ЛАБОРАТОРНА РОБОТА 4
ВИКОРИСТАННЯ КОНСТРУКТОРА КОПІЇ

Мета роботи: дослідження і вирішення проблем, що виникають при при передачі об'єкта функції або повернення об'єкту з функції і уникнення цих проблем через визначення конструктора копії.

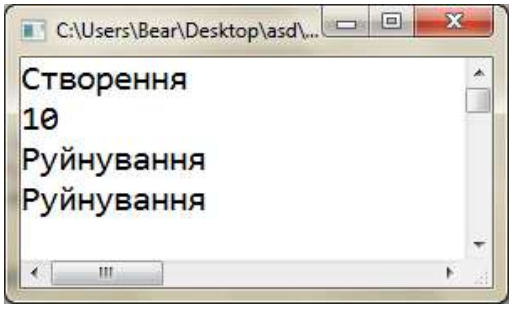
Основні теоретичні відомості

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.

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

class myclass {
int val;
public:
myclass(int i) { val = i; cout << "Створення"<<endl; }
~myclass() { cout << "Руйнування"<<endl; }
int getval() { return val; }
};

void display(myclass ob)
{
cout << ob.getval() <<endl;
}

int main()
{
setlocale(0,"Ukr");
if(true)
 {
  myclass a(10);
  display(a);
 } // вихід за межі області видимості
}
```

При виконанні ця програма виводить наступні несподівані результати.



Тут виконується одне звернення до функції конструктора (при створенні об'єкта a), але чомусь два звернення до функції деструктора.

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.

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 display (), а друге – при руйнуванні об'єкта *a* 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

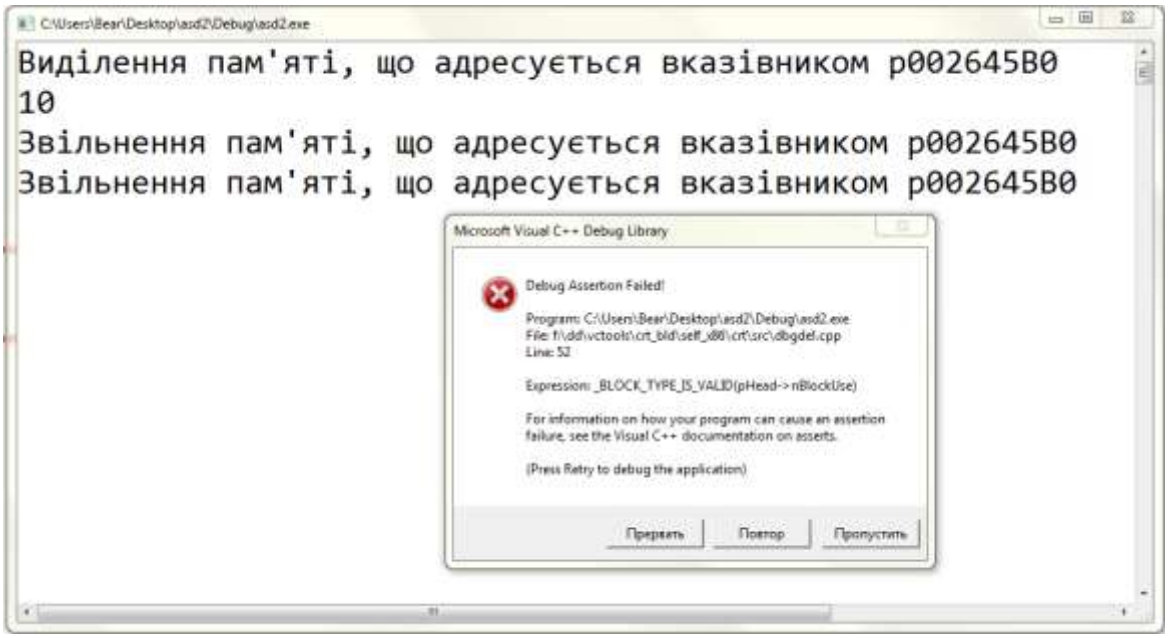
Якщо об'єкт, який використовується як аргумент, вимагає динамічного виділення пам'яті та звільняє цю пам'ять при руйнуванні, його локальна копія при виклику деструктора звільнить ту ж саму область пам'яті, яка була виділена оригінальному об'єкту.

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

    class myclass
    {
    int *p;
    public:
    myclass (int i);
    ~ myclass ();
    int getval () {return * p; }
    };

    myclass :: myclass (int i)
    {

    p = new int;
    *p = i;
    cout << "Виділення пам'яті, що адресується вказівником
p"<<p<<endl;
    }

    myclass :: ~ myclass ()
    {
    cout << "Звільнення пам'яті, що адресується вказівником
p"<<p<<endl;
    delete p;
    }
    // При виконанні цієї функції і виникає проблема.
    void display (myclass ob)
    {
```

```
cout << ob.getval () << endl;
}

int main ()
{
if(true)
   {
   myclass a (10);
   display (a);
   } // вихід за межі області видимості
}
```

Ось як виглядають результати виконання цієї програми:



В обох об'єктах (в оригіналі та його копії) вказівник p буде вказувати на одну і ту ж динамічно виділену область пам'яті. По завершенні функції display() 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.

