



Как писать на C++ (2)

Лекторы:

Аспирант МФТИ, Шер Артём Владимирович

Аспирант МФТИ, Зингеренко Михаил Владимирович

1 октября 2024

Умные указатели (smart pointers) — это объекты, которые управляют динамическими ресурсами (обычно памятью) и автоматически освобождают их, когда они больше не нужны. В C++ стандартная библиотека предоставляет три основных типа умных указателей:

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`

Этот тип умного указателя гарантирует уникальное владение объектом. То есть только один `unique_ptr` может владеть данным объектом в одно время. Особенности:

- Не поддерживает копирование.
- Поддерживает перемещение (move semantics).

unique_ptr

```
1 #include <iostream>
2 #include <memory>
3
4 class MyClass {
5 public:
6     MyClass() { std::cout << "MyClass created\n"; }
7     ~MyClass() { std::cout << "MyClass deleted\n"; }
8 };
9
10 int main() {
11     std::unique_ptr<MyClass> ptr = std::make_unique<MyClass>();
12     // MyClass created
13     // When ptr goes out of scope, the object will be deleted automatically.
14     return 0;
15 }
16 // MyClass deleted
```

Этот умный указатель позволяет разделять владение объектом между несколькими указателями. Когда последний `shared_ptr`, владеющий объектом, будет уничтожен, объект освободится.

Особенности:

- Поддерживает копирование.
- Использует счетчик ссылок (reference count), который увеличивается при копировании указателя и уменьшается при удалении.

shared_ptr

```
1 #include <iostream>
2 #include <memory>
3
4 class MyClass {
5 public:
6     MyClass() { std::cout << "MyClass created\n"; }
7     ~MyClass() { std::cout << "MyClass deleted\n"; }
8 };
9
10 int main() {
11     std::shared_ptr<MyClass> ptr1 = std::make_shared<MyClass>();
12     {
13         std::shared_ptr<MyClass> ptr2 = ptr1;
14         std::cout << "Reference count: " << ptr1.use_count() << "\n";
15     }
16     // ptr2 goes out of scope, but the object is not deleted
17     std::cout << "Reference count: " << ptr1.use_count() << "\n";
18     return 0;
19 }
20 // MyClass deleted
```

Это вспомогательный указатель, который не влияет на время жизни объекта, но может отслеживать его существование. Он используется для избежания циклических зависимостей при работе с `shared_ptr`.

```
1 #include <iostream>
2 #include <memory>
3
4 int main() {
5     std::shared_ptr<int> sharedPtr = std::make_shared<int>(42);
6     std::weak_ptr<int> weakPtr = sharedPtr;
7
8     if (auto sp = weakPtr.lock()) { // Check if the object still exists
9         std::cout << *sp << std::endl;
10    } else {
11        std::cout << "Object has been deleted\n";
12    }
13    return 0;
14 }
```


Lvalue (Left Value): Объект, который занимает определённое место в памяти и имеет адрес.

Rvalue (Right Value): Временные объекты, не имеющие адреса или которые не могут быть изменены напрямую.

```
1 int x = 10; // x - is lvalue, it has memory address
2
3 int y = x + 5; // (x + 5) - is rvalue, its just temporarily value
```

Move семантика

`std::move` преобразует `lvalue` в `rvalue`, что даёт возможность "переместить" данные вместо их копирования. Семантика перемещения позволяет "перемещать" ресурсы вместо их копирования, что экономит время и память. Это важно для временных объектов, которые можно безопасно "забрать" так как они скоро будут уничтожены

```
1 #include <iostream>
2 #include <vector>
3
4 int main() {
5     std::vector<int> vec1 = {1, 2, 3};
6     std::vector<int> vec2 = std::move(vec1); // Move data from vec1 to vec2
7
8     std::cout << "vec1 size: " << vec1.size() << "\n"; // vec1 is now empty
9     std::cout << "vec2 size: " << vec2.size() << "\n"; // vec2 contains the data
10    return 0;
11 }
```

