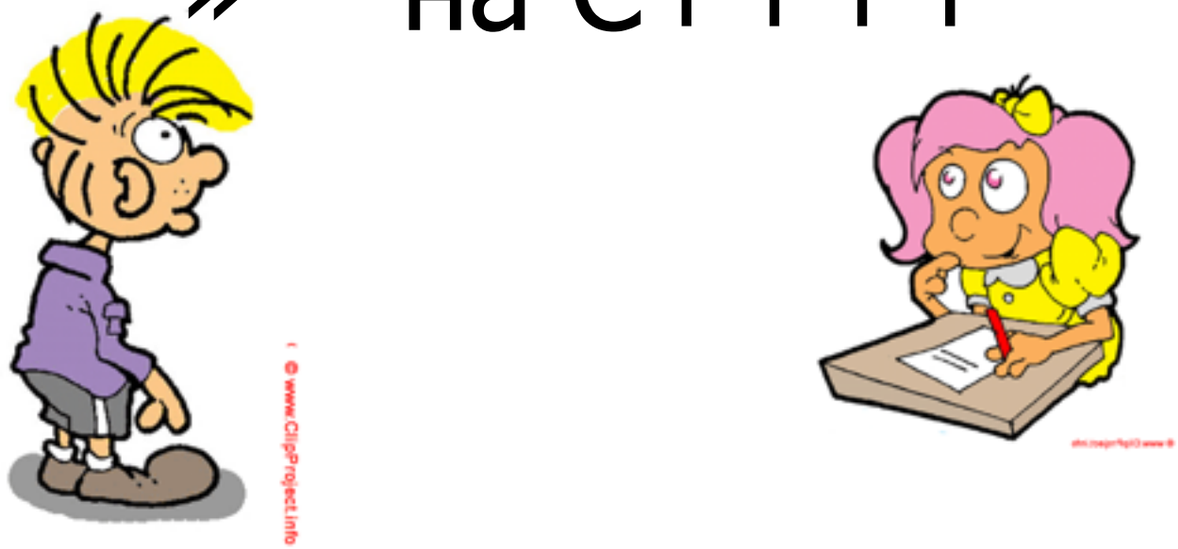
ЛЛарри и Джен делают кату «Римские цифры» » на C++++



авторы: Ольве Маудал (Olve Maudal) и Джон Джаггер (Jon Jagger)

перевод на русский: oscat.ru

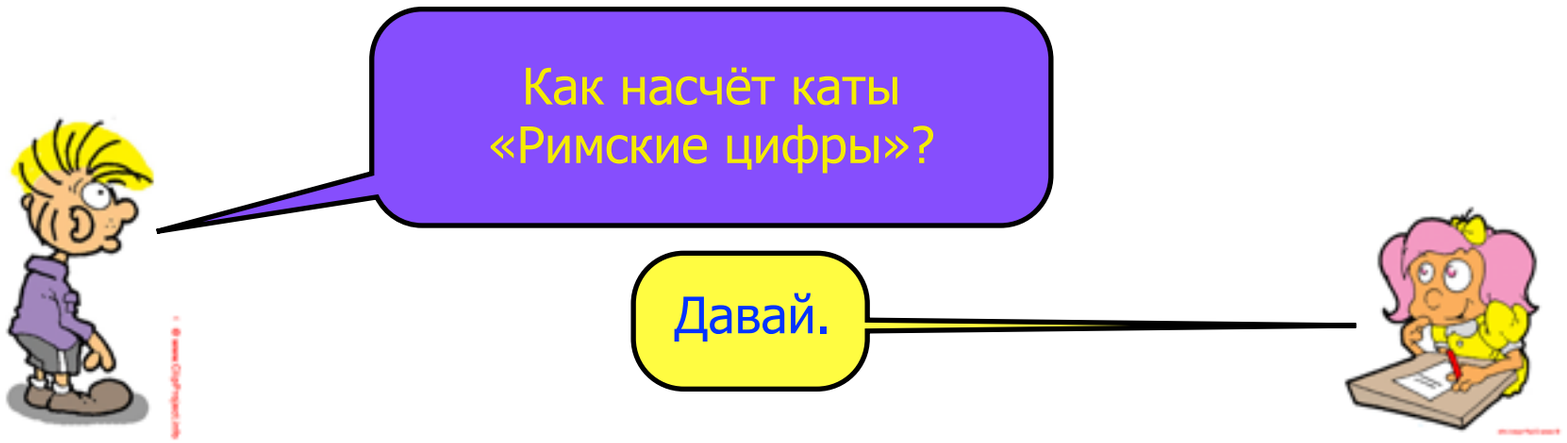
Ката — это упражнение по программировании, которое помогает программистам оттачивать свои навыки через практику и повторение. В 1999 году этот термин использовал Дэйв Томас, соавтор книги «Программист-прагматик», позаимстовав его из традиции японских боевыххисскуств (см. подробнее в Видипедии).

ЛЛарри и Джен впервые появились в презентации «Глубины C и C++»++)»

https://oscat.ru/?p=358358

Джен, поможешь мне попрактиковаться с C++?

Конечно, Ларри.



**Для введенного целого положительного числа (например, 42)
отобразите его строковое представление римскими цифрами
(для числа 42 - "XLII"). Символы римских цифр:**

```
1 ->    "I" | 10 ->    "X" | 100 ->    "C" | 1000 ->    "M"
2 ->   "II" | 20 ->   "XX" | 200 ->   "CC" | 2000 ->   "MM"
3 ->  "III" | 30 ->  "XXX" | 300 ->  "CCC" | 3000 ->  "MMM"
4 ->   "IV" | 40 ->   "XL" | 400 ->   "CD" | 4000 -> "MMMM"
5 ->    "V" | 50 ->    "L" | 500 ->    "D"
6 ->   "VI" | 60 ->   "LX" | 600 ->   "DC"
7 ->  "VII" | 70 ->  "LXX" | 700 ->  "DCC"
8 -> "VIII" | 80 -> "LXXX" | 800 -> "DCCC"
9 ->   "IX" | 90 ->   "XC" | 900 ->   "CM"
```

**Вы не можете делать так: IM для 999.
Википедия утверждает: «Современные римские цифры
записываются путем добавления каждой цифры отдельно,
начиная с крайней левой цифры и пропуском в случае нуля».**

Примеры:

```
o) 1990 -> "MCMXC"  (1000 -> "M"  + 900 -> "CM" + 90 -> "XC")
o) 2008 -> "MMVIII" (2000 -> "MM" +   8 -> "VIII")
o)   99 -> "XCIX"   (  90 -> "XC" +   9 -> "IX")
o)   47 -> "XLVII"  (  40 -> "XL" +   7 -> "VII")
```

to_roman.tests.cpp

```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>

static void test_1_is_I()
{
    assert(to_roman(1) == "I");
}

int main()
{
    std::cout << "All tests passed"
              << std::endl;
}
```

Вот мойизаголовочный файл: to_roman.hpp

А вот файл исходного кода: to_roman.cpp

Ясно.

```
#ifndef TO_ROMAN_INCLUDED
#define TO_ROMAN_INCLUDED

#include <string>

std::string to_roman(int number);

#endif
```

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    return "";
}
```

Как ты их собираешь и запускаешь?

Вот так.

Попробуй.

А?.. Что-то пошло не так. Тест должен был провалиться.!

Давай опять посмотрим на твой тест.

```
$ cc *.cpp && ./a.out
All tests passed
```


```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>

static void test_1_is_I()
{
    assert(to_roman(1) == "I");
}

int main()
{
    test_1_is_I();
    std::cout << "All tests passed"
              << std::endl;
}
```

