів, але однаковим в ієрархії успадкування конкретного методу.

Динамічні віртуальні методи реалізуються через спеціальні таблиці динамічних віртуальних методів. Таблиця динамічних віртуальних методів є

додатковою таблицею для цілого класу, і її адреса міститься у таблиці віртуальних методів класу. У таблиці динамічних віртуальних методів класу запам'ятовуються індекси та адреси тільки тих віртуальних методів, які перевизначаються для цього класу. Таблиці динамічних віртуальних методів є коротшими від таблиць нединамічних віртуальних методів, що економить оперативну пам'ять, але ускладнює доступ до віртуальних методів.

Індекси динамічних віртуальних методів опрацьовуються спеціальною підпрограмою – диспетчером викликів.

Динамічні віртуальні методи не підтримуються у Visual C++.

7.3.2. Поліморфізм динамічних віртуальних методів

Нехай у класі `A` визначено динамічний віртуальний метод `A::f()` з індексом `100`, який виводить повідомлення про його належність класу. Похідний клас `B` перевизначає динамічний віртуальний метод з тим самим індексом. Функція `main()` демонструє виклик динамічного віртуального методу за допомогою посилання та вказівника на клас `A`, проініціалізованих відповідно іменем та адресою об'єкта класу `B`. У обох випадках викликається віртуальний метод з похідного класу `B`.

```
class A
{
  public:
  virtual  void f(void)=[100];
};

void A::f(void)
{
cout<<"A::f"<<endl;
}

class B:public A
{
  public:
  virtual void f(void)=[100];
};

void B::f(void)
{
cout<<"B::f"<<endl;
}

void main()
{
B b;
```

```
A *p=&b;
p->f();        // викликається B::f
A &a=b;
a.f();         // викликається B::f
}
```

7.4. Механізм дії віртуальних методів

7.4.1. Машинний формат об'єктів

Якщо клас містить віртуальні методи (ВМ), то у відведеній для об'єкта оперативній пам'яті запам'ятовуються вказівник на таблицю віртуальних методів (ТВМ) та поля даних у порядку їх оголошення. Вказівник на ТВМ автоматично ініціалізується конструктором при створенні об'єкта. Програма C++ не має доступу до цього вказівника.

У похідному класі спочатку виділяється пам'ять для успадкованих від базового класу даних. Власні поля похідного об'єкта записуються після успадкованих полів. Схему машинного формату об'єктів базового та успадкованого класів зображено на рис. 7.1.

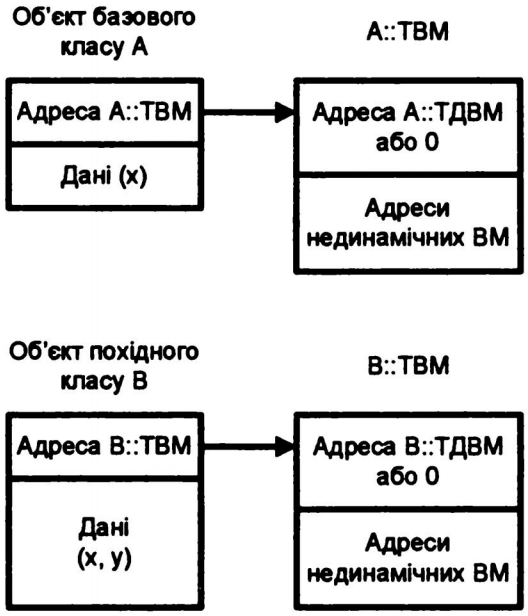

Рис. 7.1. Структура об'єктів базового та похідного класів

7.4.2. Таблиці віртуальних методів

Якщо клас оголошує або успадковує віртуальні методи, то для його об'єктів створюється ТВМ. Усі екземпляри одного класу використовують спільну ТВМ. Два різні класи використовують різні ТВМ навіть тоді, коли вони є ідентичними. Об'єкти успадкованих класів мають різні ТВМ.

ТВМ сворюються автоматично під час компіляції програми. Вказівники на ТВМ автоматично записуються конструкторами в об'єкти класу. Прямого звернення до ТВМ з програми не відбувається.

Структуру ТВМ для базового і похідного класів зображено на рис. 7.1. На початку ТВМ записується адреса таблиці динамічних віртуальних методів (ТДВМ) або нуль, якщо клас не містить динамічних віртуальних методів (ДВМ). Далі розміщуються адреси віртуальних методів класу. У ТВМ записуються адреси усіх ВМ, навіть тих, які не перевизначаються у цьому класі.

У межах поліморфічного кластеру конкретний ВМ має одне і те саме зміщення відносно початку ТВМ. Це значення використовується у коді програми для звернення до ВМ. Реальне значення адреси ВМ записано у ТВМ. Таке значення визначається під час роботи програми і може вказувати на ВМ як базового, так і похідного класу залежно від значення вказівника на об'єкт.

Звичайні ВМ викликаються у такій послідовності. Із об'єкта читається адреса ТВМ. Далі з ТВМ вибирається адреса потрібного ВМ та здійснюється його виклик.

Для роботи з ДВМ створюється додаткова таблиця ТДВМ (рис. 7.2). Адреса ТДВМ записується на початку ТВМ.

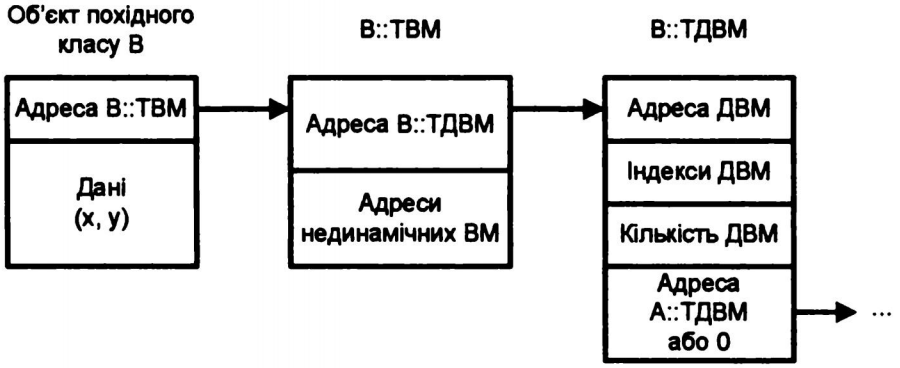

Рис. 7.2. Структури та зв'язок таблиць віртуальних методів

На початку ТДВМ розміщуються адреси тільки тих ДВМ, які визначаються або перевизначаються у класі. Далі розміщується масив зі значеннями індексів ДВМ цієї ТДВМ. Наступне слово містить число ДВМ цієї ТДВМ. В останньому слові ТДВМ записується адреса ТДВМ об'єкта-предка або нуль, якщо предок відсутній.

У ТДВМ може міститися кеш для запам'ятовування індексу та адреси ДВМ, викликаного останнім.

На відміну від ТВМ, у ТДВМ записуються тільки адреси тих ДВМ, які визначаються або перевизначаються на даному рівні успадкування класів. У результаті ТДВМ займають менший об'єм оперативної пам'яті, ніж ТВМ. Однак, для пошуку адреси ДВМ у системі таблиць ТДВМ витрачується більше часу, ніж на відповідний виклик звичайного ВМ за допомогою ТВМ.

ДВМ викликаються у такій послідовності. Із об'єкта читається адреса ТВМ, а із ТВМ – адреса ТДВМ. Далі з ТДВМ вибирається значення кешованого індексу, яке порівнюється з індексом ДВМ, який викликається. У разі збігу викликається ДВМ, адреса якого міститься у кеші для адреси. Якщо індекс ДВМ не збігається з кешованим індексом, то відбувається пошук індексу спочатку у власній ТДВМ, а потім – у ТДВМ об'єктів-предків. Пошук ТДВМ об'єкта-предка відбувається за значенням адреса, записаної у ТДВМ об'єкта-нащадка. Індекс та адреса знайденого ДВМ записується у кеш ТДВМ об'єкта. Якщо ДВМ не знайдено в ланцюжку ТДВМ, то виникає помилка.

Висновки:

- Для кожного класу, який містить віртуальні методи, компілятор будує таблицю ТВМ.
- Кожен об'єкт класу містить захований вказівник на ТВМ.
- Компілятор автоматично включає у конструктор фрагмент коду для ініціалізації вказівника на ТВМ.
- Для будь-якої заданої ієрархії класів адреса віртуального методу має однакове зміщення у всіх ТВМ.
- При виклику віртуальних методів в об'єкті знаходиться вказівник на ТВМ, за яким з ТВМ вибирається відносна адреса віртуального методу та здійснюється його виклик.
- Об'єкти одного класу мають одну (спільну) ТВМ.
- Об'єкти успадкованих класів мають різні ТВМ.
- Об'єкти класів, успадкованих від одного (спільного) класу, мають різні ТВМ.
- Два класи з однаковою структурою мають різні ТВМ.

7.5. Віртуальний деструктор

При роботі з вказівниками та посиланнями на тип базового класу, які ініціалізуються об'єктами похідних класів, у базовому класі бажано використовувати віртуальний деструктор. Віртуальний деструктор започатковує полімор-

фічний кластер. Якщо базовий клас містить віртуальний деструктор, то деструктор похідного від нього класу теж буде віртуальним.

