

Ravesli [Ravesli](#)


- [Уроки по C++](#)
- [OpenGL](#)
- [SFML](#)
- [Qt5](#)
- [RegExр](#)
- [Ассемблер](#)
- [Купить .PDF](#)


Урок №49. Глобальные переменные

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 11 Сен 2020 |

 65960

[↑](#)  10

Мы уже знаем, что переменные, объявленные внутри блока, называются [локальными](#). Они имеют локальную область видимости (используются только внутри блока, в котором объявлены) и автоматическую продолжительность жизни (создаются в точке определения и уничтожаются в конце блока).

Глобальными называются переменные, которые объявлены вне блока. Они имеют **статическую продолжительность жизни**, т.е. создаются при запуске программы и уничтожаются при её завершении. Глобальные переменные имеют **глобальную область видимости** (или «*файловую область видимости*»), т.е. их можно использовать в любом месте файла, после их объявления.

Оглавление:

1. [Определение глобальных переменных](#)
2. [Ключевые слова static и extern](#)
3. [Предварительные объявления переменных с использованием extern](#)
4. [Связи функций](#)
5. [Файловая область видимости vs. Глобальная область видимости](#)
6. [Глобальные символьные константы](#)
7. [Предостережение о \(неконстантных\) глобальных переменных](#)
8. [Заключение](#)
9. [Тест](#)

Определение глобальных переменных

Обычно глобальные переменные объявляют в верхней части кода, ниже [директив #include](#), но выше любого другого кода. Например:

```
1 | #include <iostream>
```

```
2
3 // Переменные, определенные вне блока, являются глобальными переменными
4 int g_x; // глобальная переменная g_x
5 const int g_y(3); // константная глобальная переменная g_y
6
7 void doSomething()
8 {
9     // Глобальные переменные можно использовать в любом месте программы
10    g_x = 4;
11    std::cout << g_y << "\n";
12}
13
14 int main()
15 {
16    doSomething();
17
18    // Глобальные переменные можно использовать в любом месте программы
19    g_x = 7;
20    std::cout << g_y << "\n";
21
22    return 0;
23 }
```

Подобно тому, как переменные во внутреннем блоке скрывают переменные с теми же именами во внешнем блоке, локальные переменные скрывают глобальные переменные с одинаковыми именами внутри блока, в котором они определены. Однако с помощью **оператора разрешения области видимости** (`::`), компилятору можно сообщить, какую версию переменной вы хотите использовать: глобальную или локальную. Например:

```
1 #include <iostream>
2
3 int value(4); // глобальная переменная
4
5 int main()
6 {
7     int value = 8; // эта переменная (локальная) скрывает значение глобальной переменной
8     value++; // увеличивается локальная переменная value (не глобальная)
9     ::value--; // уменьшается глобальная переменная value (не локальная)
10
11    std::cout << "Global value: " << ::value << "\n";
12    std::cout << "Local value: " << value << "\n";
13    return 0;
14 } // локальная переменная уничтожается
```

Результат выполнения программы:

```
Global value: 3
Local value: 9
```

Использовать одинаковые имена для локальных и глобальных переменных — это прямой путь к проблемам и ошибкам, поэтому подобное делать не рекомендуется. Многие разработчики добавляют к глобальным переменным префикс `g_` («g» от англ. «global»). Таким образом, можно убить сразу двух

зайцев: определить глобальные переменные и избежать конфликтов имен с локальными переменными.

Ключевые слова `static` и `extern`

В дополнение к области видимости и продолжительности жизни, переменные имеют еще одно свойство — связь. **Связь переменной** определяет, относятся ли несколько упоминаний одного идентификатора к одной и той же переменной или нет.

Переменная без связей — это переменная с локальной областью видимости, которая относится только к блоку, в котором она определена. Это обычные локальные переменные. Две переменные с одинаковыми именами, но определенные в разных функциях, не имеют никакой связи — каждая из них считается независимой единицей.

Переменная, имеющая внутренние связи, называется **внутренней переменной** (или «*статической переменной*»). Она может использоваться в любом месте файла, в котором определена, но не относится к чему-либо вне этого файла.

Переменная, имеющая внешние связи, называется **внешней переменной**. Она может использоваться как в файле, в котором определена, так и в других файлах.

Если вы хотите сделать глобальную переменную внутренней (которую можно использовать только внутри одного файла) — используйте **ключевое слово `static`**:

```
1 #include <iostream>
2
3 static int g_x; // g_x - это статическая глобальная переменная, которую можно использо
4
5 int main()
6 {
7     return 0;
8 }
```

Аналогично, если вы хотите сделать глобальную переменную внешней (которую можно использовать в любом файле программы) — используйте **ключевое слово `extern`**:

```
1 #include <iostream>
2
3 extern double g_y(9.8); // g_y - это внешняя глобальная переменная и её можно использо
4
5 int main()
6 {
7     return 0;
8 }
```

По умолчанию, неконстантные переменные, объявленные вне блока, считаются внешними. Однако константные переменные, объявленные вне блока, считаются внутренними.

