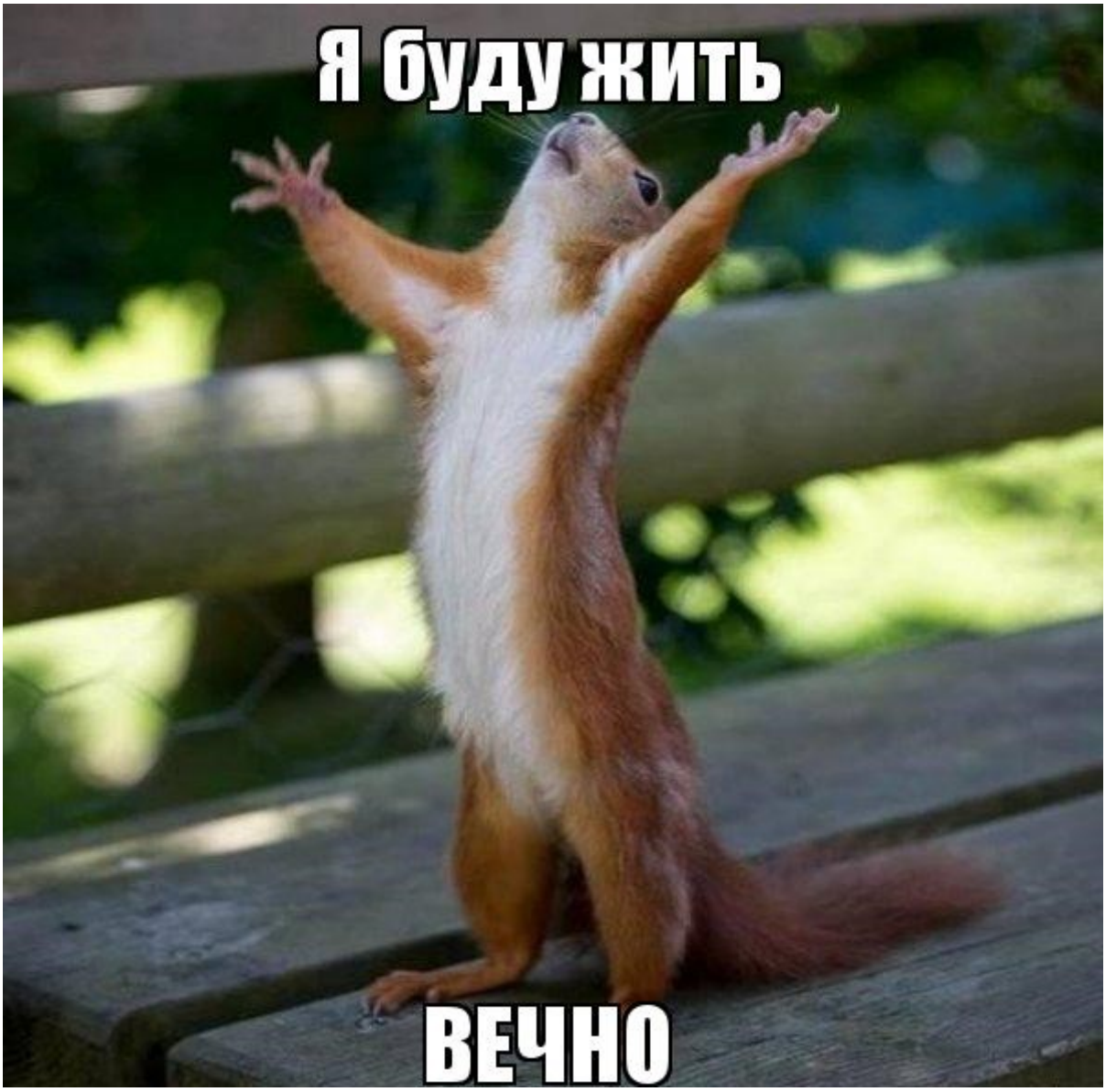
IlyaMatsuk 9 ноября 2020 в 13:30

Пока смерть не разлучит нас или всё о static в C++

Программирование *, Совершенный код *, C++ *, C *

Из песочницы

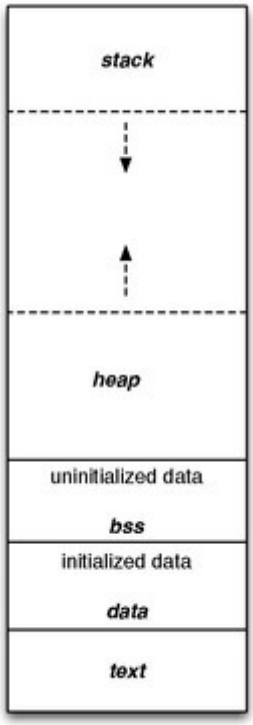


Всем привет. На одном из код-ревью я столкнулся с мыслью, что многие, а чего скрывать и я сам, не то чтобы хорошо понимаем когда нужно использовать ключевое слова **static**. В данной статье я хотел бы поделиться своими знаниями и информацией по поводу ключевого слова **static**. Статья будет полезна как начинающим программистам, так и людям, работающим с языком C++. Для понимания статьи у вас должны быть знания о процессе сборки проектов и владение языком C/C++ на базовом уровне. *Кстати, **static***

процессе сборки проектов и владение языком C/C++ на базовом уровне. *Кстати*, **static** используется не только в C++, но и в C. В этой статье я буду говорить о C++, но имейте в виду, что всё то, что не связано с объектами и классами, в основном применимо и к языку C.

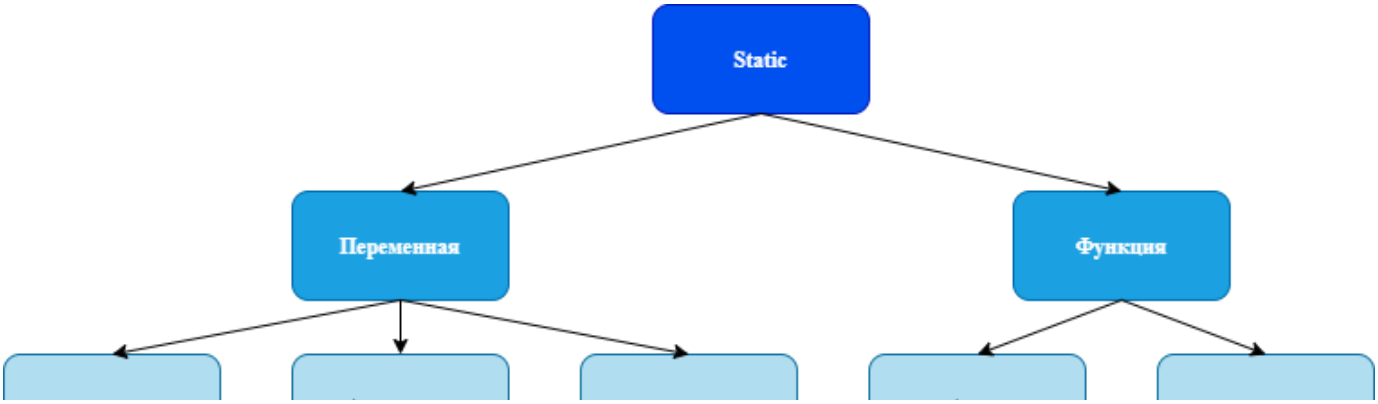
Что такое static?

Static – это ключевое слово в C++, используемое для придания элементу особых характеристик. Для статических элементов выделение памяти происходит только один раз и существуют эти элементы до завершения программы. Хранятся все эти элементы не в heap и не на stack, а в специальных сегментах памяти, которые называются *.data* и *.bss* (зависит от того инициализированы статические данные или нет). На картинке ниже показан типичный макет программной памяти.



Где используется?

Ниже приведена схема, как и где используется **static** в программе.



А теперь я постараюсь детально описать все то, что изображено на схеме. Поехали!

Статические переменные внутри функции

Статические переменные при использовании внутри функции инициализируются только один раз, а затем они сохраняют свое значение. Эти статические переменные хранятся в статической области памяти (*.data или .bss*), а не в стеке, что позволяет хранить и использовать значение переменной на протяжении всей жизни программы. Давайте рассмотрим две почти одинаковые программы и их поведение. Отличие в них только в том, что одна использует статическую переменную, а вторая нет.

