Об’єктно-орієнтоване програмування
Конспект лекцій
О.В. Коба, С.В. Пустова

1. Поняття про клас

Клас *(class)* – це складний тип даних, що визначається користувачем. Він може складатися з як з елементів-даних (змінних), так і функціональних даних (функцій, методів), які реалізують операції над ними.

Синтаксис описання класу наступний (*прим*. – квадратні дужки означають, що даний елемент може бути відсутнім в описанні класу):

```
class ім’я_класу
{ [private:]
  //оголошення закритих функцій та змінних класу
  [public:]
  //оголошення відкритих функцій та змінних класу
 } [список_об’єктів];
```

Наприклад:

```
class myclass //оголошення класу myclass
{ private:
  int x; // закрита змінна типу int
  public:
  void show(); // відкрита функція-член класу
};// крапка з комою є обов’язковою при завершенні оголошення класу
```

2. Відкриті та закриті члени класу

Функції та змінні, оголошені всередині класу, є членами *(members)* цього класу. Функції, оголошені всередині класу, називаються функціями-членами *(member functions)* або методами.

Члени класу можуть бути закритими *(private)* або відкритими *(public)*. Для оголошення закритих членів класу використовується ключове слово *private*. Закриті члени класу доступні для інших членів цього класу. Для оголошення відкритих членів класу використовується ключове слово *public*. До відкритих членів класу мають доступ як інші члени цього класу, так будь-яка інша частина програми. За умовчанням *(by default)* члени класу є закритими.

Методи оголошуються всередині класу, а для їх визначення треба зв’язати ім’я даного класу з ім’ям цього методу:

```
тип_даних_що_повертає_функція ім’я_класу :: ім’я_функції ([параметри_функції])
 {
   // тіло функції
  }
```

Наприклад, для функції *showvolume* класу *myclass*:

```
void myclass:: show(){
cout << “x=” << x <<“\n”;
}
```

Подвійною двокрапкою визначається операція розширення області бачення *(scope resolution operator)*.

Об’єкт – це змінна типу клас. Оголошення класу визначає лише тип об’єкта, який буде створено при його фактичному оголошенні. Для створення об’єкта використовується ім’я класу як специфікатор типу даних:

ім’я_класу список_об’єктів;

Тут *список_об’єктів* може складатися як з одного імені об’єкта, так і з кількох, розділених комами.

Наприклад:

myclass ob1, ob2;

litak ob3;

Оголошення класу є логічною абстракцією, яка задає новий тип даних для об’єкта, а оголошення об’єкта створює фізичну суть об’єкта такого типу (тобто при оголошенні об’єкта виділяється пам’ять, а при оголошенні класу – ні).

Доступ до відкритих членів класу можна отримати лише через об’єкт цього класу. Для доступу до відкритих членів класу використовують операцію доступу крапка (.):

ім’я_об’єкту.ім’я_функції([параметри_функції]);

Наприклад:

ob1.show(); // виклик функції *show* класу *myclass*

Для ініціалізації об’єктів класу використовують конструктори.

3. Поняття про конструктор та деструктор

Конструктор *(constructor function)* – функція-член, яка включається в описання класу і має те саме ім’я, що і клас. Конструктор створює значення типу клас. Цей процес включає в себе ініціалізацію членів-данних і, часто, розподіл вільної пам’яті. Конструктор класу викликається при створенні об’єкта автоматично. Конструктор не повертає значення.

Оголошення конструктора:

```
class ім’я_класу
{[private:]
 public:
    ім’я_класу([список_параметрів]); //конструктор
};
```

Наприклад:

```
class myclass
{int x;
 public:
    myclass(int a); //конструктор
};
```

Для глобальних об’єктів конструктор об’єкта викликається тоді, коли починається виконання програми. Для локальних об’єктів конструктор викликається кожного разу при виконанні оператора, що оголошує змінну.

Функція, обернена до конструктора, є деструктор *(destructor)*. Ця функція викликається при знищенні об’єкта. Описання деструктора задається символом ~ (тильда) з наступним ім’ям класу:

```
class ім’я_класу
{[private:]
 [public:]
    ~ім’я_класу(); //деструктор
};
```

Наприклад:

```
class myclass
{int x;
 public:
    ~myclass(); //деструктор
};
```

Локальні об’єкти знищуються тоді, коли вони виходять з області бачення. Глобальні об’єкти знищуються при завершенні програми.

Неможливо отримати вказівники на конструктор та деструктор.

4. Конструктори з параметрами

Конструктору можна передавати аргументи. Для цього просто додаються необхідні параметри в оголошення та визначення конструктора. Потім, при оголошенні об’єкта параметри задаються як аргументи:

ім’я_класу ім’я_об’єкта([список_параметрів]);

Фактично синтаксис передачі аргумента конструктору з параметром є скороченою формою запису наступного більш довгого виразу:

ім’я_класу ім’я_об’єкта = ім’я_класу ([список_параметрів]);

На відміну від конструктора, деструктор не може мати параметрів. Зміст цього зрозуміти досить просто: відсутній механізм передачі аргументів об’єкту, що знищується.

5. Поняття поліморфізму.

Поліморфізм – це процес, завдяки якому загальний інтерфейс застосовується до двох або більше схожих (але технічно разних) ситуацій, тобто реалізується філософія “один інтерфейс, багато методів”.

6. Перевантаження функцій.

Після класів важливою можливістю C++ є перевантаження функцій (function overloading). Це той механізм, завдяки якому в C++ досягається один з видів поліморфізму.

В C++ дві або більше функцій є перевантаженими, якщо вони мають одне й те саме ім’я, відрізняючись або типом, або кількістю аргументів, або і тим, і іншим.

Щоб перевантажити функцію, треба оголосити та задати всі варіанти, які можуть знадобитися. Компілятор автоматично вибере правильний варіант виклику на основі кількості та/або типу аргументів, що використовуються в функції.

Приведемо приклади перевантаження функцій. В першому з них перевантажені функції відрізняються типом аргументів, в другому - їх кількістю.

