

[« english » блог](#) | [проекты](#) | [интервью](#) | [статьи](#) | [автор](#)

# Скоростное чтение файла в STL через итераторы

Во многих современных языках программирования считать весь файл в строку можно буквально одним оператором, например, в php это делается так:

```
$lines = file_get_contents("textfile.txt");
```

или так, медленно и неэффективно, по-старинке:

```
$lines = join("", file("textfile.txt"));
```

Задумался я, как бы это так элегантно одним оператором сделать в C++. Естественно, хочется, чтобы это работало приемлемо быстро для больших файлов (например, от одного мегабайта и больше).

Первое, что сходу приходит в голову (универсальная кондовая классика):

```
std::ifstream is("testfile.txt");
std::string v;
char buf[N];
while (is) {
    is.read(buf, sizeof(buf));
    v.append(buf, is.gcount());
}
```

С размером буфера N можно еще проиграться, выбрав оптимальный.

Первое улучшение, приходящее в голову — это распределить заранее внутренний буфер для строки, минимизировав количество перераспределений буфера в процессе чтения. Для этого мы, естественно, должны заранее знать размер читаемого файла. Пусть это будет 1 мегабайт (1024\*1024).

```
std::ifstream is("testfile.txt");
std::string v;
// Резервируем внутренний буфер std::string на указанный
// размер в один мегабайт.
v.reserve(1024*1024);
char buf[N];
while (is) {
    is.read(buf, sizeof(buf));
    v.append(buf, is.gcount());
}
```

Далее на сцену выходят STL-алгоритмы и потоковые итераторы. Берем типовый пример использования алгоритма `std::copy`, который есть практически в любой книге по C++:

```
std::ifstream is("testfile.txt");
std::string v;
// Сброс данного флага необходим для отключения пропуска пробельных
// символов при форматном вводе через поток.
is.unsetf(std::ios::skipws);
std::copy(
    std::istream_iterator<char>(is),
    std::istream_iterator<char>(),
    std::back_inserter(v)
);
```

Сразу скажу, это крайне медленный метод. Первым, приходящим в голову улучшением, как всегда, является предварительное распределение буфера приемной строки:

```
std::ifstream is("testfile.txt");
std::string v;
// Резервируем внутренний буфер std::string на указанный
// размер в один мегабайт.
v.reserve(1024*1024);
is.unsetf(std::ios::skipws);
std::copy(
    std::istream_iterator<char>(is),
    std::istream_iterator<char>(),
    std::back_inserter(v)
);
```

В рассмотренном методе видно, что сначала данные изымаются из потока потоковым итератором `istream_iterator`, а потом кладутся через итератор строки `back_inserter` в саму строку. Двойная работа. Есть метод лучше — класть данные из потока напрямую в строку, используя один из специальных конструкторов класса `std::string`:

```
std::ifstream is("testfile.txt");
is.unsetf(std::ios::skipws);
// Самое интересное происходит тут: создается переменная "v"
// через конструктор, работающий напрямую с итераторами потока,
// и данные напрямую поступают во внутренний буфер строки.
std::string v(
    (std::istream_iterator<char>(is)),
    std::istream_iterator<char>()
);
```

Опытный читатель заметит казалось бы ненужные обрамляющие скобки вокруг первого параметра. Сразу скажу — без них работать не будет, а будет ошибка компиляции. Тут мы касаемся одного из “темных углов” C++. Это не самый очевидный вопрос, поэтому я посвятил ему [отдельную статью](#).

Уже лучше, но двигаемся дальше. В потоках ввода есть специальные итераторы `istreambuf_iterator`, которые работают напрямую с внутренними буферами потока в обход всех высокоуровневых функций форматирования и выходного преобразования. Именно по этому для них вызов функции `unsetf` будет уже не нужен:

```
std::ifstream is("testfile.txt");
std::string v;
// Опциональное резервирование буфера приемной строки.
v.reserve(1024*1024);
std::copy(
    std::istreambuf_iterator<char>(is),
    std::istreambuf_iterator<char>(),
    std::back_inserter(v)
);
```

И теперь вариант через конструктор класса `std::string`:

```
std::ifstream is("testfile.txt");
std::string v(
    (std::istreambuf_iterator<char>(is)),
    std::istreambuf_iterator<char>()
);
```