Наприклад:

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

class A
{
 protected:
 int *p;
 public:
 A(int x=0)
 {cout<<"A(int)"<<endl;
  p=new int(x);
 }

//virtual
  ~A()
 {cout<<"~A()"<<endl;
  delete p;
 }

 virtual void print(){cout<<*p<<endl;}
};

class B:public A
{
 int *q;
 public:
 B(int x=0,int y=0):A(x)
 {cout<<"B(int,int)"<<endl;
 q=new int(y);
 }

 ~B()
 {cout<<"~B()"<<endl;
  delete q;
 }
 virtual void print(){cout<<*p<<' '<<
 *q<<endl;}
};

void main()
{
cout<<hex;
cout<<coreleft()<<endl;
A *p=new B(1,2);
```

```
p->print();
cout<<coreleft()<<endl;
delete p;
cout<<coreleft()<<endl;
}
```

Якщо деструктор `~A()` не віртуальний, то при звільненні динамічної пам'яті, закріпленої за вказівником `p`, викликається тільки деструктор `~A()`, що може привести до втрати ресурсів похідного класу `B`.

Якщо деструктор `~A()` віртуальний, то викликаються деструктор `~B()`, а потім деструктор `~A()`, і об'єкт класу `B` повністю знищується. Зміну розміру вільної динамічної пам'яті можна відстежити за допомогою функції `coreleft()`.

7.6. Чисті віртуальні методи та абстрактні класи

7.6.1. Оголошення та властивості

Чиста віртуальна функція (pure virtual function) – це функція-член (або метод) класу, для якої оголошено лише інтерфейс, а реалізація знаходиться в одному з похідних класів. Чистий віртуальний метод оголошується за допомогою специфікатора `= 0`:

```
class A{
public:
virtual void func()=0;    // чистий віртуальний метод
};
```

Чистий віртуальний метод не може бути викликаний явно або неявно з конструктора.

Абстрактний клас (abstract class) містить один або декілька чистих віртуальних методів. Чистий абстрактний клас (pure abstract class) складається тільки з чистих віртуальних методів (не містить даних та інших методів). Основне призначення абстрактного класу – це оголошення інтерфейсу похідних від нього класів.

Може використовуватися тільки як базовий для інших класів. Неможливо оголосити об'єкт абстрактного класу, але можна оголосити вказівник на абстрактний клас. Можна оголосити посилання на абстрактний клас, якщо воно ініціалізується об'єктом похідного класу.

Якщо похідний від абстрактного клас не визначає всіх чистих віртуальних методів, то він теж є абстрактним.

```
#include <iostream>

using namespace std;

class A      // абстрактний клас
{
protected:
int x;
public:
A(int x1):x(x1){}
virtual void output() const=0; // чистий віртуальний метод
};

class B:public A   // теж абстрактний клас
{
public:
B(int x):A(x){}
virtual void output() const=0;  // можна не оголошувати
};

class C:public B   // тут визначається чистий
                   // віртуальний метод, тому
                   // клас C не є абстрактним
{
public:
C(int x):B(x){}
virtual void output() const
{ cout<<x<<endl;}
};

void main()
{
//A a(1); -  не можна оголосити об'єкт абстрактного класу
//B b(2); -  не можна оголосити об'єкт абстрактного класу

C c(3);
c.output();         // 3
A *p=&c; // можна оголосити вказівник на абстрактний клас
B *q=&c;
p->output(); // 3
q->output(); // 3
}
```

Параметри та результат функції не можуть мати тип абстрактного класу. Однак вони можуть бути вказівниками або посиланнями на абстрактний клас, наприклад:

```
#include <iostream>
```

```
using namespace std;

class B{
public:
 virtual void f()=0;
};

class D:public B{
public:
 virtual void f(){cout<<"D::f()"<<endl;}
};

B&  ref_B(B* p, B& r){
 //...
 return r;
}

void main()
{
B *p = new D;
B &r = *new D;

ref_B(p, r).f();

delete p;
delete &r;
}
```

Великі програмні системи потрібно створювати на основі абстрактних базових класів. Це зменшує залежність похідних класів від особливостей реалізації базових класів, що призводить до гнучкості організації програмної системи, забезпечує стереотипне використання програмних компонентів при внесенні змін у їх програмний код. Сумісне використання абстрактних базових класів разом з композицією та агрегуванням об'єктів забезпечують розширюваність та надійність роботи програмної системи на основі моделі повторного використання коду.

7.6.2. Застосування абстрактних класів

На основі успадкування абстрактного класу можна побудувати колекції різнотипних об'єктів. Наприклад, розглянемо побудову однонапрямленого лінійного списку, що складається з об'єктів різних класів.

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

```
class X       // абстрактний клас
{
public:
virtual void output()=0;
};

// класи об'єктів списку
class A:public X
{int info;
public:
A(int i){info=i;}
void output(){cout<<info<<endl;}
};

class B: public X
{double info;
public:
B(double d){info=d;}
void output(){cout<<info<<endl;}
};

class C: public X
{string info;
public:
C(string s){info=s;}
void output(){cout<<info<<endl;}
};

// клас елемента списку
class list{
X* ptr_obj;
list* next;
public:
list(X* p_obj, list *p){ptr_obj=p_obj; next=p;}
X* get_ptr_obj(){return ptr_obj;}
list * get_next() {return next;}
};

void main()
{
 const int n=3;
 X* ptr_obj[n]={new A(5), new B(3.14), new C("ABCDEF")};
 list *head=NULL, *p;

// Створення списку об'єктів
 for(int i=0;i<n; ++i)
   {
    p=new list(ptr_obj[i],head);
    head=p;
```

```
    }

  cout<<"Виведення списку об'єктів"<<endl;
  p=head;
  while(p!=NULL)
  {
   p->get_ptr_obj()->output();
   p=p->get_next();
  }
  cout<<endl;

 // Витирання списку об'єктів
 while(head!=NULL)
  {
   p=head;
   head=head->get_next();
   delete p->get_ptr_obj();
   delete p;
  }
 }
```

Тут X є базовим абстрактним класом, що містить чистий віртуальний метод `output()`. Класи A, B, C успадковують абстрактний клас X та визначають віртуальний метод `output()` для виведення поля даних `info`.

Клас `list` агрегує об'єкт одного з похідних від X класів за допомогою вказівника на базовий клас X. Крім цього, у класі `list` визначено вказівник `next` на наступний елемент списку. У відкритій частині класу `list` розміщено конструктор та методи читання вказівників із закритої частини.

У головній функції визначено масив вказівників на базовий клас X, елементи якого набувають значення адрес об'єктів, розміщених в області динамічної пам'яті.

Формування списку об'єктів здійснюється у динамічній пам'яті. Для створення кожного елемента викликається конструктор класу `list`, який ініціалізує вказівник на агрегований об'єкт (A, B або C) та вказівник на наступний об'єкт класу `list`.

Після створення списку виконується його виведення на екран за допомогою віртуального методу `output()`, який викликається за допомогою вказівника на базовий клас X, проініціалізованого адресою одного з похідних від нього класів. Завдяки пізньому зв'язуванню буде викликано віртуальний метод виведення елементів даних потрібного похідного класу.

Замість зовнішньої реалізації весь список об'єктів можна інкапсулювати у клас. Для цього, як і раніше, визначається абстрактний клас X та похідні від нього класи A, B, C, об'єкти яких необхідно включити у список `list`. Об'єкти

класів A, B, C агрегуються у структуру elem, яка містить вказівник на базовий клас X та вказівник на наступний об'єкт класу elem.

У закритій частині класу list визначається вказівник head на початок списку. У відкритій частині визначено конструктор і деструктор списку, методи занесення елемента на початок списку, видалення початкового елемента та виведення списку на екран.

```
// виконати оголошення класів X, A, B, C так,
// як у попередньому прикладі
class list{  // клас списку об'єктів
// елемент списку
struct elem{
 X* ptr_obj; // вказівник на агрегований об'єкт
 elem* next; // вказівник на наступний елемент списку
// конструктор елемента списку
 elem(X* p_obj, elem *p){ptr_obj=p_obj; next=p;}
 };
// вказівник на початок списку
elem * head;
public:
// конструктор
list(X* b[], int n){
       cout<<"list::list(X* [])"<<endl;
       head=NULL;
       for(int i=0;i<n;i++) {add(b[i]);}
      }
// деструктор
~list(){cout<<"list::~list()"<<endl;
       while(head){del();}
       }
// метод занесення елемента у початок список
void add(X* q){elem *p=new elem(q, head); head=p;}
// метод витирання початкового елемента списку
void del(){
 elem *p=head->next;
 delete head->ptr_obj;
 delete head;
 head=p;
}
// метод виведення списку
void print(){elem *p=head;
       while(p!=NULL)
       {
        p->ptr_obj->output();
        p=p->next;
       }
       cout<<endl;
      }
```

```
};
// головна функція
void main()
{
const int n=3;
X* ptr_obj[n]={new A(5), new B(3.14), new C("ABCDEF")};
list s(ptr_obj, n);
s.print();
}
```

Головна функція визначає масив вказівників `ptr_obj` на об'єкти класів `A`, `B`, `C`, який передається у конструктор для створення об'єкта `s` класу `list`.

Метод `print()` зчитує з кожного елемента списку вказівник на агрегований об'єкт, за яким, завдяки пізньому зв'язуванню, викликається віртуальний метод виведення даних одного з об'єктів класів `A`, `B` або `C`.

Крім списків, різнотипні об'єкти можна агрегувати в інші структури даних, наприклад, у масиви або дерева.

7.7. Ефективність поліморфізму віртуальних методів

Поліморфізм віртуальних методів є зручним механізмом для нарощування та вдосконалення програмних систем. Припустімо, що існує базовий варіант програмної системи, створеної за технологією об'єктно-орієнтованого програмування. Нехай у відкритій частині класів базової програмної системи оголошено ряд віртуальних методів. Такі віртуальні методи можуть бути перевизначені користувачем при успадкуванні базових класів системи. Метою перевизначення віртуальних методів є врахування особливостей розв'язуваної задачі. Тоді методи базової програмної системи зможуть викликати розроблені користувачем віртуальні методи за рахунок дії механізму пізнього зв'язування.

Нехай базову програмну систему побудовано на основі двох класів. Клас `A` містить оголошення віртуального методу `f()`. Клас `B` успадковує клас `A` і містить звичайний метод `work()`, який викликає віртуальний метод `f()`. Віртуальний метод `f()` базової програмної системи може бути не реалізований, а оголошений як чистий.