7. Перевантаження конструкторів

Можна перевантажувати конструктор. Деструктор перевантажувати не можна. Кожному способу оголошення об’єкта класу повинна відповідати своя версія конструктора класу.

Найбільш часте використання перевантаження конструктора – це забезпечення вибору: ініціалізувати об’єкт чи ні. Приведемо приклад.

Перевантаження конструкторів також традиційно застосовується при підтримці масивів. Для співіснування в програмі неініціалізованих масивів об’єктів поряд з

ініціалізованими використовується конструктор, який підтримує ініціалізацію та конструктор який її не підтримує. Ці ж конструктори можна використовувати для ініціалізації, або неініціалізації об’єктів. Наприклад, для класа *myclass* з попереднього приклада вірні два ці оголошення:

myclass ob(10);

myclass ob[5];

Так як існують конструктори, що використовуються і для ініціалізації, і при її відсутності, змінні можна, за мірою необхідності, ініціалізувати або ні.

8. Конструктор копіювання

Однією з найважливіших форм перевантаженого конструктора є *конструктор копій (copy constructor)*.

Коли об'єкт передається у функцію, робиться поразрядна (тобто точна) копія цього об'єкта й передається тому параметру функції, який отримує об'єкт. Однак бувають ситуації, у яких така точна копія об'єкта небажана. Наприклад, якщо об'єкт містить вказівник на виділену область пам'яті, то в копії вказівник буде посилатися на ту ж саму область пам'яті, на яку посилається вихідний вказівник. Отже, якщо копія змінює вміст області пам'яті, то ці зміни торкнуться також і вихідного об'єкта! Крім того, коли виконання функції завершується, копія видаляється, що приводить до виклику деструктора цієї копії. Подібна ситуація має місце, коли об'єкт є типом значення, який повертає функція.

Шляхом визначення конструктора копіювання можна повністю контролювати весь процес створення копії об'єкта.

Важливо розуміти, що в C++ точно виділяють два типи ситуацій, у яких значення одного об'єкта передається іншому. Перша ситуація – це присвоювання. Друга – ініціалізація, яка може мати місце в трьох випадках:

а) коли в інструкції оголошення об'єкта один об'єкт використовується для ініціалізації іншого;

б) коли об'єкт передається у функцію як параметр;

в) коли в якості значення, що повертається функцією, створюється тимчасовий об'єкт.

Конструктор копій використовується лише для ініціалізації, але не для присвоювання. Після того як конструктор копій визначений, він викликається завжди при ініціалізації одного об'єкта іншим.

Основна форма конструктора копій:

ім'я_класу (const ім'я_класу &obj)

{

// тіло конструктора

}

Тут *obj* – це посилання на об'єкт, призначений для ініціалізації іншого об'єкта. Наприклад, нехай є клас *myclass*, а *y* – це об'єкт типу *myclass*, тоді наступні інструкції могли б викликати конструктор копій *myclass*:

myclass x=y; // *y* явно ініціалізує *x*

fund (y); // *y* передається як параметр

y=func2(); // *y* отримує объект, що повертається

У двох перших випадках конструкторові копій можна було б передати посилання на об'єкт *y*. В останньому випадку конструкторові копій передається посилання на об'єкт, що повертається функцією *func*().

9. Використання аргументів за умовчанням

Можливість використання аргументів за умовчанням (default argument) зв'язана з перевантаженням функцій та дозволяє при виклику функції відповідний аргумент не задавати, а надати параметру значення за умовчанням.

Щоб передати параметру аргумент за умовчанням, треба поставити за параметром знак рівності і те значення, які ви хочете передати. Тоді, якщо при виклику функції відповідний аргумент не задано, то за умовчанням функції буде передано задане вами значення. Наприклад, в приведеній функції двом параметрам за умовчанням присвоєно значення 0:

void f (int a=0, int b=0)

Тепер цю функцію можна викликати трьома різними способами. По-перше, вона може викликатися з двома заданими аргументами. По-друге, вона може викликатися тільки з першим заданим аргументом. В цьому випадку, *b* за умовчанням буде дорівнювати нулю. Нарешті, *f( )* може викликатися без будь-яких аргументів, при цьому *a* та *b* за умовчанням, будуть дорівнювати нулю. Таким чином, всі наступні виклики *f( )* вірні:

f( );// *a=0 і b=0* за умовчанням

f(10);// *a*=10, *b=0* за умовчанням

f(10, 99);// *a*=10, *b*=99

Коли ви створюєте функції, що мають один або більше аргументів, що передаються за умовчанням, ці аргументи повинні задаватися тільки один раз: або у визначенні функції, або в її прототипі.

Всі параметри, що задаються за умовчанням повинні знаходитись правіше параметрів, що передаються звичайним шляхом. Більш того, якщо ви вже почали задавати параметри за умовчанням, то вже не можна передавати параметри звичайним шляхом.

Аргументи за умовчанням повинні бути або константами, або глобальними параметрами.

Приведемо приклад використання аргументів за умовчанням при обчисленні площі прямокутника, або квадрата за вибором. Зверніть увагу, що при цьому використовується одна функція.

Зробимо ще два зауваження стосовно використання аргументів за умовчаннями:

а) можливо передавати аргументи за умовчанням конструкторам;

б) гарним прикладом використання аргументів за умовчанням є випадки, коли треба зробити вибір варіанта.

10. Основи перевантаження операторів

Перевантаження операцій (множення, ділення, віднімання, додавання тощо) одна з важливих особливостей C++. Ця властивість дозволяє визначати зміст тих операторів C++, які зв'язані з заданими класами. Шляхом перевантаження зв'язаних з класами операторів, можна легко задавати в програмі нові типи даних. Перевантаження операторів нагадує перевантаження функцій. Більше того, перевантаження операторів, є, фактично, одним з видів перевантаження функцій. Проте при цьому вводяться деякі додаткові правила. Наприклад, оператор, що перевантажується завжди зв'язаний з класом.

Коли оператор перевантажується, то нічого з його вихідного значення не губиться. Навпаки він отримує додатковий зміст, зв'язаний з класом, для якого він задається.

