Работа с файлами — конспект темы

Такие разные потоки

Потоки:

	Стандартный ввод-вывод	Строковый поток	Файловый поток
Заголовочный файл	`<iostream>`	`<sstream>`	`<fstream>`
Поток ввода	`cin`	`istringstream`	`ifstream`
Поток вывода	`cout`, `cerr`	`ostringstream`	`ofstream`
Двунаправленный поток	—	`stringstream`	`fstream`

У потока две стороны. Одна из них активная: это функция или метод вашей программы. Задача другой — выдать или принять данные по запросу.

Поток ввода можно представить ссылкой на `istream&`. Активная сторона читает данные из него. К потокам ввода относятся `cin` и `istringstream` из библиотеки `<sstream>`.

Поток вывода представляется ссылкой на `ostream&`. Активная сторона может записывать данные в него. Этот поток содержит кэширование, ускоряющее его работу. К потокам вывода относятся `cout` и `ostringstream` из библиотеки `<sstream>`.

Ссылкой типа `iostream` можно представить поток, допускающий чтение и запись.

Для работы с файлами в стандартной библиотеке есть специальные потоки, которые станут доступны при подключении файла `<fstream>`:

- поток ввода `ifstream` для чтения из файла,
- поток вывода `ofstream` для записи в файл,
- поток `fstream` для выполнения сразу обеих операций.

Т**екущая директория** — папка, задаваемой окружением операционной системы. Текущая директория зависит от того, как пользователь запустил программу.

Ошибки и как с ними бороться

Поток `istream` будет конвертироваться в `true`, только если последняя операция чтения прошла успешно. Вот почему чтение может не получиться:

1. Неудача при открытии файла/файла нет/недостаточно прав. Критическая ситуация.

2. Ошибка ввода-вывода. Критическая ситуация.
3. В потоке находится не число. Ситуация позволяет выполнить другое чтение.
4. Достигнут конец файла. Штатная ситуация.

Флаги неудач:

- `badbit` — критическая ошибка,
- `failbit` — некритическая ошибка,
- `eofbit` — конец файла.

Читать и задавать флаги можно сеттером `clear` и геттерами. Документация по базовому классу для потоков ввода и вывода `std::basic_ios`.

Чтобы продолжить попытки чтения и заставить операцию `>>` снова работать, нужно сбросить флаги вручную методом `clear`.

Потоки не кидают исключений. Но можно попросить поток делать это

Чтобы поток кидал исключение при ошибке во время установки флага `failbit` или `badbit`, используйте метод `exceptions`.

Ошибка — ещё не конец для потока, но перед следующей операцией нужно очистить флаги.

Тонкости открытия файлов

При открытии файла конструктором или методом `open` задают особенности поведения `fstream` флагами:

- `ios::in` — разрешить чтение,
- `ios::out` — разрешить запись,
- `ios::app` — писать только в конец файла, дополняя его,
- `ios::ate` — первая запись будет в конец файла,
- `ios::trunk` — удалить содержимое файла при открытии,
- `ios::binary` — бинарный режим. Будет рассмотрен в следующих уроках.

Комбинировать флаги можно операцией `|`.

Метод `seekp` меняет позицию записи.

Метод `tellp` предназначен, чтобы узнать эту позицию.

У `tellp` есть второй параметр, обозначающий точку отсчёта. Он может принимать значения:

- `ios::beg` — отсчитывать от начала файла,
- `ios::end` — отсчитывать от конца файла,
- `ios::cur` — отсчитывать от текущего места.

При использовании `ios::end` и `ios::cur` значение позиции может быть отрицательным.

У методов `seekp` и `tellp` есть аналоги — `seekg` и `tellg`. Они влияют на операции чтения.

Бинарные файлы: читаем и пишем байты

В потоках чтения есть метод `get`. Если вызвать его без параметров, он прочтёт один байт и возвратит целое число. Обычно оно находится в пределах от 0 до 255. Но если достигнут конец файла или произошла ошибка, это число равно специальному отрицательному значению. Его можно получить вызовом `istream::char_traits::eof()`.

Для чтения определённого количества байтов у потока `istream` есть специальный метод `read`. Его результатом будет сам поток, как во многих других методах потоков.

