Министерство образования Республики Беларусь
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет прикладной математики и информатики

Учебно-методические материалы для курса
«Основы и методологии программирования» и для дисциплины
«Учебная вычислительная (ознакомительная практика)»

(НЕ ДЛЯ РАСПРОСТАНЕНИЯ – ТОЛЬКО ДЛЯ ВНУТРЕННЕГО ИСПОЛЬЗОВАНИЯ)

Разработчики:
Бутримов К.,
Казанцева О.

Минск 2023

СОДЕРЖАНИЕ

ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1 Работа с файлами формата JSON

1.1.1 Общие сведения о формате JSON

JSON (**J**ava**S**cript **O**bject **N**otation) – простой, удобный для чтения и написания формат обмена данными, разработанный Дугласом Крокфордом (соучредитель технологической консалтинговой компании State Software). Данный формат основан на подмножестве языка программирования JavaScript, определенного в стандарте ECMA-262 3rd Edition – December 1999, и полностью независим от языка реализации. Более того, во многих язык программирования предусмотрена возможность работы с данными в формате JSON.

Формат JSON поддерживает следующие типы данных:

- *Число* (целое или вещественное), записываемое в десятичной системе счисления с точкой «`.`» в качестве разделителя.
- Литералы `true` (истинное значение), `false` (ложное значение) и `null` (пустое значение).
- *Строка*. Представляет собой упорядоченное множество из нуля или более символов юникода, заключенное в двойные кавычки «`"`». Символы могут быть экранированы при помощи обратной косой черты «`\`» (например, `\"`, `\\`, `\/`, `\n`, `\r`, `\f`, `\b`).
- *Объект*. Представляет собой неупорядоченное множество пар *ключ:значение*, заключенное в фигурные скобки «`{}`». Ключ является строкой и отделяется от значения двоеточием «`:`». Пары отделяются друг от друга запятыми. Значение может иметь любой поддерживаемый тип.
- *Массив*. Представляет собой упорядоченное множество значений, заключенное в квадратные скобки «`[]`». Значения отделяются друг от друга запятыми и могут иметь любой поддерживаемый тип.

Пробелы допускаются и игнорируются вокруг и между синтаксическими элементами (но не внутри *строк*). Аналогично для символов *табуляции*, *конца строки* и *возврата каретки*. *Комментарии* в данном формате намеренно не предусмотрены. Также не допускается использование «конечных запятых» в структурах данных (например, после последнего значения в *массиве* или после последней пары в *объекте*).

1.1.2 Структура JSON файла

Содержимое файла в формате JSON представляет собой либо *объект*, либо *массив*.

Пример 1. Содержимое файла формата JSON в качестве *объекта*

```
{
    "name" : "Maksim",
```

```
    "surname" : "Kovalev",
    "status" : "student"
    "age" : 22,
    "address" :
    {
        "city" : "Minsk",
        "street" : "Rokossovsky Avenue"
        "postal code" : 220094
    },
    "favorite subjects" :
    [
        "C++",
        "Physics",
        "Mathematics"
    ],
    "GPA" : 9.3,
    "is married" : false,
    "place of work" : null
}
```

Пример 2. Содержимое файла формата JSON в качестве *массива*

```
[
    {
        "id" : 34854,
        "year" : "2020",
        "event" : "NASA launches Mars 2020 to Mars."
    },
    {
        "id" : 2357,
        "year" : "1668",
        "event" : "Isaac Newton builds the first reflecting telescope."
    },
    {
        "id" : 11429,
        "year" : "1995",
        "event" : "The first exoplanet, 51 Pegasi b, is discovered."
    },
    [
        1668,
        1995,
        2020
    ],
    34854
]
```

1.1.3 JSON и C++

C++ не предоставляет поддержки JSON ни на уровне языка, ни на уровне стандартной библиотеки. Для работы с указанным форматом существует ряд сторонних средств, например boost, Qt, RapidJSON, ujson, nlohmann/json, SimpleJSON, V8 и др. Далее будут описаны базовые концепции работы с форматом JSON при помощи библиотеки nlohmann/json.

Особенностями данной библиотеки являются:

- Интуитивно понятный интерфейс;
- Возможность использования за счет единственного заголовочного файла `json.hpp`;
- Совместимость с компиляторами GCC, Clang, Apple Clang, Intel C++ Compiler, Nvidia CUDA Compiler, Microsoft Visual C++;

– Использование стандарта C++ 11;
– Высокая степень покрытия исходного кода юнит-тестами.

1.1.4 Класс nlohmann::json

Все поддерживаемые форматом JSON типы данных представлены классом `json`, находящимся в пространстве имен `nlohmann`. Соответственно, для каждого типа существует один или несколько конструкторов данного класса.

