ЛАБОРАТОРНА РОБОТА №5

# Поліморфізм, віртуальні функції, абстрактні класи та виняткові ситуації в С++

***Мета*:** отримати навички з використання механізму поліморфізму, отримати навички з використання виняткових ситуацій в мові С++.

## Теоретичні відомості

### Поліморфізм

У загальному випадку поліморфізм являє собою здатність об'єкта змінювати форму. Поліморфний об'єкт являє собою об'єкт, який може ухвалювати різні форми. Основні концепції поліморфізму:

- Поліморфізм являє собою здатність об'єкта змінювати форму під час виконання програми.
- С++ спрощує створення поліморфних об'єктів.
- Для створення поліморфних об'єктів програми повинні використовувати віртуальні (`virtual`) функції.
- Віртуальна (`virtual`) функція — це функція базового класу, перед іменем якої стоїть ключове слово `virtual`.
- Будь-який похідний від базового клас може використовувати або перевантажувати віртуальні функції.
- Для створення поліморфного об'єкта слід використовувати покажчик на об'єкт базового класу.

### Віртуальні функції

Термін «віртуальний» означає видимий, але не існуючий у реальності. Коли використовуються віртуальні функції, програма, яка, викликає функцію одного класу, може в цей момент викликати функцію зовсім іншого класу.

Нехай, є набір об'єктів різних класів, але потрібно, щоб вони всі були в одному масиві і викликалися за допомогою того самого виразу. Наприклад, є набір класів геометричних фігур: трикутників, кіл, квадратів і т.д., і в кожному із цих класів є функція `Draw()`, відповідальна за виведення об'єкта на екран. Використовуючи механізм віртуальних функцій, можна вивевсти картинку, що містить різні геометричні фігури, за допомогою циклу:

```
Shape *pfigure[100];// Shape-Деякий базовий клас
…for (int i=0; i
```

```
    pfigure[i]->Draw();  // pfigure-масив покажчиків
//на різні геометричні фігури
```

Абсолютно різні функції виконуються за допомогою того самого виклику. Так, якщо покажчик у масиві `pfigure` вказує на кулю, викликається функція, що малює кулю, якщо він вказує на трикутник – виводиться трикутник. Це і є поліморфізм, тобто різні форми. Поліморфізм – це одна із ключових особливостей об'єктно-орієнтованого програмування, поряд із класами й спадкуванням.

Щоб використовувати поліморфізм, необхідно виконати деякі умови. По-перше, усі класи повинні бути спадкоємцями того самого базового класу, у нашому прикладі це – `Shape`. По-друге, функція `Draw()` повинна бути оголошена як віртуальна (специфікатор `virtual`) у базовому класі.

Віртуальний метод - це метод, який, будучи описаний у нащадках, заміщає собою відповідний метод скрізь, навіть у методах, описаних для похідного класу, якщо він викликається для нащадка.

Адреса віртуального методу відома лише в момент виконання програми. Коли відбувається виклик віртуального методу, його адреса береться з таблиці віртуальних методів свого класу. У такий спосіб викликається те, що потрібно.

Перевага застосування віртуальних методів полягає в тому, що при цьому використовується саме механізм пізнього зв'язування, який допускає обробку об'єктів, тип яких невідомий під час компіляції.

***Приклад 5.1*** Ілюстрація застосування віртуальних методів.

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

class vehicle  // клас " транспортний засіб"
{
int wheels; float weight;
public:
virtual void message(void)
{cout << " Транспортний засіб\n";}
/* опис віртуальної функції message класу vehicle і
реалізація цієї функції. При виклику //функції
message класу vehicle на екран монітора буде
виведений рядок " Транспортний засіб"*/
};

class car : public vehicle {
int passenger_load;
```

```
public:
void message(void) {cout << "Легкова машина\n";}
/* опис віртуальної функції message класу car і
реалізація цієї функції. При виклику функції message
класу car на екран монітора буде виведений рядок "
Легкова машина "*/
};

class truck : public vehicle
{
int passenger_load;
float payload;
public:
int passengers(void)
{return passenger_load;}
};

class boat : public vehicle
{
int passenger_load;
public:
int passengers(void)
{return passenger_load;}
void message(void) {cout << "Човен\n";}
/* опис віртуальної функції message класу boat і
реалізація цієї функції. При виклику функції message
класу boat на екран монітора  буде виведений рядок
"Човен"*/
};

void main()  {
setlocale(LC_ALL,"Ukrainian");
vehicle *unicycle;
unicycle = new vehicle; // Створюємо об'єкт класу
//vehicle, покажчик unicycle указує на цей об'єкт
unicycle->message(); //виклико методу об'єкта message
delete unicycle;  // видалення об'єкта unicycle
unicycle = new car;
unicycle->message();
delete unicycle;
unicycle = new truck;
unicycle->message();
```