Предварительные объявления переменных с использованием `extern`

Из [урока №20](#) мы уже знаем, что для использования функций, которые определены в другом файле, нужно применять предварительные объявления.

Аналогично, чтобы использовать внешнюю глобальную переменную, которая была объявлена в другом файле, нужно записать предварительное объявление переменной с использованием ключевого слова `extern` (без инициализируемого значения). Например:

global.cpp:

```
1 // Определяем две глобальные переменные
2 int g_m; // неконстантные глобальные переменные имеют внешнюю связь по умолчанию
3 int g_n(3); // неконстантные глобальные переменные имеют внешнюю связь по умолчанию
4 // g_m и g_n можно использовать в любом месте этого файла
```

main.cpp:

```
1 #include <iostream>
2
3 extern int g_m; // предварительное объявление g_m. Теперь g_m можно использовать в
4
5 int main()
6 {
7     extern int g_n; // предварительное объявление g_n. Теперь g_n можно использовать
8
9     g_m = 4;
10    std::cout << g_n; // должно вывести 3
11
12    return 0;
13 }
```

Если предварительное объявление находится вне блока, то оно применяется ко всему файлу. Если же внутри блока, то оно применяется только к нему.

Если переменная объявлена с помощью ключевого слова `static`, то получить доступ к ней с помощью предварительного объявления не получится. Например:

constants.cpp:

```
1 static const double g_gravity(9.8);
```

main.cpp:

```
1 #include <iostream>
2
3 extern const double g_gravity; // не найдет g_gravity в constants.cpp, так как g_grav
4
5 int main()
6 {
7     std::cout << g_gravity; // вызовет ошибку компиляции, так как переменная g_grav
8     return 0;
9 }
```

Обратите внимание, если вы хотите определить неинициализированную неконстантную глобальную переменную, то не используйте ключевое слово `extern`, иначе C++ будет думать, что вы пытаетесь записать предварительное объявление.

Связи функций

Функции имеют такие же свойства связи, что и переменные. По умолчанию они имеют внешнюю связь, которую можно сменить на внутреннюю с помощью ключевого слова `static`:

```
1 // Эта функция определена как static и может быть использована только внутри этого файла
2 // Попытки доступа к ней через прототип функции будут безуспешными
3 static int add(int a, int b)
4 {
5     return a + b;
6 }
```

Предварительные объявления функций не нуждаются в ключевом слове `extern`. Компилятор может определить сам (по телу функции): определяете ли вы функцию или пишете её прототип.

Файловая область видимости vs. Глобальная область видимости

Термины «файловая область видимости» и «глобальная область видимости», как правило, вызывают недоумение, и это отчасти объясняется их неофициальным использованием. В теории, в языке C++ все глобальные переменные имеют файловую область видимости. Однако, по факту, термин «файловая область видимости» чаще применяется к внутренним глобальным переменным, а «глобальная область видимости» — к внешним глобальным переменным.

Например, рассмотрим следующую программу:

global.cpp:

```
1 int g_y(3); // внешняя связь по умолчанию
```

main.cpp:

```
1 #include <iostream>
2
3 extern int g_y; // предварительное объявление g_y. Теперь g_y можно использовать в
4
5 int main()
6 {
7     std::cout << g_y; // должно вывести 3
8
9     return 0;
10 }
```

Переменная `g_y` имеет файловую область видимости внутри `global.cpp`. Доступ к этой переменной вне файла `global.cpp` отсутствует. Обратите внимание, хотя эта переменная и используется в `main.cpp`, сам `main.cpp` не видит её, он видит только предварительное объявление `g_y` (которое также имеет

файловую область видимости). Линкер отвечает за связывание определения `g_u` в `global.cpp` с использованием `g_u` в `main.cpp`.

Глобальные символьные константы

На уроке о [СИМВОЛЬНЫХ КОНСТАНТАХ](#), мы определяли их следующим образом:

`constants.h`:

```
1 #ifndef CONSTANTS_H
2 #define CONSTANTS_H
3
4 // Определяем отдельное пространство имен для хранения констант
5 namespace Constants
6 {
7     const double pi(3.14159);
8     const double avogadro(6.0221413e23);
9     const double my_gravity(9.2);
10    // ... другие константы
11 }
12 #endif
```

Хоть это просто и отлично подходит для небольших программ, но каждый раз, когда `constants.h` подключается в другой файл, каждая из этих переменных копируется в этот файл. Таким образом, если `constants.h` подключить в 20 различных файлов, то каждая из переменных продублируется 20 раз. [Header guards](#) не остановят это, так как они только предотвращают подключение заголовочного файла более одного раза в один файл. Дублирование переменных на самом деле не является проблемой (поскольку константы зачастую не занимают много памяти), но изменение значения одной константы потребует перекомпиляции каждого файла, в котором она используется, что может привести к большим временным затратам в более крупных проектах.

