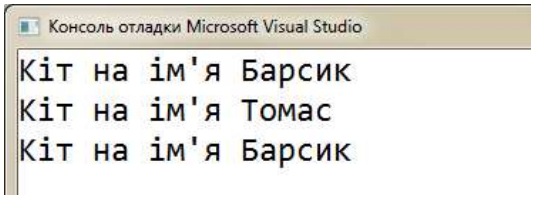
Лекція 5.1 Конструктор копії

Передача об’єктів функціям

Об'єкт можна передати функції точно так же, як значення будь-якого іншого типу даних. Функції передається не сам об'єкт, а його копія. Отже, зміни, внесені в об'єкт при виконанні функції, не мають жодного впливу на об'єкт, який використовується в якості аргументу для функції. Цей механізм демонструється в наступній програмі.

```
class Cat {
 string name;
public:
 void SetName(string Name) { name = Name; }
 void Print() { cout << "Кіт на ім'я " << name<<endl; }
};
void f(Cat cat)
{
 cat.Print(); // Виводить ім'я.
 cat.SetName("Томас"); // Встановлює тільки локальну копію.
 cat.Print();  // Виводить локальну копію Томас.
}
int main()
{
 setlocale(0, "ukr");
 Cat cat;
 cat.SetName("Барсик");
 f(cat);
 cat.Print(); // Виводить ім'я Барсик. Значення не змінилося.
}
```

Ось як виглядають результати виконання цієї програми.



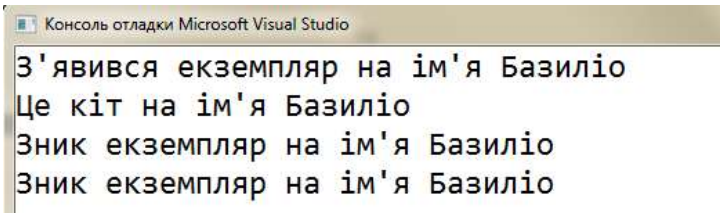
Як підтверджують ці результати, модифікація об'єкта cat в функції f () не впливає на об'єкт cat в функції main().

Конструктори, деструктори і передача об'єктів

Незважаючи на те що передача функціям нескладних об'єктів в якості аргументів – досить проста процедура, при цьому можуть відбуватися непередбачені події, що мають відношення до конструкторів і деструкторів. Щоб розібратися в цьому, розглянемо наступну програму.

```
class Cat {
 string name;
public:
 Cat(string Name) { name = Name;
      cout << "З'явився екземпляр на ім'я " << name << endl; }
 ~Cat() { cout << "Зник екземпляр на ім'я " << name << endl; }
 void Print() { cout << "Це кіт на ім'я " << name << endl; }
};

void f(Cat cat)
{
 cat.Print(); // Виводить ім'я.
}

int main()
{
 setlocale(0, "ukr");
 Cat cat("Базиліо");
 f(cat);
}
```

При виконанні ця програма виводить наступні несподівані результати.



Як бачите, тут виконується одне звернення до функції конструктора (при створенні об'єкта cat), але чомусь два звернення до функції деструктора. Давайте розбиратися, в чому тут справа.

При передачі об'єкта функції створюється його копія (і ця копія стає параметром в функції). Створення копії означає «народження» нового об'єкта. Коли виконання функції завершується, копія аргументу (тобто параметр) руйнується. Тут виникає відразу два питання. По-перше, чи викликається

конструктор об'єкта при створенні копії? По-друге, чи викликається деструктор об'єкта при руйнуванні копії? Відповіді можуть здивувати вас.

Коли при виконанні функції створюється копія аргументу, звичайний конструктор не викликається! Замість цього викликається конструктор копії об'єкта. Конструктор копії визначає, як повинна бути створена копія об'єкта. Але якщо в класі явно не визначений конструктор копії, C++ надає його за замовчуванням. Конструктор копії за замовчуванням створює побітову (тобто ідентичну) копію об'єкта. Оскільки звичайний конструктор використовується для ініціалізації об'єкта, він не повинен викликатися для створення копії вже існуючого об'єкта. Такий виклик змінив би його вміст. При передачі об'єкта функції має сенс використовувати поточний стан об'єкта, а не його початковий стан.

Але коли функція завершується і руйнується копія об'єкта, яка використовується в якості аргументу, викликається деструктор цього об'єкта. Необхідність виклику деструктора пов'язана з виходом об'єкта з області видимості. Саме тому попередня програма мала два звернення до деструктора. Перше відбулося при виході з області видимості параметра функції f(Cat cat), а друге – при руйнуванні об'єкта cat в функції main () по завершенні програми.