Для перевантаження оператора задається оператор-функція (operation function). Частіше оператор-функція є членом класу або дружньою функцією класу, для якого вона задана.

Загальна форма оператор-функції - члена класу має вигляд:

```
Тип_що_повертається ім'я_класу::operator#(список_аргументів)
    {
    //дії, що виконуються
    }
```

Частіше типом, що повертається оператором - функцією є клас, для якого вона задана. (Проте оператор-функція може повертати будь-який тип). Замість # ставиться знак оператора, що перевантажується. Наприклад, якщо перевантажується операція додавання +, то у функції повинне бути ім'я *operator*+. Зміст *списку_аргументів* сильно залежить від того, як реалізована оператор-функція і від типу оператора, що перевантажується.

Треба пам'ятати два важливих обмеження на перевантаження операторів. По-перше, не можна змінювати пріоритет операторів. По-друге, не можна змінювати число операндів оператора. Наприклад, не можна перевантажувати оператор / так, щоб в ньому використовувався тільки один оператор.

Більшість операторів C++ перевантажується. Неперевантажуваними операторами є

. :: .* ?

Не можна перевантажувати оператори препроцесора.

За виключенням оператора =, оператор-функції успадковуються похідним класом. Проте і для похідного класу можна перевантажити будь-який вибраний оператор, включаючи оператори, що вже перевантажені в базовому класі.

Хоча допустимо мати оператор-функцію для реалізації будь-якої дії, краще, якщо дії перевантажених операторів залишаються в сфері їх традиційного використання.

Проте іноді може виникнути потреба використати будь-який оператор нетрадиційним методом. Прикладом цього є оператори << та >>, які перевантажуються для вводу/виводу.

Оператор-функція не може мати параметрів, що передаються за умовчанням.

11. Перевантаження бінарних операторів

Коли оператор-функція - член класу перевантажує бінарний оператор, у функції буде тільки один параметр. Цей параметр отримає об'єкт, що розміщений справа від знаку оператора. Об'єкт зліва викликає оператор-функцію і передається неявно, через безпосереднє використання вказівника *this*.

Важливо розуміти, що для написання оператор-функцій мається багато варіантів. Приклади, які приведено нижче, не є вичерпиними, проте вони ілюструють декілька найбільш технічних приймів.

Приведемо програму, в якій перевантажено оператор + для класу *coord*. Цей клас має координати *x, y*.

```
// Перевантаження + для класу coord.
#include <iostream.h>
class coord {
        int x,y; // значення координат
public:
        coord() { x = 0; y= 0; }
        coord(int i, int j) { x = i; y = j; }
```

```
        void get_xy(int &i, int &j) { i = x; j = y; }
        coord operator+(coord ob2);
};
// Перевантаження + для класу coord.
coord coord::operator+(coord ob2)
{
coord temp;
temp.x = x +ob2.x;
temp.y = y +ob2.y;
return temp;
}
main()
{
coord o1(10, 10), o2(5, 3), o3;
int x, y;
o3 = o1 + o2;
// додавання двох об'єктів - виклик operator+()
o3.get_xy(x, y);
cout << "(o1 + o2) X: " << x << ", Y: " << y << "\n";
return 0;
}
```

В результаті роботи програми буде виведено наступний вираз:

(01+02) x : 15, x : 13

Проаналізуємо програму. Функція *operator+( )* повертає об'єкт типу *coord*, в якому сума координат *x* знаходиться в *x*, а сума координат *y* - в *y*. Відмітимо, що тимчасовий об'єкт *temp* використовується всередині *operator+( )* для зберігання результата і є об'єктом, що повертається. Більш того, жоден з операндів не змінюється. Тобто, + був перевантажений способом, що є аналогічним своєму традиційном арифметичному використанню. Завдяки цьому є можливим використання виразів:

```
03=01+02;
03=01+02+01+03;
(01+02). get_xy(x, y);
```

12. Перевантаження операторів відношення та логічних операторів

Існує можливість перевантаження операторів відношення та логічних операторів. При перевантаженні операторів відношення та логічних операторів так, щоб вони поводили себе звичайним чином, не знадобляться оператор - функції для повернення об'єкта класу, для якого вони відносяться. Замість цього, вони будуть повертати ціле, що інтерпретується як *true* або *false*. Окрім цього ці оператор-функції дозволяють також використовувати оператори відношення та логічні оператори у великих виродах, що вміщують і інші типи даних.

Приведемо програму, в якій перевантажується оператори = = та && :

13. Перевантаження унарних операторів

Перевантаження унарних операторів нагадує перевантаження бінарних, за вийнятком того, що ми маємо справу тільки з одним операндом. При перевантаженні унарного оператора з використанням функції-члена, у функції не буде параметрів.

Оскільки є тільки один операнд, він і викликає оператор-функцію. Інші параметри не потрібні.

В наступній програмі для класу *coord* перевантажується оператор інкрементації (++):

14. Використання дружніх оператор-функцій.

В дружню функцію не передається вказівник *this*. У випадку бінарного оператора це означає, що оператор-функції явно потрібно передати обидва операнди. При перевантаженні унарного оператора передається один операнд.

При використанні дружньої функції в операціях з об'єктами можна використовувати вбудовані типи даних, і при цьому вбудований тип може розміщуватись зліва від оператора (при перевантажені оператор-функції члена класу лівий операнд може бути об'єктом, а правий вбудованим типом, але не можна розміщувати вбудований тип зліва від оператора). Саме у випадку коли один з операндів є вбудованим типом і розміщення об'єкта від операції є принциповим рекомендується використовувати при перевантаженні дружню оператор-функцію.

Наведемо приклад, в якому задається перевантаження дружньої функції так, щоб лівий операнд був об'єктом, а правий - операнд іншого типу. Потім оператор перевантажується так, щоб лівий операнд мав вбудований тип, а правий був об'єктом.

15. Поняття успадкування

Успадкування *(inheritance)* – це процес, завдяки якому об'єкт може приймати властивості іншого об'єка з додаванням до них рис, властивих тільки йому.

Успадкування - один з трьох базових принципів ООП. Завдяки успадкуванню підтримується концепція ієрархії класів. Застосування ієрархії класів робить керованими великі потоки інформації.