Первая программа:

```
#include <iostream>

void counter() {
  static int count = 0; // строка 4
  std::cout << count++;
}

int main() {
  for (int i = 0; i < 10; ++i) {
    counter();
  }
  return 0;
}
```

Вывод программы:

```
0123456789
```

Вторая программа:

```
#include <iostream>

void counter() {
  int count = 0; // строка 4
  std::cout << count++;
```

```
}


int main() {
  for (int i = 0; i < 10; ++i) {
    counter();
  }
  return 0;
}
```

Вывод программы:

Если не использовать **static** в *строке 4*, выделение памяти и инициализация переменной count происходит при каждом вызове функции *counter()*, и уничтожается каждый раз, когда функция завершается. Но если мы сделаем переменную статической, после инициализации (при первом вызове функции *counter()*) область видимости *count* будет до конца функции *main()*, и переменная будет хранить свое значение между вызовами функции *counter()*.

Статические объекты класса

Статический объект класса имеет такие же свойства как и обычная статическая переменная, описанная выше, т.е. хранится в *.data или .bss* сегменте памяти, создается на старте и уничтожается при завершении программы, и инициализируется только один раз. Инициализация объекта происходит, как и обычно – через конструктор класса. Рассмотрим пример со статическим объектом класса.

```
#include <iostream>

class Base { // строка 3
public:
  Base() { // строка 5
    std::cout << "Constructor" << std::endl;
  }
  ~Base() { // строка 8
    std::cout << "Destructor" << std::endl;
  }
};

void foo() {
  static Base obj; // строка 14
```

```
} // строка 15


int main() {
  foo(); // строка 18
  std::cout << "End of main()" << std::endl;
  return 0;
}
```

Вывод программы:

```
Constructor
End of main()
Destructor
```

В *строке 3* мы создаем класс *Base* с конструктором (*строка 5*) и деструктором (*строка 8*). При вызове конструктора либо деструктора мы выводим название метода класса в консоль. В *строке 14* мы создаем статический объект *obj* класса *Base*. Создание этого статического объекта будет происходить только при первом вызове функции *foo() в строке 18*.

Из-за того, что объект статический, деструктор вызывается не при выходе из функции *foo() в строке 15*, а только при завершении программы, т.к. статический объект разрушается при завершении программы. Ниже приведен пример той же программы, за исключением того, что наш объект нестатический.

```
#include <iostream>

class Base {
public:
  Base() {
    std::cout << "Constructor" << std::endl;
  }
  ~Base() {
    std::cout << "Destructor" << std::endl;
  }
};

void foo() {
  Base obj;
} // строка 15
```

```
int main() {

  foo();
  std::cout << "End of main()" << std::endl;
  return 0;
}
```

Если мы уберем **static** при создании переменной в функции *foo()*, то разрушение объекта будет происходить в *строке 15* при каждом вызове функции. В таком случае вывод программы будет вполне ожидаемый для локальной переменной с выделенной памятью на стеке:

> Constructor
> Destructor
> End of main()

Статические члены класса

В сравнении с предыдущими вариантами использования, статические члены класса немного сложнее для понимания. Давайте разберемся, почему. Предположим, у нас есть следующая программа:

```
#include <iostream>

class A { // строка 3
public:
  A() { std::cout << "Constructor A" << std::endl; }
  ~A() { std::cout << "Destructor A" << std::endl; }
};

class B { // строка 9
public:
  B() { std::cout << "Constructor B" << std::endl; }
  ~B() { std::cout << "Destructor B" << std::endl; }

private:
  static A a; // строка 15 (объявление)
};

int main() {
  B b; // строка 19
  return 0;
```

```
}
```

В нашем примере мы создали класс *A (строка 3)* и класс *B (строка 9)* со статическими членами класса (*строка 15*). Мы предполагаем, что при создании объекта *b* в *строке 19* будет создан объект *a* в *строке 15*. Так бы и произошло, если бы мы использовали нестатические члены класса. Но вывод программы будет следующим:

> Constructor B
> Destructor B

Причиной такого поведения является то, что статические члены класса не инициализируются с помощью конструктора, поскольку они не зависят от инициализации объекта. Т.е. в *строке 15* мы только объявляем объект, а не определяем его, так как определение должно происходить вне класса с помощью *оператора разрешения области видимости (::)*. Давайте определим *члены класса B*.

```
#include <iostream>

class A {
public:
  A() { std::cout << "Constructor A" << std::endl; }
  ~A() { std::cout << "Destructor A" << std::endl; }
};

class B {
public:
  B() { std::cout << "Constructor B" << std::endl; }
  ~B() { std::cout << "Destructor B" << std::endl; }

private:
  static A a; // строка 15 (объявление)
};

A B::a; // строка 18 (определение)

int main() {
  B b;
  return 0;
}
```

Теперь, после того как мы определили наш статический член класса в строке 18, мы можем увидеть следующий результат программы:

> Constructor A
> Constructor B
> Destructor B
> Destructor A

Нужно помнить, что член класса будет **один для всех экземпляров класса B**, т.е. если мы создали три объекта класса **B**, то конструктор статического члена класса будет вызван только один раз. Вот пример того, о чем я говорю:

```
#include <iostream>

class A {
public:
  A() { std::cout << "Constructor A" << std::endl; }
  ~A() { std::cout << "Destructor A" << std::endl; }
};

class B {
public:
  B() { std::cout << "Constructor B" << count++ << std::endl; }
  ~B() { std::cout << "Destructor B" << --count << std::endl; }

private:
  static A a; // объявление
  static int count; // объявление
};

A B::a; // определение
int B::count = 1; // определение

int main() {
  B b1, b2, b3;
  return 0;
}
```

Вывод программы:

> Constructor A
> Constructor B1
> Constructor B2
> Constructor B3

> Destructor B3
> Destructor B2
> Destructor B1
> Destructor A

Статические функции

Статические функции пришли в C++ из C. По умолчанию все функции в C глобальные и, если вы захотите создать две функции с одинаковым именем в двух разных .c(.cpp) файлах одного проекта, то получите ошибку о том, что данная функция уже определена (*fatal error LNK1169: one or more multiply defined symbols found*). Ниже приведен листинг трех файлов одной программы.

```
// extend_math.cpp
int sum(int a, int b) {
  int some_coefficient = 1;
  return a + b + some_coefficient;
}
```

```
// math.cpp
int sum(int a, int b) {
  return a + b;
}
```

```
// main.cpp
 int sum(int, int); // declaration

int main() {
  int result = sum(1, 2);
  return 0;
}
```

Для того чтобы исправить данную проблему, одну из функций мы объявим статической. Например эту:

```
// extend_math.cpp
static int sum(int a, int b) {
  int some_coefficient = 1;
  return a + b + some_coefficient;
```

```
    return a + b + some_coefficient;
}
```

В этом случае вы говорите компилятору, что доступ к статическим функциям ограничен файлом, в котором они объявлены. И он имеет доступ только к функции *sum()* из *math.cpp* файла. Таким образом, используя **static** для функции, мы можем ограничить область видимости этой функции, и данная функция не будет видна в других файлах, если, конечно, это не заголовочный файл (.h).

Как известно, мы не можем определить функцию в заголовочном файле не сделав ее **inline** или **static**, потому что при повторном включении этого заголовочного файла мы получим такую же ошибку, как и при использовании двух функций с одинаковым именем. При определении статической функции в заголовочном файле мы даем возможность каждому файлу (.cpp), который сделает #include нашего заголовочного файла, иметь свое собственное определение этой функции. Это решает проблему, но влечет за собой увеличение размера выполняемого файла, т.к. директива *include* просто копирует содержимое заголовочного файла в .cpp файл.

Статические функции-члены класса (методы)

Статическую функцию-член вы можете использовать без создания объекта класса. Доступ к статическим функциям осуществляется с использованием имени класса и *оператора разрешения области видимости (::)*. При использовании статической функции-члена есть ограничения, такие как:

1. Внутри функции обращаться можно только к статическим членам данных, другим статическим функциям-членам и любым другим функциям извне класса.
2. Статические функции-члены имеют область видимости класса, в котором они находятся.
3. Вы не имеете доступа к указателю this класса, потому что мы не создаем никакого объекта для вызова этой функции.