Пример 1. Создание объектов класса `nlohmann::json`

```
#include "json.hpp"

int main() {
    // Соответствует типу "число"
    nlohmann::json j_int( 1963 );

    // Соответствует литералу "true"
    nlohmann::json j_bool( true );

    // Соответствует литералу "null" (конструктор по умолчанию)
    nlohmann::json j_null_0;

    // Соответствует литералу "null"
    nlohmann::json j_null_1( nullptr );

    // Соответствует типу "объект"
    nlohmann::json j_object( { { "key_bool", false }, { "key_int", 1964 } } );

    // Соответствует типу "массив"
    nlohmann::json j_array(  { "key_0", true, 2023, "key_1" } );

    return 0;
}
```

Для определения, какому именно типу соответствует объект класса `nlohmann::json`, используются следующие методы данного класса:

– `is_null()` определяет соответствие литералу `null`;
– `is_boolean()` определяет соответствие литералам `true` и `false`;
– `is_digit()` определяет соответствие типу «число»;
– `is_string()` определяет соответствие типу «строка»;
– `is_object()` определяет соответствие типу «объект»;
– `is_array()` определяет соответствие типу «массив».

Пример 2. Проверка соответствия типов

```
#include "json.hpp"

#include <iostream>

int main() {
    using namespace std;

    nlohmann::json j_object( { { "key_bool", false }, { "key_int", 1964 } } );

    cout << boolalpha;

    cout << "Is null: "     << j_object.is_null()       << endl;
    cout << "Is boolean: "  << j_object.is_boolean()    << endl;
    cout << "Is number: "   << j_object.is_number()     << endl;
```

```
    cout << "Is string: "    << j_object.is_string()     << endl;
    cout << "Is object: "    << j_object.is_object()     << endl;
    cout << "Is array: "     << j_object.is_array()      << endl;

    return 0;
}
```

```
Is null: false
Is boolean: false
Is number: false
Is string: false
Is object: true
Is array: false
```

1.1.5 Класс nlohmann::json и STL

Класс `nlohmann::json` предоставляет интерфейс, схожий с интерфейсами типичных STL-контейнеров (`std::vector`, `std::map`). Иными словами:

- работа с объектом класса `nlohmann::json`, соответствующим типу «объект», аналогична работе с контейнером `std::map<std::string, nlohmann::json>`;
- работа с объектом класса `nlohmann::json`, соответствующим типу «массив», аналогична работе с контейнером `std::vector<nlohmann::json>`.

Пример 1. `nlohmann::json` и `std::vector`

```
#include "json.hpp"

int main() {
    nlohmann::json j_array;

    // Заполнение json-массива при помощи метода push_back
    j_array.push_back( 1963 );
    j_array.push_back( "str" );
    j_array.push_back( false );

    // Использование итераторов
    for ( nlohmann::json::iterator it = j_array.begin();
          it != j_array.end(); ++it )
    {
        // ...
    }

    // Ranged-based for
    for ( const nlohmann::json& json_element : j_array )
    {
        // ...
    }

    // Обращение к элементам json-массива по индексу
    for ( std::size_t i = 0; i < j_array.size(); ++i )
    {
        nlohmann::json& json_element = j_array[i];
        // ...
    }

    // Получение размера json-массива
    std::size_t j_array_size = j_array.size();
```

```
    // Проверка json-массива на пустоту
    if ( j_array.empty() )
    {
        // ...
    }

    // Удаление всех элементов json-массива
    j_array.clear();

    return 0;
}
```

Пример 2. `nlohmann::json` и `std::map`

```
#include "json.hpp"

int main() {
    nlohmann::json j_object;

    // Заполнение json-объекта при помощи доступа по ключу
    j_object["key_str"] = "value_str";
    j_object["key_number"] = 1964;
    j_object["key_boolean"] = false;
    j_object["key_null"] = nullptr;

    // Использование итераторов
    for ( nlohmann::json::iterator it = j_object.begin();
          it != j_object.end(); ++it ) {
        const std::string& key = it.key();
        nlohmann::json& value = it.value();
        // ...
    }

    // Ranged-based for
    for ( auto& element : j_object.items() ) {
        const std::string& key = element.key();
        nlohmann::json& value = element.value();
        // ...
    }

    // Structured bindings (C++17)
    for ( auto& [key, value] : j_object.items() ) {
        // ...
    }

    // Проверка наличия значения по указанному ключу методом contains
    if ( j_object.contains( "some_key" ) ) {
        // ...
    }

    // Проверка наличия значения по указанному ключу методом find
    if ( j_object.find( "some_key" ) != j_object.end() ) {
        // ...
    }

    // Получение количества элементов json-бъекта
    std::size_t j_array_size = j_object.size();

    // Проверка json-объекта на пустоту
    if ( j_object.empty() ) {
        // ...
    }
    // Удаление элемента json-объекта по указанному ключу
    j_object.erase( "some_key" );
```

```
    // Удаление всех элементов json-объекта
    j_object.clear();

    return 0;
}
```

В приведенных выше примерах описаны не все возможности взаимодействия с объектами класса `nlohmann::json`. Более детальную информацию можно найти в официальной документации.

1.1.6 Методы get, get_ref и get_ptr

Для получения значений, представленных объектами класса `nlohmann::json`, а также и ссылок и указателей на эти значения используются шаблонные методы `get`, `get_ref` и `get_ptr` соответственно.