```
$ cc *.cpp && ./a.out
Assertion failed: (to_roman(1) == "I")
```


```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    return "I";
}
```

```
$ cc *.cpp && ./a.out
All tests passed
```

Давай напишем следующиййтест.
«2» должно вернуть «II».

Я думаю, сначала мы должны решить одну проблему.

Какую?

ММы исправили проблему с пропущенным вызовом функции. Но ведь мы можем забыть об этом и в следующий раз.

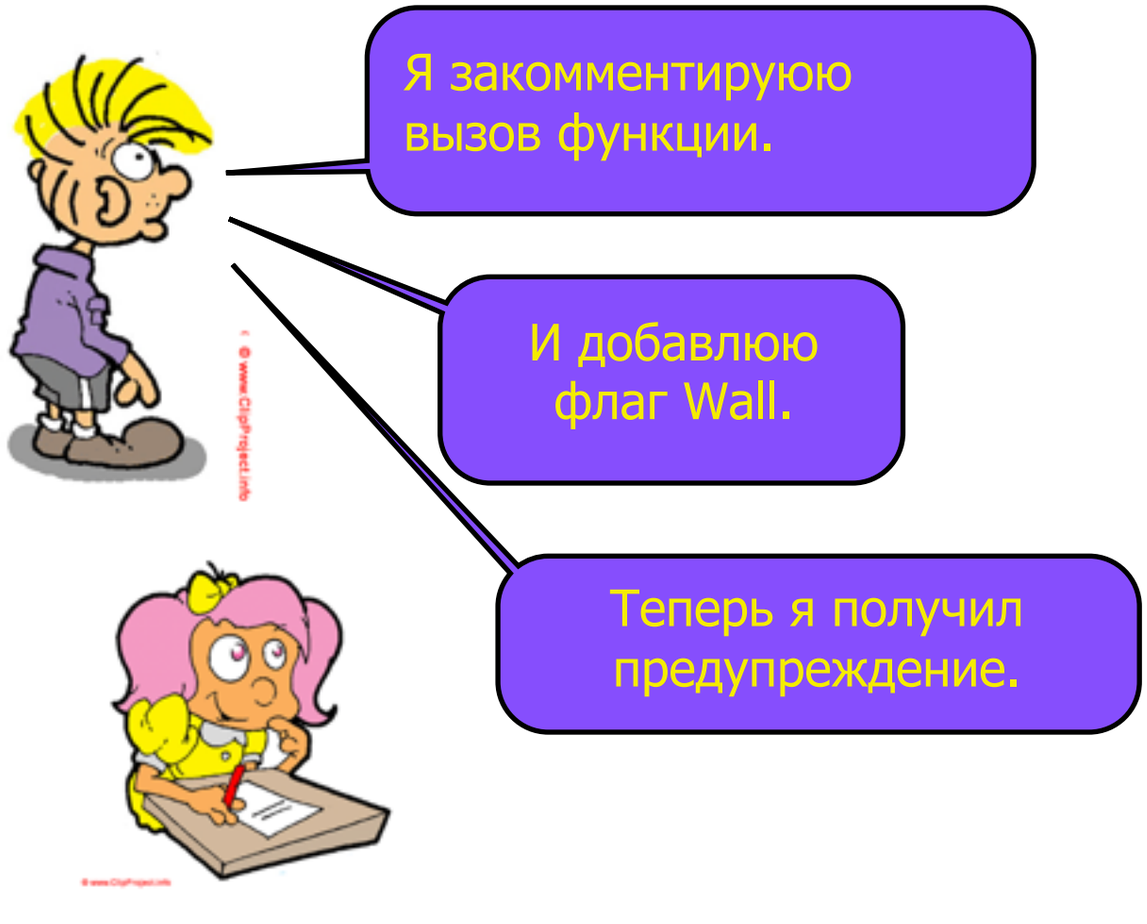
Что ты предлагаешь?

ММожет ли компилятор определить, что у нас объявлена функция, которая нигде не вызывается?

Наверное, у него есть такойй ключ командноййстроки?

Да. Компилятор может определить это при включенном флаге Wall.

Ок, давай попробуем.




```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>

static void test_1_is_I()
{
    assert(to_roman(1) == "I");
}

int main()
{
    //test_1_is_I();
    std::cout << "All tests passed"
              << std::endl;
}
```

```
$ cc -Wall *.cpp && ./a.out
warning: 'void test_1_is_I()' is
defined but not used
All tests passed
```

ЕщЕще лучше превратить это предупреждение в ошибку. Мы можем сделать это при помощи флага Werror.

Согласен.

```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>

static void test_1_is_I()
{
    assert(to_roman(1) == "I");
}

int main()
{
    //test_1_is_I();
    std::cout << "All tests passed"
              << std::endl;
}
```

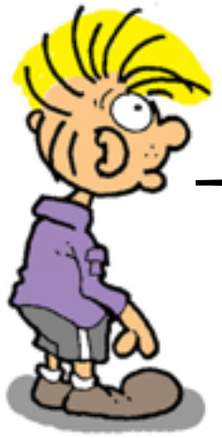
```
$ cc -Wall -Werror *.cpp && ./a.out
error: 'void test_1_is_I()' is defined
but not used
```

Я раскомментировал вызов функции, так что тест опять выполняется успешно.

ТТеперь мы можем написать следующий тест.

```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>

static void test_1_is_I()
{
    assert(to_roman(1) == "I");
}

int main()
{
    test_1_is_I();
    std::cout << "All tests passed"
              << std::endl;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```



«2«2» должно вернуть «II».

Тест должен провалиться, потому что to_roman всё еще возвращает «I».

Давайййсделаем так, чтобы он прошел.

```
...
static void test_1_is_I()
{
    assert(to_roman(1) == "I");
}

static void test_2_is_II()
{
    assert(to_roman(2) == "II");
}

int main()
{
    test_1_is_I();
    test_2_is_II();
    std::cout << "All tests passed"
              << std::endl;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
void test_2_is_II(): Assertion
`to_roman(2) == "II"' failed.
```

Как насчет такого?

ЗЗапускай!й!

А?..

А,!это потому, что если число не равно «1» или «2», то return не вызывается.

Именно.

Это легко исправить.

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    if (number == 1)
        return "I";
    if (number == 2)
        return "II";
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
error: control reaches end of non-void function
```

Что скажешь?

Выглядит неплохо.

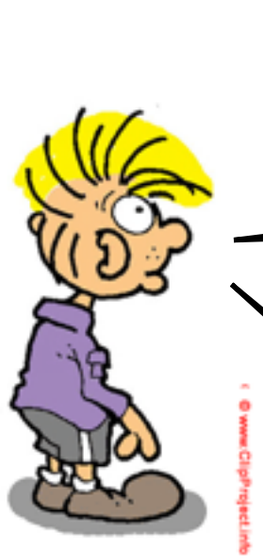
Я тоже так думаю.

Теперь следущий тест.

Ок.

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    if (number == 1)
        roman = "I";
    if (number == 2)
        roman = "II";
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```



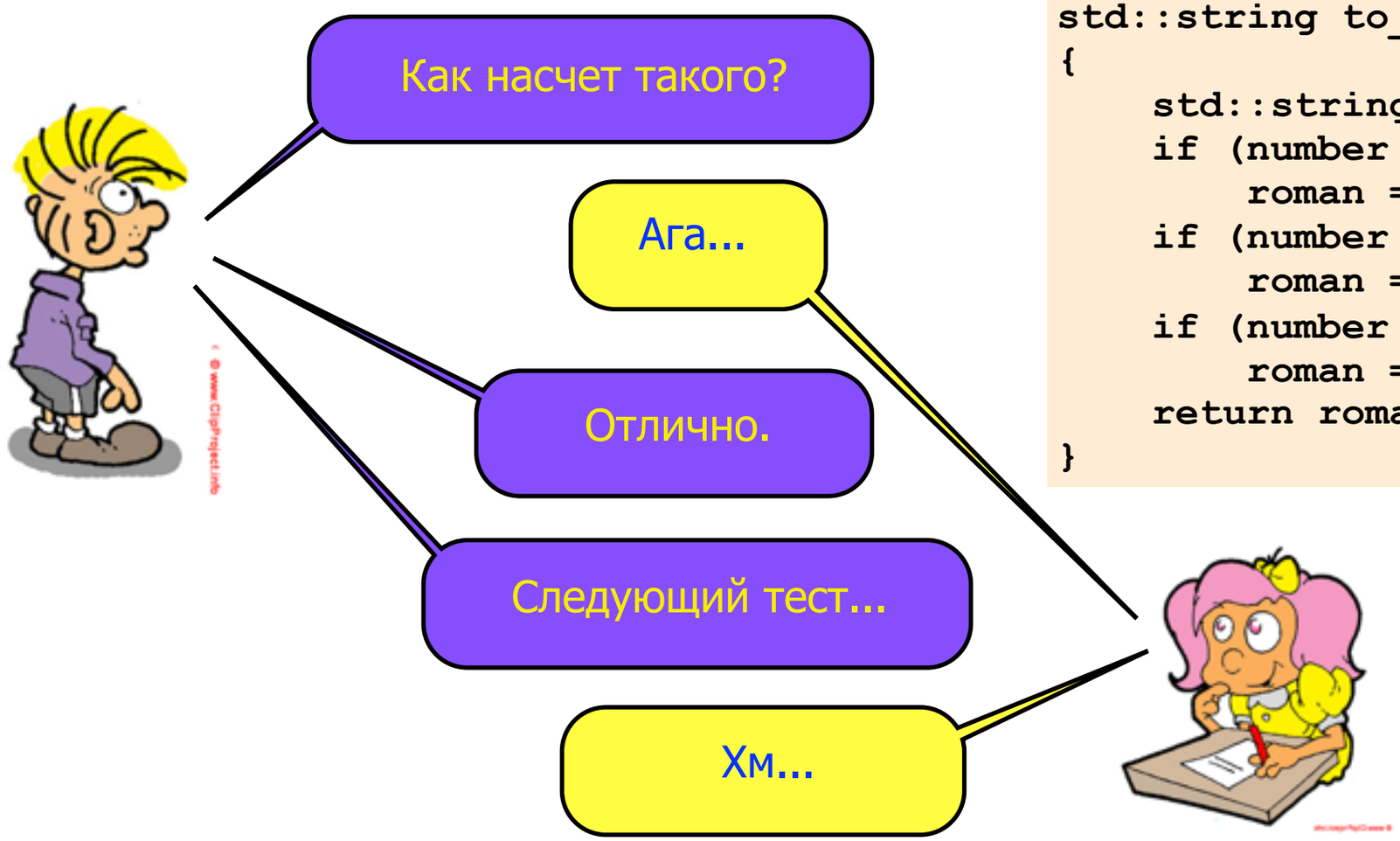
Я добавлю тест для «3».
Он должен провалиться.

А теперь я сделаю так,
чтобы он проходил.

Хорошо.