Мы уже близки к идеалу. Теперь встроим создание объекта `std::ifstream` прямо в код создания строки:

```
std::string v(
    (std::istreambuf_iterator<char>(
        std::ifstream("testfile.txt")
    )),
    std::istreambuf_iterator<char>())
);
```

Мы уже в миллиметре от идеала, но в приведенном примере есть одно большое “но”. Вызов `std::ifstream("testfile.txt")` прямо в вызове конструктора создает временный объект, который по стандарту языка всегда является константой, а первый параметр конструктора ожидает принять не константный параметр, поэтому “строгий” компилятор типа `g++` скорее всего выдаст ошибку компиляции, а менее “строгий”, например `cl.exe` от Майкрософта на такой вызов не ругается. Но мы не можем принять такое не универсальное решение, поэтому изменим код, чтобы параметр создавался динамически в куче, а для автоматического его удаления будет использоваться `std::auto_ptr`:

```
std::string v(
    (std::istreambuf_iterator<char>(
        *(std::auto_ptr<std::ifstream>(
            new std::ifstream("testfile.txt")
        )).get()
    )),
    std::istreambuf_iterator<char>())
);
```

Этот код должен работать в любом стандартном компиляторе C++.

Оглядимся назад. У нас столько вариантов — какой выбрать? Для начала, скорость. Надо понять, какой вариант работает банально быстрее. Для этого я собрал все эти варианты в тестовую программу (конечно, с использованием Google Test Framework).

Как я уже [писал](#), вы можете скачать [мою модификацию](#) этой библиотеки, которая сокращена до двух необходимых файлов `gtest/gtest.h` и `gtest-all.cc`.

