

Ravesli [Ravesli](#)


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


Урок №73. Введение в тестирование кода

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 15 Сен 2020 |

 27726

[↑](#)  8

Итак, вы написали программу, она компилируется, и даже работает! Что дальше? Есть несколько вариантов.

Оглавление:

1. [Зачем выполнять тестирование?](#)
2. [Выполнение неофициального тестирования](#)
 - [Совет №1](#)
 - [Совет №2](#)
 - [Совет №3](#)
 - [Совет №4](#)
 - [Совет №5](#)
3. [Сохранение ваших тестов](#)
4. [Автоматизация тестирования](#)
5. [Тест](#)

Зачем выполнять тестирование?

Если вы написали программу, чтобы её один раз запустить и забыть, то дальше ничего делать не нужно. Здесь не столь важно, что ваша программа может некорректно работать в некоторых случаях: если при первом запуске она работает, так как вы и ожидали, и если вы дальше запускать и использовать её не планируете, тогда всё — финиш.

Если ваша программа полностью линейна (не имеет условного ветвления: **операторов if** или **switch**), не принимает входных данных и выводит правильный результат, тогда тоже финиш. В этом случае вы уже протестировали всю программу, запустив её один раз и сверив результаты.

Но если вы написали программу, которую собираетесь запускать много раз и которая имеет циклы и условные ветвления, принимает **пользовательский ввод**, то здесь уже немного по-другому. Возможно, вы написали функцию, которую хотите повторно использовать в других программах. Возможно, вы даже намереваетесь распространять эту программу в дальнейшем. В таком случае вам действительно нужно будет проверить, как ваша программа работает в самых разных условиях.

Только потому, что она корректно выполнялась с одними значениями пользовательского ввода, совсем не значит, что она будет работать корректно и с другими значениями.

Тестирование программного обеспечения — это процесс определения работоспособности программного обеспечения согласно ожиданиям разработчика.

Прежде чем мы будем говорить о некоторых практических способах тестирования вашего кода, давайте поговорим о том, почему комплексное тестирование может быть сложным. Например, рассмотрим следующую программу:

```
1  #include <iostream>
2  #include <string>
3
4  void compare(int a, int b)
5  {
6      if (a > b)
7          std::cout << a << " is greater than " << b << '\n'; // случай №1
8      else if (a < b)
9          std::cout << a << " is less than " << b << '\n'; // случай №2
10     else
11         std::cout << a << " is equal to " << b << '\n'; // случай №3
12 }
13
14 int main()
15 {
16     std::cout << "Enter a number: ";
17     int a;
18     std::cin >> a;
19
20     std::cout << "Enter another number: ";
21     int b;
22     std::cin >> b;
23
24     compare(a, b);
25 }
```

Учитывая 4-байтовый **тип int** и его диапазон значений, для тестирования всех возможных значений нам потребуется 18 446 744 073 709 551 616 (~ 18 квинтиллионов) раз запустить эту программу. Понятно, что это абсурд.

Каждый раз, когда мы запрашиваем пользовательский ввод или используем условное ветвление в программе — мы увеличиваем в разы количество возможных способов выполнения нашей программы. Для всех программ, кроме простейших, тестировать каждую комбинацию входных данных, да еще и вручную — как-то не логично, вам не кажется?

Сейчас ваша интуиция должна подсказывать вам, что для того, чтобы убедиться в полной работоспособности программы, приведенной выше, не нужно будет её запускать 18 квинтиллионов раз. Вы можете прийти к выводу, что если код выполняется, когда выражение $x > y$ равно `true` при одной паре значений x и y , то код должен корректно работать и с любыми другими парами x и y , где $x > y$. Учитывая это, становится очевидным, что для тестирования программы нам потребуется запустить её всего лишь три раза (по одному для каждого случая: $x > y$, $x < y$, $x = y$), чтобы убедиться, что она работает корректно. Есть и другие трюки, позволяющие упростить процесс тестирования кода.