Давайте рассмотрим следующий пример:

```
#include <iostream>

class A {
public:
  A() { std::cout << "Constructor A" << std::endl; }
```

```
  ~A() { std::cout << "Destructor A" << std::endl; }


  static void foo() { // строка 8
    std::cout << "static foo()" << std::endl;
  }
};

int main() {
  A::foo(); // строка 14
  return 0;
}
```

В *классе A* в *строке 8* у нас есть статическая функция-член *foo()*. *В строке 14*, мы вызываем функцию используя имя класса и оператор разрешения области видимости и получаем следующий результат программы:

```
static foo()
```

Из вывода видно, что никакого создания объекта нет и конструктор/деструктор не вызывается.

Если бы метод *foo()* был бы нестатическим, то компилятор выдал бы ошибку на выражение *в строке 14*, т.к. нужно создать объект для того, чтобы получить доступ к его нестатическим методам.

Заключение

В одной статье в интернете я нашел совет от автора – *«Используйте static везде, где только можно»*. Я хотел бы написать, почему так делать не стоит, а стоит использовать только в случае необходимости.

Итоги:

- Статические переменные медленнее, чем нестатические переменные. Для того, чтобы обратиться к статической переменной, нам нужно сделать несколько дополнительных действий, таких как переход в другой сегмент памяти и проверка инициализации переменной. Чаще всего, быстрее выделить локальную переменную на стеке, чем делать дополнительные действия по использованию статической переменной.
- Если вы используете многопоточность, то здесь вы должны быть крайне осторожными, т.к. возможна ситуация, когда два и более потока захотят писать в одну

т.к. возможна ситуация, когда два и более потока захотят писать в одну статическую переменную. Если вы будете использовать нестатические переменные в функциях, то избежите подобного, т.к. для каждого потока будет создана собственная нестатическая переменная.

- Ключевое слово static является неотъемлемой частью порождающего шаблона проектирования Singleton, который гарантирует, что будет создан только один экземпляр этого класса. В реализации этого паттерна используется и статический объект, и статическая функция-член. На практике вы можете использовать Singleton для создания объекта трейсера, логгера или любого другого объекта, который должен быть один на всё ваше приложение.
- Иногда для того, чтобы функция отработала только один раз без хранения предыдущего состояния где-то в объекте, используют статические переменные. Пример вы можете посмотреть в разделе «Статические переменные внутри функции». Но это не очень хороший подход, и может привести к долгим часам поиска ошибки, если вы используете многопоточность.
- На практике, программисты C++ часто используют статические функции-члены как альтернативу обычным функциям, которые не требуют создания объекта для выполнения ее.

Надеюсь, вам понравилась моя статья о ключевом слове **static** в языке C++. Буду рад любой критике и советам. **Всем спасибо!**

Теги: c++, c

Хабы: Программирование, Совершенный код, C++, C

6 Карма 0 Рейтинг

Ilya Matsuk @IlyaMatsuk
Software Engineer

Комментарии 22

ПОХОЖИЕ ПУБЛИКАЦИИ

29 июля 2019 в 16:03

C++20 укомплектован, C++23 начат. Итоги встречи в Кёльне

+64 33K 77 348 +348

13 июня 2019 в 18:39

Лямбды: от C++11 до C++20. Часть 2

+41 21K 130 29 +29

20 марта 2019 в 17:26

Лямбды: от C++11 до C++20. Часть 1

+24 38K 201 8 +8

МИНУТОЧКУ ВНИМАНИЯ

Разместить

Хотите рассказать о себе в наших социальных сетях?

Промокод — твой билет в общество потребления

Проект по созданию российской базы знаний

+10 4.2K 14 36 +36

Короли инференса: PyTorch, Tensorflow или MATLAB?

Турбо

ЧИТАЮТ СЕЙЧАС

Минцифры пояснило порядок предоставления сотрудникам IT-компаний льгот по ипотеке в рамках господдержки

5.5K 26 +26

Страны с простым релокейтом: подборка для тех, кто планирует переезд

11K 13 +13

Открытая дверь

15K 229 +229

ONKALO: чудо света на все времена, забудьте о нём...

11K 32 +32

Зарубежные облачные и сетевые сервисы, службы и платформы, которые могут быть заблокированы в РФ [update 26.03]

1.3K 0

Четыре слагаемых успешной системы образования

Интересно

РАБОТА

Программист C++
114 вакансий

Программист C
41 вакансия

QT разработчик

17 вакансий

Все вакансии