Избежать этой проблемы можно, превратив эти константы в константные глобальные переменные, и изменив [заголовочный файл](#) только для хранения предварительных объявлений переменных. Например:

`constants.cpp`:

```
1 namespace Constants
2 {
3     // Фактические глобальные переменные
4     extern const double pi(3.14159);
5     extern const double avogadro(6.0221413e23);
6     extern const double my_gravity(9.2);
7 }
```

`constants.h`:

```
1 #ifndef CONSTANTS_H
2 #define CONSTANTS_H
3
4 namespace Constants
```

```
5 | {  
6 |     // Только предварительные объявления  
7 |     extern const double pi;  
8 |     extern const double avogadro;  
9 |     extern const double my_gravity;  
10 | }  
11 |  
12 | #endif
```

Их использование в коде остается неизменным:

```
1 | #include "constants.h"  
2 |  
3 | //...  
4 | double circumference = 2 * radius * Constants::pi;  
5 | //...
```

Теперь определение символьных констант выполняется только один раз (в `constants.cpp`). Любые изменения, сделанные в `constants.cpp`, потребуют перекомпиляции только (одного) этого файла.

Но есть и обратная сторона медали: такие константы больше не будут считаться константами типа `compile-time` и, поэтому, не смогут использоваться где-либо, где потребуется константа такого типа.

Поскольку глобальные символьные константы должны находиться в отдельном пространстве имен и быть доступными только для чтения, то использовать префикс `g_` уже не обязательно.

Предостережение о (неконстантных) глобальных переменных

У начинающих программистов часто возникает соблазн использовать просто множество глобальных переменных, поскольку с ними легко работать, особенно когда задействовано много функций. Тем не менее, этого следует избегать! Почему? Об этом мы поговорим на следующем уроке.

Заключение

Подытожим вышесказанное:

- ➔ Глобальные переменные имеют глобальную область видимости и могут использоваться в любом месте программы. Подобно функциям, вы должны использовать предварительные объявления (с ключевым словом `extern`), чтобы использовать глобальную переменную, определенную в другом файле.
- ➔ По умолчанию, глобальные неконстантные переменные имеют внешнюю связь. Вы можете использовать ключевое слово `static`, чтобы сделать их внутренними.
- ➔ По умолчанию, глобальные константные переменные имеют внутреннюю связь. Вы можете использовать ключевое слово `extern`, чтобы сделать их внешними.
- ➔ Используйте префикс `g_` для идентификации ваших неконстантных глобальных переменных.

Тест

В чём разница между областью видимости, продолжительностью жизни и связью переменных? Какие типы продолжительности жизни, области видимости и связи имеют глобальные переменные?

Ответ

- ➔ Область видимости определяет, где переменная доступна для использования. Продолжительность жизни определяет, где переменная создается и где уничтожается. Связь определяет, может ли переменная использоваться в другом файле или нет.
- ➔ Глобальные переменные имеют глобальную область видимости (или «файловую область видимости»), что означает, что они доступны из точки объявления до конца файла, в котором объявлены.
- ➔ Глобальные переменные имеют статическую продолжительность жизни, что означает, что они создаются при запуске программы и уничтожаются при её завершении.
- ➔ Глобальные переменные могут иметь либо внутреннюю, либо внешнюю связь (это можно изменить через использование ключевых слов `static` и `extern`, соответственно).

Оценить статью:

★★★★★ (276 оценок, среднее: 4,89 из 5)



← [Урок №48. Локальные переменные, область видимости и продолжительность жизни](#)



[Урок №50. Почему глобальные переменные – зло?](#) ➔

Комментариев: 10



1. *Slava:*

[29 октября 2020 в 18:45](#)

По поводу примера "Глобальные символьные константы". Имеет ли смысл создавать `extern const double` переменные в `cpp` файле, если можно оставить просто `const double`? Ведь `extern` уже определен в заголовочном файле.

[Ответить](#)



2. *Яна:*

[3 июня 2020 в 09:30](#)

Полезная тема. Никак не могла понять как сделать так, чтобы вводимое слово (в моем случае имя) отображалось в предложениях, которые в разных блоках. Оказалось, все так просто. Спасибо за перевод самоучителя.