```
...
static void test_1_is_I()
{
    assert(to_roman(1) == "I");
}

static void test_2_is_II()
{
    assert(to_roman(2) == "II");
}

static void test_3_is_III()
{
    assert(to_roman(3) == "III");
}

int main()
{
    test_1_is_I();
    test_2_is_II();
    test_3_is_III();
    std::cout << "All tests passed"
              << std::endl;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
void test_3_is_III(): Assertion `to_roman(3) ==
"III"' failed.
..Aborted
```




```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    if (number == 1)
        roman = "I";
    if (number == 2)
        roman = "II";
    if (number == 3)
        roman = "III";
    return roman;
}
```

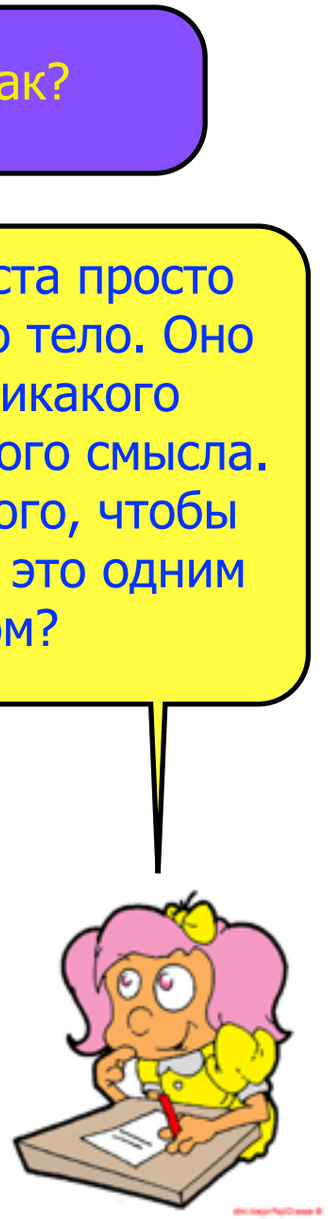
```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```


```
...
static void test_1_is_I()
{
    assert(to_roman(1) == "I");
}

static void test_2_is_II()
{
    assert(to_roman(2) == "II");
}

static void test_3_is_III()
{
    assert(to_roman(3) == "III");
}

int main()
{
    test_1_is_I();
    test_2_is_II();
    test_3_is_III();
    std::cout << "All tests passed"
              << std::endl;
}
```

А что с ним не так?

Название теста просто повторяет его тело. Оно не несет никакого дополнительного смысла. Как насчет того, чтобы заменить всё это одним тестом?

Покажешь, как это сделать?



```
...
static void test_1_is_I()
{
    assert(to_roman(1) == "I");
}

static void test_2_is_II()
{
    assert(to_roman(2) == "II");
}

static void test_3_is_III()
{
    assert(to_roman(3) == "III");
}

int main()
{
    test_1_is_I();
    test_2_is_II();
    test_3_is_III();
    std::cout << "All tests passed"
              << std::endl;
}
```

Например, вот так.

Всё работает,
как и раньше.

```
...
static void test_to_roman()
{
    assert(to_roman(1) == "I");
    assert(to_roman(2) == "II");
    assert(to_roman(3) == "III");
}

int main()
{
    test_to_roman();
    std::cout << "All tests passed"
              << std::endl;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```


```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    if (number == 1)
        roman = "I";
    if (number == 2)
        roman = "II";
    if (number == 3)
        roman = "III";
    return roman;
}
```

Для начала я изменю код так, чтобы в каждом операторе IF происходило добавление к строке одного «I».

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    if (number >= 1)
        roman += "I";
    if (number >= 2)
        roman += "I";
    if (number >= 3)
        roman += "I";
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Теперь оберну код IF-ов в фигурные скобки.

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    if (number >= 1)
    {
        roman += "I";
    }
    if (number >= 2)
    {
        roman += "I";
    }
    if (number >= 3)
    {
        roman += "I";
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Теперь я сделаю все три
IF-а одинаковыми.

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    if (number >= 1)
    {
        roman += "I";
        number--;
    }
    if (number >= 1)
    {
        roman += "I";
        number--;
    }
    if (number >= 1)
    {
        roman += "I";
        number--;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

А после этого заменю их на один цикл WHILE.

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    while (number >= 1)
    {
        roman += "I";
        number--;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Круто. И правда, набор идентичных IF-ов можно представить в виде цикла.

Именно.

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    while (number >= 1)
    {
        roman += "I";
        number--;
    }
    return roman;
}
```

Что-то ещё исправим?

Что ты предлагаешь?

Вообще, у меня больше нет идей.

Ок. Тогда продолжим.

Я добавлю тест для «4» и «IV».

Я предлагаю сразу перейти к «10» и «X».

Ладно.

Добавлю тест для «10» и «X».

Как и ожидалось -
он провалился.

```
...
static void test_to_roman()
{
    assert(to_roman(1) == "I");
    assert(to_roman(2) == "II");
    assert(to_roman(3) == "III");
    assert(to_roman(10) == "X");
}
```

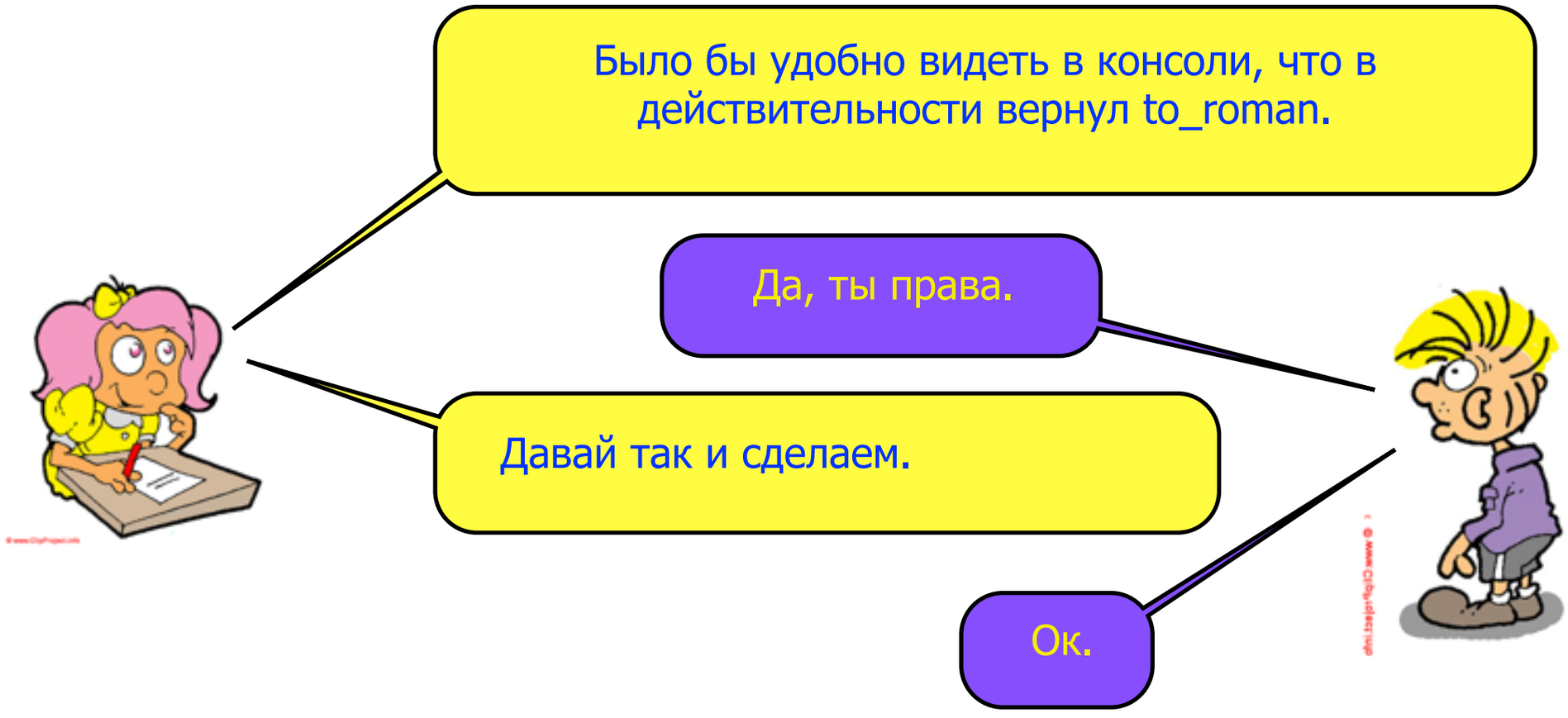
```
$ cc -Wall -Werror *.cpp && ./a.out
Assertion `to_roman(10) == "X"' failed.
Aborted
```

Чтобы он прошёл - я добавлю еще один цикл WHILE, только заменю в нём «1» и «I» на «10» и «X» соответственно.

А?.. Всё равно провалился...

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    while (number >= 1)
    {
        roman += "I";
        number--;
    }
    while (number >= 10)
    {
        roman += "X";
        number -= 10;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
Assertion `to_roman(10) == "X"' failed.
Aborted
```




```
...
static void test_to_roman()
{
    assert(to_roman(1) == "I");
    assert(to_roman(2) == "II");
    assert(to_roman(3) == "III");
    assert(to_roman(10) == "X");
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
Assertion `to_roman(10) == "X"' failed.
Aborted
```

Добавим вызов еще не существующей функции assert_to_roman.

Ага.