```
delete unicycle;
unicycle = new boat;
unicycle->message();
delete unicycle;
cin.get();
}
```

Результати роботи програми(вивід на екран):

```
Транспортний засіб
Легкова машина
Транспортний засіб
Човен
```

Розглянемо наведений приклад. Є три класи `car`, `truck` і `boat`, які є похідними від базового класу `vehicle`. У базовому класі `vehicle` описана віртуальна функція `message`. У двох із трьох класів (`car`, `boat`) також описані свої функції message, а в класі `truck` немає опису своєї функції `message`. Описано також змінну `unicycle`, як покажчик на об'єкт типу `vehicle`. Потім, створено об'єкт класу `vehicle`, змінна `unicycle` вказує на цей об'єкт. Після цього викликається метод `message` об'єкта `unicycle`, а в наступному рядку видаляємо цей об'єкт. У наступних трьох блоках по 3 рядки проведено аналогічні операції, з тою лише різницею, що робота ведеться з об'єктами класів `car`, `truck`, `boat`. Застосування покажчика дозволяє нам використовувати один і той самий покажчик для всіх похідних класів. Розглянемо виклик функції `message` для кожного з об'єктів. Якби не було вказано, що функція `message` класу `vehicle` є віртуальною(`virtual`), то компілятор статично (жорстко) зв'язав би будь-який виклик методу об'єкта покажчика `unicycle` з методом `message` класу `vehicle`, тому що при описі було сказано, що змінна `unicycle` вказує на об'єкт класу `vehicle`. Таким чином здіснено **раннє зв'язування**. Результатом роботи такої програми було б виведення чотирьох рядків " Транспортний засіб". Але за рахунок застосування віртуальної функції в класі одержано інші результати.

При роботі з об'єктами класів `car` і `boat` викликаються їхні власні методи `message`, що і підтверджується виведенням на екран відповідних повідомлень. У класу `truck` немає свого методу `message`, із цієї причини здійснюється виклик відповідного методу базового класу `vehicle`.

Дуже часто клас, що містить віртуальний метод називають поліморфним класом. Найголовніша відмінність полягає в тому, що поліморфні класи допускають обробку об'єктів, тип яких невідомий під час компіляції. Функції, описані в базовому класі як віртуальні, можуть

бути модифіковані в похідних класах, причому зв'язування відбудеться не на етапі компіляції (**раннє зв'язуванням**), а в момент звертання до даного методу (**пізніше зв'язування**).

Віртуальні методи описуються за допомогою ключового слова `virtual` у базовому класі. Це означає, що в похідному класі цей метод може бути заміщений методом, більш підходящим для цього похідного класу. Оголошений віртуальним у базовому класі, метод залишиться віртуальним для всіх похідних класів. Якщо в похідному класі віртуальний метод не буде перевизначений, то при виклику буде знайдений метод з таким іменем вверх по ієрархії класів ( тобто в базовому класі).

Специфікатор `virtual` при оголошенні функції-елемента пропонує компілятору згенерувати деяку додаткову інформацію про функцію. Якщо функція перевизначається в похідному класі і викликається з покажчиком (або посиланням) базового класу, що посилаються на представник похідного класу, ця інформація дозволяє визначити, який з варіантів функції повинен бути обраний: такий виклик буде адресований функції похідного класу.

***Приклад 5.2*** Ілюстрація застосування невіртуальних функцій.