[Ответить](#)



3. *Владимир:*
[30 ноября 2019 в 01:09](#)

Возникает вопрос в связи вот с чем:

"...Таким образом, если constants.h подключить в 20 различных файлах, то каждая из переменных продублируется 20 раз. Header guards не остановят это, так как они только предотвращают подключение заголовочного файла более одного раза в один файл. ..."

Мы изучали, что директивы условной компиляции предотвращают или позволяют именно компиляцию по заданному условию. Из них и формируются Header guards.

Как тогда могут 20 раз дублироваться переменные, если компиляция возможна лишь раз?

Дублируются эти 20 раз уже откомпилированные за первый раз переменные в виде машинного кода, или как-то еще?

[Ответить](#)



1. *Константин:*
[30 января 2020 в 20:01](#)

Подключая заголовок.h к 20 различным файлам.cpp, в тем самым в каждый файл.cpp вставляете содержимое заголовков, всего получается что по одному разу на каждый файл.cpp, поэтому получается, что у вас создается по 20 копий константных переменных, которые были в заголовке.h. Они не подключаются 20 раз в один файл, а по одному разу в каждый из 20 файлов. Как и написано в уроке, избежать этого можно написав в заголовке не определение, а объявление констант, то есть константы будут определены один раз в одном файле.cpp, а в заголовке будут определения. Так что меняя константы, вам придётся перекомпилировать лишь один файл.cpp, а всё остальное останется без изменений. Но в первом случае, вам бы пришлось перекомпилировать все 20 файлов.

[Ответить](#)



4. *Денис:*
[15 мая 2019 в 18:59](#)

Не могли бы Вы пояснить, в каких случаях применение extern является обязательным а в каких нет.

[Ответить](#)



5. *Денис:*
[15 мая 2019 в 18:53](#)

Добрый день! В данной главе сказано, что "если вы хотите сделать глобальную переменную внешней (которую можно использовать в любом файле программы) — используйте ключевое слово extern".. при этом ниже написано "По умолчанию, неконстантные переменные, объявленные вне блока, считаются внешними." Возникает недопонимание, обязательно ли в таком случае вообще использовать extern?

Более того в "Уроке №20. Многофайловые программы" приводится пример с вызовом функции из другого файла без extern, и в данной главе сказано, что "Функции имеют такие же свойства

связи, что и переменные...", в связи с чем также складывается впечатление, что использование extern с глобальными переменными необязательно.

[Ответить](#)



1. *Виталий:*

[4 сентября 2019 в 13:40](#)

В предыдущих уроках, мы подключаем сам заголовочный файл h или src в исходный файл, здесь ты это делаешь без подключения заголовочных файлов.

[Ответить](#)



6. *Алекс:*

[9 апреля 2019 в 20:50](#)

Здарова!

Уже не первый раз обращаюсь именно к этому уроку, с целью перечитать про глобальные переменные.

Мне кажется было-бы круто, в название урока добавить например extern, через запятую после основного названия.

Потому, что ищешь поиском по странице с содержанием и не находишь. Думаю я не первый и не последний такой :))

А так все круто!

PS: О! Ща куплю твою книгу!

[Ответить](#)

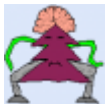


1. *Юрий:*

[9 апреля 2019 в 23:35](#)

Привет, чуть позже добавлю оглавление.

[Ответить](#)



7. *Илья:*

[23 июля 2018 в 12:30](#)

Здравствуйте. Возник вопрос, в заголовке "Файловая область видимости vs глобальная область видимости" говорится, что такая переменная не будет использоваться во всех файлах. Почему? Она получила свое определение в одном файле, не константная, следовательно, имеет видимость во всех файлах. А в файле main.cpp она предварительно объявляется с помощью extern?

Спасибо большое!

[Ответить](#)

Добавить комментарий

Ваш E-mail не будет опубликован. Обязательные поля помечены *

Имя *

Email *






Комментарий

☐ Сохранить моё Имя и E-mail. Видеть комментарии, отправленные на модерацию

☐ Получать уведомления о новых комментариях по электронной почте. Вы можете [подписаться](#) без комментирования.

[TELEGRAM](#)  [КАНАЛ](#)
[ПАБЛИК](#) 

ТОП СТАТЬИ

-  [Словарь программиста. Сленг, который должен знать каждый кодер](#)
-  [Урок №1. Введение в программирование](#)
-  [70+ бесплатных ресурсов для изучения программирования](#)
-  [Урок №1: Введение в создание игры «SameGame» на C++/MFC](#)
-  [Урок №4. Установка IDE \(Интегрированной Среды Разработки\)](#)

- [Ravesli](#)
- - [О проекте/Контакты](#) -
- - [Пользовательское Соглашение](#) -
- - [Все статьи](#) -
- Copyright © 2015 - 2020