Пример. Извлечение значений из объектов класса `nlohmann::json`

```
#include "json.hpp"

int main()
{
    nlohmann::json j_array;

    j_array.push_back( 1963 );
    j_array.push_back( false );
    j_array.push_back( "str" );
    j_array.push_back( { {"key_str", "value_str"}, {"key_bool", true} } );
    j_array.push_back( { "1", 2 , 3, "false" , true } );

    int int_value =          j_array[0].get<int>();
    bool bool_value =        j_array[1].get<bool>();
    std::string str_value = j_array[2].get<std::string>();
    std::string& str_ref =  j_array[2].get_ref<std::string&>();
    std::string* str_ptr =  j_array[2].get_ptr<std::string*>();

    // nlohmann::json::boolean_t = bool
    nlohmann::json::boolean_t boolean_t_value =
        j_array[1].get<nlohmann::json::boolean_t>();

    // nlohmann::json::string_t = std::string
    nlohmann::json::string_t string_t_value =
        j_array[2].get<nlohmann::json::string_t>();

    const nlohmann::json::string_t& string_t_const_ref =
        j_array[2].get_ref<const nlohmann::json::string_t&>();

    // nlohmann::json::object_t = std::map<std::string, nlohmann::json>
    nlohmann::json::object_t* j_inner_object_ptr =
        j_array[3].get_ptr<nlohmann::json::object_t*>();
    nlohmann::json::object_t* const j_inner_object_const_ptr =
        j_array[3].get_ptr<nlohmann::json::object_t* const>();

    // nlohmann::json::array_t = std::vector<nlohmann::json>
    nlohmann::json::array_t& j_inner_array_ref =
        j_array[4].get_ref<nlohmann::json::array_t&>();

    const nlohmann::json::array_t& j_inner_array_const_ref =
        j_array[4].get_ref<const nlohmann::json::array_t&>();

    return 0;
}
```

1.1.7 Метод dump

Статический метод `parse` класса `nlohmann::json` позволяет создавать объекты данного класса из файловых потоков, строк, а также строковых литералов.

Пример. Создание объектов класса `nlohmann::json` при помощи метода `parse`

```
#include "json.hpp"

#include <fstream>

int main()
{
    std::ifstream in( "example.json" );
    nlohmann::json json_from_file = nlohmann::json::parse( in );
    in.close();

    std::string json_str( "[ 1, 2, true, \"some_str\" ]" );
    nlohmann::json json_from_str =
        nlohmann::json::parse( json_str );

    nlohmann::json json_from_str_literal_0 =
        nlohmann::json::parse( "[ null ]" );

    nlohmann::json json_from_str_literal_1 =
        nlohmann::json::parse( R"({ "one" : 1, "two" : 2 })" );

    using namespace nlohmann::literals;
    nlohmann::json json_from_str_literal_2 =
        R"( {"date" : "2023-01-01", "time" : "00:00"} )"_json;

    return 0;
}
```

1.1.8 Метод dump

Для получения строкового представления объекта класса `nlohmann::json` используется метод `dump`. Параметры, поступающие на вход данному методу, определяют способ форматирования выходного строкового представления (объекта класса `std::string`) и имеют значения по умолчанию. Однако, для получения более удобного для чтения вида рекомендуется передавать на вход единственный параметр: 4.

Пример 2. Использование метода `dump` класса `nlohmann::json`

```
#include "json.hpp"

#include <iostream>

int main()
{
    nlohmann::json j_object;
    j_object["key_str"] = "value_str";
    j_object["key_bool"] = false;
    j_object["key_array"] = { true, 2, "3", 4, nullptr };
    j_object["key_null"] = nullptr;
    j_object["key_object"] = { {"one", 1}, {"true", true} };

    std::cout << j_object.dump( 4 ) << std::endl;
```

```
    return 0;
}
```

```
{
    "key_array": [
        true,
        2,
        "3",
        4,
        null
    ],
    "key_bool": false,
    "key_null": null,
    "key_object": {
        "one": 1,
        "true": true
    },
    "key_str": "value_str"
}
```

1.1.9 Обработка исключений

Все классы исключений, генерируемых библиотекой `nlohmann/json`, унаследованы от класса `json::exception`, который, в свою очередь, унаследован от класса `std::exception`.

Пример. Обработка исключения, генерируемого при ошибке создания json-объекта

```
#include "json.hpp"

#include <iostream>

int main()
{
    try
    {
        nlohmann::json j_object;
        j_object["key_str"] = "value_str";
        j_object["key_array"] = { true, 2, "3", 4, nullptr };
        j_object.push_back( nullptr );
        j_object["key_object"] = { {"one", 1}, {"true", true} };

        std::cout << j_object.dump( 4 ) << std::endl;
    }
    catch ( const std::exception& e )
    {
        std::cout << e.what() << std::endl;
    }

    return 0;
}
```

```
[json.exception.type_error.308] cannot use push_back() with object
```

1.2 Работа с файловой системой

1.2.1 Общие сведения

Центральным объектом файловой системы является *файл*. Кратко файл можно охарактеризовать как объект файловой системы, поддерживающий операции ввода и вывода и содержащий данные. Файлы существуют в контейнерах, называемых *каталогами*, которые могут быть вложены в другие каталоги. Для простоты каталог можно рассматривать в качестве файла. Каталог, содержащий файл, называется *родительским каталогом* для этого файла.