```
#include <iostream>
using namespace std;
class Base //Базовий клас
{
public:
void show() //Звичайна функція
{cout<<"Працює функція базового класу Base\n";}
};
class Derv1: public Base //Похідний клас 1
{
public:
void show()
    {cout<<"Працює функція похідного класу Derv1\n";}
};
class Derv2: public Base //Похідний клас 2
{
public:
void show()
    {cout<<"Працює функція похідного класу Derv2\n";}
};
void main()
{
```

```
setlocale(LC_ALL,"Ukrainian");
Derv1 dv1; //Об'єкт похідного класу 1
Derv2 dv2; //Об'єкт похідного класу 2
Base* ptr; //Покажчик на базовий клас
ptr = &dv1; //Адреса dv1 занести в покажчик
ptr->show(); //Виконати show()
ptr = &dv2; //Адреса dv2 занести в покажчик
ptr->show(); //Виконати show()
cin.get();
}
```

Класи `Derv1` і `Derv2` є нащадками класу `Base`. У кожному із цих трьох класів є метод `show()`. В `main` створено об'єкти класів `Derv1` і `Derv2`, а також покажчик на клас `Base`. Потім адреса породженого об'єкта заноситься в покажчик базового класу. Ця операція є повністю коректною, тому що покажчики на об'єкти породжених класів сумісні за типом з покажчиками на об'єкти базового класу.

В результаті, незважаючи на те, що покажчику на базовий клас присвоюються адреси похідних класів, однаково буде виконуватися метод базового класу, тобто на екран буде виведено:

```
Працює функція базового класу Base
Працює функція базового класу Base
```

Компілятор не зважає на вміст покажчика `ptr` і вибирає той метод, який задовольняє типу покажчика.

***Приклад 5.3*** Ілюстрація застосування віртуальних функцій (розміщено слово `virtual` перед оголошенням функції `show()` у базовому класі).

```
#include <iostream>
using namespace std;
class Base //Базовий клас
{
public:
      virtual void show() //Звичайна функція
      {cout<<"Працює функція базового класу Base\n";}
};
class Derv1: public Base //Похідний клас 1
{
public:
      void show()
    {cout<<"Працює функція похідного класу Derv1\n";}
};
class Derv2: public Base //Похідний клас 2
```

```
{
public:
      void show()
    {cout<<"Працює функція похідного класу Derv2\n";}
};
void main()
{
setlocale(LC_ALL,"Ukrainian");
Derv1 dv1; //Об'єкт похідного класу 1
Derv2 dv2; //Об'єкт похідного класу 2
Base* ptr; //Покажчик на базовий клас
ptr = &dv1; //Адреса dv1 занести в покажчик
ptr->show(); //Виконати show()
ptr = &dv2; //Адреса dv2 занести в покажчик
ptr->show(); //Виконати show()
cin.get();
}
```

У результаті виконання на екран буде виведено:

```
Працює функція похідного класу Derv1
Працює функція похідного класу Derv2
```

Тобто виконуються методи похідних класів, а не базового. Отже, після введення ключового слова `virtual` в оголошення функції, компілятор почав вибирати функцію, що задовольняє тому, що занесено в покажчик, а не типу покажчика, як це було в попередньому прикладі.

У момент компіляції програми компілятор не генерує виклик функції `show()`, тому що не знає, до якого класу віднести вміст покажчика `ptr`, і відкладає прийняття рішення до фактичного запуску програми. А вже в момент виконання, коли відомо, на що вказує `ptr`, викликається відповідна версія `show()`. Такий підхід називають **пізнім зв'язуванням** або **динамічним зв'язуванням**. Вибір функції у звичайному порядку, під час компіляції, називається **раннім** або **статичним зв'язуванням**. Пізнє зв'язування вимагає більше ресурсів, але дає виграш у можливостях і гнучкості.

Для віртуальних функцій можна вказати наступні правила:

• віртуальну функцію не можна оголосити як static;

• специфікатор `virtual` необов'язковий при перевизначенні функції в похідному класі;

• віртуальна функція повинна бути або визначена, або описуватися як чисто віртуальна.

Таким чином, віртуальні функції надають програмісту можливість оголосити в базовому класі функції, які можна замінити в кожному

похідному класі. Компілятор і завантажник гарантують відповідність між об'єктами й функціями, застосовуваними до них.

Для того щоб оголошення віртуальної функції працювало як інтерфейс до функцій, визначених у похідним класах, типи аргументів функцій у похідному класі не повинні відрізнятися від типів аргументів, оголошених у базовому класі, і лише незначні зміни допускаються для типу значення, що вертається. Наприклад, якщо вихідний тип значення, що вертається, був `B*`, то тип значення, що вертається функцією може бути `D*` за умови, що `B` є відкритим базовим класом для `D`. Аналогічно замість `B&` тип значення, що вертається, може бути ослаблений до `D&`.

Віртуальна функція повинна бути визначена для класу, у якому вона вперше оголошена (виняток – чисто віртуальні функції). Віртуальну функцію можна використовувати, навіть якщо в її класу немає похідних класів. Похідний клас, який не має потреби у власній версії віртуальної функції, не зобов'язаний її реалізовувати. При створенні похідного класу можна реалізувати необхідну функцію, тільки якщо в цьому є необхідність.

Для того щоб обійти механізм віртуальних функцій, при виклику використовується ім'я класу з оператором дозволу області видимості.
***Приклад 5.4*** Ілюстрація обходу механізму віртуальних функцій.

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

 #include <stdio.h>

class Base {
public:

      int nonVitr() {return 0;}
      int Vitr() {return 1;}
virtual void Virt()
{ cout<<Base::Vitr(); }
void nonVirt()
{ cout<<Base::nonVitr(); }
};

class Derived: public Base
{
public:
void Virt()
{ cout<<Derived::Vitr(); }
```

```
void nonVirt()
{ cout<<Derived::nonVitr(); }
};

int main(void)
{
/* базовый указатель, реально ссылающийся на произ-
водный класс */
Base *bp = new Derived;
// вызов виртуальной функции
bp->Virt(); // будет напечатано Derived::Virt()
bp->Base::Virt();// будет напечатано Base::Virt()
// вызов не-виртуальной функции
bp->nonVirt();// будет напечатано Base::nonVirt()
cin.get();
}
```