`filereader_unittest.cpp`:

```
#include <gtest/gtest.h>
```

```
#include <iostream>
#include <streambuf>
#include <istream>
#include <fstream>
#include <ios>
#include <iomanip>
#include <string>
#include <vector>
#include <memory>
#include <cstdlib>
```

```
// Управляющий класс для нашей среды тестирования.
```

```
class Env: public testing::Environment {
public:
```

```
    // Размер тестового файла: 1 мегабайт.
```

```
    static int testfile_sz() { return 1024 * 1024; }
```

```
    // Имя тестового файла.
```

```
    static const char* testfile() { return "testfile"; }
```

```
protected:
```

```
    // Эта функция вызывается один раз в начале тестирования.
```

```
    // Она создает тестовый файл.
```

```
    void Setup() {
```

```
std::string dummy(testfile_sz(), 'x');
std::ofstream os(testfile());
os.write(dummy.c_str(), dummy.length());
}

// Эта функция вызывается один раз после всех тестов.
// Она удаляет тестовый файл.
void TearDown() {
    std::remove(testfile());
}
};

// Функция, реализующая классический метод чтения файла кусками
// заданной длины N. При необходимости производится предварительное
// распределение буфера приемной строки.
void rawRead(int N, bool reserve) {
    std::ifstream is(Env::testfile());

    std::string v;
    if (reserve)
        v.reserve(Env::testfile_sz());

    char* buf = new char[N];
    while (is) {
        is.read(buf, sizeof(buf));
        v.append(buf, is.gcount());
    }
    delete[] buf;

    // На всякий случай проверяем размер считанного файла.
    EXPECT_EQ(Env::testfile_sz(), v.length());
}

// Классическое чтение с буфером в 100 байт.
TEST(ReaderTest, raw_100) {
    rawRead(100, false);
}

// Классическое чтение с буфером в 1 килобайт.
TEST(ReaderTest, raw_1024) {
    rawRead(1024, false);
}

// Классическое чтение с буфером в 10 килобайт.
TEST(ReaderTest, raw_10240) {
    rawRead(10240, false);
}

// Классическое чтение с буфером в 10 килобайт с предварительным
// распределением буфера приемной строки.
TEST(ReaderTest, raw_reserve_10240) {
    rawRead(10240, true);
}

// Функция, реализующая чтение через итератор istream_iterator.
// При необходимости производится предварительное распределение
// буфера приемной строки.
void check_istream_iterator(bool reserve) {
    std::ifstream is(Env::testfile());

    std::string v;
```

```
if (reserve)
    v.reserve(Env::testfile_sz());

// Принудительное игнорирование пропуска пробельных символов.
// С этим флагом двоичные данные будут читаться неверно.
is.unsetf(std::ios::skipws);
std::copy(
    std::istream_iterator<char>(is),
    std::istream_iterator<char>(),
    std::back_inserter(v)
);

// На всякий случай проверяем размер считанного файла.
EXPECT_EQ(Env::testfile_sz(), v.length());
}

// Тестируем работу через istream_iterator.
TEST(ReaderTest, istream_iterator) {
    check_istream_iterator(false);
}

// Тестируем работу через istream_iterator с предварительным
// распределением буфера приемной строки.
TEST(ReaderTest, istream_iterator_reserve) {
    check_istream_iterator(true);
}

// Тестируем работу через istream_iterator при прямом
// вызове конструктора строки, который берет данные напрямую
// из итераторов.
TEST(ReaderTest, istream_iterator_tostring) {
    std::ifstream is(Env::testfile());
    is.unsetf(std::ios::skipws);

    std::string v(
        std::istream_iterator<char>(is),
        std::istream_iterator<char>()
    );

    // На всякий случай проверяем размер считанного файла.
    EXPECT_EQ(Env::testfile_sz(), v.length());
}

// Функция, реализующая чтение через итератор istreambuf_iterator.
// При необходимости производится предварительное распределение
// буфера приемной строки. Для данного метода сброс флага
// std::ios::skipws не нужен, так как этот итератор работает
// на более низком уровне.
void check_istreambuf_iterator(bool reserve) {
    std::ifstream is(Env::testfile());

    std::string v;
    if (reserve)
        v.reserve(Env::testfile_sz());

    std::copy(
        std::istreambuf_iterator<char>(is),
        std::istreambuf_iterator<char>(),
        std::back_inserter(v)
    );
}
```

```
// На всякий случай проверяем размер считанного файла.
EXPECT_EQ(Env::testfile_sz(), v.length());
}

// Тестируем работу через istreambuf_iterator.
TEST(ReaderTest, istreambuf_iterator) {
    check_istreambuf_iterator(false);
}

// Тестируем работу через istreambuf_iterator с предварительным
// распределением буфера приемной строки.
TEST(ReaderTest, istreambuf_iterator_reserve) {
    check_istreambuf_iterator(true);
}

// Тестируем работу через istreambuf_iterator при прямом
// вызове конструктора строки, который берет данные напрямую
// из итераторов.
TEST(ReaderTest, istreambuf_iterator_tostring) {
    std::ifstream is(Env::testfile());

    std::string v(
        (std::istreambuf_iterator<char>(is)),
        std::istreambuf_iterator<char>()
    );

    // На всякий случай проверяем размер считанного файла.
    EXPECT_EQ(Env::testfile_sz(), v.length());
}

#ifdef WIN32

// Этот тест аналогичен тесту istreambuf_iterator_tostring
// за исключение создания объекта потока прямо в вызове
// конструктора строки. Работает только в cl.exe, так как
// "стандартный" компилятор запрещает передавать временные
// объекты по неконстантной ссылке, а cl.exe почему-то это
// разрешает.
TEST(ReaderTest, istreambuf_iterator_tostring_short) {
    std::string v(
        (std::istreambuf_iterator<char>(
            std::ifstream(Env::testfile())
        )),
        std::istreambuf_iterator<char>()
    );

    // На всякий случай проверяем размер считанного файла.
    EXPECT_EQ(Env::testfile_sz(), v.length());
}

#endif

// Финальный метод. Конструктор строки берет данные напрямую
// из итератора istreambuf_iterator. Объект потока создается
// динамически прямо в коде вызова конструктора строки через
// std::auto_ptr.
TEST(ReaderTest, istreambuf_iterator_tostring_short_auto_ptr) {
    std::string v(
        (std::istreambuf_iterator<char>(
            *(std::auto_ptr<std::ifstream>(
                new std::ifstream(Env::testfile())
            )
        ))
    );
}
```

```

    ))).get()
  )),
  std::istreambuf_iterator<char>()
);

// На всякий случай проверяем размер считанного файла.
EXPECT_EQ(Env::testfile_sz(), v.length());
}

// Запуск тестов.
int main(int argc, char **argv) {
  // Инициализация нашей тестовой среды.
  testing::AddGlobalTestEnvironment(new Env);
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}

```

Для компиляции вам необходимы файлы gtest/gtest.h и gtest-all.cc (см. выше).

Для начала скомпилируем в Visual Studio 2008:

```
cl /Fefilereader_vs2008.exe /DWIN32 /O2 /arch:SSE2 /I. /EHsc filereader_unittest.cpp gtest-all.cc
```

Запускаем:

```
filereader_vs2008.exe --gtest_print_time
```

Опция --gtest\_print\_time указывает Google Test выводить время работы каждого теста.