Путь – это строка, идентифицирующая конкретный файл. Пути начинаются необязательного *корневого имени*, за которым следует необязательный корневой каталог. Остальная часть пути представляет собой последовательность каталогов, разделенных определенными разделителями. Пути могут закачиваться как каталогом, так и файлом.

1.2.2 Класс std::filesystem::path

`std::filesystem::path` – это класс пространства имен `std::filesystem` для моделирования пути. Наиболее распространенными методами создания объектов такого класса являются создание через конструктор по умолчанию (создается пустой путь) и создание через строковый тип (создает путь, указанный символами в строке).

Пример. Создание объектов класса `std::filesystem::path`

```
#include <filesystem>

int main()
{
    std::filesystem::path empty_path;
    std::filesystem::path path( "C:\\Temp" );

    return 0;
}
```

В примере создаются два пути: `empty_path` при помощи конструктора по умолчанию (пустой путь) и `path` при помощи последовательности символов `C:\\Temp (const char[11])`.

1.2.3 Декомпозиция пути

В классе `std::filesystem::path` реализованы методы, позволяющие извлекать различные компоненты из пути, например:

- `root_name()` возвращает имя корня;
- `root_directory()` возвращает корневой каталог;
- `root_path()` возвращает корневой путь;
- `parent_path()` возвращает родительский путь;
- `filename()` возвращает имя файла (с расширением);
- `stem()` возвращает имя файла (без расширения);

– `extension()` возвращает расширение.

Стоит отметить, что все вышеперечисленные методы возвращают объект класса `std::filesystem::path`. При необходимости из него можно получить объект класса std::string при помощи метода `string()`.

Пример. Декомпозиция пути

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

int main()
{
    using namespace std;

    filesystem::path p( R"(C:\Temp\tmp.txt)" );
    cout <<
        "Root name: "           << p.root_name() <<
        "\nRoot directory: "    << p.root_directory() <<
        "\nRoot path: "         << p.root_path() <<
        "\nParent path: "       << p.parent_path() <<
        "\nFilename: "          << p.filename() <<
        "\nStem: "              << p.stem() <<
        "\nExtension: "         << p.extension() << endl;

    return 0;
}
```

```
--------------------------------------------------------------------------------
Root name: "C:"
Root directory: "\\"
Root path: "C:\\"
Parent path: "C:\\Temp"
Filename: "tmp.txt"
Stem: "tmp"
Extension: ".txt"
```

1.2.4 Изменение пути

В дополнение к методам декомпозиции класс `std::filesystem::path` предлагает методы, позволяющие изменять различные характеристики пути, например:

– `clear()` очищает путь;
– `remove_filename()` удаляет имя файла из пути;
– `replace_filename(p)` заменяет имя файла из текущего пути на имя `p`;
– `remove_extension()` удаляет расширение файла;
– `replace_extension(p)` заменяет расширение файла по текущему пути на расширение `p`.

Пример. Изменение параметров пути

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

int main()
{
    using namespace std;

    filesystem::path p( R"(C:\Temp\tmp.txt)" );
    cout << p << endl;
```

```
    p.replace_filename( "tmp01.txt" );
    cout << p << endl;

    p.replace_extension( ".json" );
    cout << p << endl;

    p.remove_filename();
    cout << p << endl;

    p.clear();
    cout << p << endl;

    return 0;
}
```

```
--------------------------------------------------------------------------------
"C:\\Temp\\tmp.txt"
"C:\\Temp\\tmp01.txt"
"C:\\Temp\\tmp01.json"
"C:\\Temp\\"
""
```

1.2.5 Обработка ошибок

Взаимодействие с файловой системой может привести к ошибкам, например, ненайденных файлов, недостаточных разрешений. Следовательно, функции, взаимодействующие с файловой системой, должны сообщать вызывающей стороне об ошибках (методы класса `std::filesystem::path` к таким функциям не относятся, поскольку, на самом деле, не взаимодействуют с файловой системой).

Каждая такая функция имеет две перегрузки. В первом случае в качестве параметра передается ссылка на объект класса `std::error_code`, в который функция запишет условие ошибки. Во втором случает функция сгенерирует исключение `std::filesystem::filesystem_error` (унаследован от `std::system::error`).

1.2.6 Параметры файлов и каталогов

Существует ряд функций в пространстве имен `std::filesystem`, позволяющих получить информацию о различных атрибутах файловой системы:

- `current_path()` возвращает текущий путь программы;
- `current_path(p)` устанавливает текущий путь программы равным `p`;
- `exists(p)` проверяет существование переданного пути `p`;
- `equivalent(p1, p2)` проверяет, ссылаются ли `p1` и `p2` на один и тот же объект файловой системы;
- `file_size(p)` возвращает размер файла по пути `p` в байтах.