```
...
static void test_to_roman()
{
    assert_to_roman(1,"I");
    assert_to_roman(2,"II");
    assert_to_roman(3,"III");
    assert_to_roman(10,"X");
}
```

И напишем для этой функции
вот такой код.

Теперь вывод стал гораздо понятнее.

```
static void assert_to_roman(int arabic,
                            const std::string & expected)
{
    const std::string actual = to_roman(arabic);
    if (expected != actual)
    {
        std::cerr << "expected: to_roman"
                  << '('  << arabic << ") == "
                  << '"' << expected << '"'
                  << std::endl;
        std::cerr << "  actual: to_roman"
                  << '('  << arabic << ") == "
                  << '"' << actual << '"'
                  << std::endl;
        assert(false);
    }
}
...
```

```
$ cc -Wall -Werror *.cpp && ./a.out
expected: to_roman(10) == "X"
  actual: to_roman(10) == "IIIIIIIII"
Assertion `false' failed.
Aborted
```

Но почему же to_roman не работает для «X»?

Ох, ну конечно. У нас неверный порядок операторов WHILE.

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    while (number >= 1)
    {
        roman += "I";
        number--;
    }
    while (number >= 10)
    {
        roman += "X";
        number -= 10;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
expected: to_roman(10) == "X"
  actual: to_roman(10) == "IIIIIIIII"
Assertion `false' failed.
Aborted
```

Это легко исправить.

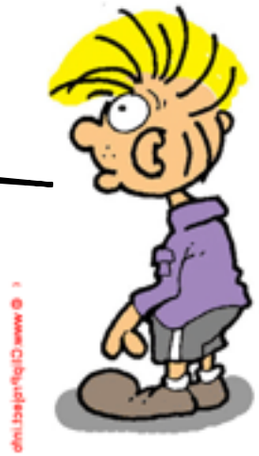
Да. И вот тест снова проходит.

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    while (number >= 10)
    {
        roman += "X";
        number -= 10;
    }
    while (number >= 1)
    {
        roman += "I";
        number--;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Думаю, тесты для «20» и «30» пройдут без проблем.

Согласна.



```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>
...
static void test_to_roman()
{
   assert_to_roman(1,"I");
   assert_to_roman(2,"II");
   assert_to_roman(3,"III");
   assert_to_roman(10,"X");
   assert_to_roman(20,"XX");
   assert_to_roman(30,"XXX");
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

И с тестом для «33» всё в порядке.

Ага.

```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>
...
static void test_to_roman()
{
   assert_to_roman( 1,"I");
   assert_to_roman( 2,"II");
   assert_to_roman( 3,"III");
   assert_to_roman(10,"X");
   assert_to_roman(20,"XX");
   assert_to_roman(30,"XXX");
   assert_to_roman(33,"XXXIII");
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

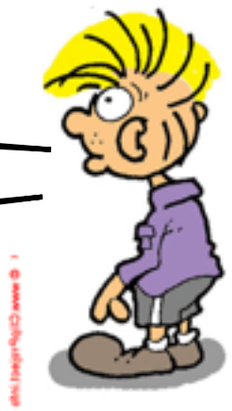
Как насчёт «100»?

Что дальше?

Для него должны получить «С».

Тест провалился - получили «10 X».

Прекрасно!



```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>
...
static void test_to_roman()
{
   assert_to_roman( 1,"I");
   assert_to_roman( 2,"II");
   assert_to_roman( 3,"III");
   assert_to_roman(10,"X");
   assert_to_roman(20,"XX");
   assert_to_roman(30,"XXX");
   assert_to_roman(33,"XXXIII");
   assert_to_roman(100,"C");
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
expected: to_roman(100) == "C"
  actual: to_roman(100) == "XXXXXXXXXX"
....int main(): Assertion `false' failed.
Aborted
```

Чтобы он прошел успешно - я просто добавлю еще один WHILE.

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    while (number >= 100)
    {
        roman += "C";
        number -= 100;
    }
    while (number >= 10)
    {
        roman += "X";
        number -= 10;
    }
    while (number >= 1)
    {
        roman += "I";
        number--;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Что думаешь?

Теперь у нас три очень похожих WHILE'а. Настало время рефакторинга!

```
#include "to_roman.hpp"

std::string to_roman(int number)
{
    std::string roman = "";
    while (number >= 100)
    {
        roman += "C";
        number -= 100;
    }
    while (number >= 10)
    {
        roman += "X";
        number -= 10;
    }
    while (number >= 1)
    {
        roman += "I";
        number--;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Для начала я создам простую структуру для представления одной римской цифры.

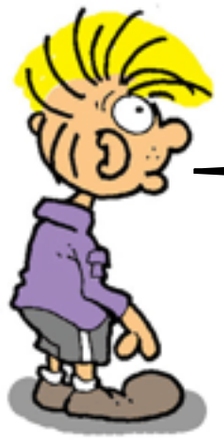
```
#include "to_roman.hpp"

struct digit
{
    int arabic;
    std::string roman;
};

std::string to_roman(int number)
{
    std::string roman = "";
    while (number >= 100)
    {
        roman += "C";
        number -= 100;
    }
    while (number >= 10)
    {
        roman += "X";
        number -= 10;
    }
    while (number >= 1)
    {
        roman += "I";
        number--;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Затем я объявлю массив таких структур и добавлю в него цифру «C» со значением «100».

```
#include "to_roman.hpp"

struct digit
{
    int arabic;
    std::string roman;
};

const digit digits[] =
{
    { 100, "C" },
};

std::string to_roman(int number)
{
    std::string roman = "";
    while (number >= 100)
    {
        roman += "C";
        number -= 100;
    }
    while (number >= 10)
    {
        roman += "X";
        number -= 10;
    }
    while (number >= 1)
    {
        roman += "I";
        number--;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```



Теперь в своем коде я заменю числа на константы из этого массива.

Отлично.

```
#include "to_roman.hpp"

struct digit
{
    int arabic;
    std::string roman;
};

const digit digits[] =
{
    { 100, "C" },
};

std::string to_roman(int number)
{
    std::string roman = "";
    while (number >= digits[0].arabic)
    {
        roman += digits[0].roman;
        number -= digits[0].arabic;
    }
    while (number >= 10)
    {
        roman += "X";
        number -= 10;
    }
    while (number >= 1)
    {
        roman += "I";
        number--;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Повторю то же самое для «X» и «10».

```
#include "to_roman.hpp"
...
const digit digits[] =
{
    { 100, "C" },
    {  10, "X" },
};

std::string to_roman(int number)
{
    std::string roman = "";
    while (number >= digits[0].arabic)
    {
        roman += digits[0].roman;
        number -= digits[0].arabic;
    }
    while (number >= digits[1].arabic)
    {
        roman += digits[1].roman;
        number -= digits[1].arabic;
    }
    while (number >= 1)
    {
        roman += "I";
        number--;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

И, наконец, для «I» и «1».

```
#include "to_roman.hpp"
...
const digit digits[] =
{
    { 100, "C" },
    {  10, "X" },
    {   1, "I" },
};

std::string to_roman(int number)
{
    std::string roman = "";
    while (number >= digits[0].arabic)
    {
        roman += digits[0].roman;
        number -= digits[0].arabic;
    }
    while (number >= digits[1].arabic)
    {
        roman += digits[1].roman;
        number -= digits[1].arabic;
    }
    while (number >= digits[2].arabic)
    {
        roman += digits[2].roman;
        number -= digits[2].arabic;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Теперь при обращении к массиву я могу заменить индекс 0 на переменную i.

```
...
std::string to_roman(int number)
{
    std::string roman = "";
    std::size_t i = 0;
    while (number >= digits[i].arabic)
    {
        roman += digits[i].roman;
        number -= digits[i].arabic;
    }
    while (number >= digits[1].arabic)
    {
        roman += digits[1].roman;
        number -= digits[1].arabic;
    }
    while (number >= digits[2].arabic)
    {
        roman += digits[2].roman;
        number -= digits[2].arabic;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

И индекс 1 тоже.

```
...
std::string to_roman(int number)
{
    std::string roman = "";
    std::size_t i = 0;
    while (number >= digits[i].arabic)
    {
        roman += digits[i].roman;
        number -= digits[i].arabic;
    }
    ++i;
    while (number >= digits[i].arabic)
    {
        roman += digits[i].roman;
        number -= digits[i].arabic;
    }
    while (number >= digits[2].arabic)
    {
        roman += digits[2].roman;
        number -= digits[2].arabic;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

И индекс 2.

```
...
std::string to_roman(int number)
{
    std::string roman = "";
    std::size_t i = 0;
    while (number >= digits[i].arabic)
    {
        roman += digits[i].roman;
        number -= digits[i].arabic;
    }
    ++i;
    while (number >= digits[i].arabic)
    {
        roman += digits[i].roman;
        number -= digits[i].arabic;
    }
    ++i;
    while (number >= digits[i].arabic)
    {
        roman += digits[i].roman;
        number -= digits[i].arabic;
    }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

А теперь превращаю все 3 WHILE'a в один.