16. Доступ до елементів базового класу

Клас, властивості якого успадковуються називається базовим, а клас, який успадковує властивості базового називається похідним.

При успадкуванні використовується така загальна форма:

```
class ім'я_похідного_класу :доступ ім'я_базового_класу
{
        // елементи похідного класу
}
```

Тут доступ - одне з трьох ключових слів: *public, private* або *protected.*

Специфікатор доступу визначає те, як елементи базового класу успадковуються похідним класом. Якщо специфікатором доступу успадкованого базового класу є *public,* то всі відкриті члени базового класу стають відкритими і в похідному. Якщо специфікатором доступу успадкованого базового класу є *private*, то всі відкриті члени базового класу стають закритими в похідному. В обох випадках, всі закриті члени базового класу залишаються закритими і недосяжними для похідного класу.

Важливо розуміти, що якщо специфікатором доступу є *private*, то відкриті члени базового класу стають закритими в похідному, проте ці члени залишаються доступними для функцій-членів похідного класу.

Наведемо приклади з приводу вищезгаданого.

17. Захищені члени класу

Іноді можлива ситуація, коли члени базового класу, залишаючись закритими, були досяжні для похідного класу. Для реалізації цієї ідеї в C++ введено специфікатор доступу protected (захищений).

Специфікатор доступу protected еквівалентний *private* з єдиною різницею: захищені члени базового класу досяжні для членів всіх похідних класів цього базовим класом. Поза базового класу або похідних класів захищені члени недосяжні.

Специфікатор доступу *protected* може знаходитись у будь-якому місці оголошення класу, хоча, як правило, його розміщують після оголошення закритих членів та перед оголошенням відкрити членів. Повна загальна форма оголошення класу має наступний вигляд:

```
class ім’я_класу
{
        // закриті члени
protected:
        //захищені члени
public:
        // відкриті члени
};
```

Коли захищений член базового класу успадковується похідним класом як відкритий, він стає захищеним членом похідного класу. Якщо базовий клас успадковується як закритий, то захищений член базового класу стає закритим членом похідного класу.

Базовий клас може також успадковуватись похідним класом як захищений. У цьому випадку відкриті та захищені члени базового класу стають захищеними членами похідного класу. Закриті члени базового класу залишаються закритими, і вони не досяжні для похідного класу. Приведемо приклади з приводу вищезазначеного.

18. Конструктори, деструктори та успадкування

Базовий клас, похідний клас або обидва можуть мати конструктори і / або деструктори.

Якщо і у базового, і у похідного класів маються конструктори та деструктори, то конструктори виконуються у порядку успадкування, а деструктори - у зворотньому порядку. Таким чином, конструктор базового класу виконується раніше конструктора похідного класу. Для деструкторів правильний зворотній порядок: деструктор похідного класу виконується раніше деструктора базового класу.

19. Передача аргументів для конструкторів базавого та похідного класів

Розглядаючи поняття успадкування необхідно звернути увагу на питання передачі аргументів для конструкторів похідного та базового класів. Коли ініціалізація проводиться тільки в похідному класі, аргументи передаються звичайним шляхом. Проте, при передачі аргумента конструктору базового класу виникають деякі складнощі . По-перше, всі необхідні аргументи базового та похідного класів передаються конструктору похідного класу. Потім, використовуючи розширену форму оголошення конструктора похідного класу, відповідні аргументи передаються далі в базовий клас. Синтаксис передачі аргументів з похідного в базовий клас наступний:

```
конструктор_похідного_класу(список_арг):
        конструктор_базового_класу(список_арг)
{
```

```
        //тіло конструктора похідного класу
    }
```

Для базового та похідного класів допустимо використовувати одні і ті самі аргументи. Для похідного класу допустимо також ігнорування всіх аргументів та передача їх напряму в базовий клас. Проілюструємо вищесказане на прикладах.

```
#include <iostream.h>
class base
{
    public:
       base() { cout << "Робота конструктора базового класу \n"; }
       ~base() { cout << "Робота деструктора базового класу \n"; }
};
class derived : public base
{
        int j;
    public:
       derived(int n)
        {cout << "Робота конструктора похідного класу \n";
         j = n;
        }
        ~derived()
        { cout << "Робота деструктора похідного класу \n"; }
        void showj() { cout << j << '\n'; }
};
void main()
{
        derived o(10);
        o.showj();
}
```

У цьому прикладі у конструкторів похідного та базового класів мається один і той самий аргументи. Обидва конструктори використовують його, і похідний клас передає цей аргумент в базовий.

```
#include <iostream.h>
class base
    {int i;
public:
    base(int n)
    {
        cout << "Робота конструктора базового класу \n";
            i = n;
        }
        ~base()
    {
        cout << "Робота деструктора базового класу \n";
    }
        void showi() { cout << i << '\n'; }
};
class derived : public base
```

```
{
        int j;
public:
        derived(int n) : base(n)
{ // Передача аргумента в базовий клас
                cout << "Робота конструктора похідного класу \n";
                j = n;
        }
        ~derived()
{
        cout << "Робота деструктора похідного класу \n"; }
        void showj() { cout << j << '\n'; }
};
void main()
{
        derived o(10);
        o.showi();
        o.showj();
}
```

Зверніть особливу увагу на оголошення конструктора похідного класу. Відмітьте, як параметр *n* (який отримує аргумент при ініціалізації) використовується в *derived()* і передається в *base()*.

В більшості випадків конструктори базового та похідного класів не використовують один і той самий аргумент. Тоді, при необхідності передачі кожному конструктору класу одного або більше параметрів, ви повинні передати конструктору похідного класу всі аргументи, необхідні конструкторам цих двох класів. Потім конструктор похідного класу просто передає конструктору базового ті аргументи, які йому потрібні.

