



# Lesson #4

## Smart Pointers

## Недостатки обычных указателей

1. Объявление не дает информации о том, ссылается ли указатель на объект или на массив
2. Объявление указателя не говорит о том, как он должен быть уничтожен (ошибка выбора формы ведет к неопределенному поведению)
3. Нет способа выяснить, не ссылается ли указатель на область памяти, которая больше не хранить объект.

```
4  int main()  
5  {  
6      setlocale(LC_ALL, "Russian");  
7  
8      // Объявление и инициализация указателей  
9  
10     int* pointer1 = new int(123);  
11     // *pointer1 = 123;  
12  
13     int* pointer2 = new int[10];  
14     for (int i = 0; i < 10; i++)  
15         pointer2[i] = i + 1;  
16  
17     std::cout << "Адрес:" << pointer1 << std::endl;  
18     std::cout << "Значение:" << * pointer1 << std::endl;  
19  
20     // Освобождение памяти  
21     delete pointer1;  
22     delete[] pointer2;  
23  
24     return 0;
```

# Проблема утечки памяти

Методика обнаружения утечек

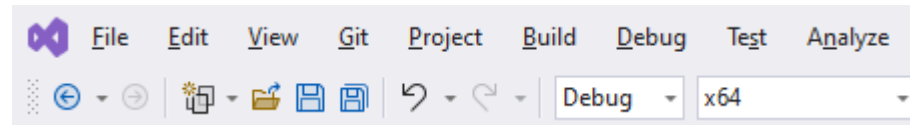
## Find memory leaks with the CRT library

<https://learn.microsoft.com/en-us/cpp/c-runtime-library/find-memory-leaks-using-the-crt-library?view=msvc-170&viewFallbackFrom=vs-2019#interpret-the-memory-leak-report>

1. Подключить библиотеку для работы с отладчиком

```
1  #include <crtdbg.h>
2  #include <iostream>
```