```
// ======== базова програмна система ========
class A
{
  virtual  void f(void);
};

void A::f(void)
```

```
{
cout<<"A::f"<<endl;
}

class B:public A
{
  public:
  virtual void f(void);
  void work(void);
};

void B::f(void)
{
cout<<"B::f"<<endl;
}

void B::work(void)
{ f(); }


// ==== користувацька частина програмної системи ====
class C:public B
{
  public:
  void f(void);
};

void C::f(void)
{
cout<<"C::f"<<endl;
}

void main()
{
C c;

c.work();     // для  virtual викликається C::f()
              // без  virtual  викликається B::f()
B &b=c;
b.work();     // для  virtual викликається C::f(),
              // але без посилання або без  virtual
              // викликається B::f()
}
```

У цьому прикладі виклик віртуального методу `f()` з методу `work()` здійснюється із залученням вказівника на віртуальний метод (тут неявно). Як відомо, вказівники на віртуальні методи теж підтримують пізнє зв'язування.

Побудована на класі C користувацька частина програмної системи успадковує клас B базової програмної системи і визначає власний віртуальний метод f(). Тоді виклик базового методу work() з користувацької частини програмної системи забезпечить звернення до розробленого користувачем віртуального методу f(). Так за рахунок пізнього зв'язування відбувається налаштування базової програмної системи під потреби користувача.

7.8. Приклад програми

Оголосити абстрактний клас з віртуальною функцією Площа. Оголосити похідні класи – Трикутник, Прямокутник та Круг, у яких визначити функції обчислення площі:

- *трикутника* – $S=\sqrt{p(p-a)(p-b)(p-c)}$, *де* $p=(a+b+c)/2$, a,b,c – *сторони трикутника;*
- *прямокутника* – $S=xy$, *де* x,y – *сторони прямокутника;*
- *круга* – $S=\pi r^2$, *де* r – *радіус.*

Продемонструвати механізм пізнього зв'язування за допомогою масиву вказівників на абстрактний клас, елементам якого присвоєно адреси об'єктів похідних неабстрактних класів. Використати вказівники для виклику віртуального методу.

```
#include <iostream>
#define _USE_MATH_DEFINES
#include <cmath>
#include <iomanip>

using namespace std;

class Figure{
 public:
 virtual float Area()=0;
};

class Triangle:public Figure
{float a, b, c;
 public:
 Triangle(float a1=0, float b1=0, float c1=0)
 {a=a1; b=b1; c=c1;}
 virtual float Area()
 { cout<<"Площа трикутника: ";
  float p=(a+b+c)/2;
  return sqrt(p*(p-a)*(p-b)*(p-c));
```

```
 }
};

class Rectangle:public Figure
{
 float x, y;
 public:
 Rectangle(float x1=0, float y1=0)
 {x=x1; y=y1;}

 virtual float Area()
 {
 cout<<"Площа прямокутника: ";
 return x*y;
 }
};

 class Circle:public Figure
 {
   float r;
   public:
   Circle(float r1=0) {r=r1;}

 virtual float Area()
 {
 cout<<"Площа круга: ";
 return M_PI*r*r;
 }
};

float GetArea(Figure * f)
{
return f->Area();
}

void main()
{
 Figure* p[3]={
               new Triangle(5, 8, 7),
               new Rectangle(3, 6),
               new Circle(4)
              };

 cout.setf(ios::fixed);
 cout.precision(2);

 for(int i=0; i<3; ++i)
   cout << GetArea(p[i]) << endl;
}
```

У цій програмі для отримання площі геометричних фігур використовується функція `GetArea(Figure*)`, параметром якої є вказівник на абстрактний клас `Figure`.

Функція `main()` звертається до функції `GetArea(Figure*)` з фактичним аргументом – адресою кожного із об'єктів фігур. Завдяки пізньому зв'язуванню, функція `GetArea(Figure*)` забезпечує правильний виклик віртуальної функції `Area()` для визначення площі конкретної геометричної фігури. Результатом роботи програми є такі виведення:

```
Площа трикутника: 17.32
Площа прямокутника:18.00
Площа круга: 50.2
```

Розділ 8
МНОЖИННЕ УСПАДКУВАННЯ КЛАСІВ

8.1. Особливості множинного успадкування класів

8.1.1. Оголошення множинного успадкування

Один клас C++ може успадковувати елементи декількох інших класів. Це забезпечує поєднання властивостей декількох класів в одному класі.

Оголошення множинного успадкування:

```
class Derived:     режим_успадкування_1    Base1,
                   /* ..., */
                   режим_успадкування_N    BaseN
{/*...*/};
```

Тут `Base1, ..., BaseN` – назви базових класів, а режим успадкування може набувати значень `private`, `protected`, `public`. Дія режимів успадкування така ж, як і для одинарного успадкування.

Семантично правильне використання множинного успадкування повинно ґрунтуватися на відношенні "є" : похідний клас у деякому сенсі є екземпляром одночасно усіх базових класів. Наприклад, підменю є меню та пункт меню, тому клас "підменю" можна успадкувати від класів "меню" та "пункт меню". Однак, успадкування класу "комп'ютер" від класів "системний блок", "екран", "клавіатура" та "миша" не зовсім правильне, оскільки "комп'ютер" не є тільки екраном, клавіатурою чи мишею окремо. У цьому випадку краще використовувати композицію об'єктів.

Поєднання різних режимів забезпечує гнучкість множинного успадкування, але може призвести до ускладнень та неоднозначностей доступу до елементів класу.

8.1.2. Порядок викликів конструкторів і деструкторів при множинному успадкуванні класів

За множинного успадкування конструктор похідного класу викликає конструктори базових класів для ініціалізації успадкованих елементів даних. Порядок виклику конструкторів є таким:

1) викликаються конструктори базових класів за послідовністю їх множинного успадкування; віртуальні базові класи (див. п. 8.1.6) ініціалізуються перед невіртуальними базовими класами;

2) якщо базовий клас є контейнерним, то спочатку будуть викликані конструктори для інкапсульованих у нього об'єктів за послідовністю їх розміщення у протоколі класу, а потім – відповідний конструктор базового класу;
3) для тих базових класів, які не вказані у списку ініціалізації конструктора похідного класу, викликаються конструктори за замовчуванням (без параметрів);
4) викликаються конструктори об'єктів, інкапсульованих у похідний клас;
5) на завершення викликається конструктор похідного класу.

Деструктори викликаються у зворотному порядку.

Розглянемо приклад програми множинного успадкування класом `D` класів `B` та `C`. Крім того, нехай кожен з класів `B`, `C`, `D` є контейнером для класу `A`. У класах визначено конструктори і деструктори, які виводять ідентифікаційну інформацію про свій виклик.

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

class A{
public:
 A(){cout<<"A()"<<"  ";}
 ~A(){cout<<"~A()"<<"  ";}
};

class B{
 A a;
public:
 B(){cout<<"B()"<<"  ";}
 ~B(){cout<<"~B()"<<"  ";}
};

class C{
 A a;
public:
 C(){cout<<"C()"<<"  ";}
 ~C(){cout<<"~C()"<<"  ";}
};

class D:public B, public C
{
 A a;
public:
 D(){cout<<"D()"<<"  ";}
 ~D(){cout<<"\n~D()"<<"  ";}
};
```

```
void main()
{
 D d;
}
```

Результат роботи програми отримаємо у вигляді такого виведення:

```
A()   B()   A()   C()   A()   D()
~D()  ~A()  ~C()  ~A()  ~B()  ~A()
```

8.1.3. Доступ до перекритих елементів класів

Оголошуючи власні елементи даних та методи, похідний клас може перекривати однойменні елементи, успадковані від базових класів. Тоді доступ до перекритих елементів відбувається через імена базових класів та операції дозволу доступу (::).

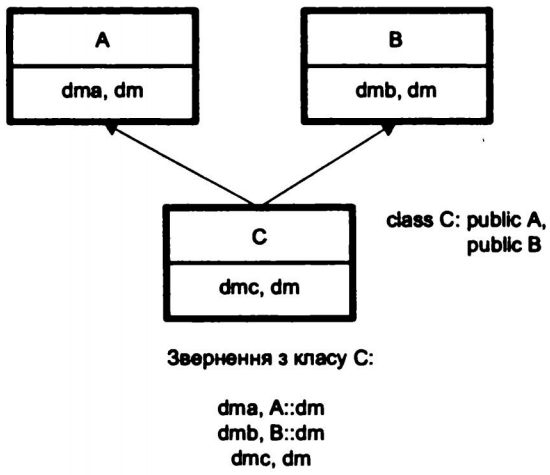

Рис. 8.1. Діаграма множинного успадкування класів

Наступна програма демонструє організацію доступу до перекритих полів даних при множинному успадкуванні класів так, як це зображено на рис. 8.1. Варіанти доступу до власних та успадкованих елементів наведено у конструкторі похідного класу. Як і для одинарного успадкувавання, при множинному успадкуванні об'єкти базових класів можуть бути проініціалізовані об'єктом класу-нащадка. Наприклад:

```
class A
{
protected:
int dma;
int dm;
public:
```

```
A(int x,int  y):dma(x),dm(y){cout<<"A::A(int,int)"<<endl;}
};

class B
{
protected:
int dmb;
int dm;
public:
B(int x,int  y):dmb(x),dm(y){cout<<"B::B(int,int)"<<endl;}
};

class C:public A, public B
{
int dmc;
int dm;
public:
C(int x1, int x2,
  int x3, int x4,
  int x5, int x6): A(x1,x2),B(x3,x4),dmc(x5),dm(x6)
{cout<<"C::C(int,int,int,int,int,int)"<<endl;}
void output()
{
cout<<dma  <<' '
    <<A::dm<<' '
    <<dmb  <<' '
    <<B::dm<<' '
    <<dmc  <<' '
    <<dm <<endl;
  }
};

void main()
{
C c(1,2,3,4,5,6);
c.output();

A a=c; // так можна
B b=c;
}
```

8.1.4. Неоднозначність параметричного перевантаження методів

У разі множинного успадкування методів, що мають однакове ім'я, але відрізняються типами параметрів, може виникати неоднозначність їх виклику. Наприклад, нехай визначено метод `A::output(int)` у базовому класі `A` та метод `B::output(int)` у базовому класі `B`. Клас `C` успадковує класи `A` та `B`.

```
#include <iostream>
#include <string>

using namespace std;

class A{
public:
 void output(int i)
 {cout<<"A::output()="<<i<<endl;}
};

class B{
public:
 void output(double d)
 {cout<<"B::output()="<<d<<endl;}
};

class C: public A, public B
{
// ...
};

void main()
{
C c;
c.output(5);        // Помилка
c.output(3.14);     // Помилка
}
```

За множинного успадкування виникає неоднозначність виклику методів `output(int)` та `output(double)` з класу `C` у зв'язку з неявним перетворенням типів їх параметрів. Оскільки значення цілого типу автоматично перетворюється до дійсного, а дійсне значення – до цілого, то компілятор не може визначити, який з двох методів необхідно викликати.

Для уникнення неоднозначності необхідно у класі `C` перевизначити успадковані методи:

```
class C: public A, public B
{
public:
void output(int i)
  {A::output(i);}
void output(double d)
  {B::output(d);}
};
```

Таке уточнення є інформацією компілятору про те, що перевантаження методів виконано осмислено. У результаті виклики методів відбудуться правильно.