Отже, коли об'єкт передається функції як аргумент, звичайний конструктор не викликається. Замість нього викликається конструктор копії, який за замовчуванням створює побітову (ідентичну) копію цього об'єкта. Але коли ця копія руйнується (зазвичай при виході за межі області видимості по завершенні функції), обов'язково викликається деструктор.

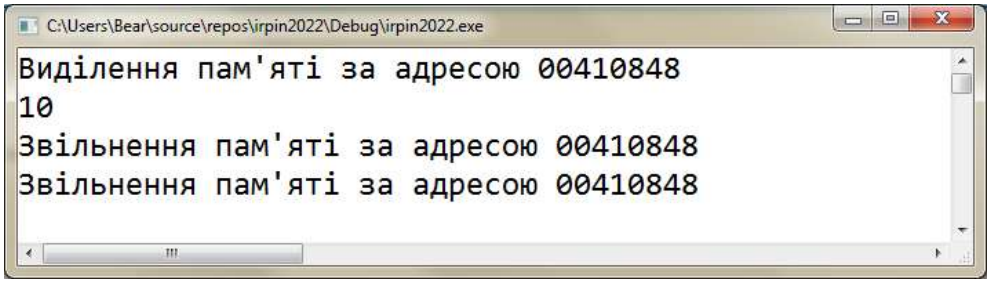
Потенційні проблеми при передачі параметрів

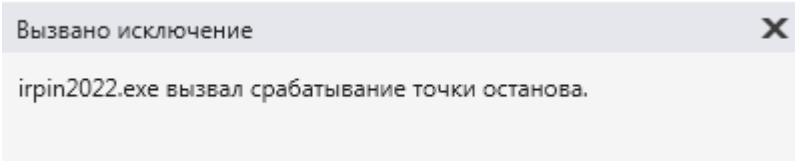
Незважаючи на те що об'єкти передаються функцій «за значенням», тобто за допомогою звичайного механізму передачі параметрів, який теоретично захищає аргумент, тут все-таки можливий побічний ефект або навіть загроза для «життя» об'єкта, який використовується в якості аргументу. Наприклад, якщо об'єкт, який використовується як аргумент, вимагає динамічного виділення пам'яті та звільняє цю пам'ять при руйнуванні, його локальна копія

при виклику деструктора звільнить ту ж саму область пам'яті, яка була виділена оригінальному об'єкту. І цей факт стає вже цілою проблемою, оскільки оригінальний об'єкт все ще використовує цю (вже звільнену) область пам'яті. Описана ситуація робить вихідний об'єкт «неповноцінним» і, по суті, непридатним для використання. Розглянемо наступну просту програму.

```
// Демонстрація проблеми, можливої при передачі об'єктів функцій.
class MyCl
{
 int *p;
public:
 MyCl(int i)
 {
       p = new int;
       *p = i;
       cout << "Виділення пам'яті за адресою " << p << endl;
 }

 ~MyCl()
 {
       cout <<  "Звільнення пам'яті за адресою " << p << endl;
       delete p;
 }
 int GetVal() { return *p; }
};
// При виконанні цієї функції і виникає проблема.
void GetInfo(MyCl ob)
{
 cout << ob.GetVal() << endl;
}
int main()
{
 setlocale(0, "Ukr");
 MyCl a(10);
 GetInfo(a);
}
```

Ось як виглядають результати виконання цієї програми:





Ця програма містить принципову помилку. І ось чому: при створенні в функції main() об'єкта a виділяється область пам'яті, адреса якої присвоюється вказівнику p об’єкта a. При передачі функції GetInfo() об'єкт a копіюється в параметр ob. Це означає, що обидва об'єкти (a і ob) матимуть однакове значення для вказівника p.

Іншими словами, в обох об'єктах (в оригіналі та його копії) член даних p буде вказувати на одну і ту ж динамічно виділену область пам'яті. По завершенні функції GetInfo() об'єкт ob руйнується, і його руйнування супроводжується викликом деструктора. Деструктор звільняє область пам'яті, що адресується вказівником ob.p. Але ж ця (вже звільнена) область пам'яті – та ж сама область, на яку все ще вказує член даних (вихідного об'єкта) a.p! У наявності серйозна помилка.

Насправді справи йдуть ще гірше. По завершенні програми руйнується об'єкт a, і динамічно виділена (ще при його створенні) пам'ять звільняється вдруге. Справа в тому, що звільнення однієї і тієї ж області динамічно виділеної пам'яті вдруге вважається невизначеною операцією, яка, як правило (в залежності від того, як реалізована система динамічного розподілу пам'яті), викликає непоправну помилку.