2. Установить режим сборки **Debug**

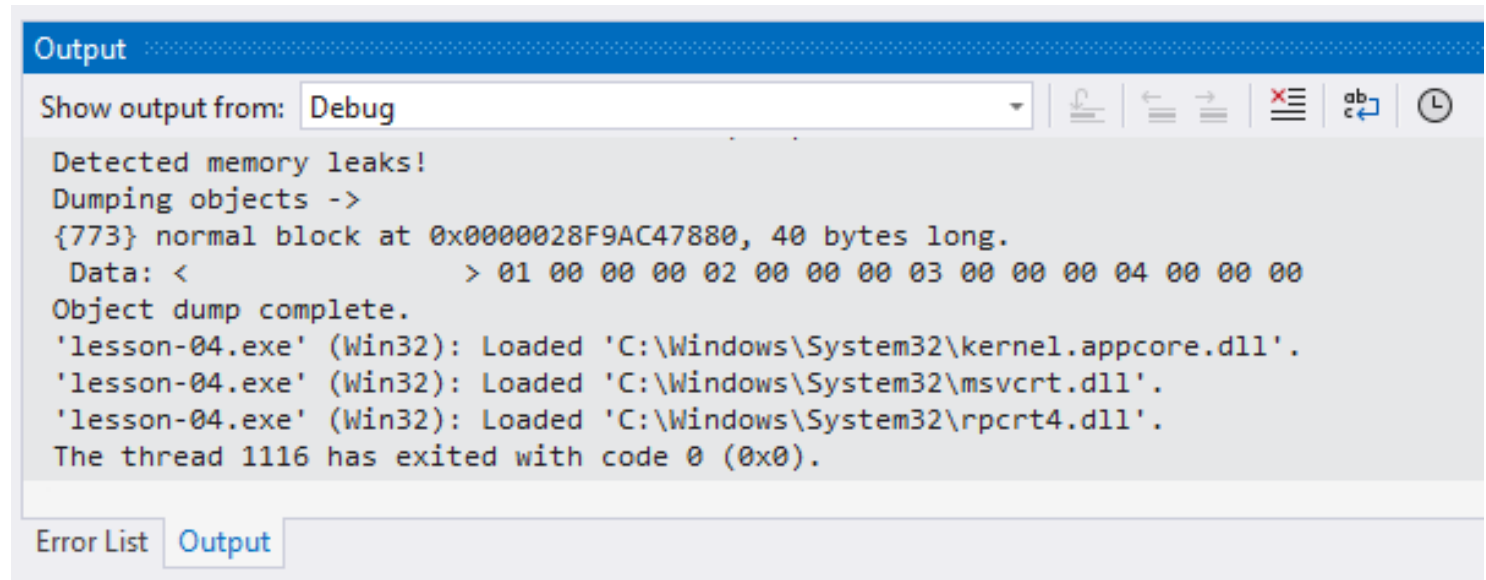


3. Перед завершением программы добавить вызов метода  
`_CrtDumpMemoryLeaks();`

**Правило** для каждого динамического выделения объекта **new** должно гарантироваться освобождение памяти **delete**

SMART POINTERS

```
28 ///////////////////////////////////////////////////.
29 // Проблема утечки памяти
30 ///////////////////////////////////////////////////.
31 int* p = nullptr;
32
33 {
34     int* p = new int[10];
35     for (int i = 0; i < 10; i++)
36         p[i] = i + 1;
37
38     // delete[] p;
39 }
40
41 _CrtDumpMemoryLeaks();
42 }
```



The screenshot shows the 'Output' window in Visual Studio. The 'Show output from:' dropdown is set to 'Debug'. The output text is as follows:

```
Detected memory leaks!
Dumping objects ->
{773} normal block at 0x0000028F9AC47880, 40 bytes long.
Data: <                                > 01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00
Object dump complete.
'lesson-04.exe' (Win32): Loaded 'C:\Windows\System32\kernel.appcore.dll'.
'lesson-04.exe' (Win32): Loaded 'C:\Windows\System32\msvcrt.dll'.
'lesson-04.exe' (Win32): Loaded 'C:\Windows\System32\rpcrt4.dll'.
The thread 1116 has exited with code 0 (0x0).
```

At the bottom of the window, there are tabs for 'Error List' and 'Output', with 'Output' currently selected.

## Суть умных указателей

Умные указатели обеспечивают автоматическое управление памятью: когда умный указатель больше не используется (выходит из области видимости) память, на которую он указывает, автоматически высвобождается

Умные указатели – «обертка» над `new` и `delete`.

При выходе из области видимости срабатывает деструктор и происходит очистка памяти. Эта техника называется **Resource Acquisition Is Initialization (RAII)**

Таким образом гарантировано можно избежать утечки ресурсов.

Умные указатели можно рассматривать как примитивную реализацию сборки мусора.

## Умные указатели

Умные указатели появились в C++ 11.

- `std::unique_ptr` – умный указатель, владеющий динамически выделенным ресурсом;
- `std::shared_ptr` – умный указатель, владеющий разделяемым динамически выделенным ресурсом. Несколько `std::shared_ptr` могут владеть одним и тем же ресурсом, и внутренний счетчик ведет их учет;
- `std::weak_ptr` – подобен `std::shared_ptr`, но не увеличивает счетчик.

~~`std::auto_ptr`~~ – устарел и не рекомендуется к использованию

\* *Определение умных указателей содержится в заголовочном файле `<memory>`*

## `std::unique_ptr`

*std::unique\_ptr* владеет объектом, на который он указывает, и никакие другие умные указатели не могут на него указывать. Когда *std::unique\_ptr* выходит из области видимости, объект удаляется. Это полезно, когда вы работаете с временным, динамически выделенным ресурсом, который может быть уничтожен после выхода из области действия [1].

Когда *std::unique\_ptr* выходит из области видимости, утечки памяти не происходит, потому что в своем деструкторе умный указатель вызывает `delete` для объекта на который ссылается, высвобождая тем самым память.

[1] <https://habr.com/ru/companies/piter/articles/706866/>

## Методы создания `std::unique_ptr`

1. С помощью оператора `new`

`std::unique_ptr<T> p(new T)`

```
112 | std::unique_ptr<int> pInt1(new int);  
113 | std::unique_ptr<int[]> pIntA1(new int[10]);  
114 | std::unique_ptr<Student> pStud1(new Student("Сергей", "Петров", "123-456-789 01", 2010, 4.5));
```

2. С помощью функции `std::make_unique`

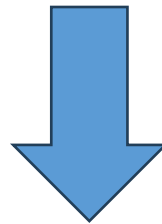
```
116 | std::unique_ptr<int> pInt2 = std::make_unique<int>( );  
117 | std::unique_ptr<int[]> pIntA2 = std::make_unique<int[]>(10);  
118 | std::unique_ptr<Student> pStud2 = std::make_unique<Student>("Сергей", "Петров", "123-456-789 01", 2010, 4.5);
```