Результат:

```

[=====] Running 12 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 12 tests from ReaderTest
[ RUN    ] ReaderTest.raw_100
[ OK     ] ReaderTest.raw_100 (141 ms)
[ RUN    ] ReaderTest.raw_1024
[ OK     ] ReaderTest.raw_1024 (94 ms)
[ RUN    ] ReaderTest.raw_10240
[ OK     ] ReaderTest.raw_10240 (109 ms)
[ RUN    ] ReaderTest.raw_reserve_10240
[ OK     ] ReaderTest.raw_reserve_10240 (94 ms)
[ RUN    ] ReaderTest.istream_iterator
[ OK     ] ReaderTest.istream_iterator (359 ms)
[ RUN    ] ReaderTest.istream_iterator_reserve
[ OK     ] ReaderTest.istream_iterator_reserve (344 ms)
[ RUN    ] ReaderTest.istream_iterator_tostring
[ OK     ] ReaderTest.istream_iterator_tostring (281 ms)
[ RUN    ] ReaderTest.istreambuf_iterator
[ OK     ] ReaderTest.istreambuf_iterator (141 ms)
[ RUN    ] ReaderTest.istreambuf_iterator_reserve
[ OK     ] ReaderTest.istreambuf_iterator_reserve (125 ms)
[ RUN    ] ReaderTest.istreambuf_iterator_tostring
[ OK     ] ReaderTest.istreambuf_iterator_tostring (78 ms)
[ RUN    ] ReaderTest.istreambuf_iterator_tostring_short
[ OK     ] ReaderTest.istreambuf_iterator_tostring_short (67 ms)
[ RUN    ] ReaderTest.istreambuf_iterator_tostring_short_auto_ptr
[ OK     ] ReaderTest.istreambuf_iterator_tostring_short_auto_ptr (78 ms)
[-----] 12 tests from ReaderTest (1891 ms total)

[-----] Global test environment tear-down
[=====] 12 tests from 1 test case ran. (1906 ms total)
[ PASSED ] 12 tests.

```

Давайте проанализируем результаты.



Мы не будем сравнивать абсолютные времена компиляторов друг против друга. Сейчас не об этом.

**Вывод первый.** Предварительное резервирование буфера приемной строки (метод `reserve()`) не дает никакого эффекта в нашем случае. Может это из-за того, что стратегия расширения буфера при простом линейном добавлении данных итак весьма эффективна в классе `std::string`.

**Вывод второй.** Размер буфера чтения в методе чтения файла явными кусками установленного размера не дал четкой картины. Не очевидно, какой размер буфера может быть потенциально оптимальным. Тут может и дисковый кэш повлиял, может внутреннее буферизирование в классе `std::ifstream`, может что-то еще.

**Вывод третий.** Работа через итератор `istream_iterator` является крайне медленной. Возможно это связано с накладными расходами на форматные преобразования, производимые данным классом и совершенно ненужные в нашей задаче. Для реального использования данный метод практически непригоден.

**Вывод четвертый.** Использование конструктора класса `std::string`, работающего напрямую с итераторами потока, заметно быстрее, чем использование алгоритма `std::copy` (`ReaderTest.istream_iterator` заметно медленнее `ReaderTest.istream_iterator_tostring` и `ReaderTest.iterator` заметно медленнее `ReaderTest.istreambuf_iterator_tostring`). И понятно почему — данные напрямую поступают в буфер строки без ненужного промежуточного копирования.

**Вывод пятый (основной).** Метод чтения через итератор `istreambuf_iterator` с использованием конструктора строки, работающего напрямую с итераторами потока (тест `ReaderTest.istreambuf_iterator_tostring` для “нестроого” компилятора и тест `ReaderTest.istreambuf_iterator_tostring_auto_ptr` для компилятора, следующего стандартам), является весьма эффективным и может конкурировать с ручным блочным чтением. Конечно, текст данного метода весьма непрост и может запутан для понимания на первый взгляд, особенно для начинающих, а блочное чтения прозрачно и ясно, но при почти равной эффективности этих методов нет причин отказываться от работы через итераторы, так как данный метод весь фактически предоставляется библиотекой STL, а значит быть может оптимизирован независимо, без затрагивания кода уже использующей его программы.