Повернення об'єктів функціями

Якщо об'єкти можна передавати функціям, то «з таким же успіхом» функції можуть повертати об'єкти. Щоб функція могла повернути об'єкт, по-перше, необхідно оголосити в якості типу значення що повертається нею тип відповідного класу. По-друге, потрібно забезпечити повернення об'єкта цього типу за допомогою звичайної інструкції return. Розглянемо приклад функції, яка повертає об'єкт.

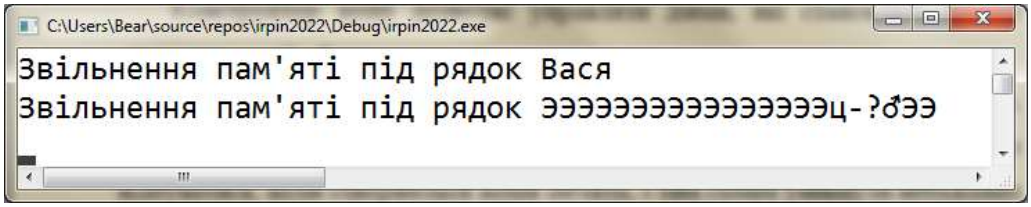
Щодо повернення об'єктів функціями важливо розуміти наступне. Якщо функція повертає об'єкт класу, вона автоматично створює тимчасовий об'єкт,

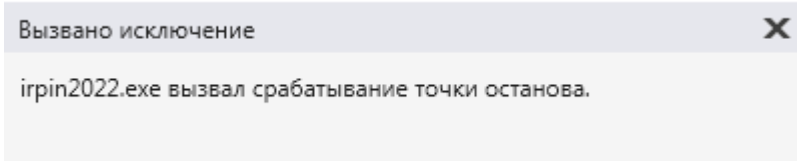
який зберігає значення, що повертається. Саме цей об'єкт реально і повертається функцією. Після повернення значення об'єкт руйнується. Руйнування тимчасового об'єкта в деяких ситуаціях може викликати непередбачені побічні ефекти. Наприклад, якщо об'єкт, що повертається функцією, має деструктор, який звільняє пам'ять, що динамічно виділяється. Ця пам'ять буде звільнена навіть в тому випадку, якщо об'єкт, який одержує значення, що повертається функцією, все ще її використовує. Розглянемо програму:

```
// Помилка, що генерується при поверненні об'єкта функцією.
class Text {
 char *Ptext;
public:
 Text() { Ptext = nullptr; }
 ~Text() {
       if(Ptext)
       {
             cout << "Звільнення пам'яті під рядок " << Ptext << endl;
             delete[] Ptext;
       }
 }
 void Print() { cout << Ptext << endl; }
 // Завантаження рядка.
 void SetStr(char *str)
 {
       Ptext = new char[strlen(str) + 1];
       strcpy(Ptext, str);
 }
};
// Ця функція повертає об'єкт типу Text.
Text input()
{
 char instr[]="Вася";
 Text str;
 str.SetStr(instr);
 return str;
}


int main()

{

 setlocale(0, "Ukr");

 Text ob;

 // Надаємо об'єкт ob, що повертається функцією input()
```

```
 ob = input(); // Ця інструкція генерує помилку!!!!
 ob.Print(); // Відображення "сміття".
}
```

Результати виконання цієї програми виглядають таким чином:





Коли функція повертає об'єкт, автоматично генерується невидимий тимчасовий об'єкт, який зберігає значення, що повертається. В даному випадку цей об'єкт просто є побітову копію об'єкта str, який є значенням, що повертається функцією. Отже, після повернення з функції виконується деструктор тимчасового об'єкта. Оскільки область пам'яті для зберігання рядка, що вводиться користувачем, вже була звільнена, при виконанні функції Print() на екран виведеться «сміття».

Для вирішення цієї проблеми замість повернення об'єкта з функції використовується повернення вказівника або посилання на об'єкт. Але це не завжди можливо. Ще один спосіб вирішення цієї проблеми включає використання конструктора копії, якому присвячений наступний розділ.

Створення і використання конструктора копії

Одним з найбільш важливих форматів перевантаженого конструктора є конструктор копії. Як було показано в попередніх прикладах, при передачі об'єкта функції або повернення об'єкту з функції можуть виникати проблеми. Один із способів уникнути цих проблем полягає у визначенні конструктора копії, який являє собою спеціальний тип перевантаженого конструктора.

Конструктор копії дозволяє управляти діями, які становлять процес створення копії об'єкта.

В основі розглянутих проблем лежить створення побітової копії об'єкта. Щоб запобігти їх виникненню, необхідно точно визначити, що повинно відбуватися, коли створюється копія об'єкта, і тим самим уникнути небажаних побічних ефектів. Цього можна домогтися шляхом створення конструктора копії.