## Особенности `std::unique_ptr`

Одна из самых популярных причин использования этого указателя – динамический полиморфизм.

```
13  void Group::setStrategy(std::unique_ptr<StrategyExportData>&& strategy)
14  {
15      if (strategy)
16          _strategy = std::move(strategy);
17  }
```



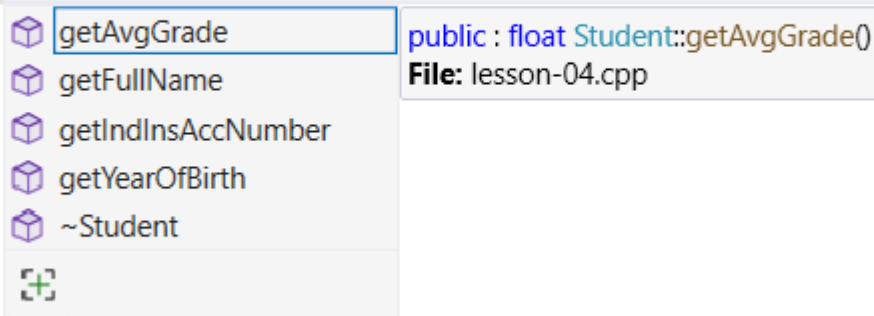
```
23      // Назначение стратегии - экспорт в CSV
24      uts21->setStrategy(std::make_unique<StrategyExportDataToCSV>());
25      std::string outCSV = uts21->exportData();
26
27      // Назначение стратегии - экспорт вTXT
28      uts21->setStrategy(std::make_unique<StrategyExportDataToTXT>());
29      std::string outTXT = uts21->exportData();
```

## Особенности `std::unique_ptr`

Класс `std::unique_ptr` перегружает оператор `->`, что позволяет обращаться к полям класса и вызывать его методы, словно мы работаем с обычным указателем

```
std::unique_ptr<Student> pStud2 = std::make_unique<Student>("Сергей", "Петров", "123-456-789 01", 2010, 4.5);
```

pStud2->



## Особенности `std::unique_ptr`

Указатель `std::unique_ptr` запрещено копировать

```
// Копирование указателя запрещено  
auto pInt3 = pInt2;  
auto pIntA3 = pIntA2;  
std::unique_ptr<Student> pStud3 = pStud2;
```

## Особенности `std::unique_ptr`

Но можно передать владение объектом другому указателю с помощью функции `std::move`

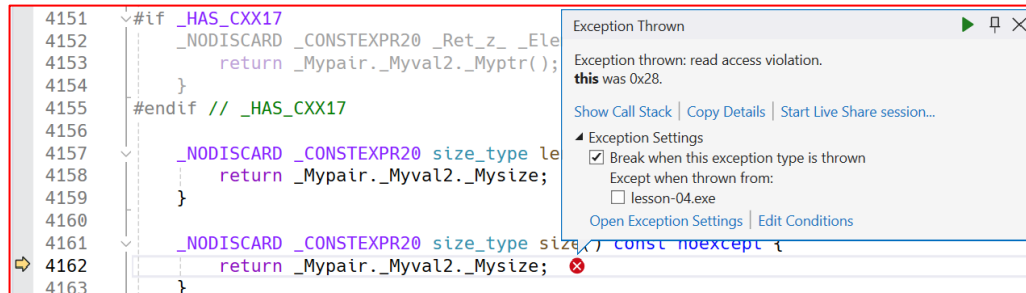
```
// Передача владения с помощью move
auto pStud3 = std::move(pStud2);
std::cout << pStud3->getFullName() << std::endl;
std::cout << pStud2->getFullName() << std::endl;
```

```
std::cout << pStud2->getFullName() << std::endl;
```

 (local variable) `std::unique_ptr<Student> pStud2`

[Search Online](#)

**C26800:** Use of a moved from object: "pStud2" (lifetime.1).



# Передача в функцию в качестве параметра

Функция для вычисления суммы элементов массива

```
static void sumA(std::unique_ptr<int[]> B)
{
    int sum = 0;
    for (int i = 0; i < 5; i++)
        sum += B[i];

    std::cout << "SUM: " << sum << std::endl;
}
```