8.1.5. Множинне успадкування класів із загальною базою

На практиці можливі випадки, коли декілька класів успадковують один і той самий клас. Такий варіант називається успадкуванням із загальною базою (спільним суперкласом). За множинного успадкування таких класів у похідний клас потрапить декілька екземплярів елементів загального базового класу. У такому випадку для звернення до множинно успадкованих елементів загального базового класу можна використати назву одного з похідних класів та операцію дозволу доступу.

Варіант множинного успадкування із загальною базою наведено на рис. 8.2. Графічно множинне успадкування із загальною базою позначають ромбоподібною діаграмою класів.

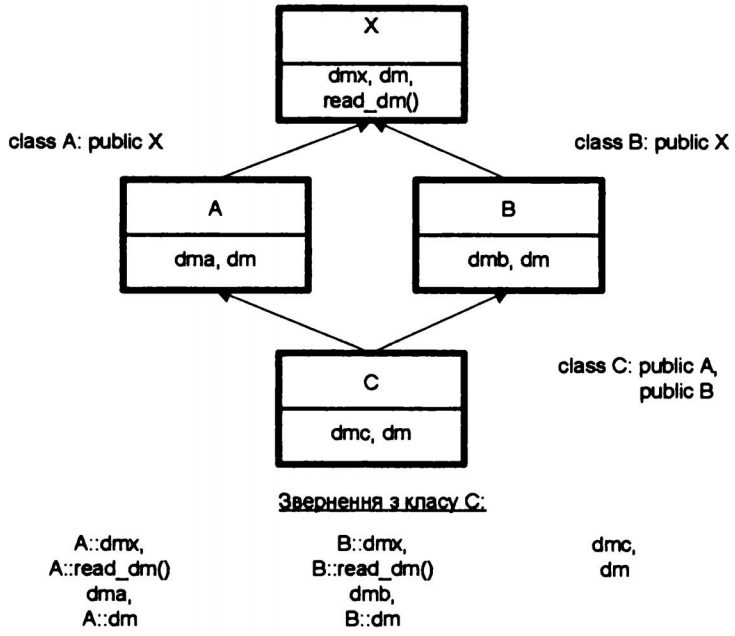

Рис. 8.2. Діаграма множинного успадкування класів із загальною базою

У результаті клас `C` матиме 10 елементів даних: два власні, по два успадкованих від класів `A` та `B` і двічі успадковані елементи класу `X` через проміжні класи `A` та `B`.

Для звернення до поля `dmx` класу `X` з класу `C` необхідно зазначити, через який клас відбулося успадкування, наприклад: `A::dmx`, `B::dmx`.

Ситуація ускладнюється, коли перекриваються однойменні елементи даних або однойменні методи класів. У наведеному прикладі в усіх класах знаходиться однойменне поле `dm`. Звернутися до поля `dm` класу `X` через ім'я проміжного класу `A` або `B` неможливо, адже `A::dm`, `B::dm` позначатиме наявне у класах `A` та `B` поле `dm`.

Тому для забезпечення доступу з класу C до поля dm класу X необхідна наявність методу X::read_dm(), який не перекривається у класах A та B.

Нижче наведено відповідну до рис. 8.2 програму множинного успадкування класів із загальною базою. Варіанти доступу до даних класу C, успадкованих від класів X, A та B, демонструє метод C::output().

```
#include <iostream>

using namespace std;

class X
{
protected:
int dmx;
int dm;
public:
X(int x1,int x2):dmx(x1),dm(x2)
       {cout<<"X::X()"<<endl;}
int read_dm(void){return dm;}
};

class A: public X
{
protected:
int dma;
int dm;
public:
A(int x1,int  x2, int x3, int x4):X(x1, x2),dma(x3),dm(x4)
       {cout<<"A::A()"<<endl;}
};

class B:public X
{
protected:
int dmb;
int dm;
public:
B(int x1,int x2, int x3, int x4):X(x1, x2),dmb(x3),dm(x4)
       {cout<<"B::B()"<<endl;}
};

class C:public A, public B
{
int dmc;
int dm;
public:
C(int x1, int x2,
  int x3, int x4,
```

```
  int x5, int x6,
  int x7, int x8,
  int x9, int x10): A(x1,x2,x3,x4), B(x5,x6,x7,x8),
                    dmc(x9),dm(x10)
       {cout<<"C::C()"<<endl;}

void output()
{cout<<A::dmx<<' '          // 1
<<A::read_dm()<<' '         // 2
<<dma<<' '                  // 3
<<A::dm<<' '                // 4

<<B::dmx<<' '               // 5
<<B::read_dm()<<' '         // 6
<<dmb<<' '                  // 7
<<B::dm<<' '                // 8

<<dmc  <<' '                // 9
<<dm<<endl;                 // 10
}

};

void main()
{
C    c(1,2,3,4,5,6,7,8,9,10);
c.output();

A    a(1,2,3,4);
B    b(1,2,3,4);
X    x1=a;  // так можна
X    x2=b;  // так можна

//X    x3=c; // так не можна
}
```

Клас C не може виконати пряме успадкування класу X. Крім того, об'єкту класу X не можна присвоїти об'єкт класу C. Посиланню на клас X не можна присвоїти об'єкт класу C, а вказівнику на клас X – адресу об'єкта класу C. Однак, для public-успадкувань дозволяється ланцюжок присвоєнь від похідного класу C множинного успадкування через проміжний клас A або B до загального базового класу X , наприклад:

```
X x3=a=c; // або X x3=b=c;
```

Під час створення об'єкта класу C викликаються конструктори базових класів у такій послідовності: X::X(), A::A(), X::X(), B::B(), C::C(). Деструктори викликаються у зворотному порядку.

Як видно із виведення функції `C::output()`, дані класу `X` потрапили у клас `C` у двох екземплярах.

Це справедливо не тільки для елементів-даних, але і для методів. Так у результаті компіляції наступної програми виникне помилка неоднозначності виклику методу `X::func()`:

```
class X
{
public:
void func(){cout<<"X::func()"<<endl;}
};

class A: public X
{
};

class B: public X
{
};

class C:public A, public B
{
};

void main()
{
C c;
A*p=&c;
c.func();
}
```

8.1.6. Віртуальне успадкування класів

Щоби існував тільки один екземпляр елементів базового класу при множинному успадкуванні з загальною базою, використовують механізм віртуального успадкування. Для цього при прямому успадкуванні базового класу `X` класами `A` та `B` разом з режимом успадкування вказується режим `virtual`. Відповідну схему віртуального успадкування базового класу зображено на рис. 8.3.

Тепер для доступу до елементів віртуального базового класу не потрібно використовувати операцію доступу через проміжний клас, оскільки вони існують тільки в одному екземплярі. Варіанти звернень до усіх елементів класу `C` подано на рис. 8.3. Звернення до поля `dmx` класу `X` здійснюється безпосередньо за іменем. Для звернення до перекритого поля `dm` необхідно явно вказати базовий клас `X::dm`.

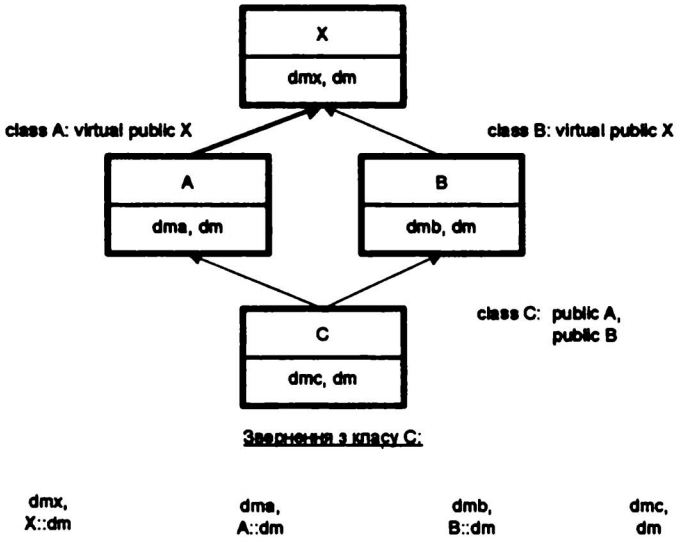

Рис. 8.3. Діаграма множинного успадкування з віртуальним базовим класом

За спроби у класі C додатково виконати успадкування базового класу X виводиться зауваження про те, що клас X вже є базовим класом для A.

На відміну від звичайного успадкування з загальною базою у разі віртуального успадкування об'єкт класу X можна проініціалізувати об'єктом класу C. Наприклад:

```
#include <iostream>

using namespace std;

class X
{
protected:
int dmx;
int dm;
public:
X(int x1=0,int  x2=0):dmx(x1),dm(x2)
       {cout<<"X::X()"<<endl;}
};

class A: virtual public X
{
protected:
int dma;
int dm;
public:
A(int x1,int  x2):dma(x1),dm(x2)
       {cout<<"A::A()"<<endl;}
};
```

```
class B: virtual public X
{
protected:
int dmb;
int dm;
public:
B(int x1,int x2):dmb(x1),dm(x2)
       {cout<<"B::B()"<<endl;}
};

class C:public A, public B
{
int dmc;
int dm;
public:
C(int x1, int x2,
  int x3, int x4,
  int x5, int x6): A(x1,x2), B(x3,x4),
             dmc(x5),dm(x6)
       {cout<<"C::C()"<<endl;}

void output()
{cout<<dmx<<' '     // 0
<<X::dm<<' '        // 0

<<dma<<' '          // 1
<<A::dm<<' '        // 2

<<dmb<<' '          // 3
<<B::dm<<' '        // 4

<<dmc  <<' '        // 5
<<dm<<endl;         // 6
}

};

void main()
{
C    c(1,2,3,4,5,6);
c.output();

X    x=c; // так можна для віртуального базового класу
}
```

Як видно із виведення функції `C::output()`, дані класу `X` потрапили у клас `C` в одному екземплярі: `X::dmx=0`, `X::dm=0` . Елементи даних класу `X` проініціалізовані конструктором з параметрами за замовчуванням.

Якщо віртуальний базовий клас X має хоча б один конструктор, то він повинен мати конструктор без параметрів (void-конструктор), який викликається конструктором класу множинного успадкування C. Конструктори віртуальних базових класів викликаються перед конструкторами невіртуальних базових класів.