Результат:

```
110
```

За винятком випадків, коли явно вказано, за допомогою оператора дозволу області видимості, яка версія віртуальної функції повинна викликатися, активізується найбільш підходящий для об'єкта варіант заміщеної функції.

Віртуальні функції реалізуються з використанням таблиці переходів у такий спосіб.

• Для кожного класу, що містить віртуальні функції, компілятор будує таблицю адрес цих функцій - таблицю віртуальних методів.

**Віртуальні класи**

Хоча операція дозволу видимості і представляє обхідний шлях, що дозволяє уникнути неоднозначності при складному наслідуванні, можна здійснити успадкування похідним класом лише одного екземпляру деякого базового класу. Цього можна досягти застосуванням ключового слова `virtual` при специфікації наслідуваного класу. Наприклад:

```
class A
{
protected:
int idata;
public:
void f();
…};
class B: virtual public A
{ … };
```

```
class C: virtual public A
{ … };
class D: public B, public C
{
// містить тільки один екземпляр класу A
…
};

int main(void)
{
D d;
d.f();
d.idata=10;//помилка: елемент даних недоступний
}
```

Якщо спробувати використовувати множинне спадкування без використання ключового слова `virtual`, то компілятор видасть повідомлення про помилку, тому що класи B і C у цьому випадку наслідують свою копію класу A. Така копія називається підоб'єктом. Кожний із двох підоб'єктів містить власну копію базового класу A, включаючи `f()`. Отже, виникне неоднозначність відносного того, у який із двох копій класу A викликати функцію `f()`. Для усунення неоднозначності використовують ключове слово `virtual`, що забезпечує спадкування класами B і C єдиного загального підоб'єкта базового класу A. Оскільки тепер є лише одна копія класу A, то неоднозначність при звертанні до базового класу усувається.

**Віртуальні деструктори**

Деструктори базового класу обов'язково повинні бути віртуальними. Наприклад, для видалення об'єкта породженого класу, виконано `delete` над покажчиком базового класу, що вказують на породжений клас. Якщо деструктор базового класу не є віртуальним, тоді `delete`, будучи звичайним методом, викличе деструктор для базового класу, замість того щоб запустити деструктор для породженого класу. Це призведе до того, що буде видалена тільки та частина об'єкта, яка відноситься до базового класу.

***Приклад 5.5*** Ілюстрація застосування віртуальних деструкторів.

```
#include <iostream>
using namespace std;
class Base
{
public:
~base() //невіртуальний деструктор
```

```
// virtual ~base() //віртуальний деструктор
{ cout << "Base видалений\n"; }
};
class Derv: public Base
{
public:
~derv()
{ cout << "Derv видалений\n"; }
};
void main()
{
      setlocale(LC_ALL,"Ukrainian");
Base* pbase = new Derv;
delete pbase;
cin.get();
}
```

У результаті виконання програми на екрані буде надруковано:

```
Base вилучений
```

Це говорить про те, що деструктор для `Derv` не викликався взагалі. До такого результату привело те, що деструктор базового класу в наведеному фрагменті коду невіртуальний. Виправити це можна, закоментувавши перший рядок визначення деструктора і активізувавши другий. Тепер результатом роботи програми є:

```
Derv видалений
Base видалений
```

Тільки тепер обидві частини об'єкта породженого класу видалені коректно. Тому в загальному випадку, щоб бути впевненим у тому, що об'єкти породжених класів правильно видаляються, слід завжди робити деструктори в базових класах віртуальними.