Повернення об'єктів функціями

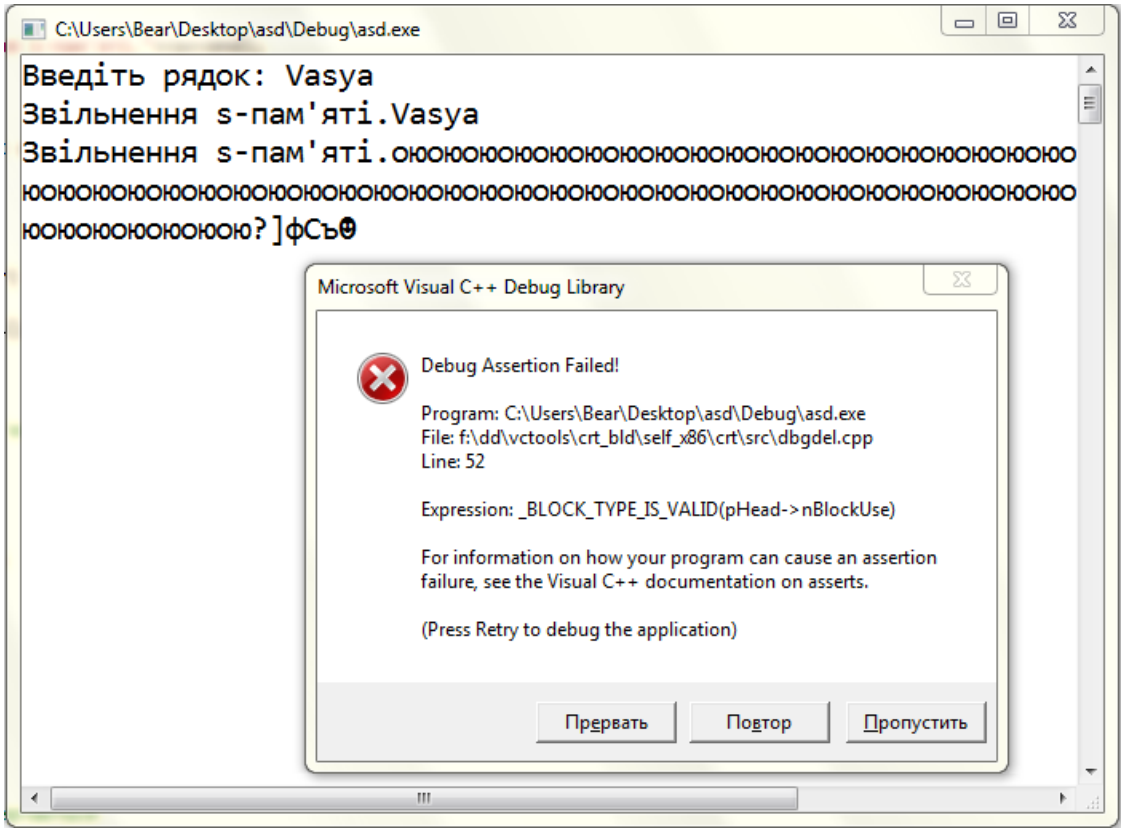
Щодо повернення об'єктів функціями важливо розуміти наступне. Якщо функція повертає об'єкт класу, вона автоматично створює тимчасовий об'єкт, який зберігає значення, що повертається. Саме цей об'єкт реально і повертається функцією. Після повернення значення об'єкт руйнується. Руйнування тимчасового об'єкта 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.
#include <iostream>
#include <string.h>
#include <conio.h>
using namespace std;

class sample {
char *s;
public:
sample() { s = 0; }
~sample() {
if(s)
 {
      cout << "Звільнення s-пам'яті."<<s<<endl;
      delete [] s;
    }
}
void show() { cout << s << endl; }
void set(char *str);
};

// Завантаження рядка.
void sample::set(char *str)
{
s = new char[strlen(str)+1];
strcpy(s, str);
}

// Ця функція повертає об'єкт типу sample.
sample input()
{
char instr[80];
sample str;
cout << "Введіть рядок: ";
cin >> instr;
str.set(instr);
return str;
}
```