При створенні об'єкта класу C будуть викликані конструктори класів у такій послідовності: X::X(), A::A(), B::B(), C::C(). Виклик деструкторів відбудеться у зворотному порядку.

За необхідності ініціалізації даних віртуального базового класу X можна викликати його конструктор у списку ініціалізації конструктора похідного класу C, наприклад:

```
C::C(int x1, int x2, int x3, int x4, int x5, int x6):
       A(x1,x2), B(x3,x4), X(-1, -2), dmc(x5),dm(x6)
       {cout<<"C::C()"<<endl;}
```

Тоді поля базового класу X набудуть значення: X::dmx=-1, X::dm=-2. Це єдина ситуація, коли у класі можна використати конструктор іншого класу, який не є його безпосереднім предком.

Послідовність виклику конструкторів та деструкторів залишиться попередньою.

8.2. Віртуальні методи множинного успадкування

8.2.1. Безпосереднє множинне успадкування

Множинне успадкування класів підтримує механізм пізнього зв'язування за допомогою віртуальних методів, вказівників або посилань на клас. Нехай у базових класах A та B визначено віртуальні методи output(), які виводять на екран значення елементів даних цих класів. Клас C множинно успадковує класи A та B і визначає власний віртуальний метод C::output(). Для підтримання пізнього зв'язування використовується тільки public-успадкування базових класів.

У функції main() посилання на базовий клас A та посилання на базовий клас B ініціалізуються назвою об'єкта класу C і використовуються для виклику віртуального методу output(). Завдяки пізньому зв'язуванню ці посилання вказують на похідний клас C, і за їх допомогою викликається віртуальний метод C::output(). Підтвердженням цього є відповідні виведення на екран.

```
#include <iostream>
#include <typeinfo.h>
```

```
using namespace std;

class A
{
protected:
int dma;
public:
A(int x):dma(x){cout<<"A::A(int)"<<endl;}
virtual void output()
 {cout<<dma<<endl;}
};

class B
{
protected:
int dmb;
public:
B(int x):dmb(x){cout<<"B::B(int)"<<endl;}
virtual void output()
 {cout<<dmb<<endl;}
};

class C:public A, public B
{
int dmc;
public:
C(int x1, int x2, int x3): A(x1),B(x2),dmc(x3)
{cout<<"C::C(int,int,int)"<<endl;}

virtual void output()
 {cout<<dma<<' '<<dmb<<' '<<dmc<<endl;}
};

void main()
{
C c(1,2,3);
A& a=c;
B& b=c;

cout<<typeid(a).name()<<endl;  // class C
cout<<typeid(b).name()<<endl;  // class C

a.output();          // 1  2  3
b.output();          // 1  2  3

c.A::output();       // 1
c.B::output();       // 2

c.output();          // 1  2  3
}
```

8.2.2. Множинне успадкування зі спільним базовим класом

Нехай задано схему множинного успадкування класом C класів A та B, які, своєю чергою, успадковують спільний базовий клас X. Кожен клас визначає віртуальний метод output() для виведення на екран його елементів даних.

Так само, як у попередньому прикладі, посилання на класи A та B, проініціалізовані іменем об'єкта класу C, викликають віртуальний метод C::output() з похідного класу C.

Посилання на клас X не може бути проініціалізоване об'єктом класу C у зв'язку з неоднозначністю перетворення типу, оскільки успадкування класу X відбулося двома шляхами – через клас A і через клас B.

Однак можливим є ланцюжок присвоєнь: посилання на базовий клас X можна проініціалізувати посиланням на клас A або B, які, своєю чергою, ініціалізуються об'єктом класу C. Так проініціалізоване посилання на клас X вказуватиме на клас C і за його допомогою буде викликаний віртуальний метод C::output() класу множинного успадкування C. Наприклад:

```
#include <iostream>
#include <typeinfo.h>

using namespace std;

class X
{
protected:
int dmx;
public:
X(int x):dmx(x){cout<<"X::X()"<<endl;}
virtual void output()
 {cout<<dmx<<endl;}
};

class A: public X
{
protected:
int dma;
public:
A(int x1, int x2):X(x1), dma(x2){cout<<"A::A()"<<endl;}
virtual void output()
 {cout<<dmx<<' '<<dma<<endl;}
};

class B: public X
{
protected:
```

```
int dmb;
public:
B(int x1, int x2):X(x1), dmb(x2){cout<<"B::B()"<<endl;}
virtual void output()
 {cout<<dmx<<' '<<dmb<<endl;}
};

class C:public A, public B
{
int dmc;
public:
C(int x1, int x2, int x3, int x4, int x5):
   A(x1, x2),B(x3, x4),dmc(x5)
 {cout<<"C::C()"<<endl;}

virtual void output()
{cout<<A::dmx<<' '<<dma<<' '<<B::dmx<<' '<<dmb
             <<' '<<dmc<<endl;}
};

void main()
{
C c(1,2,3,4,5);
A& a=c;
B& b=c;

a.output();          // 1  2  3  4  5
b.output();          // 1  2  3  4  5

c.A::output();       // 1  2
c.B::output();       // 3  4

c.output();          // 1  2  3  4  5
//X& x1=c;    // помилка: неоднозначність
             // перетворення типів
//x1.output();

X& x2=a;       // посиланням на клас C
x2.output();         // 1  2  3  4  5

X& x3=b;       // посиланням на клас C
x3.output();         // 1  2  3  4  5
}
```

8.2.3. Множинне успадкування з віртуальним базовим класом

На відміну від попереднього, за множинного успадкування з віртуальним базовим класом посилання на клас X може бути проініціалізоване об'єктом класу C. Неоднозначності перетворення типів не виникає, оскільки успадкування класу X відбулося тільки один раз.

Так проініціалізоване посилання на клас X можна використати для виклику віртуального методу з класу C. Замість посилання можна використати вказівник на клас X, проініціалізований адресою об'єкта класу C.

```
#include <iostream>
#include <typeinfo.h>

using namespace std;

class X
{
protected:
int dmx;
public:
X(int x=0):dmx(x){cout<<"X::X()"<<endl;}
virtual void output()
 {cout<<dmx<<endl;}
};

class A: virtual public X
{
protected:
int dma;
public:
A(int x1, int x2):X(x1), dma(x2){cout<<"A::A()"<<endl;}
virtual void output()
 {cout<<dmx<<' '<<dma<<endl;}
};

class B: virtual public X
{
protected:
int dmb;
public:
B(int x1, int x2):X(x1), dmb(x2){cout<<"B::B()"<<endl;}
virtual void output()
 {cout<<dmx<<' '<<dmb<<endl;}
};

class C:public A, public B
{
```

```
int dmc;
public:
C(int x1, int x2, int x3, int x4, int x5):
   A(x1, x2),B(x3, x4),dmc(x5)
 {cout<<"C::C()"<<endl;}

virtual void output()
{cout<<A::dmx<<' '<<dma<<' '<<B::dmx<<' '<<dmb
       <<' '<<dmc<<endl;}
};

void main()
{
C c(1,2,3,4,5);
A& a=c;
B& b=c;

a.output();          // 0  2  0  4  5
b.output();          // 0  2  0  4  5

c.output();          // 0  2  0  4  5

X& x1=c;
x1.output();         // 0  2  0  4  5

x1.X::output();      // 0
}
```

8.3. Приклад програми

Створити два класи: Рамка (координати головної діагоналі) та Меню (масив рядків символів з назвами команд меню та індекс активної команди меню).

Використовуючи множинне успадкування цих класів, створити новий клас – Вікно з рамкою та меню. Додатково задати назву заголовка вікна та колір фону (рядки символів). Визначити необхідні дані, методи для роботи з даними, конструктори та деструктори, операторні функції введення–виведення даних.

```
#include <iostream>
#include <cstring>

using namespace std;

// клас рамки вікна
class Frame{
```

```
protected:
 int x1, y1, x2, y2; // координати рамки
public:
// конструктор
 Frame(int x1, int y1, int x2, int y2)
 {
 this->x1=x1;
 this->y1=y1;
 this->x2=x2;
 this->y2=y2;
 }
// конструктор копіювання
 Frame(Frame& f)
 {
 x1 = f.x1;
 y1 = f.y1;
 x2 = f.x2;
 y2 = f.y2;
 }
// операція введення з потоку
 friend istream& operator>>(istream& is, Frame& f)
 {is>>f.x1>>f.y1>>f.x2>>f.y2;
 return is;
 }
// операція виведення у потік
 friend ostream& operator<<(ostream& os, Frame& f)
 {os<<f.x1<<' '<<f.y1<<' '<<f.x2<<' '<<f.y2<<endl;
 return os;
 }
};

// клас меню
class Menu{
protected:
 int count;              // кількість команд
 int active;             // номер активної команди
 char ** command;        // назви команд
public:
// конструктор
 Menu(int count, int active, char **command)
 {
 this->count=count;
 this->active=active;

 this->command = new char*[count];
 for(int i=0; i<count; i++)
 {
 this->command[i] = new char[20];
 strcpy(this->command[i], command[i]);
```

```
 }
 }
// конструктор копіювання
 Menu(Menu& m)
 {
 count=m.count;
 active=m.active;

 command = new char*[m.count];
 for(int i=0; i<count; i++)
 {
 command[i] = new char[20];
 strcpy(command[i], m.command[i]);
 }
 }
// деструктор
 ~Menu()
 {for(int i=0; i<count; i++)
  delete []command[i];
 }
// операція введення з потоку
 friend istream& operator>>(istream& is, Menu& m)
 {
 for(int i=0; i<m.count; i++) is>>m.command[i];
 return is;
 }
// операція виведення у потік
 friend ostream& operator<<(ostream& os, Menu& m)
 {
 for(int i=0; i<m.count; i++) os<<m.command[i]<<endl;
 return os;
 }
};

// клас вікна
class Window: public Frame, public Menu
 {
protected:
 char *title;       // заголовок вікна
 int bkcolor;       // колір фону
public:
// конструктор
 Window(Frame f, Menu m, char *title, int bkcolor):
                                  Frame(f), Menu(m)
 {
 this->title = new char[100];
 strcpy(this->title, title);
 this->bkcolor = bkcolor;
 }
```

```
// деструктор
 ~Window()
 {delete []title;}
// операція введення з потоку
 friend istream& operator>>(istream& is, Window& w)
 {
 is>>w.x1>>w.y1>>w.x2>>w.y2;
 for(int i=0; i<w.count; i++)is>>w.command[i];
 is>>w.title;
 is>>w.bkcolor;
 return is;
 }
// операція виведення у потік
 friend ostream& operator<<(ostream& os, Window& w)
 {
 os<<Frame(w.x1, w.y1, w. x2, w.y2);
 os<<Menu(w.count, w.active, w.command);
 os<<w.title<<endl;
 os<<w.bkcolor<<endl;
 return os;
 }
};
// головна функція
void main()
{
char * command[] = {"File", "Edit", "View",
                    "Project","Build","Help"};
enum colors {RED, GREEN, BLUE, WHITE, BLACK};

Frame f(1, 1, 80, 25);
cout<<"Склад об'єкта Frame\n"<<f;

Menu m(6, 0, command);
cout<<"Склад об'єкта Menu\n"<<m;

Window w(f, m, "MyWindow", WHITE);
cout<<"Склад об'єкта Window\n"<<w;
}
```

Розділ 10
ШАБЛОНИ

10.1. Шаблонні функції

10.1.1. Оголошення шаблонних функцій

Шаблонні функції – узагальнене визначення функції, за яким компілятор автоматично згенерує код конкретного екземпляра функції. Шаблонні функції інакше називаються параметризованими, або template-функціями. Використовуються у випадку, коли необхідно виконати обчислення для декількох різних типів даних за одним і тим самим алгоритмом.