Приведемо приклад, в якому показано, як передати один аргумент конструктору похідного класу, а другий - конструктору базового.

```
#include <iostream.h>
class base
{
        int i;
public:
        base(int n)
        {
        cout << ”Робота конструктора базового класу\n”
        i = n;
        }
        ~base()
        {
        cout<<”Робота деструктора базового класу\n”;
        }
        void showi() {cout << i << ’\n’; }
}
class derived : public base
{
```

```
        int j;
public:
        derived(int n; int m) : base(m)
        {// Передача аргумента в базовий клас
        cout << ”Робота конструктора похідного класу\n”;
        j = n;
        }
        ~derived()
        {
        cout << ”Робота деструктора похідного класу\n”;
        void showj() {cout << j << ‘\n’;
        }
};
void main()
{
        derived o(10,20);
        o.showi();
        o.showj();
}
```

Важливо розуміти, що якщо похідному класу деякий аргумент не потрібен, він його ігнорує і просто передає в базовий клас.

20. Множинне успадкування.

Існує два способи, завдяки яким похідний клас може успадковувати більш як один базовий клас. По-перше, похідний клас може використовуватися як базовий для іншого похідного класу, створюючи багаторівневу ієрархію класів. В цьому випадку говорять, що вихідний базовий клас є непрямим (inderect) базовим класом для іншого похідного класу. По-друге, похідний клас може прямо успадковувати більш одного базового класу. В такій ситуації, комбінація двох або більше базових класів допомагає створенню похідного класу.

Коли клас використовується як базовий для похідного, в свою чергу, є базовим для іншого похідного класу, конструктори всіх трьох класів викликаються в порядку успадкування. Деструктори викликаються у зворотньому порядку.

Якщо похідний клас напряму успадковує множину базових класів, використовується таке розширене оголошення:

```
class ім’я_похідногокласу :
                спец_доступу ім’я_базового_класу1,
                спец_доступу ім’я_базового_класу2,
        . . .  спец_доступу ім’я базавого_класуN
{
// тіло класу
}
```

Тут ім’я_базавого класу1, ... , ім’я_базового класуN - імена базових класів, спец_доступу - специфікатор доступу, який може бути різним для кожного базового класу. Коли успадковується множина базових класів, конструктори використовуються зліва направо у порядку, що задається в огошенні похідного класу. Деструктори виконуються у зворотньому порядку.

Коли клас успадковує множину базових класів, конструктором яких необхідні аргументи, похідний клас передає ці аргументи, використовуючи розширену форму оголошення конструктора похідного класу:

```
констр_похід_класу(список_арг):
        ім'я_базового_класу1(список_арг),
        ім'я_базового_класу2(список_арг),
                . . .
        ім'я_базового_класуN(список_арг)
{
 // тіло конструктора похідного класу
}
```

Коли похідний клас успадковує ієрархію класів, кожний похідний клас повинен передавати попередньому базовому класу по ланцюжку необхідні аргументи.

Приведемо приклади.

21. Вказівники на похідні класи.

Поліморфізм підтримується двома способами, а саме при компіляції він підтримується шляхом перевантаження операторів та функцій (статичний поліморфізм), при виконанні програм він підтримується шляхом віртуальних функцій (динамічний поліморфізм).

Основою практичного застосування віртуальних функцій та динамічного поліморфізму є вказівники на похідні класи.

Важливою особливістю вказівників в C++ є те, що вказівник, оголошений як вказівник на базовий клас також може використовуватися як вказівник на будь-який клас, похідний від цього базового. В такій ситуації наступні оператори є вірними.

```
base *p; // вказівник базового класу
base base_ob; // об'єкт базового класу
derived derived_ob; // об'єкт похідного класу
p=&base_ob; // p вказує на об'єкт базового класу
p=&derived_ob; // p вказує на об'єкт похідного класу
```

Як відмічено, вказівник базового класу може вказувати на об'єкт будь-якого класу, похідного від цього базового, без генерації помилки невідповідності типів.

Вказівник базового класу може використовуватися для вказівки на об'єкт похідного класу, проте зворотній порядок недійний. Вказівник на похідний клас неможливо використати для доступу до об'єктів похідного класу. Приведемо приклад.

22. Віртуальні функції.

Віртуальна функція (virtual function) є функцією - членом класу. Вона оголошується всередині базового класу та перевизначається в похідному класі. Для створення віртуальної функції перед оголошенням функції ставиться ключеве слово *virtual*. Якщо клас, що має віртуальну функцію, успадковується, то в похідному класі віртуальна функція перевизначається.

По суті, віртуальна функція реалізує ідею "один інтерфейс, багато методів", яка лежить в основі поліморфізму. Віртуальна функція всередині базового класу задає інтерфейс цієї функції. Кожне перевизначення віртуальної функції в похідному класі визначає її реалізацію, яка зв'язана з специфікою похідного класу. Таким чином,

перевизначення створює конкретний метод. При перевизначенні віртуальної функції в похідному класі, ключеве слово *virtual* не потрібне.

Віртуальна функція може викликатися так само, як і будь-яка інша функція-член. Проте найбільш цікавим є виклик віртуальної функції через вказівник, завдяки чому підтримується динамічний поліморфізм. Як було відмічено, вказівник базового класу можна використовувати як вказівник на об'єкт похідного класу. Якщо вказівник базового класу вказує на об'єкт похідного класу, який має віртуальну функцію, і для якого віртуальна функція викликається через цей вказівник, то компілятор визначає, яку версію віртуальної функцію викликати на основі типу об'єкта, на який вказує вказівник. Цей процес є реалізацією принципа динамічного поліморфізму. Клас, що має віртуальну функцію, називають поліморфним класом (polymorphic class).

Розглянемо приклади.

23. Чисто віртуальні функції

Іноді, коли віртуальна функція оголошується в базовому класі, вона не виконує ніяких значимих дій. Це звичайна ситуація, оскільки часто базовий клас не визначає закінчений тип. Замість цього, він просто вміщує базовий набір функцій-членів та зміних, для яких похідний клас задає все, чого не вистачає. Коли у віртуальній функції базового класу відсутня значима дія, в реалізації будь-якого похідного класу ця функція повинна перевизначатись. Для реалізації цього положення C++ підтримує чисто віртуальні функції (pure virtual function).