```
...
std::string to_roman(int number)
{
    std::string roman = "";
    for (std::size_t i = 0; i != 3; ++i)
        while (number >= digits[i].arabic)
        {
            roman += digits[i].roman;
            number -= digits[i].arabic;
        }
    return roman;
}
```

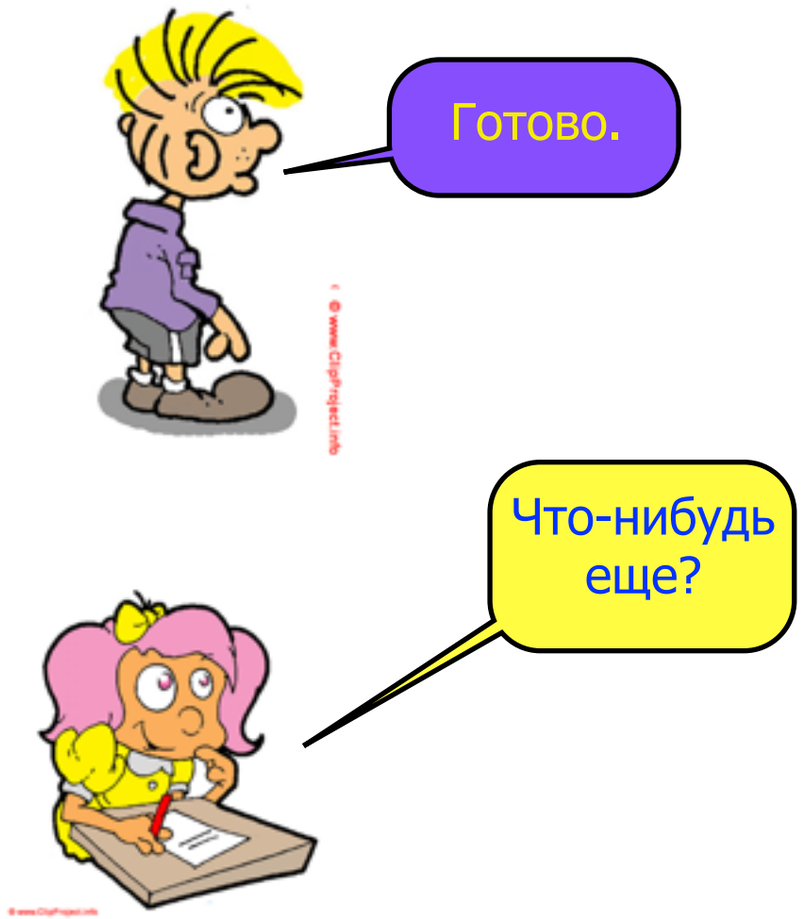
```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Вместо числа «3» лучше использовать число элементов массива.

Прекрасно.

```
#include "to_roman.hpp"
...
const digit digits[] =
{
    { 100, "C" },
    {  10, "X" },
    {   1, "I" },
};

const std::size_t n =
    sizeof digits / sizeof digits[0];

std::string to_roman(int number)
{
    std::string roman = "";
    for (std::size_t i = 0; i != n; ++i)
        while (number >= digits[i].arabic)
        {
            roman += digits[i].roman;
            number -= digits[i].arabic;
        }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```




```
#include "to_roman.hpp"

struct digit
{
    int arabic;
    std::string roman;
};

const digit digits[] =
{
    { 100, "C" },
    {  10, "X" },
    {   1, "I" },
};

const std::size_t n =
    sizeof digits / sizeof digits[0];

std::string to_roman(int number)
{
    std::string roman = "";
    for (std::size_t i = 0; i != n; ++i)
        while (number >= digits[i].arabic)
        {
            roman += digits[i].roman;
            number -= digits[i].arabic;
        }
    return roman;
}
```

Твои предложения?

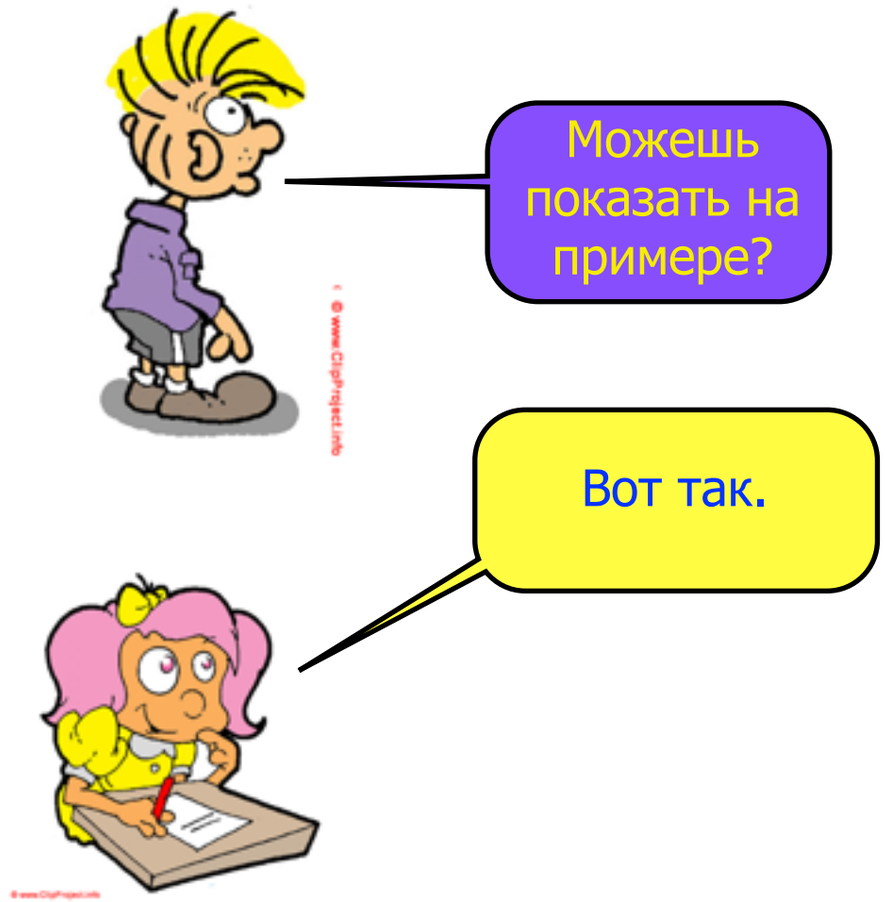
digits и n имеют внутреннее связывание, потому что это константы, но тип digit имеет внешнее связывание.

```
#include "to_roman.hpp"

struct digit
{
    int arabic;
    std::string roman;
};

const digit digits[] =
{
    { 100, "C" },
    {  10, "X" },
    {   1, "I" },
};

const std::size_t n =
    sizeof digits / sizeof digits[0];

std::string to_roman(int number)
{
    std::string roman = "";
    for (std::size_t i = 0; i != n; ++i)
        while (number >= digits[i].arabic)
        {
            roman += digits[i].roman;
            number -= digits[i].arabic;
        }
    return roman;
}
```

Как я могу задать для типа внутренее связывание?

С помощью анонимного пространства имен.

```
#include "to_roman.hpp"

struct digit
{
    int arabic;
    std::string roman;
};

const digit digits[] =
{
    { 100, "C" },
    {  10, "X" },
    {   1, "I" },
};

const std::size_t n =
    sizeof digits / sizeof digits[0];

std::string to_roman(int number)
{
    std::string roman = "";
    for (std::size_t i = 0; i != n; ++i)
        while (number >= digits[i].arabic)
        {
            roman += digits[i].roman;
            number -= digits[i].arabic;
        }
    return roman;
}
```




```
#include "to_roman.hpp"

namespace
{
    struct digit
    {
        int arabic;
        std::string roman;
    };

    const digit digits[] =
    {
        { 100, "C" },
        {  10, "X" },
        {   1, "I" },
    };

    const std::size_t n =
        sizeof digits / sizeof digits[0];

}
...
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```


```
#include "to_roman.hpp"

namespace
{
...
}

std::string to_roman(int arabic)
{
    std::string roman = "";
    for (std::size_t i = 0; i != n; ++i)
        while (arabic >= digits[i].arabic)
        {
            roman += digits[i].roman;
            arabic -= digits[i].arabic;
        }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Запустим наши тесты.

Думаю, «200» и «300» должны пройти.

«333» тоже прошел.

Отлично.

```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>
...
static void test_to_roman()
{
   assert_to_roman(  1, "I");
   assert_to_roman(  2, "II");
   assert_to_roman(  3, "III");
   assert_to_roman( 10, "X");
   assert_to_roman( 20, "XX");
   assert_to_roman( 30, "XXX");
   assert_to_roman( 33, "XXXIII");
   assert_to_roman(100, "C");
   assert_to_roman(200, "CC");
   assert_to_roman(300, "CCC");
   assert_to_roman(333, "CCCXXXIII");
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Вызовы наших тестов фактически дублируют друг друга. Давай проведем рефакторинг.

Не уверена, что оно того стоит.