Шаблонні функції оголошують за допомогою кваліфікатора template:

```
template < class T1 /*, ... , class Tn*/ >
тип_функції  ім'я_функції ( T1 t1 /* , ... , Tn tn,
                            параметри інших типів*/ );
```

У кутових дужках після слів class вказуються формальні параметри типів. Замість слова class можна використати typename. Параметри типів можуть бути використані для оголошень формальних параметрів та локальних даних функції.

Приклад шаблонної функції для обміну значень двох змінних:

```
template <class T>
void swap(T& t1,T& t2)
{  T t=t1;
   t1=t2;
   t2=t;
}
```

Шаблонна функція може бути перевантажена, наприклад, для циклічного обміну трьох значень:

```
template <class T>
void swap(T& t1,T& t2,T& t3)
{  T t=t1;
   t1=t2;
   t2=t3;
   t3=t;
}
```

Опрацювання даних деяких типів може вимагати спеціальних операцій, наприклад, застосування бібліотечних функцій для роботи з рядками символів. У такому випадку необхідно виконати спеціалізацію шаблонної функції, наприклад:

```
#include <cstring>
void swap(char *t1,char *t2)
{  char t[10];
   strcpy(t,t1);
   strcpy(t1,t2);
   strcpy(t2,t);
}
```

10.1.2. Виклики шаблонних функцій

Під час виклику шаблонної функції її формальні параметри типів набувають значення відповідних фактичних аргументів. Якщо фактичний аргумент є об'єктом класу, то, як правило, виникає необхідність у перевантаженні деяких операцій.

Порядок виклику функцій:

1) знайти нешаблонну функцію, параметри якої мають типи, вказані при виклику;
2) якщо функцію не знайдено, то знайти шаблон з точною відповідністю параметрів (для шаблонних параметрів перетворення типів не відбувається);
3) якщо жоден шаблон не забезпечує точної відповідності типів, то розглянути звичайні функції на предмет можливого неявного перетворення типів.

Приклад викликів шаблонних функцій:

```
void main()
{
// викликається шаблонна функція з двома параметрами
int a=3, b=4;
swap(a,b);
cout<<a<<' '<<b<<endl; // 4 3

float x=5, y=6;
swap(x,y);
cout<<x<<' '<<y<<endl; // 6 5

// викликається перевантажена шаблонна
// функція з трьома параметрами
char c1='1' ,c2='2', c3='3';
swap(c1, c2, c3);
cout<<c1<<' '<<c2<<' '<<c3<<endl; // '2' '3' '1'

// викликається спеціалізована шаблонна функція
char s1[10]="11111",s2[10]="22222";
swap(s1,s2);
cout<<s1<<' '<<s2<<endl;       //"22222"   "11111"
}
```

10.2. Шаблонні класи

Шаблонний клас – це узагальнене визначення множини класів, з якого компілятор автоматично згенерує код конкретного класу. Шаблонні класи інакше називаються параметризованими, або template-класами. Використовуються у випадку, коли програма будується на основі декількох класів з однаковою структурою, які відрізняються типами даних або типами параметрів методів. Шаблонні класи C++ реалізують один із варіантів поліморфізму – поліморфізм типів.

Оголошення шаблонного класу:

```
template <список формальних параметрів шаблону>
class ім'я_класу
{
 протокол класу, який використовує параметри шаблону;
};
```

Параметри шаблону оголошуються одним із двох способів (табл. 10.1).

Таблиця 10.1

Відповідність параметрів шаблону класу

Формальний параметр	Фактичний аргумент
тип ідентифікатор	константа відповідного типу (цілий, переліковий, вказівник, посилання або вказівник на елемент класу). Значення дійсних та класових типів не можуть бути аргументами. Дозволяється вказівник на класовий тип. Константний рядок символів не може бути аргументом шаблону.
class ідентифікатор параметризованого типу	ідентифікатор будь-якого типу

Для прикладу розглянемо шаблонний клас з двома можливими способами оголошення формальних параметрів:

```
template <class T, int k>
class A {
 T *p;
 int size;
public:
 A(int n);
 ~A();
 //...
};
```

Якщо метод шаблонного класу визначається поза протоколом класу, то у заголовку методу вказуються оголошення формальних параметрів, наприклад,

template <class T, int k>, та належність методу параметризованому класу A<T, k>, наприклад:

```
template <class T, int k>
A<T, k>::A(int n)
{p=new T[size=n];
for(int i=0;i<size;i++)
p[i]=k;
}

template <class T, int k>
A<T, k>::~A()
{delete []p;}
```

При оголошенні об'єктів параметризованого класу після назви класу у кутових дужках задається список фактичних аргументів, наприклад:

```
void main()
{
A<int, 0>    a(5);
A<double, 3.14>  b(10);
}
```

Якщо параметризований тип відрізняється значенням фактичного аргумента (константою або назвою типу), то утворюються різні типи. Наприклад, наступні оголошення вказівників ptr1 та ptr2 мають різні базові типи:

```
A<int,0> *ptr1=&a;        // правильно
A<int,100> *ptr2=&a;      // помилка: несумісність типів
```

Вказівник ptr1 можна проініціалізувати адресою об'єкта a, оскільки їхні базові типи є тотожними. Але вказівник ptr2 та об'єкт a мають різні базові типи, тому ініціалізація вказівника ptr2 адресою об'єкта a не дозволяється.

Конкретний тип шаблонного класу у програмі можна визначити за допомогою операції typeid, яка здійснює динамічну ідентифікацію типів, наприклад

```
A<int, 0>    a(5);
cout<<typeid(a).name()<<endl;  // A<int, 0>
```

Параметри шаблону класу можуть набувати значення за замовчуванням. Якщо один з параметрів набуває значення за замовчуванням, то усі наступні повинні бути параметрами за замовчуванням. Наприклад:

```
template <class T=int, int k=0>
class A {
 T *p;
 int size;
public:
```

```
 A(int n);
 ~A();
 //...
};
```

Тоді при оголошенні об'єкта фактичні аргументи шаблону, які набувають значення за замовчуванням, можна не задавати, наприклад:

```
void main()
{
A<>  a(5);
A<unsigned>  b(10);
A<short, 7>  c(20);
}
```

Вибіркове задання параметрів за замовчуванням не дозволяється.

Параметри типу можуть набувати стандартних значень або бути типами, визначеними користувачем. Можна визначити параметр-посилання на змінну заданого типу:

```
template <int & ref>
 class X
{
public:
   void print()
   {
      cout<<ref<<endl;
   }

};

int  n = 1;

int main()
{
   X<n> x;
   x.print();
}
```

Допускається вкладеність шаблонів при оголошенні об'єктів параметризованих класів:

```
template <class T>
class X
{
};

template <int n>
class Y
{
```

```
};

int main()
{
 X<Y<5>> x;  // вкладені шаблони
}
```

Окремі методи параметризованого класу допускають спеціалізацію типу, наприклад:

```
A<float, 0>::A(int n)
{p=new float[size=n];
for(int i=0;i<size;i++)
p[i]=0;
}
```

Можна визначити спеціалізацію усього класу для специфічних параметрів типів, наприклад, для масиву рядків символів. Спеціалізація класу разом з усіма перевизначеними методами розміщується після шаблонного класу. Спеціалізований шаблонний клас оголошується за допомогою специфікації `template <>`:

```
template <> class A<char *> {
 char** p;
 int size;
public:
 A(int n);
 ~A();
 //...
};

A<char*>::A(int n)
{p=new char*[size=n];
for(int i=0;i<size;i++)
p[i]=new char[100];
}

A<char*>::~A()
{
for(int i=0;i<size;i++)
 delete []p[i];

delete []p;
}
```

Відповідно до раніше зробленого оголошення шаблонного класу `A<T, k>`, у цьому прикладі другий параметр спеціалізованого шаблону набуває значення за замовчуванням, яке не використовується.

Шаблонні класи не допускають перевантаження. Наприклад, для класу

```
template <class T>  class A{};
```

не допускається таке перевантаження:

```
template <class T1, class T2>
class A {
 T1 *p;
 int size;
public:
 A(int n);
 ~A();
 //...
};
```

Методи шаблонного класу можуть бути шаблонними функціями:

```
template <class T1, class T2>
class X
{
public:
   template <class U, class V>
     void mf(const U &u, const V &v){}
};
```

Визначаючи шаблонний метод поза класом, необхідно повторити template-оголошення для кожної групи параметрів типу:

```
template <class T1, class T2>
template <class U, class V>
      void X<T1, T2>::mf(const U &u, const V &v){}
```

Локальні класи не можуть бути шаблонними та не можуть мати template-елементів.