Чтобы узнать количество прочитанных символов, после `read` вызовите метод `gcount`.

В этом примере метод `read` читает со скоростью примерно 280 Мбит/с, а многократные `get` показывают всего 11 Мбит/с:

```
#include "log_duration.h"

#include <fstream>
#include <iostream>
#include <random>
#include <string>

using namespace std;

size_t ReadExact(istream& input, char* dst, size_t count) {
    for (size_t i = 0; i < count; ++i) {
        int c = input.get();
        if (c == istream::traits_type::eof() || !input) {
            return i;
        }
        dst[i] = static_cast<char>(c);
    }

    return count;
}

string GenerateRandomString(size_t size) {
```

```
    string random_str(size, 0);
    static mt19937 engine;

    for (char& c : random_str) {
        c = static_cast<char>(uniform_int_distribution<int>('A', 'Z')(engine));
    }

    return random_str;
}

int main() {
    // размер файла 10 мегабайт
    static const int FILE_SIZE = 10 * 1024 * 1024;
    static const int READ_COUNT = 10;

    // создаём файл нужного размера
    {
        ofstream test_out("test.txt");
        test_out << GenerateRandomString(FILE_SIZE);
    }

    vector<char> buffer(FILE_SIZE);

    // прочитаем его заданное количество раз двумя способами
    {
        LOG_DURATION("multiple get");
        ifstream test_in("test.txt");
        for (int i = 0; i < READ_COUNT; ++i) {
            test_in.seekg(0);
            ReadExact(test_in, buffer.data(), FILE_SIZE);
        }
    }

    {
        LOG_DURATION("stream read");
        ifstream test_in("test.txt");
        for (int i = 0; i < READ_COUNT; ++i) {
            test_in.seekg(0);
            test_in.read(buffer.data(), FILE_SIZE);
        }
    }
}
```

`get` хорош, если нужен один символ, от которого зависит, что делать дальше. `get` неприемлем, если нужно прочитать файл целиком и что-нибудь сделать с каждым символом. Вместо этого используйте буфер. Хороший размер буфера — один килобайт.

Каждая потоковая операция выполняет несколько вспомогательных вызовов, обрабатывает флаги ошибок и требует немалых накладных расходов. Поэтому количество потоковых операций надо свести к минимуму.

Аналоги `get` и `read` для записи — методы `put` и `write`. На них распространяются те же правила, что и на методы чтения: не следует использовать `put` для записи многих

символов подряд.

Надёжный способ узнать, сколько символов прочитано, — метод `gcount`. Даже если случилась ошибка во время чтения, его значение будет 0, и `write` ничего не запишет.

Если программа работает некорректно и не может скопировать бинарный файл, это опасная ошибка. Чтобы её не произошло, добавьте флаг `binary` при открытии обоих файлов — входного и выходного. Используйте `binary` везде, кроме случаев, когда вы уверены, что придётся работать именно с текстовым файлом.

Тернистые пути: работаем с файловой системой

Для операций с файлами существует стандартная библиотека filesystem. При компоновке программ с использованием filesystem могут потребоваться дополнительные опции компиляции. Для `gcc` до версии 9.1 нужен ключ `-lstdc++fs`.

filesystem предназначена, чтобы дать наиболее универсальный интерфейс файловой системы, который бы работал корректно на разных платформах.

Базовое понятие filesystem — **путь**. Он представляется классом `std::filesystem::path`. Он способен задавать существующий или несуществующий файл.

Путь состоит из **секций** — промежуточных папок. При комбинировании путей можно не задумываться о том, какой слеш нужен в данной операционной системе, и использовать операцию `/`:

```
#include <filesystem>
#include <iostream>

using namespace std;

int main() {
    using filesystem::path;

    path p = path("a") / path("folder") / path("and") / path("..") / path("a") / path("file.txt");

    // выводим естественное представление пути в std::string
    cout << p.string() << endl;
}
```

Строковые литералы с суффиксом `s` или `sv` позволяют создавать объект `string` или `string_view` эффективно и коротко. Аналогичного суффикса для создания путей в стандартной библиотеке нет. Но его можно написать самостоятельно.