Чисто віртуальні функції не визначаються в базовому класі. В нього включаються тільки прототипи цих функцій. Для чисто віртуальної функції використовується така загальна форма:

virtual mun ім'я_функції (список_параметрів) = 0;

Прирівняння функції нулю - це повідомлення компілятору, що в базовому класі не існує тіла функції. Якщо функція задається як чисто віртуальна, то вона повинна перевизначатися в кожному похідному класі. Якщо цього немає, то при компіляції виникає помилка. Таким чином, створення чисто віртуальних функцій - це шлях, який гарантує, що похідні класи забезпечуть їх перевизначення.

Якщо клас має хоча б одну чисто віртуальну функцію, то про нього кажуть, як про абстрактний клас (abstract class). Оскільки в абстрактному класі мається хоча б одна функція, у якої відсутнє тіло, технічно цей клас є неповним і жодного об'єкту цього класу створити не можна. Таким чином абстрактні класи можуть бути тільки успадковуваними. Можна створювати вказівники абстрактного класу, завдяки яким досягається динамічний поліморфізм.

24. Родові функції.

Родова функція визначає базовий набір операцій, які будуть застосовуватися до різних типів даних. Родова функція оперує з тим типом даних, який вона отримує як параметр. За допомогою цього механізму одна й та сама процедура може застосовуватись до різних даних.

Як відомо, багато алгоритмів логічно однакові, незалежно від того, для обробки яких типів даних вони призначені. Наприклад, алгоритм швидкого сортування є однаковим як для масивів цілих, так і для масивів дійсних чисел. Це саме той випадок, коли дані, що сортуються відрізняються типами. Завдяки створенню родової функції можна незалежно від даних визначити суть алгоритма. У випадку родової функції компілятор автоматично генерує правильний код для фактичних типів даних. По суті,

при створенні родової функції створюється функція, яка може автоматично перевантажити сама себе.

Родова функція створюється за допомогою ключевого слова *template* (шаблон). Воно призначене для створення шаблону (або каркасу), який описує те, що буде робити функція, при цьому компілятор доповнить при необхідності деталі.

Приведемо типову форму визначення родової функції:

```
template <class Ttype> значення__що__повертається
                              ім'я__функції(список__параметрів)
{
. . . // Тіло функції
};
```

Тут на місці *Ttype* вказується тип даних, що використовується функцією. Це фіктивне ім'я, яке компілятор автоматично замінює іменем реального типу даних при створенні конкретної версії функції.

Наведемо приклад, в якому створюється родова функція, яка міняє місцями значення двух змінних, що передаються їй як параметри. Оскільки в своїй основі процес обміну двух значень не залежить від типу змінних, цей процес вдало реалізується за допомогою родової функції.

24. Родові класи

Родовий клас - це клас, в якому визначені всі алгоритми цього класу, а фактичні типи даних, що обробляються, задаються як параметри пізніше, при створенні об'єктів цього класу.

Загальна форма оголошення родового класу наступна:

```
template <class Ttype> class ім'я_класу
{
...//Тіло класу
}
```

Тут *Ttype* - це фіктивне ім'я імені типу, який буде задано при створенні екземпляру класу. При необхідності можна визначити більше одного родового типу даних, що розділяються комами.

Після створення родового класу можна створити конкретний екземпляр цього класу за допомогою наступної форми:

```
ім'я_класу <type> об'єкт;
```

Тут *type* - це ім'я типу даних, з якими буде оперувати клас.

Функції-члени родового класу автоматично стають родовими. Для них не обов'язково явно задавати ключеве слово *template.*

Створюючи родовий (параметризований) клас, ви створюєте ціле сімейство родинних класів, які можна застосувати до будь-якого типу даних.

25.Контейнерні класи.

Найбільш широке застосування шаблони класів знаходять при створенні контейнерних класів.

Контейнерними класами (контейнерами) в загальному випадку називаються класи, в яких зберігаються організовані дані.

Перевага визначення параметризованих контейнерних класів полягає в тому, що як тільки логіка, необхідна для підтримки контейнера визначена, він може застосовуватися до будь-яких типів даних без необхідності його переписування. Завдяки цьому, один раз написаний та відлажений контейнерний клас можна використовувати безліч разів.

Останні версії *Borland C* мають великі бібліотеки контейнерів.

Частіше за інші використовуються контейнери наступних типів: обмежений (“захищений”) масив, черга, стек, зв’язаний список, бінарне дерево. Кожний з цих контейнерів виконує конкретні операції збереження та вилучення інформації, отримання запитів.

26. Виняткові ситуації – виникнення непередбачених помилкових умов (наприклад, ділення на ноль при операціях з плаваючою крапкою). Як правило ці умови завершають програму користувача з системним повідомленням про помилку. C++ дає програмісту можливість відновити програму за цих умов та продовжити її виконання. Це вбудований механізм, який називається обробкою виняткових ситуацій (exception handling).

Обробка виняткових ситуацій в C++ будується за допомогою трьох ключових слів: *try, catch* та *throw*. Оператори програми, під час виконання яких ви хочете забезпечити обробку виняткових ситуацій розміщуються в блоці *try.* Якщо виняткова ситуація (тобто помилка) має місце в середтні блоку *try*, вона генерується за додопогою оператора *throw.* Перехоплюється та обробляється виняткова ситуація за допомогою оператора *catch,* що іде безпосередньо за блоком *try.* З блоком *try* може бути зв’язано більше одного оператора *catch.* Блок *try* може вміщувати як декілька операторів всередині однієї функції, так і всі оператори функції *main().*

Загальна форма операторів *try* і *catch* наступна:

```
try{
…//блок try
}
catch(type1 arg){
…//блок catch
}
catch(type2 arg){
…//блок catch
}
.
.
.
catch(typeN arg){
…//блок catch
}
```

Як видно з блоком *try* може бути зв’язано більше одного опратора *catch.* Те, який конкретно оператор *catch* буде виконуватися, залежить від типу виняткової ситуації.

Тобто, якщо тип даних, вказаний в операторі *catch,* відповідає типу виняткової ситуації, то виконується даний оператор *catch.* Всі інші оператори *catch* пропускаються. Якщо виняткова ситуація перехоплена, аргумент *arg* отримує її значення. Можна перехоплювати будь-які типи даних, включаючи і створені вами класи.