Но давай попробуем и посмотрим, станет ли лучше.

```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>
...
static void test_to_roman()
{
   assert_to_roman(  1, "I");
   assert_to_roman(  2, "II");
   assert_to_roman(  3, "III");
   assert_to_roman( 10, "X");
   assert_to_roman( 20, "XX");
   assert_to_roman( 30, "XXX");
   assert_to_roman( 33, "XXXIII");
   assert_to_roman(100, "C");
   assert_to_roman(200, "CC");
   assert_to_roman(300, "CCC");
   assert_to_roman(333, "CCCXXXIII");
}
```

Как и раньше - создам структуру. В ней добавлю поля типа int и string.

```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>
...

struct test_data
{
    int arabic;
    std::string roman;
};

static void test_to_roman()
{
   assert_to_roman(  1, "I");
   assert_to_roman(  2, "II");
   assert_to_roman(  3, "III");
   assert_to_roman( 10, "X");
   assert_to_roman( 20, "XX");
   assert_to_roman( 30, "XXX");
   assert_to_roman( 33, "XXXIII");
   assert_to_roman(100, "C");
   assert_to_roman(200, "CC");
   assert_to_roman(300, "CCC");
   assert_to_roman(333, "CCCXXXIII");
}
```

Теперь объявлю массив таких структур. Начну с двух.

```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>
...
struct test_data
{
    int arabic;
    std::string roman;
};

const test_data tests[] =
{
    { 1, "I" },
    { 2, "II" },
};

static void test_to_roman()
{
   assert_to_roman(  1, "I");
   assert_to_roman(  2, "II");
   assert_to_roman(  3, "III");
   assert_to_roman( 10, "X");
   assert_to_roman( 20, "XX");
   assert_to_roman( 30, "XXX");
   assert_to_roman( 33, "XXXIII");
   assert_to_roman(100, "C");
   assert_to_roman(200, "CC");
   assert_to_roman(300, "CCC");
   assert_to_roman(333, "CCCXXXIII");
}
```

```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>
...
struct test_data
{
    int arabic;
    std::string roman;
};

const test_data tests[] =
{
    { 1, "I" },
    { 2, "II" },
};

int main()
{
    for (size_t i = 0; i != 2; ++i)
        assert_to_roman(
            tests[i].arabic,
            tests[i].roman);

    test_to_roman();
    std::cout << "All tests passed"
              << std::endl;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>
...
struct test_data
{
    int arabic;
    std::string roman;
};

const test_data tests[] =
{
    { 1, "I" },
    { 2, "II" },
};

int main()
{
    const size_t n =
        sizeof tests / sizeof tests[0];

    for (size_t i = 0; i != n; ++i)
        assert_to_roman(
            tests[i].arabic,
            tests[i].roman);

    test_to_roman();
    std::cout << "All tests passed"
              << std::endl;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>
...
struct test_data
{
    int arabic;
    std::string roman;
};

const test_data tests[] =
{
    {   1, "I" },
    {   2, "II" },
    {   3, "III" },
    {  10, "X" },
    {  20, "XX" },
    {  30, "XXX" },
    {  33, "XXXIII" },
    { 100, "C" },
    { 200, "CC" },
    { 300, "CCC" },
    { 333, "CCCXXXIII" },
};
...
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Now I can delete test_to_roman() and its call.

ok

```
...
const test_data tests[] =
{
    {   1, "I" },
    ...
    { 333, "CCCXXXIII" },
};

int main()
{
    const size_t n =
        sizeof tests / sizeof tests[0];

    for (size_t i = 0; i != n; ++i)
        assert_to_roman(
            tests[i].arabic,
            tests[i].roman);

    std::cout << "All tests passed"
              << std::endl;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Can you see anything else?

Perhaps make assert_to_roman() a method of test_data?

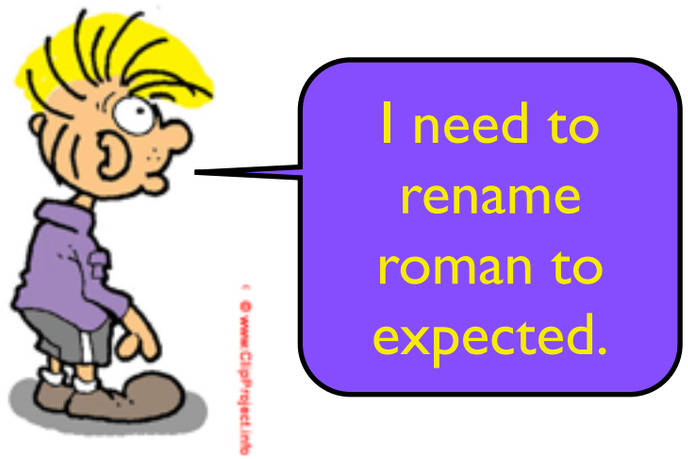
```
...
struct test_data
{
    int arabic;
    std::string roman;
};

const test_data tests[] =
{
    {   1, "I" },
    ...
    { 333, "CCCXXXIII" },
};

int main()
{
    const size_t n =
        sizeof tests / sizeof tests[0];

    for (size_t i = 0; i != n; ++i)
        assert_to_roman(
            tests[i].arabic,
            tests[i].roman);

    std::cout << "All tests passed"
              << std::endl;
}
```

```
...
struct test_data
{
    int arabic;
    std::string roman;
};

const test_data tests[] =
{
    {   1, "I" },
    ...
    { 333, "CCCXXXIII" },
};

int main()
{
    const size_t n =
        sizeof tests / sizeof tests[0];

    for (size_t i = 0; i != n; ++i)
        tests[i].assert_equal();


    std::cout << "All tests passed"
              << std::endl;
}
```

Now I'll add a declaration.