Про методологии тестирования можно долго рассказывать, но так как эта тема не является специфической именно для языка C++, то мы будем придерживаться краткого и понятного изложения материала для начинающего разработчика, который тестирует свой собственный код.

Выполнение неофициального тестирования

Большинство разработчиков проводят **неофициальное тестирование**, когда пишут свои программы. После написания части кода (функции, класса или какого-либо другого «куска кода») разработчик пишет некий код для проверки только что добавленной части, и, если тест пройден успешно, разработчик удаляет код этого теста. Например, для следующей функции `isLowerVowel()` мы можем написать следующий код для проверки:

```
1  #include <iostream>
2
3  bool isLowerVowel(char c)
4  {
5      switch (c)
6      {
7          case 'a':
8          case 'e':
9          case 'i':
10         case 'o':
11         case 'u':
12             return true;
13         default:
14             return false;
15     }
16 }
17
18 int main()
19 {
20     std::cout << isLowerVowel('a'); // временный тестовый код, результатом которого до
21     std::cout << isLowerVowel('q'); // временный тестовый код, результатом которого до
22 }
```

Если при выполнении программы вы получите 1 и 0, тогда всё хорошо. Вы знаете, что ваша функция работает, поэтому можно удалить временный тестовый код и продолжить процесс программирования.

Совет №1: Пишите свою программу по частям: в небольших, чётко определенных единицах (функциях)

Возьмем, к примеру, автопроизводителя, который создает автомобиль. Как вы думаете, что из следующего он делает?

- ➔ Создает (или покупает) и проверяет каждый компонент автомобиля отдельно перед его установкой. Как только компонент успешно проходит проверку, автопроизводитель интегрирует его в автомобиль и повторяет проверку, чтобы убедиться, что интеграция прошла успешно. В конце, перед презентацией, проводится генеральный тест работоспособности всего автомобиля.
- ➔ Создает автомобиль из всех компонентов без какой-либо предварительной проверки (за один присест). Затем, в конце, проводится первое и окончательное тестирование работоспособности уже собранного автомобиля.

Не кажется вам, что более правильным является первый вариант? И все же большинство начинающих программистов пишут свой код в соответствии со вторым вариантом!

Во втором случае, если какая-либо из частей автомобиля будет работать неправильно, то механику придется провести диагностику всего автомобиля, чтобы определить, что пошло не так — проблема может находиться где угодно. Например, автомобиль может не заводиться из-за неисправной свечи зажигания, аккумулятора, топливного насоса или чего-то еще. Это приведет к потере большого количества потраченного впустую времени в попытках точного определения корня проблемы. И, если проблема будет найдена, последствия могут быть катастрофическими: изменения в одной части автомобиля могут привести к «эффекту бабочки» — серьезным изменениям в других частях автомобиля. Например, слишком маленький топливный насос может привести к изменению двигателя, что приведет к реорганизации каркаса автомобиля. В конечном итоге вам придется переделывать большую часть авто, просто чтобы исправить то, что изначально было небольшой проблемой!

В первом случае автопроизводитель проверяет все детали по мере поступления. Если какой-либо из компонентов оказался бракованным, то механики сразу понимают, в чем проблема и как её решить. Ничто не интегрируется в автомобиль, пока не будет успешно протестировано. К тому времени, когда они уже соберут весь автомобиль, у них будет разумная уверенность в его работоспособности — в конце концов, все его части были успешно протестированы. Все же есть вероятность, что что-то может пойти не так при соединении всех частей, но по сравнению со вторым вариантом — это очень малая вероятность, о которой и не следует серьезно беспокоиться.

Вышеупомянутая аналогия справедлива и для программистов, хотя, по некоторым причинам, новички часто этого не осознают. Гораздо лучше писать небольшие функции, а затем сразу их компилировать и тестировать. Таким образом, если вы допустили ошибку, вы будете знать, что она находится в небольшом количестве кода, который вы только что написали/изменили. А это, в свою очередь, означает, что площадь поиска ошибки невелика, и времени на отладку будет потрачено намного меньше.

