

Ravesli [Ravesli](#)


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

## Урок №108. Обработка ошибок, cerr и exit()

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 8 Сен 2020 |

 22130

[↓](#)  5

При написании программ возникновение ошибок почти неизбежно. Ошибки в языке C++ делятся на две категории: синтаксические и семантические.

Оглавление:

1. [Синтаксические ошибки](#)
2. [Семантические ошибки](#)
3. [Определение ложных предположений](#)
4. [Обработка ложных предположений](#)

### Синтаксические ошибки

**Синтаксическая ошибка** возникает при нарушении правил грамматики языка C++. Например:

если 7 не равно 8, то пишем "not equal";

Хотя этот стейтмент нам (людям) понятен, компьютер не сможет его корректно обработать. В соответствии с правилами грамматики языка C++, корректно будет:

```
1 if (7 != 8)
2     std::cout << "not equal";
```

Синтаксические ошибки почти всегда улавливаются компилятором и их обычно легко исправить. Следовательно, о них слишком беспокоиться не стоит.

# Семантические ошибки

**Семантическая** (или «*смысловая*») **ошибка** возникает, когда код синтаксически правильный, но выполняет не то, что нужно программисту. Например:

```
1 for (int count=0; count <= 4; ++count)
2     std::cout << count << " ";
```

Возможно, программист хотел, чтобы вывелось 0 1 2 3, но на самом деле выведется 0 1 2 3 4.

Семантические ошибки не улавливаются компилятором и могут иметь разное влияние: некоторые могут вообще не отображаться, что приведет к неверным результатам, к повреждению данных или вообще к сбою программы. Поэтому о семантических ошибках беспокоиться уже придется.

Они могут возникать несколькими способами. Одной из наиболее распространенных семантических ошибок является логическая ошибка. **Логическая ошибка** возникает, когда программист неправильно программирует логику выполнения кода. Например, вышеприведенный фрагмент кода имеет логическую ошибку. Вот еще один пример:

```
1 if (x >= 4)
2     std::cout << "x is greater than 4";
```

Что произойдет, если x будет равен 4? Условие выполнится как true, а программа выведет x is greater than 4. Логические ошибки иногда бывает довольно-таки трудно обнаружить.

Другой распространенной семантической ошибкой является ложное предположение. **Ложное предположение** возникает, когда программист предполагает, что что-то будет истинным или ложным, а оказывается наоборот. Например:

```
1 std::string hello = "Hello, world!";
2 std::cout << "Enter an index: ";
3
4 int index;
5 std::cin >> index;
6
7 std::cout << "Letter #" << index << " is " << hello[index] << std::endl;
```

Заметили потенциальную проблему здесь? Предполагается, что пользователь введет значение между 0 и длиной строки Hello, world!. Если же пользователь введет отрицательное число или число, которое больше длины указанной строки, то index окажется за пределами диапазона массива. В этом случае, поскольку мы просто выводим значение по индексу, результатом будет вывод мусора (при условии, что пользователь введет число вне диапазона). Но в других случаях ложное предположение может привести и к изменениям значений переменных, и к сбою в программе.

**Безопасное программирование** — это методика разработки программ, которая включает анализ областей, где могут быть допущены ложные предположения, и написание кода, который обнаруживает и обрабатывает любой случай такого нарушения, чтобы свести к минимуму риск возникновения сбоя или повреждения программы.

# Определение ложных предположений

Оказывается, мы можем найти почти все предположения, которые необходимо проверить в одном из следующих 3-х мест:

- При вызове функции, когда caller может передать некорректные или семантически бессмысленные аргументы.
- При **возврате значения** функцией, когда возвращаемое значение может быть индикатором выполнения (произошла ли ошибка или нет).
- При обработке данных ввода (либо от пользователя, либо из файла), когда эти данные могут быть не того типа, что нужно.

Поэтому, придерживаясь безопасного программирования, нужно следовать следующим 3-м правилам:

- В верхней части каждой функции убедитесь, что все параметры имеют соответствующие значения.
- После возврата функцией значения, проверьте возвращаемое значение (если оно есть) и любые другие механизмы сообщения об ошибках на предмет того, произошла ли ошибка.
- Проверяйте данные ввода на соответствие ожидаемому типу данных и его диапазону.