Пример. Извлечение некоторых атрибутов файловой системы

```
int main() {
    using namespace std;

    filesystem::path p1( R"(C:\Temp\tmp01.txt)" );
    filesystem::path p2( R"(C:\Temp\tmp02.txt)" );

    cout <<
```

```
        boolalpha <<
        filesystem::exists( p1 )            << endl <<
        filesystem::exists( p2 )            << endl <<
        filesystem::file_size( p1 )         << endl <<
        filesystem::file_size( p2 )         << endl <<
        filesystem::equivalent( p1, p2 )    << endl;

    return 0;
}
```

```
true
true
726
1020
false
```

1.2.7 Управление файлами и каталогами

Существует ряд функций в пространстве имен `std::filesystem` для управления файлами и каталогами:

- `copy(p1, p2)` копирует файлы и каталоги из `p1` в `p2`. Можно предоставить опцию `std::filesystem::copy_options` для настройки поведения (необязательный параметр функции). Возможные значения данного класса перечисления: значение по умолчанию `none`, `skip_existing` (существующий файл по пути `p2` не изменяется), `overwrite_existing` (существующий файл по пути `p2` перезаписывается файлом по пути `p2`) и `update_existing` (существующий файл по пути `p2` перезаписывается по пути если файл по пути `p1` новее);
- `create_directory(p)` создает каталог по пути `p` (не создает вложенных каталогов);
- `create_directories(p)` создает каталог по пути `p` (создает вложенные каталоги);
- `remove(p)` удаляет файл или пустой каталог по пути `p`;
- `remove_all(p)` рекурсивно удаляет файл или каталог по пути `p`;
- `rename(p1, p2)` переименовывает (перемещает) объект файловой системы по пути `p1` в `p2`;
- `resize_file(p,new_size)` изменяет размер файла по пути `p` на `new_size` (размер указывается в байтах). Если файл увеличивается, свободное пространство заполняется нулями. В противном случае файл обрезается с конца.

Пример. Некоторые операции с файлами и каталогами

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

int main()
{
    using namespace std;

    filesystem::path p1( R"(C:\Temp\tmp.txt)" );
    filesystem::path p2( R"(C:\lib\dll)" );
```

```
    filesystem::path p3( R"(C:\lib\dll\tmp.txt)" );
    filesystem::path p4( R"(C:\lib\dll\tmp.dll)" );

    filesystem::create_directories( p2 );
    filesystem::copy( p1, p2 );
    filesystem::rename( p3, p4 );
    filesystem::remove( p4 );

    return 0;
}
```

В данном примере файл `tmp.txt` копируется в созданный каталог по пути `p2`, переименовывается в `tmp.dll` и затем удаляется (только сам файл, созданные каталоги `C:\lib` и `C:\lib\dll` остаются без изменений).

1.2.8 Итераторы каталогов

Пространство имен `std::filesystem` предоставляет два класса для итерирования по содержимому каталога: `std::directory_iterator` и `std::recursive_directory_iterator`. Разница между ними заключается в функционале: `std::recursive_directory_iterator` позволяет итерироваться в том числе и по содержимому вложенных каталогов (сами вложенные каталоги перечисляются как отдельные элементы). Поэтому далее будет рассматриваться только класс `std::directory_iterator`.

Для создания объекта класса `std::directory_iterator` существует конструктор, принимающий на вход объект класса `std::filesystem::path` по константной ссылке (указывает на каталог, по содержимому которого необходимо проитерироваться). Также, при помощи необязательных параметров имеется возможность предоставить опцию `std::filesystem::directory_options` для настройки поведения и `std::error_code` для сохранения ошибки вместо генерирования исключения.

1.2.9 Содержимое каталогов

В процессе итерирования по содержимому каталога при помощи классов `std::directory_iterator` и `std::recursive_directory_iterator` для каждого вложенного объекта файловой системы создается объект класса `std::filesystem::directory_entry`, который хранит путь к текущему объекту файловой системы вместе с некоторыми атрибутами этого пути. Доступ к атрибутам осуществляется при помощи соответствующих методов:

- `path()` возвращает путь к текущему объекту файловой системы;
- `exists()` проверяет существование объекта файловой системы;
- `is_directory()` проверяет, является ли текущий объект файловой системы каталогом;
- `file_size()` возвращает размер текущего файла.

Выше перечислены далеко не все методы класса `std::filesystem::directory_entry`. За дополнительной информацией можно обратиться к официальной документации.

Пример. Вывод размера содержимого каталога (только файлов)

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

int main()
{
    using namespace std;

    filesystem::path p( R"(c:\libs)" );

    for ( const auto& dir_entry :
          filesystem::recursive_directory_iterator( p ) )
    {
        cout << dir_entry.path();

        if ( !filesystem::is_directory( dir_entry ) )
        {
            cout << '\t' << filesystem::file_size( dir_entry );
        }

        cout << endl;
    }

    return 0;
}
```

```
"c:\\libs\\dll"
"c:\\libs\\dll\\asferror.dll"   2560
"c:\\libs\\dll\\asycfilt.dll"   89088
"c:\\libs\\libcurl.lib" 20216
"c:\\libs\\zlib.lib"    354870
```

В данном примере для каждого вложенного в каталог `c:\libs` объекта файловой системы выводится его полный путь и размер в байтах, если данный объект не является директорией.