Правило: Часто компилируйте свой код и всегда тестируйте все нетривиальные функции, которые вы пишете.

Совет №2: Нацеливайтесь на 100%-ное покрытие кода

Термин «**покрытие кода**» относится к количеству исходного кода программы, который был задействован во время тестирования. Есть много разных показателей покрытия кода, но лишь о нескольких из них стоит упомянуть.

Покрытие стейтментов — это процент стейтментов в вашем коде, которые были задействованы во время выполнения тестирования. Например:

```
1 int boo(int a, int b)
2 {
3     bool z = b;
4     if (a > b)
5     {
6         z = a;
7     }
8     return z;
9 }
```

Вызов `boo(1, 0)` даст вам полный охват стейтментов этой функции, так как выполнится каждая строка кода.

В случае с функцией `isLowerVowel()`:

```
1 bool isLowerVowel(char c)
2 {
3     switch (c) // стейтмент №1
4     {
5         case 'a':
6         case 'e':
7         case 'i':
8         case 'o':
9         case 'u':
10            return true; // стейтмент №2
11        default:
12            return false; // стейтмент №3
13    }
14 }
```

Здесь потребуется два вызова для проверки всех стейтментов, так как определить работу стейтментов №2 и №3 в одном вызове функции мы не сможем.

Правило: Убедитесь, что во время тестирования задействованы все стейтменты вашей функции.

Совет №3: Нацеливайтесь на 100%-ное покрытие ветвлений

Термин «**покрытие ветвлений**» относится к проценту ветвлений, которые были выполнены в каждом случае (положительном и отрицательном) отдельно. Оператор `switch` может иметь много ветвлений. Оператор `if` имеет два ветвления: случай `true` и случай `false` (даже если нет оператора `else`). Например:

```
1 int boo(int a, int b)
2 {
3     bool z = b;
4     if (a > b)
5     {
6         z = a;
7     }
8     return z;
9 }
```

Предыдущий вызов `boo(1, 0)` дал нам 100%-ный охват стейтментов и ветвление `true`. Но это всего лишь 50%-ный охват ветвлений. Нам нужен еще один вызов — `boo(0, 1)`, чтобы протестировать ветвление `false`.

В функции `isLowerVowel()` нужны два вызова (например, `isLowerVowel('a')` и `isLowerVowel('q')`), чтобы убедиться в 100%-ном охвате ветвлений (все буквы, которые находятся в `switch`, тестировать не обязательно, если сработала одна — сработают и другие):

```
1 #include <iostream>
2
3 bool isLowerVowel(char c)
4 {
5     switch (c)
6     {
7         case 'a':
8         case 'e':
9         case 'i':
10        case 'o':
11        case 'u':
12            return true;
13        default:
14            return false;
15    }
16 }
17
18 int main()
19 {
20     std::cout << isLowerVowel('a'); // временный тестовый код, результатом которого до
21     std::cout << isLowerVowel('q'); // временный тестовый код, результатом которого до
22 }
```

Пересмотрим функцию сравнения из первого примера данного урока:

```
1 void compare(int a, int b)
2 {
```

```
3 | if (a > b)
4 |     std::cout << a << " is greater than " << b << '\n'; // случай №1
5 | else if (a < b)
6 |     std::cout << a << " is less than " << b << '\n'; // случай №2
7 | else
8 |     std::cout << a << " is equal to " << b << '\n'; // случай №3
9 | }
```

Здесь необходимы 3 вызова функции, чтобы получить 100%-ный охват ветвлений:

- ➔ `compare(1, 0)` проверяет вариант `true` для первого оператора `if`.
- ➔ `compare(0, 1)` проверяет вариант `false` для первого оператора `if` и вариант `true` для второго оператора `if` (`else if`).
- ➔ `compare(0, 0)` проверяет вариант `false` для второго оператора `if` и выполняет инструкцию `else`.