Forwarding позволяет передавать параметры любой категории (rvalue, lvalue) в функции, сохраняя их изначальные свойства.

```
1 #include <iostream>
2 #include <utility>
3
4 void process(int& x) {
5     std::cout << "Lvalue: " << x << std::endl;
6 }
7
8 void process(int&& x) {
9     std::cout << "Rvalue: " << x << std::endl;
10 }
11
12 template <typename T>
13 void forwarder(T&& arg) {
14     process(std::forward<T>(arg));
15 }
16
17 int main() {
18     int a = 5;
19     forwarder(a);           // calls Lvalue
20     forwarder(10);          // calls Rvalue
21     return 0;
22 }
```

const

const — ключевое слово, которое делает переменную, указатель или метод неизменяемым.

```
1 const int a = 10; // 'a' cannot be modified after initialization
2 int x = 5;
3 const int* ptr = &x; // Pointer to a constant value, the value cannot be
   changed through the pointer
4 class MyClass {
5 public:
6     int getValue() const {
7         return value; // The method cannot modify class data
8     }
9 private:
10    int value;
11 };
```

Ключевые моменты:

- Используйте const для защиты данных от непреднамеренного изменения.
- Константные методы могут быть вызваны даже для объектов, объявленных как не const.

constexpr — ключевое слово, которое позволяет вычислять значение на этапе компиляции, если это возможно.

```
1 constexpr int square(int x) {  
2     return x * x; // The compiler can compute this during compilation  
3 }  
4 constexpr int result = square(5); // result is computed at compile time  
5 constexpr int size = 10;  
6 int arr[size]; // Array size must be known at compile time
```

Ключевые моменты:

- Функции или переменные, объявленные как constexpr, должны быть полностью определены на этапе компиляции.
- Можно комбинировать с const для неизменяемости.

Разница между `const` и `constexpr`

Characteristic	<code>const</code>	<code>constexpr</code>
Evaluation Time	Runtime	Compile-time
Usage	Makes a value immutable	Ensures value is computed at compile-time
Applies to Functions	No	Yes

`volatile` — это ключевое слово, которое указывает, что значение переменной может измениться неожиданно (например, операционной системой или оборудованием) и не должно быть оптимизировано компилятором.

Основные моменты:

- Компилятор не будет кэшировать значение переменной `volatile`.
- Переменная всегда читается из памяти, обеспечивая использование актуального значения.
- `volatile` не обеспечивает безопасность для работы с потоками — могут потребоваться дополнительные механизмы синхронизации.

Примеры использования:

- Многопоточность (общие переменные между потоками)
- Доступ к оборудованию (память, сопоставленная с устройствами ввода-вывода, регистры)
- Обработка сигналов (прерывания и т.д.)

```
1 volatile bool stop = false;
2
3 void threadFunction() {
4     while (!stop) {
5         // Do some work
6     }
7 }
8
9 int main() {
10     std::thread worker(threadFunction);
11     // Signal the worker thread to stop
12     stop = true;
13     worker.join();
14     return 0;
15 }
```


`mutable` — это ключевое слово, которое позволяет изменять переменную-член в `const` объекте. Оно используется, когда необходимо разрешить модификацию конкретных данных в `const` объекте или методе.

Основные моменты:

- `mutable` позволяет изменять переменную-член в `const` методах или объектах.
- Используется, когда внутреннее состояние должно быть изменяемым, даже если объект логически `const`, например, для кеширования или ведения статистики

Примеры использования:

- mutable используется, когда нужно разрешить изменение переменной-члена, даже если метод/объект является const.

```
1  class MyClass {
2  public:
3      MyClass() : counter(0) {}
4
5      void increment() const {
6          counter++; // Allowed because 'counter' is mutable
7      }
8
9      int getCounter() const {
10         return counter;
11     }
12
13 private:
14     mutable int counter; // Can be modified even in const methods
15 };
16
17 int main() {
18     const MyClass obj;
19     obj.increment(); // Modifies the mutable member 'counter'
20     std::cout << obj.getCounter() << std::endl;
21     return 0;
22 }
```

`static` — ключевое слово, которое изменяет область видимости и время жизни переменных и методов. Ключевые моменты:

- Для локальных переменных: сохраняет значение между вызовами функции или для всех экземпляров класса (scope).
- Для методов: статические методы могут быть вызваны без создания объекта класса (scope и duration).
- Для глобальных переменных: ограничивает область видимости переменной текущим файлом (scope и linkage).
- Пока смерть не разлучит нас или всё о `static` в C++

static

```
1 void myFunction() {
2     static int counter = 0; // Variable retains its value between function
    calls
3     counter++;
4 }
5 class MyClass {
6 public:
7     static int count;
8 };
9
10 int MyClass::count = 0; // Static member initialization outside the class
11 class MyClass {
12 public:
13     static void showCount() {
14         std::cout << count << std::endl;
15     }
16 private:
17     static int count;
18 };
```

До следующей лекции!