Рассмотрим примеры проблем:

**Проблема №1:** При вызове функции caller может передать некорректные или семантически бессмысленные аргументы:

```
1 void printString(const char *cstring)
2 {
3     std::cout << cstring;
4 }
```

Можете ли вы определить потенциальную проблему здесь? Дело в том, что caller может передать **нулевой указатель** вместо допустимой **строки C-style**. Если это произойдет, то в программе будет сбой. Вот как правильно (с проверкой параметра функции на то, не является ли он нулевым):

```
1 void printString(const char *cstring)
2 {
3     // Выводим cstring при условии, что он не нулевой
4     if (cstring)
5         std::cout << cstring;
6 }
```

**Проблема №2:** Возвращаемое значение может указывать на возникшую ошибку:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
```

```
6   std::string hello = "Hello, world!";
7   std::cout << "Enter a letter: ";
8
9   char ch;
10  std::cin >> ch;
11
12  int index = hello.find(ch);
13  std::cout << ch << " was found at index " << index << '\n';
14
15  return 0;
16 }
```

Можете ли вы определить потенциальную проблему здесь? Пользователь может ввести символ, который не находится в строке `hello`. Если это произойдет, то функция `find()` возвратит индекс `-1`, который и выведется. Правильно:

```
1  #include <iostream>
2  #include <string>
3
4  int main()
5  {
6      std::string hello = "Hello, world!";
7      std::cout << "Enter a letter: ";
8
9      char ch;
10     std::cin >> ch;
11
12     int index = hello.find(ch);
13     if (index != -1) // обрабатываем случай, когда функция find() не нашла символ в строке
14         std::cout << ch << " was found at index " << index << '\n';
15     else
16         std::cout << ch << " wasn't found" << '\n';
17
18     return 0;
19 }
```

**Проблема №3:** При обработке данных ввода (либо от пользователя, либо из файла), эти данные могут быть не того типа и диапазона, что нужно. Разберем программу из предыдущего примера: данный код позволяет проиллюстрировать ситуацию с обработкой ввода.

```
1  #include <iostream>
2  #include <string>
3
4  int main()
5  {
6      std::string hello = "Hello, world!";
7      std::cout << "Enter an index: ";
8
9      int index;
```

```
10     std::cin >> index;
11
12     std::cout << "Letter #" << index << " is " << hello [index] << std::endl;
13
14     return 0;
15 }
```

Вот как правильно ([с проверкой пользовательского ввода](#)):

```
1  #include <iostream>
2  #include <string>
3
4  int main()
5  {
6      std::string hello = "Hello, world!";
7      int index;
8
9      do
10     {
11         std::cout << "Enter an index: ";
12         std::cin >> index;
13
14         // Обрабатываем случай, когда пользователь ввел нецелочисленное значение
15         if (std::cin.fail())
16         {
17             std::cin.clear();
18             std::cin.ignore(32767, '\n');
19             index = -1; // убеждаемся, что index имеет недопустимое значение, чтобы цикл не завис
20             continue; // этот continue может показаться здесь лишним, но он явно указывает на ошибку
21         }
22
23     } while (index < 0 || index >= hello.size()); // обрабатываем случай, когда пользователь ввел значение вне диапазона
24
25     std::cout << "Letter #" << index << " is " << hello [index] << std::endl;
26
27     return 0;
28 }
```

Обратите внимание, здесь проверка двухуровневая:

- ➔ Во-первых, мы должны убедиться, что пользователь введет значение того типа данных, который мы используем.
- ➔ Во-вторых, это значение должно находиться в диапазоне массива.

## Обработка ложных предположений

Теперь, когда вы знаете, где обычно возникают ложные предположения, давайте поговорим о способах, позволяющих избежать их. Одного универсального способа исправления всех ошибок нет, всё зависит от характера проблемы.

Но все же есть несколько способов обработки ложных предположений:

**Способ №1:** Пропустите код, который зависит напрямую от правильности предположения:

```
1 void printString(const char *cstring)
2 {
3     // Выводим cstring только при условии, что он не нулевой
4     if (cstring)
5         std::cout << cstring;
6 }
```

В примере, приведенном выше, если `cstring` окажется `NULL`, то мы ничего не будем выводить. Мы пропустили тот код, который напрямую зависит от значения `cstring` и который с ним работает (в коде мы просто выводим этот `cstring`). Это может быть хорошим вариантом, если пропущенный стейтмент не является критическим и не влияет на логику программы. Основной недостаток при этом заключается в том, что `caller` или пользователь не имеет возможности определить, что что-то пошло не так.

**Способ №2:** Из функции возвращайте код ошибки обратно в `caller` и позволяйте `caller`-у обработать эту ошибку:

```
1 int getArrayValue(const std::array &array, int index)
2 {
3     // Используем условие if для обнаружения ложного предположения
4     if (index < 0 || index >= array.size())
5         return -1; // возвращаем код ошибки обратно в caller
6
7     return array[index];
8 }
```

Здесь функция возвратит `-1`, если `caller` передаст некорректный `index`. Возврат [перечислителя](#) в качестве кода ошибки будет еще лучшим вариантом.

**Способ №3:** Если нужно немедленно завершить программу, то используйте функцию `exit()`, которая находится в [заголовочном файле](#) `cstdlib`, для возврата кода ошибки обратно в операционную систему:

```
1 #include <cstdlib> // for exit()
2
3 int getArrayValue(const std::array &array, int index)
4 {
5     // Используем условие if для обнаружения ложного предположения
6     if (index < 0 || index >= array.size())
7         exit(2); // завершаем программу и возвращаем код ошибки 2 обратно в ОС
8
9     return array[index];
10 }
```

Если caller передаст некорректный `index`, то программа немедленно завершит свое выполнение и передаст код ошибки 2 обратно в операционную систему.

**Способ №4:** Если пользователь ввел данные не того типа, что нужно — попросите пользователя ввести данные еще раз:

```
1  #include <iostream>
2  #include <string>
3
4  int main()
5  {
6      std::string hello = "Hello, world!";
7      int index;
8
9      do
10     {
11         std::cout << "Enter an index: ";
12         std::cin >> index;
13
14         // Обрабатываем случай, когда пользователь ввел нецелочисленное значение
15         if (std::cin.fail())
16         {
17             std::cin.clear();
18             std::cin.ignore(32767, '\n');
19             index = -1; // убеждаемся, что index имеет недопустимое значение, чтобы цикл
20             continue; // этот continue может показаться здесь лишним, но он явно указывает
21         }
22
23     } while (index < 0 || index >= hello.size()); // обрабатываем случай, когда пользо
24
25     std::cout << "Letter #" << index << " is " << hello [index] << std::endl;
26
27     return 0;
28 }
```

**Способ №5:** Используйте `cerr`. `cerr` — это объект вывода (как и `cout`), который находится в заголовочном файле `iostream` и выводит сообщения об ошибках в консоль (как и `cout`), но только эти сообщения можно еще и перенаправить в отдельный файл с ошибками. Т.е. основное отличие `cerr` от `cout` заключается в том, что `cerr` целенаправленно используется для вывода сообщений об ошибках, тогда как `cout` — для вывода всего остального. Например:

```
1  void printString(const char *cstring)
2  {
3      // Выводим cstring при условии, что он не нулевой
4      if (cstring)
5          std::cout << cstring;
6      else
7          std::cerr << "function printString() received a null parameter";
8  }
```

В примере, приведенном выше, мы не только пропускаем код, который напрямую зависит от правильности предположения, но также регистрируем ошибку, чтобы пользователь мог позже определить, почему программа выполняется не так, как нужно.


**Способ №6:** Если вы работаете в какой-то графической среде, то распространенной практикой является вывод всплывающего окна с кодом ошибки, а затем немедленное завершение программы. То, как это сделать, зависит от конкретной среды разработки.

Оценить статью:

 (180 оценок, среднее: 4,91 из 5)



[← Урок №107. Рекурсия и Числа Фибоначчи](#)

[Урок №109. assert и static\\_assert](#) 

## Комментариев: 5



1. *Антонида:*  
[6 августа 2020 в 17:14](#)

Большое спасибо за труд! Запустила код с проверкой пользовательского ввода и ввела число 3.5, вывод был следующий: "Letter 3 is I", хотя ожидалось, что программа попросит ввести число еще раз, так как 3.5 не является целым числом. Почему так происходит? cin привел это число к int?

[Ответить](#)



1. *Кетчуп:*  
[19 августа 2020 в 15:52](#)

std::cin сначала взял 3 и присвоил её переменной index, но внутри std::cin ещё осталось '.' и 5. Если вы потом попытаетесь поместить значение из std::cin в переменную типа char, то вам даже не предложат ничего вводить и возьмут то самое значение '.'. Чтобы такого избежать, в случае успеха с index нужно проигнорировать все хранимые значения в std::cin. Можно сделать это так:

```
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
```

[Ответить](#)



2. *Алексей:*  
[2 сентября 2019 в 17:51](#)



Отличный урок, очень пригодиться.

Я только не одну неделю не могу найти способ определить пустую строку или пустой ввод. В bash это всего лишь "if( -z \$variable)". C++ вообще без понятия.

[Ответить](#)



3. Алексей:

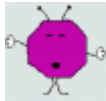
[2 сентября 2019 в 17:07](#)

```
1 #include <iostream>
2 #include <string>
3 #include <stdlib.h>
4
5 int main()
6 {
7     std::string hello = "Hello, world!";
8     int index;
9
10    do
11    {
12        std::cout << "Enter an index: ";
13        std::cin >> index;
14
15        // Обрабатываем случай, когда пользователь ввёл не целочисленное значение
16        if (std::cin.fail())
17        {
18            std::cin.clear();
19            std::cin.ignore(32767, '\n');
20            index = -1; // убеждаемся, что index имеет недопустимое значение, чтоо
21        }
22        system("cls");
23
24        } while (index < 0 || index >= hello.size()); // обрабатываем случай, когда п
25
26        std::cout << "Letter #" << index << " is " << hello[index] << std::endl;
27
28        return 0;
29 }
```

Вроде как без continue работает, нехватает очистки консоли.

Сижу и думаю, массив, почему while >=13, потом присмотрелся "="...

[Ответить](#)



4. Александр:

[1 марта 2019 в 14:29](#)

а где же throw-try-catch? или дальше будет?

вроде как логичный и удобный способ как сообщать об ошибках, так и "ловить" их...

[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