**Чисто віртуальні функції**

Для створення поліморфного об'єкта програми визначають один або кілька методів базового класу як віртуальні функції. Похідний клас може визначити свою власну функцію, яка виконується замість віртуальної функції базового класу, або використовувати базову функцію (похідний клас може і не визначати свій власний метод). Залежно від програми іноді не має змісту визначати віртуальну функцію в базовому класі. Наприклад, об'єкти похідних типів можуть настільки сильно відрізнятися, що їм не потрібно буде використовувати метод базового класу. У таких випадках замість визначення операторів для віртуальної функції базового класу програми можуть створити чисто віртуальну функцію, яка не містить операторів.

Для створення чисто віртуальної функції програма вказує прототип функції, але не вказує її оператори. Замість них програма присвоює функції значення нуль, як показано нижче:

```
class phone
{
 public:
   virtual void dial (char *number) =0; // Чисто
                                  //віртуальна функція
   void answer(void)
      { cout << "Очікування відповіді" << endl; }
 void hangup(void)
   {cout<<"Дзвінок виконано-покладіть трубку"<<endl;}
void ring(void)
      { cout << "Дзвінок, дзвінок, дзвінок" << endl;}
 phone(char *number)
      { strcpy(phone::number, number); };
 protected:
   char number[13];
 };
```

Кожний похідний клас, визначений у програмі, повинен визначити свою функцію замість чисто віртуальної функції базового класу. Якщо похідний клас вилучить визначення функції для чисто віртуальної функції, то компілятор C++ повідомить про синтаксичні помилки.

**Абстрактний клас**

Абстрактний клас є класом, який може використовуватися тільки в якості базового для інших класів. Абстрактний клас містить одну або кілька чисто віртуальних функцій. Чисто віртуальна функція може розглядатися як вбудована функція, тіло якої визначено як =0 (чистий специфікатор). Для чисто віртуальної функції не потрібно виконувати дійсне визначення; передбачається, що вона перевизначається в похідних класах.

До абстрактних класів застосовні наступні правила:

• абстрактний клас не може використовуватися в якості типу аргументу функції або типу значення, що вертається;

• абстрактний клас не можна використовувати в явному перетворенні;

• не можна визначити представник абстрактного класу (локальну/глобальну змінну або елемент даних);

• можна визначати покажчик або посилання на абстрактний клас;

• якщо клас, похідний від абстрактного, не визначає всі чисто віртуальні функції абстрактного класу, він також є абстрактним.

Наприклад:

```
class Bird
{
public:
void virtual Sing() = 0;
};
//клас Eagle - також абстрактний
class Eagle : public Bird
{};
// Goldeneagle - не є абстрактним
class Goldeneagle: public Bird
{
void Sign()
{…}
};
```

Важливим прикладом використання абстрактних класів є надання інтерфейсу з повною відсутністю деталей реалізації. Наприклад, операційна система може сховати деталі реалізації драйверів пристроїв за інтерфейсом абстрактного класу.

***Приклад 5.6*** Ілюстрація застосування абстрактних класів.

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

class CA { // Абстрактний клас
public:
CA ( void )
{ cout << "Цей обєкт належить класу "; }
virtual void Abstr(void)=0; /* Чисто віртуальна
функція */
void fun(void)
{ cout << "Реалізацяя не буде наслідуватися!";}
    ~CA ()
{ cout << "Деструктор  у дії" << endl; cin.get(); }
//Викликається у зворотньому порядку конструкторів
  };

class CB : public CA {
  public:
    CB (void)
      { cout << "CB"<<endl; }
     void Abstr(void)
```