1.2.10 Совместимость с `fstream`

Файловые потоки `std::ifstream` и `std::ofstream` возможно создавать, используя объекты классов `std::filesystem::path` и `std::filesystem_directory_entry` в дополнение к строковым типам.

Пример. Совместимость `std::ifstream` и `std::filesystem::path`

```
#include <iostream>
#include <fstream>
#include <string>
#include <filesystem>

int main()
{
    using namespace std;

    filesystem::path p( R"(c:\libs)" );

    for ( const auto& dir_entry :
          filesystem::recursive_directory_iterator( p ) )
    {
        cout << dir_entry.path();

        if ( !filesystem::is_directory( dir_entry ) )
```

```
        {
            cout << '\t';

            ifstream in( dir_entry.path(), ios::binary);
            string file_content;
            file_content.resize( filesystem::file_size( dir_entry ) );
            in.read( file_content.data(), file_content.length() );
            in.close();

            cout << hash<string>()( file_content );
        }

        cout << endl;
    }

    return 0;
}
```

```
"c:\\libs\\dll"
"c:\\libs\\dll\\asferror.dll"   7239708782046785653
"c:\\libs\\dll\\asycfilt.dll"   2754258958906944785
"c:\\libs\\libcurl.lib" 17555371235351538164
"c:\\libs\\zlib.lib"    2493138298377219748
```

В данном примере для каждого вложенного в каталог `c:\libs` объекта файловой системы выводится полный путь и, если он не является директорией, хэш-значение от содержимого файла. Для этого содержимое файла считывается в объект класса `std::string` , от которого при помощи `operator()` структуры `std::hash<std::string>` и вычисляется хэш-значение типа `std::size_t`.

ГЛАВА 2. ПРАКТИЧЕСКАЯ ЧАСТЬ

Лабораторная работа №1. Проверка структуры файлов формата JSON

Цель работы

Сформировать умения по использованию библиотеки nlohmann/json для получения информации о структуре файлов формата JSON.

Задание

Разработать консольное приложение, проверяющее соответствие структуры файла формата JSON структуре, заданной в соответствующем варианте задания. Порядок следования пар *ключ:значение* в JSON-объектах не учитывается. Обозначения `{…}` и `[…]` соответствуют JSON-объектам и JSON-массивам с произвольным содержимым. Входные данные приложения: полный путь к объекту файловой системы (файлу формата JSON).

Связанные темы

1. Функции. Объявление и определение функций.
2. Пространства имен.
3. Организация ввода/вывода при помощи потоковых классов.
4. Обработка исключительных ситуаций.
5. Многофайловые проекты.
6. Работа с файловой системой.

Общие рекомендации

1. Вспомните материал по связанным темам.
2. Изучите теоретические сведения, касающиеся основ работы с библиотекой nlohmann/json. Используйте официальную документацию: https://json.nlohmann.me.

Порядок выполнения

1. В глобальной области видимости реализовать функцию со следующим прототипом:

`void CheckArgumentsAmount(int arguments_amount).`

Данная функция принимает на вход количество аргументов командной строки и проверяет его на корректность. В случае некорректного значения необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Invalid command line arguments amount: current – `*`указать текущее количество аргументов`*`, required – `*`указать требуемое!`*».

2. В глобальной области видимости реализовать функцию со следующим прототипом:

`void CheckInputPath (const std::filesystem::path& path_to_filesystem_object).`

Данная функция принимает на вход путь к объекту файловой системы по константной ссылке и проверяет его корректность: существует ли такой объект, является ли он регулярным файлом и имеет ли расширение «.json». В случае некорректности пути необходимо сгенерировать исключение std::invalid_argument, содержащее соответствующее сообщение: «Filesystem object by path *указать путь* is not exists!», либо «Filesystem object by path *указать путь* is not a regular file!», либо «Filesystem object by path *указать путь* has invalid extension!». Использовать:

- функцию std::filesystem::exists;
- функцию std::filesystem::is_regular_file;
- метод has_extension класса std::filesystem::path;
- метод extension класса std::filesystem::path.

3. В глобальной области видимости реализовать функцию со следующим прототипом:

bool IsJsonCorrect(const nlohmann::json& json).

Данная функция принимает на вход объект класса nlohmann::json по константной ссылке и определяет, соответствует ли его структура описанной в соответствующем варианте задания лабораторной работы. Использовать функционал класса nlohmann::json, описанный в теоретических сведениях.

4. При помощи реализованных функций CheckArgumentsAmount и ChekInputPath проверить корректность входных данных приложения.

5. При помощи функционала класса std::ifstream и функции nlohmann::json::parse получить объект класса nlohmann::json, представляющий содержимое входного файла.

6. При помощи функции IsJsonCorrect проверить корректность структуры объекта класса nlohmann::json, полученного ранее. В зависимости от результата вывести на экран соответствующее сообщение: «Structure of JSON-file by path *указать путь к файлу* is correct.» либо «Structure of JSON-file by path *указать путь к файлу* is incorrect.».