Таким образом, мы можем сказать, что эту функцию можно протестировать с помощью всего лишь 3-х вызовов функции (а не 18 квинтиллионов раз).

Правило: Тестируйте каждый случай ветвления в вашей программе.

Совет №4: Нацеливайтесь на 100%-ное покрытие циклов

Покрывание циклов (неофициально называемый «*тест 0, 1, 2*») сообщает, что если у вас есть цикл в коде, то, чтобы убедиться в его работоспособности, нужно его выполнить 0, 1 и 2 раза. Если он работает корректно во второй итерации, то должен работать корректно и для всех последующих итераций (3, 4, 10, 100 и т.д.). Например:

```
1 | #include <iostream>
2 |
3 | int spam(int timesToPrint)
4 | {
5 |     for (int count=0; count < timesToPrint; ++count)
6 |         std::cout << "Spam!!!";
7 | }
```

Чтобы протестировать цикл внутри функции, нам придется вызвать его 3 раза:

- ➔ `spam(0)`, чтобы проверить случай нулевой итерации.
- ➔ `spam(1)` для проверки итерации №1 и `spam(2)` для проверки итерации №2.
- ➔ Если `spam(2)` работает, тогда и `spam(n)` будет работать (где $n > 2$).

Правило: Используйте «тест 0, 1, 2» для проверки циклов на корректную работу с разным количеством итераций.

Совет №5: Убедитесь, что вы тестируете разные типы ввода

Когда вы пишете функции, которые принимают параметры или пользовательский ввод, то посмотрите, что происходит с разными типами ввода. Например, если я написал функцию вычисления квадратного корня из целого числа, то какие значения имело бы смысл протестировать? Вероятнее всего, вы бы начали с обычных значений, например, с 4. Но также было бы неплохо протестировать и 0, и какое-нибудь отрицательное число.

Вот несколько основных рекомендаций по тестированию разных типов ввода:

- ➔ Для целых чисел убедитесь, что вы проверили, как ваша функция обрабатывает 0, отрицательные и положительные значения. При наличии пользовательского ввода вы также должны проверить вариант возникновения переполнения.
- ➔ Для чисел типа с плавающей запятой убедитесь, что вы рассмотрели варианты, как ваша функция обрабатывает значения, которые имеют неточности (значения, которые немного больше/меньше ожидаемых). Хорошие тестовые значения — это 0.1 и -0.1 (для проверки чисел, которые немного больше ожидаемых) и 0.6 и -0.6 (для проверки чисел, которые немного меньше ожидаемых).
- ➔ Для строк убедитесь, что вы рассмотрели вариант, как ваша функция обрабатывает пустую строку, строку с допустимыми значениями, строку с пробелами и строку, содержанием которой являются одни пробелы.

Правило: Тестируйте разные типы ввода, чтобы убедиться, что ваш «кусочек кода» правильно их обрабатывает.

Сохранение ваших тестов

Хотя написание тестов и последующее их удаление — достаточно хороший вариант для быстрого и временного тестирования, но для кода, который вы намереваетесь повторно использовать или модифицировать в будущем, имеет смысл сохранять эти тесты. Например, вместо удаления вашего временного теста, вы можете переместить его в функцию test():

```
1 #include <iostream>
2
3 bool isLowerVowel(char c)
4 {
5     switch (c)
6     {
7         case 'a':
8         case 'e':
9         case 'i':
10        case 'o':
11        case 'u':
12            return true;
13        default:
14            return false;
```



```
15     }
16 }
17
18 // Эта функция сейчас нигде не вызывается, но находится здесь в случае, если вы захотите
19 void test()
20 {
21     std::cout << isLowerVowel('a'); // временный тестовый код, результатом которого до
22     std::cout << isLowerVowel('q'); // временный тестовый код, результатом которого до
23 }
24
25 int main()
26 {
27     return 0;
28 }
```