```
{ cout << " Виклик функції cb.Abstr()"<<endl;}
//Підмінююча функція.
    void fun (void)
{ cout << " Виклик функції cb.fun()"<<endl; }
    ~CB ()
      {     cin.get();  } //Невірно для абстр. класу
~CB(){~CA();}
  };
class CC : public CA {
  public:
    CC (void)
{cout << "CC"<<endl; }
void Abstr(void)
{cout<<" Виклик функції cc.Abstr()"<<endl;}
//Підмінююча функція.
void fun(void)
{ cout << " Виклик функції cc.fun()"<<endl; }
      ~CC(){cin.get();}//Невірно для абстр. кл.
~CC(){ ~CA();}
};
void main ()
{
setlocale(LC_ALL,"Ukrainian");
cout << "Програма:" <<endl;
  CB cb;
  cb.Abstr();
  cb.fun();
  cb.~CB();
  CC cc;
  cc.Abstr();
  cc.fun();
  cc.~CC();
 cin.get();
  }
```

Результат роботи програми:

```
Програма:
Цей об'єкт належить класу CB
 Виклик функції cb.Abstr()
 Виклик функції cb.fun()
Деструктор у дії
Цей об'єкт належить класу CC
 Виклик функції cc.Abstr()
```

```
 Виклик функції cc.fun()
Деструктор у дії
Деструктор у дії
деструктор у дії
```

Отже, поліморфний об'єкт являє собою такий об'єкт, який може змінювати форму під час виконання програми. Наприклад, необхідно написати програму, яка емулює телефонні операції. Для цього можна вибрати загальні операції, такі як набір номера, дзвінок, роз'єднання і індикація зайнятості. Визначимо можливість емуляції дискового, кнопкового або платного телефону на вибір. Інакше кажучи, для одного дзвінка об'єкт-телефон міг би представляти кнопковий апарат, для іншого виступав би як платний телефон і т.д. Інакше кажучи, від одного дзвінка до іншого об'єкт-телефон повинен змінювати форму.

У цих різних класах телефону існує єдина відмінна функція – це метод `dial`. Для створення поліморфного об'єкта спочатку слід визначаєти функції базового класу, які відрізняються від функцій похідних класів тим, що вони віртуальні, випереджаючи їх прототипи ключовим словом `virtual`, як показано нижче:

```
class phone
{
 public:
   virtual void dial(char number)
       {
       cout << "Набір номера " << number << endl;
       }
       void answer(void)
       {cout << "Очікування відповіді" << endl; }
       void hangup(void)
  {cout<<"Дзвінок виконаний-покладіть трубку"<<endl;}
       void ring(void)
       { cout << "Дзвінок, дзвінок, дзвінок" << endl;}
       phone(char *number)
       {strcpy(phone::number, number); };
protected:
       char number[13];
};
```

Далі необхідно створити покажчик на об'єкт базового класу. Для телефонної програми створити покажчик на базовий клас `phone`:

```
       phone *poly_phone;
```

Для зміни форми об'єкта достатньо присвоїти цьому покажчику адреса об'єкта похідного класу, як показано нижче:

```
       poly_phone = (phone *) &home_phone;
```

Символи (phone *), які ідуть за оператором присвоювання, є оператором приведення типів, який повідомляє компілятора C++, що необхідно присвоїти адресу змінної одного типу (touch_tone) покажчику на змінну іншого типу (phone). Оскільки програма може присвоювати покажчику об'єкта poly_phone адреси різних об'єктів, то цей об'єкт може змінювати форму, а отже, є поліморфним.

***Приклад 5.7*** Створення об'єкта-телефону. Після запуску програми об'єкт poly_phone змінює форму з дискового телефону на кнопковий, а потім на платний.

```
#include <iostream>
#include <cstring>
using namespace std;
class phone
{
public:
      virtual void dial(char *number)
      { cout << "Набір номера " << number << endl; }
      void answer(void)
       { cout << "Очікування відповіді" << endl; }
      void hangup(void)
      { cout << "Дзвінок виконаний - покладіть
трубку" << endl; }
      void ring(void)
      { cout << "Дзвінок, дзвінок, дзвінок" << endl;}
      phone(char *number)
      { strcpy(phone::number, number);
      };
protected:
      char number[13] ;
};
class touch_tone : phone
{
public:
      void dial(char * number) { cout << "Пік пік
Набір номера " << number << endl; }
      touch_tone(char *number) : phone(number) { }
};
class pay_phone: phone
{
public:
```