Требования

1. Объявления и реализации функций должны находиться в разных файлах.

2. Исключения, генерируемые реализованными функциями, а также функциями и методами классов библиотеки nlohmann/json, должны быть обработаны.

Варианты заданий

Вариант 1.

```
{
   "object" :
   {
       "string" : значение типа строка,
       "number" : значение типа число,
       "inner_array" : []
```

```
    },
    "array" :
    [
        false,
        null,
        {...}
    ]
}
```

Вариант 2.

```
[
    [
        значение типа число,
        false,
        {}
    ],
    {
        "array" : [...],
        "number" : 10,
        "empty" : null
    }
]
```

Вариант 3.

```
{
    "string" : значение типа строка,
    "empty" : null,
    "array" :
    [
        false,
        2023,
        {}
    ],
    "number" : значение типа число,
    "object" : null
}
```

Вариант 4.

```
{
    "string_0" : "some string",
    "number" : 10,
    "object" :
    {
        "boolean" : true,
        "array" : []
    },
    "string_1" : null,
    "array" : null
}
```

Вариант 5.

```
[
    {
        "string" : значение типа строка,
        "array" : [...],
        "empty" : null
    },
    null,
    "2023",
    true
```

```
]
```

Вариант 6.

```
[
    null,
    [
        false,
        2023,
        {...}
    ],
    значение типа строка,
    false,
    null
]
```

Лабораторная работа №2. Изменение структуры файлов формата JSON

Цель работы

Сформировать умения по использованию библиотеки nlohmann/json для изменения структуры файлов формата JSON.

Задание

Разработать консольное приложение, формирующее из одного входного файла в формате JSON несколько аналогичных файлов специальным образом. Входной файл представляет собой json-объект, каждому ключу которого соответствует также json-объект. Для каждой пары *ключ:json-объект* необходимо создать файл с именем *ключ*. В данный файл необходимо записать соответствующий *ключу json-объект*, содержимое которого должно быть предварительно отредактировано в соответствии с вариантом задания. Входные данные приложения: полный путь к объекту файловой системы (входному файлу в формате JSON).

Связанные темы

1. Функции. Объявление и определение функций.
2. Пространства имен.
3. Организация ввода/вывода при помощи потоковых классов.
4. Обработка исключительных ситуаций.
5. Многофайловые проекты.
6. Работа с файловой системой.

Общие рекомендации

1. Вспомните материал по связанным темам.

2. Изучите теоретические сведения, касающиеся основ работы с библиотекой nlohmann/json. Используйте официальную документацию: https://json.nlohmann.me.

Порядок выполнения

1. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void CheckArgumentsAmount(int arguments_amount).
```

Данная функция принимает на вход количество аргументов командной строки и проверяет его на корректность. В случае некорректного значения необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Invalid command line arguments amount: current – указать текущее количество аргументов, required – указать требуемое!`».

2. В глобальной области видимости реализовать функцию со следующим прототипом:

`void CheckInputPath (const std::filesystem::path& path_to_filesystem_object).`

Данная функция принимает на вход путь к объекту файловой системы по константной ссылке и проверяет его корректность: существует ли такой объект, является ли он регулярным файлом и имеет ли расширение «`.json`». В случае некорректности пути необходимо сгенерировать исключение `std::invalid_argument`, содержащее соответствующее сообщение: «`Filesystem object by path указать путь is not exists!`», либо «`Filesystem object by path указать путь is not a regular file!`», либо «`Filesystem object by path указать путь has invalid extension!`». Использовать:

- функцию std::filesystem::exists;
- функцию std::filesystem::is_regular_file;
- метод has_extension класса std::filesystem::path;
- метод extension класса std::filesystem::path.

3. В глобальной области видимости реализовать функцию следующим прототипом:

`void ModifyJsonObject(nlohmann::json& json_object).`

Данная функция принимает на вход объект класса `nlohmann::json` по ссылке и редактирует его содержимое в соответствии с вариантом задания. Использовать функционал класса `nlohmann::json`, описанный в теоретических сведениях.

4. При помощи реализованных функций `CheckArgumentsAmount` и `ChekInputPath` проверить корректность входных данных приложения.

5. При помощи функционала класса `std::ifstream` и функции `nlohmann::json::parse` получить объект класса `nlohmann::json`, представляющий содержимое входного файла.

6. Для каждой пары *ключ:json-объект*, содержащейся в объекте класса `nlohmann::json`, полученном выше, выполнить следующие действия:

- отредактировать *json-объект* при помощи функции `ModifyJsonObject`;
- создать файл с именем *ключ* в каталоге, в котором находится входной файл в формате JSON;
- записать отредактированный *json-объект* в созданный выше файл.

Использовать:

– функционал класса `nlohmann::json`, описанный в теоретических сведениях;
– метод parent_path класса std::filesystem::path;
– функционал класса `std::ofstream`.

Требования

1. Объявления и реализации функций должны находиться в разных файлах.

2. Исключения, генерируемые реализованными функциями, а также функциями и методами классов библиотеки nlohmann/json, должны быть обработаны.

Варианты заданий

Вариант 1. Удалить все пары *ключ:значение*, где значению соответствует литерал `null`.

Вариант 2. Удалить все пары *ключ:значение*, где значению соответствует логический литерал (`true` либо `false`).

Вариант 3. Удалить все пары *ключ:значение*, где значением является четное число.

Вариант 4. Удалить все пары *ключ:значение*, где значением является нечетное число.

Вариант 5. Удалить все пары *ключ:значение*, где значению соответствует тип «строка».

Вариант 6. Удалить все пары *ключ:значение*, где значению соответствует тип «объект».

Вариант 7. Удалить все пары *ключ:значение*, где значению соответствует тип «массив».

Вариант 8.*Удалить все пары *ключ:значение*, где значением является массив, содержащий не менее 2 (двух) json-объектов.

Вариант 9*.Удалить все пары *ключ:значение*, где значением является json-объект, у которого хотя бы 3 (трем) ключам соответствует тип «объект».

Лабораторная работа №3. Сериализация даты и времени в формат JSON

Цель работы

Сформировать умения по использованию библиотеки nlohmann/json и стандартной библиотеки языка программирования C++ для представления даты и времени в формате JSON.

Задание

Разработать консольное приложение, модифицирующее текущие значения даты и времени в соответствии с вариантом задания. Результат необходимо представить в формате JSON, вывести его на экран, а также записать в соответствующий файл. Входные данные приложения: полный путь к объекту

файловой системы (файлу, в котором будут записаны значения даты и времени в формате JSON).

Связанные темы

1. Функции. Объявление и определение функций.
2. Пространства имен.
3. Организация ввода/вывода при помощи потоковых классов.
4. Обработка исключительных ситуаций.
5. Многофайловые проекты.
6. Работа с файловой системой.

Общие рекомендации

1. Вспомните материал по связанным темам.

2. Изучите теоретические сведения, касающиеся основ работы с библиотекой nlohmann/json. Используйте официальную документацию: https://json.nlohmann.me.

Порядок выполнения

1. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void CheckArgumentsAmount(int arguments_amount).
```