У межах класу дозволяється спеціалізація шаблонних методів та перевантаження шаблонних і нешаблонних методів, наприклад:

```
template <class T>
class A
{
 T t;
public:
 A(T t){this->t = t;}
 void func(){}
 void func(T& t){}
 template <class X>
    void func(X &x){}
 template <class X, class Y>
    void func(X &x, Y& y){}
};
```

```
void main()
{
A<int> a(5);
int x=7;
a.func(x);

float y=3.14;
a.func(y);
}
```

10.3. Вкладені шаблонні класи

Шаблонні класи можуть бути вкладеними у звичайні або інші шаблонні класи.

Приклад вкладення шаблонного класу у нешаблонний клас:

```
class X
{
public:
   template <class T>
   class Y
   {public:
      T t;
      Y(T t1): t(t1) { }
   void print()
   { cout<<t<<endl;}
   };

   Y<int> y_int;
   Y<char> y_char;

   X(int i, char c) : y_int(i), y_char(c) { }
   void print()
   {
      cout << y_int.t << " " << y_char.t << endl;
   }
};

void main()
{
   X x(5, '*');     // об'єкт зовнішнього класу
   x.print();

   X::Y<int> y(7); // об'єкт внутрішнього класу
   y.print();
}
```

Методи вкладеного класу можна визначити поза охоплюючим класом:

```
template <class T>
 X::Y<T>::Y(T t1): t(t1) { }
```

Приклад вкладення шаблонного класу у шаблонний клас:

```
template <class T>
class X
{
public:
 template <class U>
   class Y
   {
      U u;
   public:
   Y(){}
      Y(U u1);
     ~Y();
      void print();
   };

   Y<T> y;

   X(Y<T> t) { y = t; }
   void print() { y.print(); }
};

template <class T>
template <class U>
X<T>::Y<U>::Y(U u1)
{
   u = u1;
}

template <class T>
template <class U>
void X<T>::Y<U>::print()
{
   cout << u << endl;
}

template <class T>
template <class U>
X<T>::Y<U>::~Y()
{
}

void main()
{
```

```
// об'єкти зовнішнього класу у динамічній пам'яті
   X<int>* ptr1 = new X<int>(5);
   X<char>* ptr2 = new X<char>('*');
   ptr1->print();
   ptr2->print();
// об'єкт внутрішнього класу у динамічній пам'яті

   X<int>::Y<float>*ptr3 = new X<int>::Y<float>(3.14);
   ptr3->print();
 // вилучення об'єктів з динамічної пам'яті
   delete ptr1;
   delete ptr2;
   delete ptr3;
}
```

10.4. Статичні елементи шаблонних класів

Поля статичних даних є спільними для усіх об'єктів конкретного екземпляра шаблонного класу (об'єктів з однаковими значеннями аргументів шаблону).

Статичні дані визначаються у тому самому файлі, де було зроблено оголошення шаблонного класу.

Наприклад:

```
template <class T>
class A
{public:
   static int x;
   static T y;
public:
    void print(){cout<<x<<' '<<y<<endl;}
    void inc(){x++; y++;}
};

template <class T> int A<T>::x=5;
template <class T> T A<T>::y=7;

void main()
{
A<int> a;
a.print();   // 5  7
cout<<A<int>::x<<' '<<A<int>::y<<endl;     // 5  7

a.inc();
a.print();   // 6  8
cout<<A<int>::x<<' '<<A<int>::y<<endl;     // 6  8
```

```
A<float> b;
b.print();    // 5  7
cout<<A<int>::x<<' '<<A<int>::y<<endl;     // 6  8
cout<<A<float>::x<<' '<<A<float>::y<<endl; // 5  7
}
```

Як видно з результатів роботи програми, зміна статичних полів об'єкта класу `A<int>` не впливає на значення статичних полів об'єкта класу `A<float>`.

10.5. Друзі шаблонних класів

10.5.1. Дружні зовнішні функції

Друзями шаблонного класу можуть бути зовнішні функції. Якщо дружня функція використовує об'єкт шаблонного класу, то перед її заголовком необхідно розмістити `template`-оголошення. Наприклад:

```
template <class T>
class A{
 friend void f1();
 template <class T>
       friend void f2(A<T>&);
 template <class U>
       friend void f3(U&);
};

// звичайна функція, друг множини класів, відношення 1:n
void f1() { }

// функція-друг одного класу типу T, відношення 1:1
template <class T>
void f2(A<T>& a) { }

// параметризована дружня функція, відношення m:n
template <class U>
void f3(U& u) { }

void main()
{
 f1();
 A<int> a;
 f2(a);
 double x;
 f3(x);
}
```

Звичайна функція `f1()` є дружньою до всіх класів `A<T>` для усіх типів `T`, наприклад:

`f1()` —*friend*→ `A<int>, A<char>, A<float>, ...`

Тут стрілка позначає відношення між функцією і шаблонними класами.

Функція `f2(A<T>&)` є дружньою до класу `A<T>` для кожного конкретного типу `T`:

`f2(A<int>&)` —*friend*→ `A<int>`
`f2(A<char>&)` —*friend*→ `A<char>`
`f2(A<float>&)` —*friend*→ `A<float>`
`...`

Для кожного `U` шаблонна функція `f3(U&)` є дружньою до класу `A<T>` для кожного типу `T`:

`f3(int&)` —*friend*→ `A<int>, A<char>, A<float>, ...`
`f3(char&)` —*friend*→ `A<int>, A<char>, A<float>, ...`
`f3(float&)` —*friend*→ `A<int>, A<char>, A<float>, ...`

10.5.2. Дружні методи класів

Друзями шаблонного класу можуть бути методи інших класів, наприклад:

```
template <class T> class A;

template <class T>
class D{
public:
template <class U>
  void f3(U& u) // шаблонний метод шаблонного класу
    { }
};

template <class T>
class C{
public:
void f2(A<T>& a) // звичайний метод шаблонного класу
  { }
};

class B{
public:
```

```
    void f1()     // звичайний метод нешаблонного класу
      { }
};

template <class T>
class A{
 friend void B::f1();
 friend void C::f2(A<T>&);
 template <class T>
    template <class U>
       friend void D<U>::f3(U&);
};

void main()
{
}
```

Функція `B::f1()` є дружньою до `A<T>` для усіх типів `T`:

`B::f1()` —*friend*→ `A<int>, A<char>, A<float>, ...`

Функція `C<T>::f2(A<T>&)` є дружньою до класу `A<T>` для кожного конкретного типу `T`:

`C<int>::f2(A<int>&)` —*friend*→ `A<int>`
`C<char>::f2(A<char>&)` —*friend*→ `A<char>`
`C<float>::f2(A<float>&)` —*friend*→ `A<float>`
`...`

Для кожного `U` функція `D<U>::f3(U&)` є дружньою до класу `A<T>` для кожного типу `T`:

`D<int>::f3(int&)` —*friend*→ `A<int>, A<char>, A<float>, ...`
`D<char>::f3(char&)` —*friend*→ `A<int>, A<char>, A<float>, ...`
`D<float>::f3(float&)` —*friend*→ `A<int>, A<char>, A<float>, ...`

10.5.3. Дружні класи

Друзями шаблонного класу можуть бути цілі класи, наприклад:

```
template <class U>
class D{
};

template <class T>
class C{
};
```

```
class B{
};

template <class T>
class A{
 friend class B;
 friend class C<T>;
 template <class U>
    friend class  D;
};

void main()
{
}
```

Усі методи класу B є дружніми до A<T> для усіх типів T:

B —*friend*→ A<int>, A<char>, A<float>, ...

Усі методи класу C<T> є дружніми до класу A<T> для кожного конкретного типу T:

C<int> —*friend*→ A<int>
C<char> —*friend*→ A<char>
C<float> —*friend*→ A<float>
...

Для кожного U усі методи класу D<U> є дружніми до класу A<T> для кожного типу T:

D<int> —*friend*→ A<int>, A<char>, A<float>, ...
D<char> —*friend*→ A<int>, A<char>, A<float>, ...
D<float> —*friend*→ A<int>, A<char>, A<float>, ...

10.6. Перевантаження операцій шаблонних класів

Під час роботи з шаблонними класами часто виникає необхідність у перевантаженні операцій. Для прикладу розглянемо перевантаження бінарної операції operator+ як члена класу:

```
template <class T>
class A
{  T x;
public:
    A(){}
```

```
    A(T x){this->x=x;}
    A<T> operator+(A<T>&);
    void print(){cout<<x<<endl;}
};

template <class T>  A<T> A<T>::operator+(A<T>& a)
{
return A<T>(x+a.x);
}

void main()
{
A<int> a(2), b(3), c;
c=a+b;
c.print();
}
```

Операторні функції можуть бути друзями класу. Наведемо приклад застосування дружніх функцій для перевантаження бінарної операції operator- та операцій потокового введення–виведення об'єкта шаблонного класу:

```
template <class T>
class A
{       T x;
public:
        A(){}
        A(T x){this->x=x;}
 template <class T>
        friend A<T> operator-(A<T>&, A<T>&);
 template <class T>
        friend istream& operator>>(istream&, A<T>&);
 template <class T>
        friend ostream& operator<<(ostream&, A<T>&);
};

template <class T>
 A<T> operator-(A<T>& a, A<T>& b)
{
A<T> c;
c.x=a.x-b.x;
return c;
}

template <class T>
 istream& operator>>(istream& is, A<T>& a)
 {
 is>>a.x;
 return is;
 }
```

```
template <class T>
 ostream& operator<<(ostream& os, A<T>& a)
 {
 os<<a.x<<endl;
 return os;
 }

void main()
{
A<int> a(2), b(3), c;
c=a-b;
cout<<"Результат a-b"<<endl;
cout<<c;
}
```

Незважаючи на те, що друзі не належать шаблонному класу, вони використовують його параметри.

Операторні методи та операторні друзі класу можуть бути шаблонними, наприклад:

```
template <class T>
class A{
 T x;
public:
 A(){}
 A(T x1){x=x1;}
 // операторний метод додавання
 A<T> operator+ <>(A<T>);
 // дружня операторна функція віднімання
 friend  A<T> operator- <>(A<T>&, A<T>&);
 // дружня операторна функція потокового виведення
 friend ostream& operator<< <>(ostream&, A<T>&);
};

template <class T>
  A<T> A<T>::operator+(A<T> a)
 {
 return A<T>(x+a.x);
 }

 template <class T>
  A<T> operator-(A<T>& a1, A<T>& a2)
  {
  return A<T>(a1.x-a2.x);
  }

  template <class T>
  ostream& operator<<(ostream& os, A<T>& a)
   {
```

```
        os<<a.x<<endl;
        return os;
    }

void main()
{
 A<int> a(7), b(5), c;
 c=a+b;
 cout<<c<<endl;    // 12
 c=a-b;
 cout<<c<<endl;    // 2
}
```

Для позначення того, що операторна функція є шаблонною, використовується пара кутових дужок <> з порожнім вмістом.