 (local variable) std::unique\_ptr<int []> A

[Search Online](#)

function "std::unique\_ptr<\_Ty [], \_Dx>::unique\_ptr(const std::unique\_ptr<\_Ty [], \_Dx> &) [with \_Ty=int, \_Dx=std::default\_delete<int []>]" (declared at line 3422 of "C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.39.33519\include\memory") cannot be referenced -- it is a deleted function

[Search Online](#)

```
std::unique_ptr<int[]> A = std::make_unique<int[]>(5);
A[0] = 10; A[1] = 20; A[2] = 30; A[3] = 40; A[4] = 50;
for (int i = 0; i < 5; i++)
    std::cout << A[i] << std::endl;

sumA(A); // Ошибка! A передается копией,
// а копирование unique_ptr запрещено
```

# Передача в функцию в качестве параметра

Функция для вычисления суммы элементов массива

```
static void sumB(int* B)
{
    int sum = 0;
    for (int i = 0; i < 5; i++)
        sum += B[i];

    std::cout << "SUM: " << sum << std::endl;
}
```

```
sumB(A.get()); // Ошибки нет
for (int i = 0; i < 5; i++)
    std::cout << A[i] << std::endl;
```

Метод **.get()** позволяет получить сырой (обычный / raw) указатель на объект.

# Передача в функцию в качестве параметра

Функция для вычисления суммы элементов массива (с использованием move)

```
// Использование семантики move
sumA(std::move(A));
for (int i = 0; i < 5; i++)
    std::cout << A[i] << std::endl;
```

```
10
20
30
40
50
SUM: 150
10
20
30
40
50
SUM: 150
```

 (local variable) std::unique\_ptr<int []> A

[Search Online](#)

[C26800](#): Use of a moved from object: "A" (lifetime.1).

```
static void sumA(std::unique_ptr<int []> B)
{
    int sum = 0;
    for (int i = 0; i < 5; i++)
        sum += B[i];

    std::cout << "SUM: " << sum << std::endl;
}
```

## Exception Thrown

Exception thrown: read access violation.

**std::unique\_ptr<int [0],std::default\_delete<int [0]> >::operator[]**  
(...) returned nullptr.

[Show Call Stack](#) | [Copy Details](#) | [Start Live Share session...](#)

### Exception Settings

- ☒ Break when this exception type is thrown
- Except when thrown from:
  - ☐ lesson-04.exe

[Open Exception Settings](#) | [Edit Conditions](#)



## Освобождение `unique_ptr`

`std::unique_ptr` уничтожает управляемый объект когда указатель покидает область видимости. Для удаления объекта и назначения (при необходимости) нового объекта в управление `std::unique_ptr` используйте метод **`reset()`**.

```
// Принудительное освобождение указателей  
pInt1.reset();  
pIntA1.reset();  
pStud1.reset();  
  
pInt2.reset();  
pIntA2.reset();  
pStud2.reset();  
  
A.reset();
```



## unique\_ptr::reset vs unique\_ptr::release

**reset()** – уничтожает объект, которым в данный момент управляет unique\_ptr (если есть), и становится владельцем p (освобождает текущий объект).

```
1 // unique_ptr::reset example
2 #include <iostream>
3 #include <memory>
4
5 int main () {
6     std::unique_ptr<int> up; // empty
7
8     up.reset (new int);      // takes ownership of pointer
9     *up=5;
10    std::cout << *up << '\n';
11
12    up.reset (new int);      // deletes managed object, acquires new pointer
13    *up=10;
14    std::cout << *up << '\n';
15
16    up.reset();              // deletes managed object
17
18    return 0;
19 }
```

[https://cplusplus.com/reference/memory/unique\\_ptr/reset/](https://cplusplus.com/reference/memory/unique_ptr/reset/)

## unique\_ptr::reset vs unique\_ptr::release

**release()** – забирает право собственности на сохраненный указатель, возвращая его значение и заменяя его нулевым указателем (не освобождает объект).

```
1 // unique_ptr::release example
2 #include <iostream>
3 #include <memory>
4
5 int main () {
6     std::unique_ptr<int> auto_pointer (new int);
7     int * manual_pointer;
8
9     *auto_pointer=10;
10
11     manual_pointer = auto_pointer.release();
12     // (auto_pointer is now empty)
13
14     std::cout << "manual_pointer points to " << *manual_pointer << '\n';
15
16     delete manual_pointer;
17
18     return 0;
19 }
```

[https://cplusplus.com/reference/memory/unique\\_ptr/release/](https://cplusplus.com/reference/memory/unique_ptr/release/)