```
      void dial(char *number) { cout << " Будь ласка,
оплатіть " << amount << " гривень" << endl; cout <<
"Набір номера "       << number << endl; }
      pay_phone(char *number, int amount) :
phone(number) { pay_phone::amount = amount; }
private:
      int amount;
};
void main(void)
{
      setlocale(LC_ALL,"Ukrainian");
      pay_phone city_phone("702-555-1212", 25);
      touch_tone home_phone("555-1212");
      phone rotary("201-555-1212") ;
      // Зробити об'єкт дисковим телефоном
      phone *poly_phone = &rotary;
      poly_phone->dial("818-555-1212");
      // Замінити форму об'єкта на кнопковий телефон
      poly_phone = (phone *) &home_phone;
      poly_phone->dial("303-555-1212");
      // Замінити форму об'єкта на платний телефон
      poly_phone = (phone *) &city_phone;
      poly_phone->dial("212-555-1212");
      cin.get();
}
```

Результати роботи програми:

```
Набір номера 818-555-1212
Пік пік Набір номера 303-555-1212
Будь ласка, оплатите 25 гривень
Набір номера 212-555-1212
```

Оскільки об'єкт `poly_phone` змінює форму в процесі виконання програми, він є поліморфним.

### Виключні ситуації

Після того як створено і й підлагоджено (виправлено помилки) в програмі, можливим є передбачення помилок, які можуть в ній зустрітися. Наприклад, якщо програма читає інформацію з файлу, їй необхідно перевірити, чи існує файл і чи може програма його відкрити. Аналогічно, якщо програма використовує оператор `new` для виділення пам'яті, їй необхідно перевірити й відреагувати на можливу відсутність пам'яті. У міру збільшення розміру і складності програм може виявитися, що необхідно включити в програму багато таких перевірок в усій програмі.

Для цього використовуються така звані виняткові ситуації в C++ для спрощення перевірки й обробки помилок.

Виняткова ситуація (exception) являє собою несподівану подію – помилку – у програмі. У програмах виняткові ситуації визначаються як класи. Для відстеження виняткових ситуацій необхідно використовувати оператор try, а для виявлення певної виняткової ситуації використовують оператор catch.

Для генерації виняткової ситуації при виникненні помилки використовують оператор C++ throw.

Якщо програма виявляє виняткову ситуацію, вона викликає спеціальну (характерну для даної виняткової ситуації) функцію, яка називається обробником виняткової ситуації.

### C++ і виняткові ситуації

Метою використання виняткових ситуацій C++ є спрощення виявлення і обробки помилок у програмах. Якщо програма виявляє несподівану помилку (виняткову ситуацію), то здійснюється обробка такої ситуації замість того, щоб просто припинити виконання.

У програмах визначається кожна виняткова ситуація як клас. Наприклад, виняткові ситуації для роботи з файлами можна описати:

```
class file_open_error {};
class file_read_error {};
```

### Перевірка виняткових ситуацій

Перш ніж програми можуть виявити і відреагувати на виняткову ситуацію, необхідно використати оператор C++ try для дозволу виявлення виняткової ситуації. Наприклад, оператор try дозволяє виявлення виняткової ситуації для виклику функції file_copy:

```
try
{
   file_copy("SOURCE.TXT", "TARGET.TXT") ;
};
```

За оператором try програма повинна розмістити один або кілька операторів catch, щоб визначити, яка виняткова ситуація мала місце (якщо вона взагалі була):

```
try
{
   file_copy("SOURCE.TXT", "TARGET.TXT") ;
};
catch (file_open_error)
{
```

```
        cerr << "Помилка відкриття вихідного або
цільового файлу" << endl;
        exit(1);
     }
     catch (file_read_error)
     {
     cerr <<"Помилка читання вихідного файлу" <<endl;
        exit(1);
      }
     catch (file_write_error)
     {
     cerr << "Помилка запису цільового файлу"<< endl;
        exit(1);
     }
```

Даний код перевіряє виникнення виняткових ситуацій роботи з файлами, створених раніше. У цьому випадку незалежно від типу помилки код просто виводить повідомлення і завершує програму. Найкраще описувати виключні ситуації зі спробою визначення причини помилки і повтору операції. Якщо виклик функції пройшов успішно і виняткова ситуація не виявлена, C++ просто ігнорує оператори `catch`.

### Використання оператора `throw` для генерації виняткової ситуації

Сам C++ не генерує виняткові ситуації. Їх генерують програми, використовуючи оператор C++ `throw`. Наприклад, всередині функції `file_copy` програма може перевірити умову виникнення помилки і згенерувати виняткову ситуацію:

