Лабораторна робота №15. Розумні покажчики

Тема. Розумні покажчики.

Мета. По результатах практичної роботи порівняти розумні покажчики бібліотеки STL.

1 ВИМОГИ

1.1 Розробник

Інформація про розробника:

- Кононенко Дмитро Олексійович
- НТУ “ХПІ”,
- КІТ 102.8а

1.2 Завдання

Створити STL контейнер, що містить у собі об’єкти ієрархії класів, використати розумні покажчики:

- auto_ptr;
- unique_ptr;
- shared_ptr;
- weak_ptr.

Додаткове завдання на оцінку «відмінно»:
Створити власний розумний покажчик, що представлений у вигляді шаблонного класу, який:

- має перевантажений оператор * та -> для отримання фактичного об’єкта та його покажчика;
- дозволяє підраховувати кількість покажчиків на об’єкт. Продемонструвати дії, коли виникає інкремент та декремент кількості покажчиків;
- контролювати виток пам’яті при виникненні виняткової ситуації.

2. Опис програми

2.1 Призначення

Програма призначена для ознайомлення з розумними покажчиками та їх недоліками.

2.2 Опис логічної структури

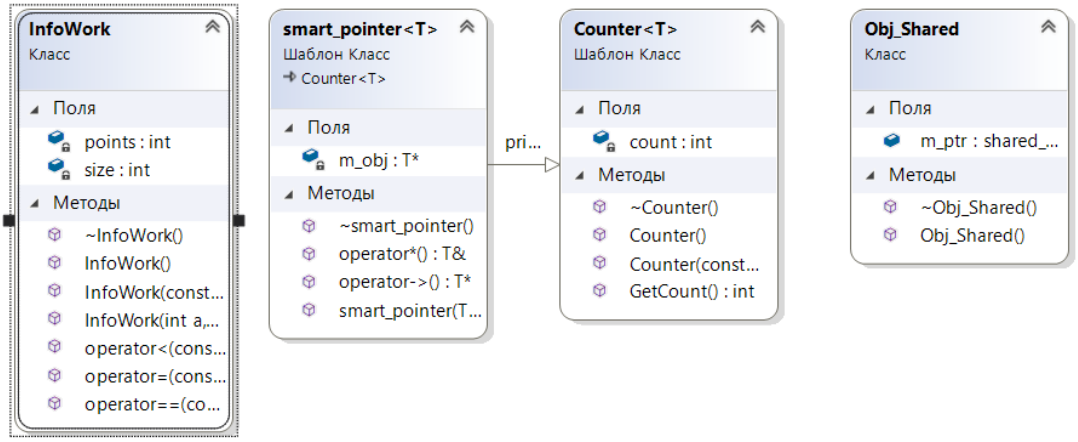

Рисунок 2.2.1 — створені класи

Class InfoWork містить такі поля як:
1. int points – оцінка за кваліфікаційну роботу
2. int size – кількість сторінок в роботі

Class smart_pointer клас який створений для виконання додаткового завдання.

Class Counter створений для підрахунку створених елементів класу smart_pointer.

Class Obj_Shared створений лише для зображення головної проблеми shared_ptr

3. Варіанти використання

```
20  void func_auto_ptr() {
21      std::auto_ptr<InfoWork> p1(new InfoWork(100,5));
22      std::auto_ptr<InfoWork> p2;
23
24      ///p1 now point to NULL and p2 now point to p1
25      p2 = p1;
26
27      ///if we try to print p1 now, we will got an exception
28      if (p1.get() != NULL) {
29          cout << "P1: " << *p1;
30      }
31      cout << "P2: " << *p2;
32
33  }
```

Рисунок 3.1 — функція func_auto_ptr(), в якій зображено роботу з auto_ptr

```
35  void func_unique_ptr() {
36          unique_ptr<InfoWork> work1(new InfoWork(95,5));
37          unique_ptr<InfoWork> work2;
38
39          cout << "work1 is " << (static_cast<bool>(work1) ? "not null\n" : "null\n");
40          cout << "work2 is " << (static_cast<bool>(work2) ? "not null\n" : "null\n");
41
42          ///unique_ptr has the same problem like auto_ptr, he can`t be appropriated to another unique_ptr
43          ///But we can use std::move to appropriat those two pointers, but work1 will be empty
44          work2 = std::move(work1);
45
46          cout << "Ownership transferred to work2\n";
47
48          cout << "work1 is " << (static_cast<bool>(work1) ? "not null\n" : "null\n");
49          cout << "work2 is " << (static_cast<bool>(work2) ? "not null\n" : "null\n");
50
51          ////////////////////////////////////////////////////////////////////////////////
52
53          ///Also unique_ptr can be used to create arrays
54
55          auto array_work = make_unique<InfoWork[]>(3);
56
57          for (int i = 0; i < 3; i++) {
58              cout << array_work[i];
59          }
60
61          ///At the end whole array will be deleted
62  }
```