```
    int main()
    {
    sample ob;
    // Надаємо об'єкт, що повертається
    // функцією input (), об'єкту ob.
    ob = input(); // Ця інструкція генерує помилку!!!!
    ob.show(); // Відображення "сміття".
    _getch();
    return 0;
}
```

Результати виконання цієї програми виглядають таким чином:



Створення і використання конструктора копії

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 привласнення.

Ось як виглядає найпоширеніший формат конструктора копії.

ім'я_класу (const ім'я_класу & 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.

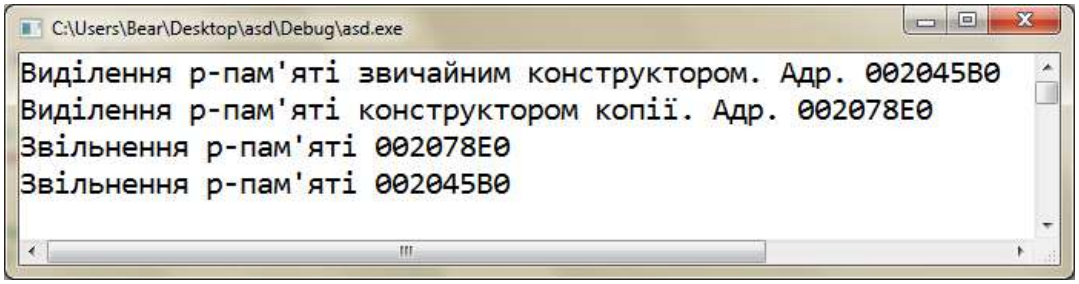
Конструктор копії також викликається в разі, коли один об'єкт використовується для ініціалізації іншого.

Розглянемо наступну просту програму.

```
// Виклик конструктора копії для ініціалізації об'єкта.
#include <iostream>
#include <string.h>
#include <conio.h>
using namespace std;

class myclass {
int *p;
public:
myclass(int i); // звичайний конструктор
myclass(const myclass &ob); // конструктор копії
~myclass();
int getval() { return *p; }
};

// конструктор копії.
```

```
    myclass::myclass(const myclass &ob)
    {
    p = new int;
    *p = *ob.p; // значення копії
    cout << "Виділення р-пам'яті конструктором копії. Адр.
"<<p<<endl;
    }

    // звичайний конструктор.
    myclass::myclass(int i)
    {
    p = new int;
    *p = i;
    cout << "Виділення р-пам'яті звичайним конструктором. Адр.
"<<p<<endl;
    }

    myclass::~myclass()
    {
    cout << "Звільнення р-пам'яті "<<p<<endl;
    delete p;
    }

    int main()
    {
    setlocale(0, "Ukr");
    if(true)
       {
       myclass a(10); // Викликається звичайний конструктор.
       myclass b = a; // Викликається конструктор копії.
       } // вихід за межі області видимості
    }
```

Результати виконання цієї програми такі



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

Завдання до лабораторної роботи:

Створити клас, конструктор якого генерує динамічний масив чисел (див. варіанти) Розмір масиву задається параметром конструктора. Деструктор звільняє пам’ять, що виділена під масив. Створити функцію, параметром якої є створений клас. Функція виконує дії над масивом певного типу відповідно варіанту (таблиця 4.1).

Варіанти завдань

Таблиця 4.1

№ варіанта	Тип даних	Кількість елементів	Вираз для обчислення елемента масиву		Вказівки до обробки
			парного	непарного	
1.	Дійсний	15	i + 4.1	i – 1.0	Знайти суму усіх парних елементів
2.	Цілий	8	i + 5	i – 2	Знайти кількість непарних елементів
3.	Цілий	9	i	i – 4	“Перевернути” масив (1234 -> 4321)
4.	Дійсний	13	i - 3.8	i + 1.5	Знайти суму всіх додатних елементів
5.	Дійсний	12	i - 6.0	i	Знайти кількість від’ємних елементів
6.	Цілий	11	i + 3	i – 7	Знайти суму кожного третього елемента
7.	Цілий	8	i - 4	i – 6	Знайти добуток непарних елементів
8.	Дійсний	14	i - 1.9	i	“Перевернути” масив (1234 -> 4321)
9.	Цілий	9	i - 2	i + 5	Знайти суму усіх непарних елементів
10.	Цілий	10	i	i – 1	Знайти добуток кожного третього елемента
11.	Дійсний	12	i + 5.1	i – 4.2	Знайти суму кожного другого елемента
12.	Цілий	11	i - 5	i	Знайти кількість парних елементів
13.	Цілий	8	i	i – 6	Знайти добуток усіх від’ємних елементів
14.	Цілий	14	i - 1	i	Знайти суму усіх додатних елементів
15.	Дійсний	14	i - 1.9	i	“Перевернути” масив (4321 ->1234)

Контрольні питання

1. Коли необхідно визначати конструктор копії?
2. Назвіть випадки, коли викликається конструктор копії?
3. Що створює конструктор копії за замовчуванням?
4. Що створюється при передачі об'єкта функції в якості аргумента?
5. Коли не викликається конструктор копії?