У шаблонному класі можна визначити операцію перетворення типу:

```
template <class T>
class A
{  T x;
public:
    A(){}
    A(T x){this->x=x;}
    operator T();
    void print(){cout<<x<<endl;}
};

template <class T>  A<T>::operator T()
{
return x;
}

void main()
{
A<int> a(5);
int x=a;
cout<<x<<endl;
}
```

Аналогічно можна визначити операцію перетворення одного параметризованого типу до іншого:

```
template <class T>
class A
{  T x;
public:
    A(){}
    A(T x){this->x=x;}
    void print(){cout<<x<<endl;}
};
```

```
template <class T>
class B
{  A<T> x;
public:
    B(T x=0){this->x=x;}
    operator A<T>();
    void print(){x.print();}
};

template <class T>  B<T>::operator A<T>()
{
return x;
}

void main()
{
B<int> b(65);
A<char> a=b;
a.print();
}
```

Операція перетворення типу може бути шаблонним методом класу:

```
template <class T>
class S
{
public:
template <class U> operator S<U>()
{
 return S<U>();     // викликається конструктор класу S<U>
}
};

void main()
{
   S<int> s1;
   S<long> s2 = s1;       // перетворення типу S<int>
                          // до типу S<long>
}
```

10.7. Успадкування шаблонних класів

10.7.1. Одинарне успадкування шаблонних класів

Шаблонні класи допускають успадкування. Шаблонний клас може успадковувати нешаблонний, шаблонний клас – шаблонний, нешаблонний клас – шаблонний. Можливі варіанти успадкування шаблонних класів демонструє така програма:

```
// нешаблонний клас
class A
{
protected:
  int x;
public:
  A(int x=0){this->x=x;}
};

// шаблонний клас
template <class T>
class B
{
protected:
  T x;
public:
  B(){}
  B(T x){this->x=x;}
};

//шаблон : нешаблон
template <class T>
 class C:  public A
 {
public:
  C(){}
  C(T x):A(x){}
 };

// шаблон : шаблон
template <class T>
 class D:  public B<T>
 {
public:
 D(){}
 D(T x):B<T>(x){}
 };
```

```
// нешаблон : шаблон
class E:  public B<int>
 {
public:
  E(int x=0):B<int>(x){}
 };

void main()
{
C<int> c;
D<int> d;
E e;
}
```

Нешаблонний клас може успадкувати тільки конкретну реалізацію шаблонного класу.

10.7.2. Множинне успадкування шаблонних класів із загальною базою

При множинному успадкуванні шаблонних класів із загальною базою елементи базового класу потрапляють у фінальний похідний клас кожним шляхом їх успадкування. Для звернення до таких елементів використовується ім'я класу, через який відбулося успадкування.

Для існування тільки одного екземпляра елементів базового класу у фінальному похідному класі множинного успадкування необхідно виконати віртуальне успадкування спільного базового класу. Наприклад:

```
template <class T>
class X
{ protected:
  T x;
public:
  X(T x){this->x=x;}
  void print(){cout<<x<<endl;}
};

template <class T>
class A: virtual public X<T>
{ protected:
  T y;
public:
  A(T x, T y):X<T>(x){this->y=y;}
  void print(){cout<<x<<' '<<y<<endl;}
};

template <class T>
```

```
class B: virtual public X<T>
{ protected:
  T y;
public:
  B(T x=0, T y=0):X<T>(x){this->y=y;}
  void print(){cout<<x<<' '<<y<<endl;}
};

template <class T>
class C: public A<T>, public B<T>
{
  T z;
public:
  C(T x, T y, T z): X<T>(x), A<T>(x,y), B<T>(x,y)
    {this->z=z;}
  void print(){cout<<X<T>::x<<' '

          <<A<T>::y<<' '

          <<B<T>::y<<' '

          <<z<<endl;}
};

void main()
{
C<int> c(1,2,3);
c.print();          // 1   2   2   3
}
```

Клас множинного успадкування не може виконати безпосереднє успадкування базового класу. Однак, при віртуальному успадкуванні базового класу конструктор фінального класу множинного успадкування повинен викликати конструктор базового класу або має існувати і бути доступним конструктор без параметрів.

10.8. Віртуальні методи шаблонних класів

Шаблонний клас може містити оголошення віртуальних методів. Шаблонні методи (методи з оголошенням `template`) не можуть бути віртуальними.

Віртуальні методи забезпечують пізнє зв'язування, якщо вони викликаються за допомогою вказівників (або посилань) на базовий клас, проініціалізованих адресою (або ідентифікатором об'єкта – для посилань) похідного класу з `public`-

успадкуванням. Для шаблонних класів вказівник (або посилання) на базовий клас та об'єкт похідного класу повинні мати однакову параметризацію.

Приклад поліморфізму віртуальних методів шаблонних класів:

```
#include <iostream>
#include <typeinfo.h>

using namespace std;

// базовий шаблонний клас
template <class T>
class B
{
protected:
  T x;
public:
  B(){}
  B(T x){this->x=x;}
 virtual void func() {cout << typeid(*this).name()
                        << "::func()" << endl;}
};

// похідний шаблонний клас
template <class T>
 class D:  public B<T>
 {
public:
 D(){}
 D(T x):B<T>(x){}
 virtual void func(){cout << typeid(*this).name()
                        << "::func()" <<endl;}
 };

void main()
{
B<int>* p = new D<int>;
p->func();          // class D<int>::func()

B<double> *q = new D<double>;
q->func();          // class D<double>::func()
}
```

Вказівник на базовий шаблонний клас `B<int>`, проініціалізовано адресою об'єкта похідного шаблонного класу `D<int>`. Цей вказівник використано для виклику віртуального методу `func()`. Завдяки пізньому зв'язуванню буде викликано віртуальний метод похідного класу `D<int>`.

Відповідно, вказівник на клас `B<double>`, проініціалізований адресою об'єкта похідного класу `D<double>`, викличе віртуальний метод з класу `D<double>`.

10.9. Приклад програми

Створити шаблонний клас `vector` *з двома параметрами: перший є параметром типу, а другий – цілочисловим значенням для ініціалізації елементів вектора. Визначити конструктор, деструктор, методи пошуку екстремальних значень (мінімуму та максимуму), впорядкування, обчислення евклідової норми. Перевантажити операції* `+, =, [], <<, >>` *відповідно для додавання, присвоєння, контролю діапазону індексу, виведення та введення об'єктів.*

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <math.h>

using namespace std;

template <class T, int k> class vector
  {
    T *v;
    int size;

    public:
    vector(int newsize);
   ~vector();

    T  extr(char *);
    vector& sort(char *);
    double  norma(void);

    vector& operator+(vector&);
    T&   operator[](int index);
    vector& operator=(const vector&);

    template <class T>
       friend ostream& operator<<(ostream&, vector&);
     template <class T>
       friend istream& operator>>(istream&, vector&);
  };

    template <class T,int k>
       vector<T,k>::vector(int newsize)
    {
      v=new T[size=newsize];

      for(int i=0;i<size;i++)
       v[i]=k;
    }
```

```
   template <class T, int k>
vector<T,k>::~vector()
   {
   delete[] v;
   }

    template <class T, int k>
      T vector<T,k>::extr(char * MinOrMax)
   {
    T  ExtrElem=v[0];
    for(int i=0;i<size;i++)
    {if(!_strcmpi(MinOrMax,"min"))
   {if(v[i]<ExtrElem) ExtrElem=v[i];}
     else
     if(v[i]>ExtrElem) ExtrElem=v[i];
    }
     return ExtrElem;
   }

   template <class T, int k>
      vector<T,k>& vector<T,k>::sort(char * UpOrDown)
   { T x;
     for(int i=0;i<size-1;i++)
      for(int j=i+1;j<size;j++)
      if(!_strcmpi(UpOrDown,"up"))
{if(v[i]>v[j]) { x=v[i];
             v[i]=v[j];
             v[j]=x;
             }
}
else
 if(v[i]<v[j]) { x=v[i];
             v[i]=v[j];
             v[j]=x;
             }
return *this;
    }

   template <class T, int k>
      double vector<T,k>::norma(void)
   { double s=0;
    for(int i=0;i<size;i++)
     s+=v[i]*v[i];
    return sqrt(s);
   }

  template <class T,int k>
      vector<T,k>& vector<T,k>::operator+(vector<T,k>& y)
   {
```

```
 if(size!=y.size) false;
 for(int i=0;i<size;i++)
 v[i]+=y.v[i];
 return *this;
    }

   template <class T, int k>
       T& vector<T,k>::operator[](int index)
    {
     if(index<0 || index>=size)
      throw "Індекс за межами діапазону можливих значень";

      return v[index];
    }

template <class T, int k>
    vector<T,k>& vector<T,k>::operator=(
                                const vector<T,k>& x)
    {
     if(this!=&x)
     {
      delete[]v;
      v=new T[size=x.size];
      for(int i=0;i<size;i++)
 v[i]=x.v[i];
     }
     return *this;
    }

 template <class T>
    istream& operator>>(istream& is, vector<T,0>& x)
    {
     cout<<"Введіть "<<x.size<<" елементів вектора\n";
     for(int i=0;i<x.size;i++)
       is>>x.v[i];
       return is;
    }

template <class T>
    ostream& operator<<(ostream& os, vector<T,0>& x)
{
     for(int i=0;i<x.size;i++)
      os<<x.v[i]<<' ';
      os<<endl;
      return os;
}

    void main()
 {
```

```
int n=5;
vector<int,0> V(n),U(n);

vector<int,0> *Z=new vector<int,0>(n);

try{
cin>>V;

cin>>U;

*Z=V+U;
cout<<*Z;

cout<<Z->extr("min")<<endl;

Z->sort("up");
cout<<*Z;

cout<<Z->norma()<<endl;
}
catch(bool)
{
cout<<"Різна кількість елементів векторів"<<endl;
}
catch(char *s)
{
 cout<<s<<endl;
}
}
```