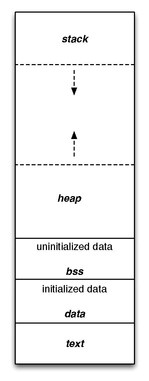
Ключевое слово static

**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:



**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.

Первая программа:

#include <iostream>

void counter() {

static int count = 0; // строка 3

std::cout << count++;

}

int main() {

for (int i = 0; i < 10; ++i) {

counter();

}

}

Вторая программа:

#include <iostream>

void counter() {

int count = 0; // строка 3

std::cout << count++;

}

int main() {

for (int i = 0; i < 10; ++i) {

counter();

}

}

По итогу обе **программы** выведут **разный** результат: 0123456789 для первой и 0000000000 для второй. Если **не использовать** static в строке 3, 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**

Второе использование static — **внутри** определения **класса**. Хотя **большинство** переменных, объявленных внутри класса, могут иметь **разное значение** в **каждом экземпляре** класса, **статические** поля класса будут иметь **то же значение для всех** экземпляров данного класса и даже **не обязательно создавать** экземпляр этого класса. Полезно представить себе, что статические переменные класса содержат информацию, необходимую для создания новых объектов. Например, если вы хотите **пронумеровать экземпляры** класса, можно использовать статическую переменную **для отслеживания последнего** используемого номера.

Пусть есть класс Base имеющий только конструктор и деструктор. В качестве примера будем работать с void foo().

#include <iostream>

class Base {

public:

Base() {

std::cout << "Constructor" << std::endl;

}

~Base() {

std::cout << "Destructor" << std::endl;

}

};

void foo(); // Объявление функции

int main() {

foo();

std::cout << "End of main()" << std::endl;

}

Реализация функции foo() в первом случае:

void foo() {

Base obj;

}

Во втором случае:

void foo() {

static Base obj;

}

В первом случае программа выведет:

Constructor

Destructor

End of main()

Во втором:

Constructor

End of main()

Destructor

Из-за того, что объект **статический**, деструктор **вызываетс**я не при выходе из функции foo(), а только **при завершении** программы, т.к. статический объект **разрушается при завершении** программы. Если мы уберем static при создании переменной в функции foo(), то разрушение объекта будет происходить в конце каждого вызова функции.

**Важно** отметить, что хорошим тоном при использовании статических переменных класса является **использование** class\_name::х, а не instance\_of\_class.x. Это **помогает напомнить** программисту, что статические переменные **не принадлежат к одному** экземпляру класса, и что вам **не обязательно создавать** экземпляр этого класса. Как вы уже, наверное, заметили, для доступа к static можно использовать оператор области видимости, ::, когда вы обращаетесь к нему через имя класса.

**Статические функции**

Статические функции пришли в **С++** из **С**. По умолчанию все функции в **С** 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). Для того чтобы исправить данную проблему, одну из функций необходимо **объявлять статической**. Вы говорите компилятору, что доступ к статическим функциям **ограничен файлом**, в котором они объявлены. 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).

**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.

Источники:

* <https://habr.com/ru/articles/527044/>
* <http://cppstudio.com/post/3298/>

Подготовил **Занин Вячеслав ВМК-21**