Для чего используют пути:

- вместо `string` в конструкторах файловых потоков `fstream`, `ofstream`, `ifstream`, чтобы открыть файл;
- получение абсолютного пути из относительного;
- создание папки. В filesystem есть две функции для создания папок: `create_directory` и `create_directories`;
- различение файла и папки;
- получение всех файлов в папке.

Больше о путях

У пути два формата строкового представления:

- native,
- generic.

В native-формате путь выглядит так, как он должен быть представлен в операционной системе. Метод `path::string` возвращает путь в этом представлении. Оно рекомендуется в большинстве случаев.

Формат generic одинаковый на всех операционных системах. Чтобы получить generic-представление, вызовите метод `generic_string`. Это метод класса `path`.

Под Windows путь в native-формате иногда содержит оба вида слешей, но предпочтительнее обратный.

Можно явно заменить в переменной типа `path` все слеши на предпочтительные методом `make_preferred`:

```
path p = path("a/folder/and/../a/file.txt"s, path::generic_format);

cout << "p.string():                      "sv << p.string() << endl;
p.make_preferred();
cout << "p.string() после make_preferred: "sv << p.string() << endl;
cout << "p.generic_string():              "sv << p.generic_string() << endl;
```

Возможный вывод:

```
p.string():                      a/folder/and/../a/file.txt
p.string() после make_preferred: a\folder\and\..\a\file.txt
p.generic_string():              a/folder/and/../a/file.txt
```

Рассмотрим другие операции с путями, которые не вошли в прошлый урок.

Пути, содержащие элементы папки `.` или несколько слешей подряд, можно упростить методом `lexically_normal`:

```
path p = "a"_p / "folder"_p / "and"_p / ".."_p / "a"_p / "file.txt"_p;

cout << "Исходный вид: "sv << p.string() << endl;
p = p.lexically_normal();
cout << "После lexically_normal(): "sv << p.string() << endl;
```

Возможный вывод:

```
Исходный вид: a/folder/and/../a/file.txt
После lexically_normal(): a/folder/a/file.txt
```

Полученный путь называется нормальным.

Чтобы получить родительскую папку, используйте метод пути `parent_path`.

В библиотеке filesystem есть и другие функции, например:

- `exists` — проверка существования файла или папки. Может принимать статус или путь.
- `copy`, `copy_file` — копирование файлов и папок.
- `current_path` — получение и изменение текущего пути.
- `equivalent` — проверка, что два пути указывают на один объект.
- `file_size` — получение размера файла.
- `remove`, `remove_all` — удаление файлов и папок.
- `resize_file` — изменение размера файлов. Увеличивать размер можно потоком `ofstream`, а уменьшать — этой функцией.

Полный список — на сайте cppreference.com.

Свой препроцессор

Регулярное выражение — специальный паттерн, который применяют для поиска строки определённого вида, замен в тексте, проверки на соответствие. Функции для работы с регулярными выражениями располагаются в `<regex>`. Они подробно описаны в документации.

Для проверки строки на соответствие регулярному выражению используйте алгоритм `regex_match`, принимающий параметры:

- `str` — строка или литерал, которую нужно проверять на соответствие. Можно также указывать два итератора на символы;
- `m` (необязательный) — ссылка на объект типа `smatch`, в который будет записан результат сопоставления;
- `e` — само регулярное выражение;
- `flags` (необязательный) — параметры соответствия.

Функция возвращает `true`, если соответствие найдено. При этом метод `empty` параметра `m` будет возвращать `false`.

Сырой литерал — разновидность строкового литерала, позволяющая не экранировать спецсимволы: слеши, переводы строк и даже кавычки.

Чтобы задать **сырой литерал**, поставьте перед кавычкой букву R. Литерал позволяет задать произвольную последовательность конца литерала, которая пишется между кавычкой и открывающей скобкой.

Чтобы показать, где литерал закончился, закройте скобку и повторите эту последовательность. Для регулярного выражения можно использовать последовательность конца из одного символа `/`:

```
static regex num_reg(R"/(\s*([+-]?[0-9]+)\.([0-9]*)(e[+-]?[0-9]+)?\s*)/");
```