```
     void file_copy(char *source, char *target)
     {
        char line[256];
        ifstream input_file(source);
        ofstream output_file(target);
        if (input_file.fail())
            throw(file_open_error);
        else
            if (output_file.fail())
throw(file_open_error);
        else
        {
           while ((! input_file.eof()) && (!
                              input_file.fail()))
           {
             input_file.getline(line, sizeof(line)) ;
```

```
            if (! input_file.fail()) output_file <<
                                        line << endl;
            else throw(file_read_error);
            if (output_file.fail()) throw
                                (file_write_error);
        }
    }
}
```

Дана функція `file_copy` використовує оператор `throw` для генерації певних виняткових ситуацій.

**Як працюють виняткові ситуації**

Коли використовуються виняткові ситуації, програма перевіряє умову виникнення помилки і, якщо необхідно, генерує виняткову ситуацію, використовуючи оператор `throw`. Коли C++ зустрічає оператор `throw`, він активізує відповідний обробник виняткової ситуації (функцію, оператори якої визначені в класі виняткової ситуації). Після завершення функції обробки виняткової ситуації C++ повертає керування першому оператору, що іде за оператором `try`, який дозволив виявлення виняткової ситуації. Далі, використовуючи оператори `catch`, програма може визначити, яка саме виняткова ситуація виникла, і відреагувати відповідним чином.

**Визначення обробника виняткової ситуації**

Коли програма генерує виняткову ситуацію, C++ запускає обробник виняткової ситуації (функцію), оператори яких визначено в класі виняткової ситуації. Наприклад, є клас виняткової ситуації `nuke_meltdown` з операторами обробника виняткової ситуації у функції `nuke_meltdown`:

```
class nuke_meltdown
{
public:
   nuke_meltdown(void)
{ cerr << "Ура! Працюю! Працюю!" << endl;}
};
```

У цьому випадку, коли програма згенерує виняткову ситуацію `nuke_meltdown`, C++ запустить оператори функції `nuke_meltdown`, перш ніж поверне керування першому оператору, що слідує за оператором `try`, що дозволяє виявлення виняткової ситуації.

***Приклад 5.8.*** Ілюстрація використання оператора `try` для виявлення виняткової ситуації. Програма викликає функцію `add_u232` з параметром `amount`. Якщо значення цього параметра менше 255, то

функція виконується успішно. Якщо ж значення параметра перевищує 255, функція генерує виняткову ситуацію nuke_meltdown.

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

class nuke_meltdown
{
public:
   nuke_meltdown(void);
};

nuke_meltdown::nuke_meltdown(void)
   {
         cerr << "Ура! Працюю! Працюю!" << endl;
   }

void add_u232(int amount)
{
   if (amount < 255)
       cout << "Параметр add_u232 у порядку" << endl;
   else throw nuke_meltdown();
}
void main(void)
{
   setlocale(LC_ALL,"Ukrainian");
      try
   {
      add_u232(255);
   }
   catch (nuke_meltdown)
   {
      cerr << "Програма стійка" << endl;
   }
   cin.get();
}
```

Результати роботи програми:

```
Ура! Працюю! Працюю!
Програма стійка
```

Якщо простежити початковий код, який генерує кожне з повідомлень, то можна переконатися, що потік керування при виникненні виняткової ситуації проходить через обробник виняткової ситуації і назад до оператора catch. Так, перший рядок виводу генерується обробником

виняткової ситуації, тобто функцією nuke_meltdown. Другий рядок виводу генерується в операторові catch, який виявив виняткову ситуацію.

**Виняткові ситуації і класи**

При створенні класу можна визначити виняткові ситуації, характерні для даного класу. Для цього необхідно включити цю виняткову ситуацію в якості одного із загальних (public) елементів класу. Наприклад опис класу string визначає дві виняткові ситуації:

```
class string
{
public:
   string(char *str);
   void fill_string(*str);
   void show_string(void);
   int string_length(void);
   class string_empty { };
   class string_overflow {};
private:
   int length;
   char string[255];
};
```

Як видно, даний клас визначає виняткові ситуації string_empty і string_overflow. У програмі можна перевірити наявність виняткової ситуації, використовуючи оператор глобального дозволу та імена класу, як показано нижче:

```
try
{ some_string.fill_string(some_long_string);  };
catch (string::string_overflow)
{
   cerr << "Перевищена довжина рядка, символи
відкинуті" << endl;  }
```