Загальна форма оператора *throw* наступна:

throw виняткова_ситуація

Оператор *throw* повинен виконуватися або всередині блоку *try,* або в будь-якій функції, яку цей блок викликає (прямо чи непрямо). В загалльній формі виняткова_ситуація – це виняткова ситуація, що генерується оператором. Наведемо приклад:

В деяких випадках необхідно нак настроїти оброюку виняткових ситуацій, щоб перехоплювати всі виняткові ситуації, незалежно від їх типу. Для цього використовується наступна форма оператора *catch*:

```
catch( …){
        …//Обробка всіх виняткових ситуацій
}
```

Дуже зручно оператор *catch(...)* використовувати як останній в групі операторів *catch.* Він за умовчанням стає оператором, який “перехоплює все”. Наведемо приклад.

26.Базові положення системи вводу/виводу С++.

Система вводу/виводу С++ діє через потоки (streams).

Потік – це логічний устрій, який видає та приймає інформацію. Потік зв’язаний з фізичним устроєм за допомогою системи вводу/виводу С++.

Коли починається програма на С++, автоматично відкриваються чотири потоки (інформацію про них наведено в таблиці 3).

Таблиця 3

Потік	Значення	Устрій за умовчанням
cin	Стандартний ввод	Клавіатура
cout	Стандартний вивід	Екран
cerr	Стандартна помилка	Екран
clog	Буферизована версія cerr	Екран

В С++ система вводу/виводу підтримується заголовочним файлом *iostream.h*. В цьому файлі для підтримки вводу/виводу задана ієрархія класів. Клас нижнього рівня вводу/виводу називається *streambuf*. Цей клас забезпечує базові дії по вводу/виводу та використовується в основному як базовий для інших класів. Наступним класом в ієрархії є *ios*. Цей клас забезпечкє форматування, контроль помилок та інформацію про стан потоків вводу/виводу. *Ios* використовується як базовий для трьох класів: *istream, ostream* та *iostream*. Ці класи застосовуються для створення потоків, сумісних, відповідно, з вводом, виводом та вводом/виводом.

Клас *ios* має багато функцій та змінних – членів класу, які управляють та контролюють основну роботу потока. Коли файл *iostram.h* включено в програму, ви отримуєте доступ до класу *ios*.

27. Форматний ввод/вивод. Інформацію можна виводити в широкому діапазоні форм. Кожний потік C++ зв'язаний з набором прапорів формата, які задають те, як відображаються дані. Прапори формата закодовані в довге ціле і мають свої ідентифікатори. Нижче приведено опис прапорів вводу/виводу у вигляді перерахування (enumeration), що знаходиться в середині класу *ios*:

```
//Прапори формата ios
enum{skipws=0x0001, //зкидання невидимих початкових
                    //символів
left=0x0002, //вирівнювання по лівому краю
right=0x0004, //вирівнювання по правому краю
internal=0x0008, //вставка пробіла між знаком і числом
dec=0x0010, //вивід у десятковій системі числення
oct=0x0020, //вивід у вісімковій системі числення
hex=0x0040, //вивід у шістнадцятковій системі числення
showbase=0x0080, //вивід основи системи числення
showpoint=0x0100, //вивід десяткової крапки і наступних"0"
uppercase=0x0200, //вивід "e" і "x" у науковій нотації у
          //верхньому регістрі
showpos=0x0400, //вивід "+" перед додатнім числом
scientific=0x0800, //вивід чисел з плаваючою крапкою у
                   //науковій нотації
fixed=0x1000, //вивід чисел з плаваючою крапкою з
                  //фіксованою крапкою
unitbuf=0x2000, //система в/в флешує буфер потока виводу
stdio=0x4000      //потоки stdout і stderr після кожної //операції виводу
                  флешуються
}
```

Для встановлення прапора формата користуються функцією *setf()*. Ця функція є членом класу *ios*. Приведемо її типову форму:

long setf(long прапори);

Функція *setf()* повертає попередні установки прапорів формату і встановлює задані прапори. Наприклад, для встановлення прапора showpos ви можете скористатися оператором:

потік.setf(ios::showpos);

Тут потік – це той потік, на який ви хочете вплинути; *showpos* – це константа, що входить в *enumeration* всередині класу *ios*. Тому, щоб повідомити компілятору про це, необхідно поставити перед *showpos* ім'я класу і операцію розширення області бачення.

Замість декількох викликів *setf(),*можна встановлювати більш одного прапора за один виклик. Для того, щоб об'єднати необхідні прапори, використовується операція OR. Наприклад,

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

Для зкинення одного або декількох прапорів формата, використовується функція *unsetf()*:

```
long unsetf(long  прапори);
```

В *ios* також внесено функцію-член *flags()*, яка повертає поточний стан прапорів формату в змінній типу *long*.

Наведемо приклад роботи функцій *setf()* і *unsetf()*. В програмі спочатку встановлюється прапори *uppercase, showbase* і *hex*. Потім виводиться 88 з використанням наукової нотації. В цьому випадку шістнадцяткове "x" виводиться у верхньому регістрі. Далі *unsetf()* зкидає прапор *uppercase* і знову виводиться шістнадцяткове 88. Тепер "x" буде у нижньому регістрі.

```
#include <iostream.h>
main()
{
        cout.setf(ios::uppercase|ios::showbase|ios::hex);
        cout << 88 << '\n';
        cout.unsetf(ios::uppercase);
        cout << 88 << '\n';
        return 0;
}
```

28. Використання функцій *width(), precision()* і *fill()*. Крім прапорів формату існують три функції-члена, що визначені в класі *ios*, які визначають параметри формата: ширину поля, точність і символ заповнення. Це, відповідно, функції *width(), precision()* і *fill()*.

За умовчанням при виводі якого-небудь значення, це значення займає число позицій, що відповідає числу символів, що виводяться. Проте, викоритовуючи функцію width(), можна задати мінімальну ширину поля. Прототип функції наступний:

```
int width(int w);
```

Тут *w* – ширина поля, а функція повертає попередню ширину поля.