Рисунок 3.2 — функція func_unique_ptr(), в якій зображено роботу з unique_ptr

```
64  void func_shared_ptr() {
65      {   shared_ptr<InfoWork> obj_test;
66          {
67              auto pointer1 = make_shared<InfoWork>(60, 4);
68              {
69                  obj_test = pointer1;
70                  auto pointer2 = pointer1;
71                  cout << "Amount of points to pointer1: " << pointer1.use_count();
72                  cout << " Pointer2: " << *pointer2;
73
74              }
75              cout << endl << "Pointer1: " << *pointer1 << endl;
76              cout << "Amount of points to pointer1: " << pointer1.use_count();
77          }
78          ///If we use ordinal pointer, we won`t be able to use memory which stored in pointer1, because it would be deleted
79          ///The memory for pointer1 still in use,because there still one more pointer obj_test which point to the same slot of memory
80          ///So we can use obj_test
81          cout << " Pointer2: " << *obj_test;
82      }
83
```

Рисунок 3.3 — частина функції func_shared_ptr(),в якій зображено роботу з shared_ptr та weak_ptr

```
84      int choose;
85      {
86          weak_ptr<InfoWork> pointer3;
87          auto pointer1 = make_shared<InfoWork>(100, 5);
88          {
89              auto pointer2 = pointer1;
90              cout << "Amount of points to pointer1: " << pointer1.use_count();
91              pointer3 = pointer1;
92              ///The number of points to pointer1 won`t increase,because weak_ptr is not considered the owner
93              cout << " Pointer2: " << *pointer2;
94
95          }
96          cout << endl << "Pointer1: " << *pointer1 << endl;
97          cout << "Amount of points to pointer1: " << pointer1.use_count();
98      }
99
100     ///The shared pointer has a pretty big problem, circular dependency, if this happens, shared_ptr will not be removed
101     ///And here such example
102     ///At the end we will see leak of memory
103     cout << "Do you want to perfome an exception? : ";
104     cin >> choose;
105     if (choose == 1) {
106         auto pointer1 = make_shared<Obj_Shared>();
107         pointer1->m_ptr = pointer1;
108     }
109     ///To handle this problem we need to use weak_ptr
```

Рисунок 3.4 — інша частина функції func_shared_ptr(), де зображена головна проблема shared_ptr

```
3
4       template <class T>  <T>
5       class smart_pointer : private Counter<T> {
6           T *m_obj;
7       public:
8           using Counter<T>::GetCount;
9           smart_pointer(T *obj)
10              : m_obj(obj)
11          {
12          }
13          ~smart_pointer() {
14              delete m_obj;
15          }
16          T* operator->() { return m_obj; }
17          T& operator* () { return *m_obj; }
18      };
```

Рисунок 3.5 — розроблений розумний покажчик в якому перевантажені оператори “*” та “→”

```
120     int main()
121     {
122         {
123             smart_pointer<InfoWork> pInt(new InfoWork);
124             smart_pointer<InfoWork> pInt2(new InfoWork);
125             smart_pointer<InfoWork> pInt3(new InfoWork);
126             smart_pointer<InfoWork> pInt4(new InfoWork);
127             smart_pointer<InfoWork> pInt5(new InfoWork);
128             smart_pointer<InfoWork> pInt6(new InfoWork);
129             cout<<"Alive:" << Counter<InfoWork>::GetCount();
130         }
131
132         cout<<"Alive: " << Counter<InfoWork>::GetCount();
133
134         //system("cls");
135         func_shared_ptr();
136         func_auto_ptr();
137         func_unique_ptr();
```

Рисунок 3.6 — використання лічильника для підрахування “smart_pointer”.

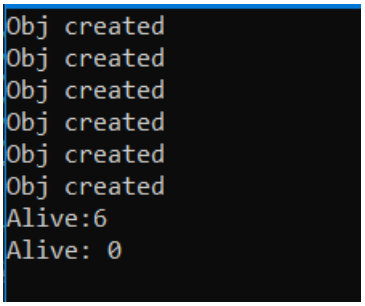

Рисунок 3.7 — результат роботи лічильника

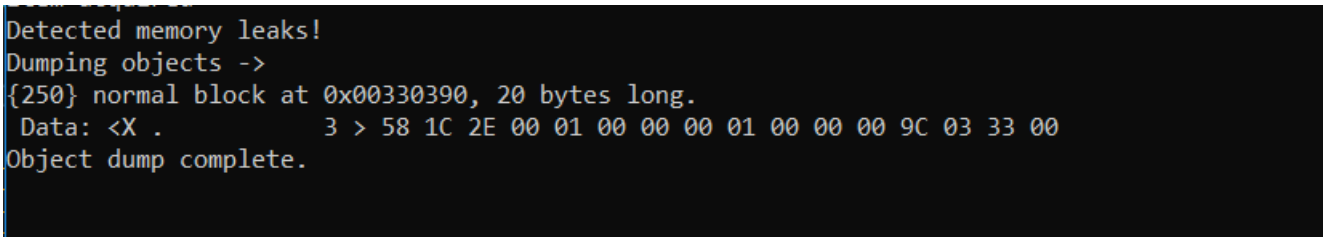

Рисунок 3.8 — результат навмисно створеного виключення з shared_ptr

Висновок: навчився користуватися розумними покажчиками, зайшов деякі проблеми покажчивків.