Автоматизация тестирования

Одна из проблем с вышеупомянутой тестовой функцией заключается в том, что вам придется вручную проверять результаты теста. А можно сделать лучше — добавить к тесту правильные ожидаемые результаты, которые должны получиться при успешном тестировании:

```
1  #include <iostream>
2
3  bool isLowerVowel(char c)
4  {
5      switch (c)
6      {
7          case 'a':
8          case 'e':
9          case 'i':
10         case 'o':
11         case 'u':
12             return true;
13         default:
14             return false;
15     }
16 }
17
18 // Возвращается номер теста, который не был пройден или 0, если все тесты были пройдены
19 int test()
20 {
21     if (isLowerVowel('a') != true) return 1;
22     if (isLowerVowel('q') != false) return 2;
23
24     return 0;
25 }
26
27 int main()
```

```
28 | {  
29 |     return 0;  
30 | }
```

Теперь вы можете вызывать `test()` в любое время и функция сама всё сделает за вас.

Тест

Задание №1

Когда вы должны начинать тестировать свой код?

Ответ №1

Сразу, как только написали нетривиальную функцию.

Задание №2

Сколько тестов потребуется для минимального подтверждения работоспособности следующей функции?

```
1 | bool isLowerVowel(char c, bool yIsVowel)  
2 | {  
3 |     switch (c)  
4 |     {  
5 |         case 'a':  
6 |         case 'e':  
7 |         case 'i':  
8 |         case 'o':  
9 |         case 'u':  
10 |             return true;  
11 |         case 'y':  
12 |             return (yIsVowel ? true : false);  
13 |         default:  
14 |             return false;  
15 |     }  
16 | }
```

Ответ №2

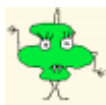
4-х тестов будет достаточно:

- ➔ Один для проверки случаев `a/e/i/o/u`.
- ➔ Один для проверки случая по умолчанию.
- ➔ Один для тестирования `isLowerVowel('y', true)`.
- ➔ И один для тестирования `isLowerVowel('y', false)`.

Оценить статью:

 (209 оценок, среднее: 4,87 из 5)[← Урок №72. Обработка некорректного пользовательского ввода](#)[Глава №5. Итоговый тест](#)

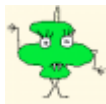
Комментариев: 8

1. *Алексей:*[23 июля 2019 в 15:01](#)

За что и люблю RegEX, хоть он и сложный бывает.

Выдал число — другое попадет, определенные символы — другое не найдет/пройдет (если пользоваться тем же "егер").

Мембрана электронная.

[Ответить](#)2. *Алексей:*[23 июля 2019 в 14:57](#)

Так и делаю обычно.

Все простое — гениальное.

Проще с малого на большее, не наоборот.

Частями обычно и программирую, может это другая сфера, то тоже шаги, не бег.

[Ответить](#)3. *Giveun:*[13 сентября 2018 в 21:12](#)

В ответе на тест 2 объясните пожалуйста что такое случай по умолчанию?

[Ответить](#)1. *Вит:*[21 сентября 2018 в 13:51](#)

default:

[Ответить](#)4. *Oleksiy:*[23 августа 2018 в 14:42](#)

"Совет при тестировании №4..."

Функция `spam(int timesToPrint)` должна быть типа `void`.

[Ответить](#)5. *korvell:*[12 июня 2018 в 23:51](#)

Вопрос: почему в 4 байтах максимальное число 18 квинтиллионов?

Если в 1 байте максимальное число FF, а в 4 — FF'FF'FF'FF, что равно 4 294 967 296

[Ответить](#)1. *Юрий:*[14 июня 2018 в 22:44](#)

Потому что каждое число из диапазона сравнивается со всеми значениями диапазона (1 с 1, 1 с 2, 1 с 3, 1 с 123 и т.д., затем 2 с 1, 2 с 2 и т.д.). И получается что диапазон нужно поделить на диапазон ($4\,294\,967\,295 * 4\,294\,967\,295$), дальше уже математика.

[Ответить](#)1. *korvell:*[16 июня 2018 в 14:29](#)

понял, спасибо

[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.

Отправить комментарий

[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