Как писать на C++ (2)

Лекторы:
Аспирант МФТИ, Шер Артём Владимирович
Аспирант МФТИ, Зингеренко Михаил Владимирович

1 октября 2024

Умные указатели

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

unique_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

```cpp
#include <iostream>
#include <memory>

class MyClass {
public:
    MyClass() { std::cout << "MyClass created\n"; }
    ~MyClass() { std::cout << "MyClass deleted\n"; }
};

int main() {
    std::unique_ptr<MyClass> ptr = std::make_unique<MyClass>();
    // MyClass created
    // When ptr goes out of scope, the object will be deleted automatically.
    return 0;
}
// MyClass deleted
```

shared_ptr

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

```cpp
#include <iostream>
#include <memory>

class MyClass {
public:
    MyClass() { std::cout << "MyClass created\n"; }
    ~MyClass() { std::cout << "MyClass deleted\n"; }
};

int main() {
    std::shared_ptr<MyClass> ptr1 = std::make_shared<MyClass>();
    {
        std::shared_ptr<MyClass> ptr2 = ptr1;
        std::cout << "Reference count: " << ptr1.use_count() << "\n";
    }
    // ptr2 goes out of scope, but the object is not deleted
    std::cout << "Reference count: " << ptr1.use_count() << "\n";
    return 0;
}
// MyClass deleted
```

weak_ptr

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.

weak_ptr

```cpp
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> sharedPtr = std::make_shared<int>(42);
    std::weak_ptr<int> weakPtr = sharedPtr;

    if (auto sp = weakPtr.lock()) { // Check if the object still exists
        std::cout << *sp << std::endl;
    } else {
        std::cout << "Object has been deleted\n";
    }
    return 0;
}
```

Lvalue и Rvalue

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 memmory addres
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

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec1 = {1, 2, 3};
    std::vector<int> vec2 = std::move(vec1);  // Move data from vec1 to vec2

    std::cout << "vec1 size: " << vec1.size() << "\n"; // vec1 is now empty
    std::cout << "vec2 size: " << vec2.size() << "\n"; // vec2 contains the data
    return 0;
}
```

Forward семантика

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

```cpp
#include <iostream>
#include <utility>

void process(int& x) {
    std::cout << "Lvalue: " << x << std::endl;
}

void process(int&& x) {
    std::cout << "Rvalue: " << x << std::endl;
}

template <typename T>
void forwarder(T&& arg) {
    process(std::forward<T>(arg));
}

int main() {
    int a = 5;
    forwarder(a);         // calls Lvalue
    forwarder(10);        // calls Rvalue
    return 0;
}
```

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

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

```
constexpr int square(int x) {
    return x * x;  // The compiler can compute this during compilation
}
constexpr int result = square(5);  // result is computed at compile time
constexpr int size = 10;
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	const	constexpr
Evaluation Time	Runtime	Compile-time
Usage	Makes a value immutable	Ensures value is computed at compile-time
Applies to Functions	No	Yes

volatile

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.

volatile

Примеры использования:

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

```
volatile bool stop = false;

void threadFunction() {
    while (!stop) {
        // Do some work
    }
}

int main() {
    std::thread worker(threadFunction);
    // Signal the worker thread to stop
    stop = true;
    worker.join();
    return 0;
}
```

mutable

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

mutable

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

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!