Після встановлення мінімальної ширини поля, якщо значення, що виводиться потребує меншу ширину поля, остача заповнюється поточним символом заповнення (за умовчанням пробілом). Проте, якщо розмір значення, що виводитьяс, перевищує мінімальну ширину поля, буде зайнято стільки символів, скільки треба. Значення, що виводяться не усікається.

За умовчанням, при виводі значень з плаваючою крапкою, після десяткової крапки ставиться шість цифр. Використовуючи фунцію *precision()*, це число можна змінити. Прототип функції наступний:

int precision(int p);

Тут *p* – це точність (число цифр, що виводяться після коми), сама функція повертає попередню точність.

За умовчанням, при заповненні поля використовуються пробіли. Проте можна змінити символ заповнення, використовуючи функцію *fill()*. Її прототип наступний:

char fill(char ch);

Після виклику функції *fill()* ch становиться символом заповнення, а функція повертає попередній символ заповнення.

29. Файловий ввод/вивід. Для реалізації файлового вводу/виводу необхідно включити в програму заголовочний файл *fstream.h*. В ньому визначено декілька класів, включаючи *ifstream, ofstream* і *fstream*. Ці класи є похідними класів *istream* і *ostream*. В С++ файл відкривається за допомогою його зв'язування з потоком. Є три види потоків: вхідний, вихідний і вхідний/вихідний. Перед тим, як відкрити файл, по-перше, треба створити потік. Для створення вхідного потока необхідно оголосити потік класу *ifstream*. Для створення вихідного – оголосити потік класу *ofstream*. Потоки, які реалізують і ввод, і вивід, повинні оголошуватися як об'єкти класу *fstream*. Наприклад,

```
ifstream in; // ввод
ofstream on; // вивід
fstream io; // ввод і вивід
```

Після створення потоку, одним з способів зв'язати його з файлом є функція *open()*. Ця функція є членом кожного з трьох початкових класів. Її прототип настпупний:

void open(char *filename, int mode, int access);

Тут *filename* – ім'я файлу, в який можна включити шлях. Величина *mode* задає те, як відкривається файл. Вона може приймати одне (або більше) з наступних значень (що успадковуються класом *fstream.h*):

```
ios::app
ios::ate
ios::binary
ios::in
ios::nocreate
ios::noreplace
ios::out
ios::trunc
```

Включення *ios::app* викликає відкриття файла в режимі додавання в кінець файла. Ця величина може застосовуватися тільки до файлів, що відкриваються для виводу. Включення *ios::ate* викликає позиціювання в кінець файла. Значення *ios::binary* визиває відкриття файлу у двійковому режимі. За умовчанням, всі файли відкриваються у текстовому режимі. Режим *ios::in* визначає відкриття файла для виводу. Прапор *ios::out* показує, що файл відкривається для виводу. Створення потоку з використанням *ifstream* забезпечує ввод, а створення потоку з використанням *ofstream* забезпечує вивід, тому в таких випадках, зазначені величини використовуються необов'язково. Використання *ios::nocreate* призводить до того, що функція *open()* повертає помилку, якщо файл досі не існує. Використання *ios::noreplace* призводить до того, що функція *open()* повертає помилку, якщо файл вже існує. Використання *ios::trunc* призводить до знищення змісту раніше існуючого файлу з тією ж назвою і усіченню його до нульової довжини.

Параметр *access* задає права доступу до файла. За умовчанням значення *access* дорівнює *filebuf::openprot* і дорівнює 0x644 для середовища UNIX, що означає звичайний клас. В середовищі DOS/WINDOWS, *access* відповідає кодам атрибітів файлів DOS/WINDOWS. Вони дорівнюють (дивись таблицю 5):

Таблиця 5

Атрибут	Значення
0	Звичайний файл з вільним доступом
1	Файл тільки для читання
2	Зкритий файл
4	Системний файл
8	Архівний файл

Використовуючи оператор OR можна об'єднати два або більше значення. Для DOS/WINDOWS значення *access* для звичайних файлів дорівнює нулю.

Наведемо приклад відкриття звичайного вихідного файла в середовищі DOS/WINDOWS:

```
ofstream out;
out open("test", ios::out, 0);
```

Проте, як правило, параметри *mode* і *access* задаються за умовчанням: для *ifstream mode* дорівнює *ios::in,* а для *ofstream* дорівнює *ios::out*; *access* має значення 0. Тому попередній фрагмент перепишеться у вигляді:

```
out.open("test");
```

Щоб відкрити потік для вводу і виводу, необхідно задати для *mode* обидва значення: *ios::in* і *ios::out*:

```
fstream mystream;
mystream.open("test", ios::in | ios::out);
```

Якщо *open()* закінчилася з помилкою, потік буде дорівнювати нулю. Тому перед використанням файла, рекомендується переконатися у його вдалому відкритті, як показано нижче:

```
if (!mystream) {
  cout<<”Файл відкрити неможливо\n”;//Помилка відкриття файла
                    }
```

Хоча використовувати функцію *open()* для відкриття файла в цілому правильно, на практиці частіше це не робиться, оскільки у класів *ifstream, ofstream* і *fstream* є конструктори, які відкривають файл автоматично. Конструктори мають ті ж параметри, в тому числі і ті, що задаються за умовчанням, що у функції *open()*.

Приклад:

```
ifstream mystream(”myfile”);//Відкриття файла вводу
```

Для закриття файла використовується функція-член *close()*. Наприклад, щоб закрити файл, зв’язаний з потоком *mystream*, використовується оператор:

```
mystream.close();
```

Функція *close()* не має параметрів та значення, що повертається.

Використовуючи функцію-член *eof()*, можна визначити, чи був досягнутий кінець файла при вводі.

Прототип функції наступний:

```
int eof();
```

Вона повертає ненульове значення в тому випадку, якщо було досягнуто кінець файла; в противному випадку функція повертає нуль.

Після відкриття файла для вводу/виводу інформації використовуються операції “<<” і ” >>” і зв‘язаний з файлом потік.