Другие посты по теме:

- [Темные углы C++](#)
- [Unit-тестирование для параллельных потоков](#)

---

[Оригинальный пост](#) | [Disclaimer](#)

## Комментарии

11 Comments

Программирование - это просто!

 [Disqus' Privacy Policy](#)

 [Login](#) ▾

 Recommend 2

 Tweet

 Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 

Name



Igor • 10 years ago





Используйте отображение файла в память. Это самый быстрый способ "прочитать весь файл" Windows: CreateFileMapping. Linux: mmap

2 ^ | v • Reply • Share ›



**Evgeniy Gusar** • 9 years ago • edited

Насчет кода на RSDN:

```
#include <vector>
#include <string>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <iterator>

int main(int argc, char* argv[])
{
    std::vector<std::string> v;
    std::ifstream f("e:/test.txt");
    if(f.is_open())
    {
        std::copy(
            std::istream_iterator<std::string>(f),
            std::istream_iterator<std::string>(),
            v.begin(), v.end());
    }
}
```

[see more](#)

1 ^ | v • Reply • Share ›



**zugr** • 10 years ago

Если вместо std::string использовать std::vector то получится презабавная штука... std::string оказывается очень не эффективный при использовании std::back\_inserter

1 ^ | v • Reply • Share ›



**Evgeniy Gusar** • 9 years ago • edited

Спасибо, очень полезная статья!

Я только знакомлюсь с C++, прочел обсуждение на RSDN:

<http://www.rsdn.ru/forum/cp...>

искал расшифровку предложенного там кода с использованием STL-алгоритмов и потоковых итераторов. Твоя статья внесла ясность!

^ | v • Reply • Share ›



**Aleskey** • 10 years ago

```
TEST(ReaderTest, while_getline) {
    std::ifstream is(Env::testfile());
    std::string tmp;
    std::string v;
    while(std::getline(is, tmp))
        v += tmp;

    EXPECT_EQ(Env::testfile_sz(), v.length());
}
```

}

Результаты тестов:

```
[ RUN ] ReaderTest.istreambuf_iterator_tostring_short_auto_ptr
[ OK ] ReaderTest.istreambuf_iterator_tostring_short_auto_ptr (61 ms)
[ RUN ] ReaderTest.while_getline
[ OK ] ReaderTest.while_getline (28 ms)
```

^ | v • Reply • Share ›



**Alex** • 10 years ago

Чтение файла в строку.

<http://groups.google.com/gr...>

Замечание. Один из самых простых и эффективных - использование `rdbuf()`

^ | v • Reply • Share ›



**Alex** • 10 years ago

Simple C/C++ Perfometer: Copying Files

<http://groups.google.co.il/...>

^ | v • Reply • Share ›



**Александр** • 11 years ago

Oleg, ваша правда. Вот что значит copy and paste.

^ | v • Reply • Share ›



**Oleg** • 11 years ago

....

```
char* buf = new char[N]
```

...

`sizeof( buf )` к сожалению будет равно 4 ( или 8 ),  
но не N.

Так, что в некоторых тестах вы померили не совсем то.

^ | v • Reply • Share ›



**Александр** • 11 years ago

Именно из-за неоднозначности многих экспериментов я решился сделать свои, которые и описал тут.

Вообще, сравнивать эффективность кода C++ без включенной оптимизации нельзя. Без оптимизации код C++ может быть в разы медленнее, а вот когда она включена, то все можно работать очень быстро. Про C такого нельзя сказать так явно. Оптимизация конечно ускоряет код C, но это не так критично, как для C++.

У меня получилось, что при работе через `istreambuf_iterator` и через конструктор строки, работающий напрямую с итераторами, скорость ничем не хуже, чем через неформатный ввод в потоке, например, через `read`.

Конечно форматный ввод `"ifstream::operator >>"` может уступать `scanf`у по скорости из-за

перегруженности защитными действиями и большей абстрактностью, но может это приемлемая цена за надежность и отсутствие работы с указателями.

Есть одна ложка дегтя тут: например на SunOS 5.2 версия STLport'a не имеет в классе `std::string` конструктора от итераторов потока, поэтому мой самый быстрый способ чтения на этой платформе оказался не применим. Увы. Но для MSVC и gcc -- все работает отлично.

^ | v • Reply • Share ›



**Andrey** • 11 years ago

Это все конечно красиво и удобно но библиотека ввода/вывода C++ все-таки медленней

© 2009-2013 [Александр Дёмин](#) | [RSS](#)