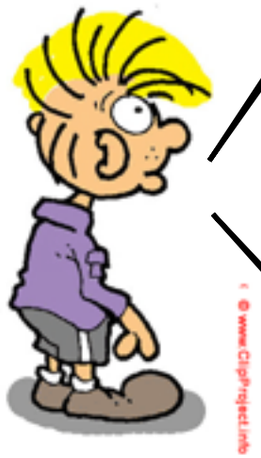
```
...
struct test_data
{
    int arabic;
    std::string roman;
    void assert_equal() const;
};

int main()
{
    const size_t n =
        sizeof tests / sizeof tests[0];

    for (size_t i = 0; i != n; ++i)
       tests[i].assert_equal();

    std::cout << "All tests passed"
              << std::endl;
}
```

```
...
struct test_data
{
    int arabic;
    std::string roman;
    void assert_equal() const;
};


void test_data::assert_equal() const
{
    const std::string actual = to_roman(arabic);
    if (expected != actual)
    {
        std::cerr << "expected: to_roman"
                  << '('  << arabic << ") == "
                  << '"' << expected << '"'
                  << std::endl;
        std::cerr << "  actual: to_roman"
                  << '('  << arabic << ") == "
                  << '"' << actual << '"'
                  << std::endl;
        assert(false);
    }
}
```




```
...
struct test_data
{
    int arabic;
    std::string expected;
    void assert_equal() const;
};

void test_data::assert_equal() const
{
    const std::string actual = to_roman(arabic);
    if (expected != actual)
    {
        std::cerr << "expected: to_roman"
                  << '('  << arabic << ") == "
                  << '"' << expected << '"'
                  << std::endl;
        std::cerr << "  actual: to_roman"
                  << '('  << arabic << ") == "
                  << '"' << actual << '"'
                  << std::endl;
        assert(false);
    }
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```



Anything else?

The name test_data is a bit weak.

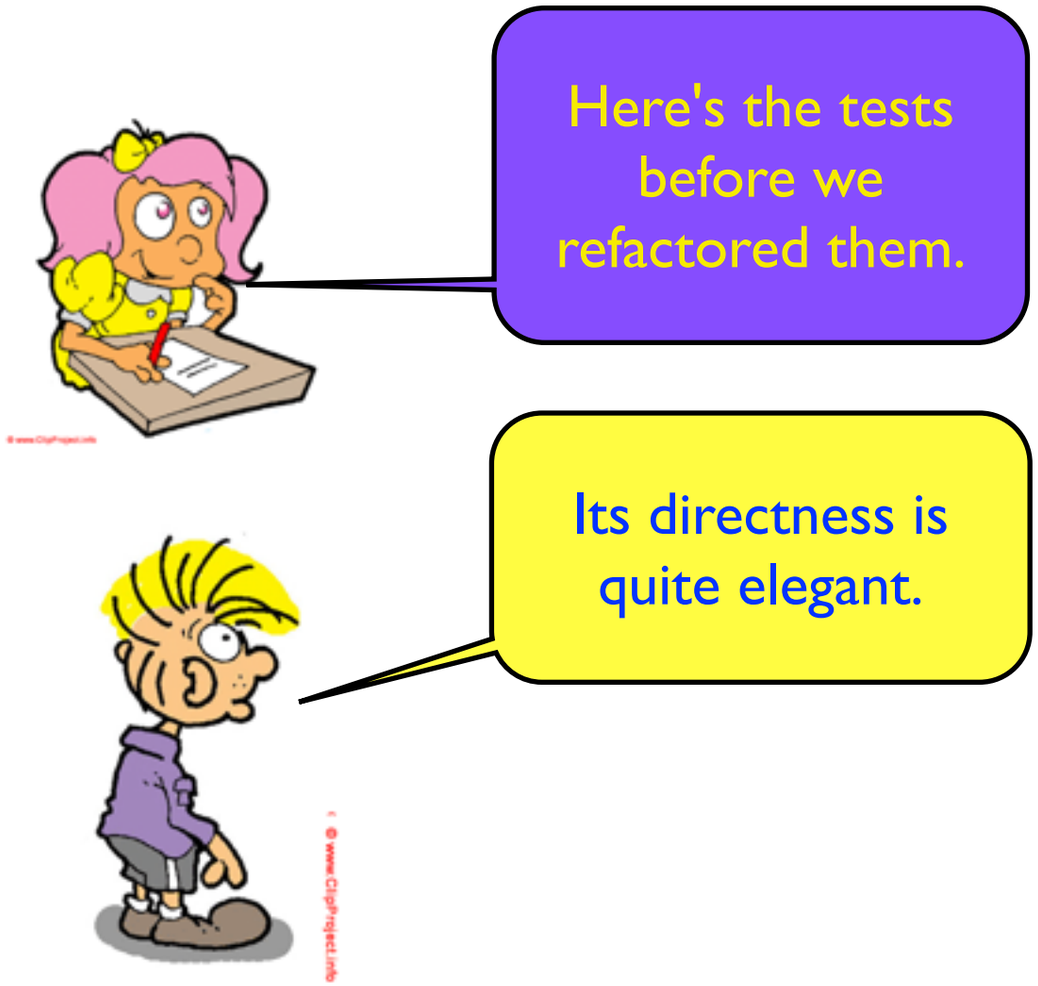
And it should not have external linkage.

Agreed.

But we can come back to that later.

Let's compare the tests before and after the refactoring

Yeah! I forgot about that.

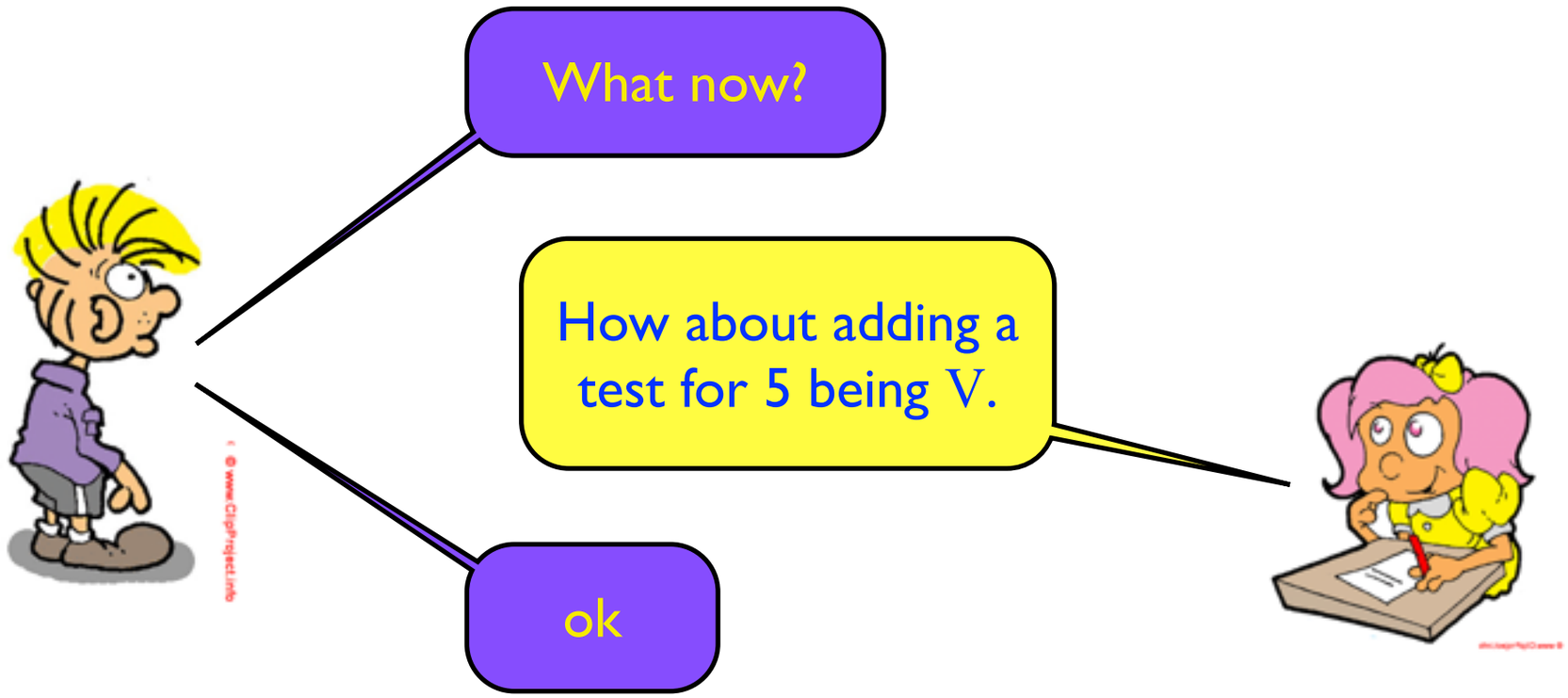



```
#include "to_roman.hpp"
#include <cassert>
#include <iostream>
...
static void test_to_roman()
{
   assert_to_roman(  1, "I");
   assert_to_roman(  2, "II");
   assert_to_roman(  3, "III");
   assert_to_roman( 10, "X");
   assert_to_roman( 20, "XX");
   assert_to_roman( 30, "XXX");
   assert_to_roman( 33, "XXXIII");
   assert_to_roman(100, "C");
   assert_to_roman(200, "CC");
   assert_to_roman(300, "CCC");
   assert_to_roman(333, "CCCXXXIII");
}

int main()
{
    test_to_roman();
    std::cout << "All tests passed"
              << std::endl;
}
```

Here's the tests after we refactored them.

It feels quite different. And there's more lines of code! I'm not sure which I prefer.

```
...
struct test_data
{
    int arabic;
    std::string expected;
    void assert_equal() const;
};

const test_data tests[] =
{
    {   1, "I" },
    {   2, "II" },
    {   3, "III" },
    {  10, "X" },
    {  20, "XX" },
    {  30, "XXX" },
    {  33, "XXXIII" },
    { 100, "C" },
    { 200, "CC" },
    { 300, "CCC" },
    { 333, "CCCXXXIII" },
};

int main()
{
    const size_t n =
        sizeof tests / sizeof tests[0];

    for (size_t i = 0; i != n; ++i)
        tests[i].assert_equal();

    std::cout << "All tests passed"
              << std::endl;
}
```

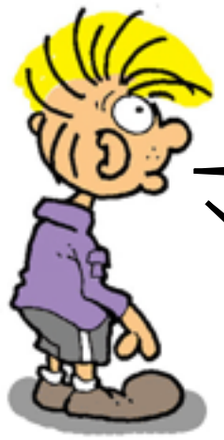

What now?
How about adding a test for 5 being V.
ok

I just have to add one more entry to the tests array.

It fails as expected.

```
...
const test_data tests[] =
{
    {   1, "I" },
    {   2, "II" },
    {   3, "III" },
    {   5, "V" },
    {  10, "X" },
    {  20, "XX" },
    {  30, "XXX" },
    {  33, "XXXIII" },
    { 100, "C" },
    { 200, "CC" },
    { 300, "CCC" },
    { 333, "CCCXXXIII" },
};

int main()
{
    const size_t n =
        sizeof tests / sizeof tests[0];

    for (size_t i = 0; i != n; ++i)
        tests[i].assert_equal();

    std::cout << "All tests passed"
              << std::endl;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
expected: to_roman(5) == "V"
  actual: to_roman(5) == "IIIII"
Assertion failed: (false), ...
```



Now to make it pass.

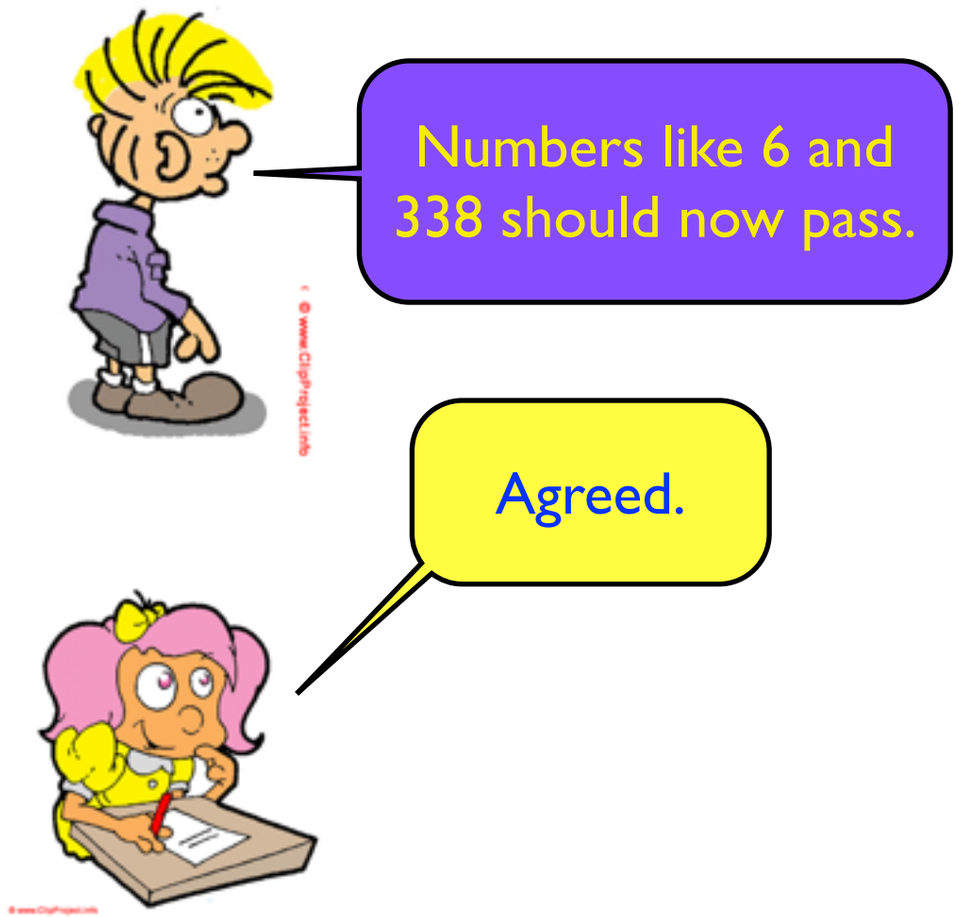
I just need to say the digit 5 is V.

Excellent