Перш ніж докладніше знайомитися з використанням конструктора копії, важливо розуміти, що в C++ визначено два окремі види ситуацій, в яких значення одного об'єкта передається іншому. Першою такою ситуацією є присвоювання, а другий – ініціалізація. Ініціалізація може виконуватися трьома способами, тобто у випадках, коли:

- один об'єкт явно ініціалізує інший об'єкт, як, наприклад, в оголошенні;

- копія об'єкта передається параметру функції;

- генерується тимчасовий об'єкт (найчастіше в якості значення, що повертається функцією).

Конструктор копії застосовується тільки до ініціалізації. Він не застосовується до присвоювання.

Ось як виглядає найпоширеніший формат конструктора копії:

```
ClassName(const ClassName & obj) {
 // тіло конструктора
}
```

Тут елемент obj означає посилання на об'єкт, який використовується для ініціалізації іншого об'єкта.

При передачі об'єкта функції в якості аргументу створюється копія цього об'єкта. Якщо в класі визначено конструктор копії, то саме він і викликається для створення копії.

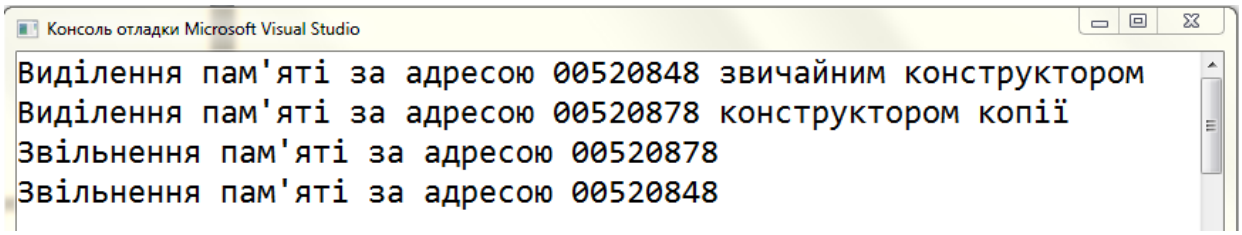
Конструктор копії також викликається в разі, коли один об'єкт використовується для ініціалізації іншого.

Розглянемо наступну просту програму.

```
class MyCl
{
 int *p;
public:
```

```
 // звичайний конструктор.
 MyCl(int i)
 {
       p = new int;
       *p = i;
       cout << "Виділення пам'яті за адресою " << p;
       cout << " звичайним конструктором "<< endl;
 }
 // конструктор копії
 MyCl(const MyCl &obj)
 {
       p = new int;
       *p = *obj.p; // значення копії
       cout << "Виділення пам'яті за адресою " << p;
       cout << " конструктором копії" << endl;
 }
 ~MyCl()
 {
       cout << "Звільнення пам'яті за адресою " << p << endl;
       delete p;
 }
};


int main()
{
 setlocale(0, "Ukr");
 MyCl a(10); // Викликається звичайний конструктор.
 MyCl b = a; // Викликається конструктор копії.
}
```

Результати виконання цієї програми такі:



Слід мати на увазі, що конструктор копії викликається тільки в разі виконання ініціалізації. Наприклад, наступна послідовність інструкцій не викличе конструктор копії, визначений у попередній програмі:

```
MyCl a (2), b (3);
// ...
b = a;
```

Тут інструкція b = a виконує операцію присвоювання, а не операцію копіювання.

Конструктор копії також викликається при створенні тимчасового об'єкта, який є результатом повернення функцією об'єкта.

Початківці у програмуванні часто не розуміють, чому такий важливий конструктор копії. Для багатьох не відразу стає очевидним відповідь на питання: коли потрібен конструктор копії, а коли – ні. Ця ситуація часто виражається в такій формі: «А чи не існує більш простого способу?». Відповідь також непроста: і так, і ні!

Такі мови, як Java і C #, не мають конструкторів копії, оскільки в жодному з них не створюються побітові копії об'єктів. Справа в тому, що як Java, так і C# динамічно виділяють пам'ять для всіх об'єктів, а програміст оперує цими об'єктами виключно через посилання. Тому при передачі об'єктів в якості параметрів функції або при поверненні їх з функцій в копіях об'єктів немає ніякої необхідності.

Той факт, що а ні Java, а ні C# не потребують конструкторів копії, робить ці мови простіше, але за простоту теж потрібно платити. Робота з об'єктами виключно за допомогою посилань (а не безпосередньо, як в C++) накладає обмеження на тип операцій, які може виконувати програміст.