```
...
namespace
{
    struct digit
    {
        int arabic;
        std::string roman;
    };

    const digit digits[] =
    {
        { 100, "C" },
        {  10, "X" },
        {   5, "V" },
        {   1, "I" },
    };

    const std::size_t n =
        sizeof digits / sizeof digits[0];
}
```

```
std::string to_roman(int arabic)
{
    std::string roman = "";
    for (std::size_t i = 0; i != n; ++i)
        while (arabic >= digits[i].arabic)
        {
            roman += digits[i].roman;
            arabic -= digits[i].arabic;
        }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```




```
...
const test_data tests[] =
{
    {   1, "I" },
    {   2, "II" },
    {   3, "III" },
    {   5, "V" },
    {   6, "VI" },
    {  10, "X" },
    {  20, "XX" },
    {  30, "XXX" },
    {  33, "XXXIII" },
    { 100, "C" },
    { 200, "CC" },
    { 300, "CCC" },
    { 333, "CCCXXXIII" },
    { 338, "CCCXXXVIII" },
};

int main()
{
    const size_t n =
        sizeof tests / sizeof tests[0];

    for (size_t i = 0; i != n; ++i)
        tests[i].assert_equal();

    std::cout << "All tests passed"
              << std::endl;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

Let's do 4 is IV now.

ok

We already have 1 is I and 5 is V. Perhaps we could somehow combine them to make IV.

Hmmm...

I think it will get quite tricky.

Yes...

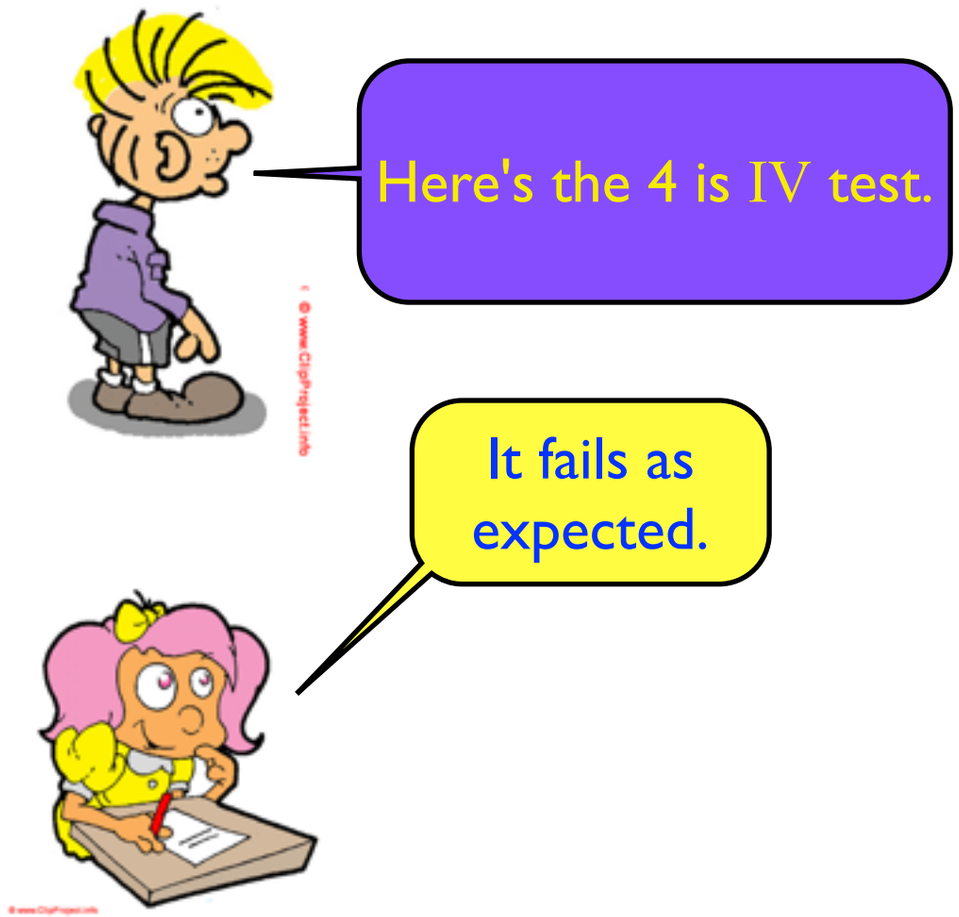
What are you thinking?

I'm thinking that when things get difficult it can be a sign that you're doing it wrong.

We could just add a new 4 is IV entry in the digits array! It seems like a hack though.

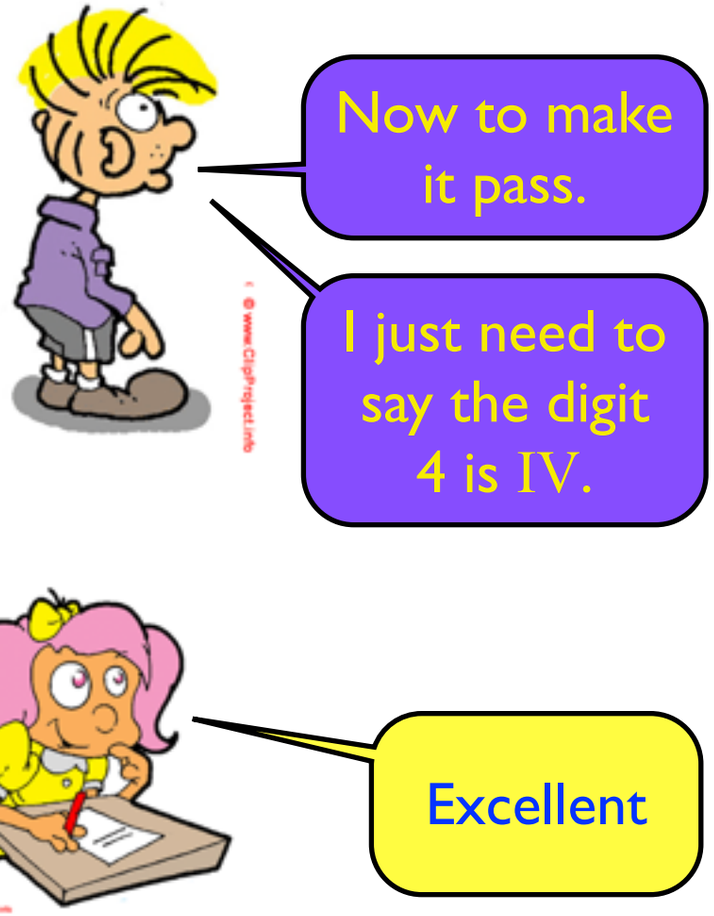
I think that's a great idea. I don't think it's a hack at all! It's the simplest thing that works.

ok!




```
...
const test_data tests[] =
{
    {   1, "I" },
    {   2, "II" },
    {   3, "III" },
    {   4, "IV" },
    {   5, "V" },
    {   6, "VI" },
    {  10, "X" },
    {  20, "XX" },
    {  30, "XXX" },
    {  33, "XXXIII" },
    { 100, "C" },
    { 200, "CC" },
    { 300, "CCC" },
    { 333, "CCCXXXIII" },
    { 338, "CCCXXXVIII" },
};
```

```
$ cc -Wall -Werror *.cpp && ./a.out
expected: to_roman(4) == "IV"
  actual: to_roman(4) == "IIII"
Assertion failed: (false), ...
```




```
...
namespace
{
   struct digit
   {
       int arabic;
       std::string roman;
   };

   const digit digits[] =
   {
       { 100, "C" },
       {  10, "X" },
       {   5, "V" },
       {   4, "IV" },
       {   1, "I" },
   };

   const std::size_t n =
       sizeof digits / sizeof digits[0];
}

std::string to_roman(int arabic)
{
    std::string roman = "";
    for (std::size_t i = 0; i != n; ++i)
        while (arabic >= digits[i].arabic)
        {
            roman += digits[i].roman;
            arabic -= digits[i].arabic;
        }
    return roman;
}
```

```
$ cc -Wall -Werror *.cpp && ./a.out
All tests passed
```

We can repeat that idea for all the digits such as 9 being IX and 40 being XL.

Agreed.

So we've done it. It's just more of the same now.

Yes. I'm sure there'll be a few more small refactorings though. And we've an external linkage todo from earlier.

Thanks for your help.

Any time.