Данная функция принимает на вход количество аргументов командной строки и проверяет его на корректность. В случае некорректного значения необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Invalid command line arguments amount: current – указать текущее количество аргументов, required – указать требуемое!`».

2. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void CheckInputPath
(const std::filesystem::path& path_to_filesystem_object).
```

Данная функция принимает на вход путь к объекту файловой системы по константной ссылке и проверяет его корректность: имеет ли он расширение «`.json`». В случае некорректности пути необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Filesystem object by path указать путь has invalid extension!`». Использовать:

- метод has_extension класса std::filesystem::path;
- метод extension класса std::filesystem::path.

3. В глобальной области видимости реализовать функцию следующим прототипом:

```
tm GetRequiredDateTime().
```

Данная функция возвращает модифицированные в соответствии с вариантом задания значения текущих даты и времени в качестве структуры `tm`. Использовать функции `time` и `localtime_s`.

4. В глобальной области видимости реализовать функцию со следующим прототипом:

```
nlohmann::json TmToJson(tm date_time).
```

Данная функция принимает на вход объект структуры `tm` по значению и возвращает значения её полей, сериализованных в формат JSON. Результатом является объект класса `nlohmann::json`, который соответствует типу «объект». Ключам и значениям ставится в соответствие имена и значения полей структуры `tm`. Использовать функционал класса `nlohmann::json`, описанный в теоретических сведениях.

5. При помощи реализованных функций `CheckArgumentsAmount` и `ChekInputPath` проверить корректность входных данных приложения.

6. При помощи функций `GetRequiredDateTime` и `TmToJson` получите требуемые значения даты и времени в качестве объекта класса `nlohmann::json`.

7. Выведите содержимое полученного выше экземпляра класса `nlohmann::json` на экран, а также запишите его в файл по указанному во входном параметре пути. Если каталога, в котором должен располагаться файл, не существует, воспользоваться функцией `create_directories` пространства имен `std::filesystem`.

Требования

1. Объявления и реализации функций должны находиться в разных файлах.

2. Исключения, генерируемые реализованными функциями, а также функциями и методами классов библиотеки nlohmann/json, должны быть обработаны.

Варианты заданий

Вариант 1. Получить значения даты и времени, больше текущих на 365 суток.

Вариант 2. Получить значения даты и времени, меньше текущих на 87 часов.

Вариант 3. Получить значения даты и времени, соответствующих началу текущего года (полночь 1 января).

Вариант 4. Получить значения даты и времени, соответствующих концу прошлого года (23:59:59 31 декабря).

Вариант 5. Получить значения даты и времени, на сутки меньше текущих.

Лабораторная работа №4. Сериализация информации об объектах файловой системы в формат JSON

Цель работы

Сформировать умения по использованию библиотеки nlohmann/json и стандартной библиотеки языка программирования C++ для получения информации об объектах файловой системы и ее представления в формате JSON.

Задание

Разработать консольное приложение, выводящее на экран информацию об объекте файловой системы в формате JSON, а также записывающее эту

информацию в файл. Входные данные приложения: полный путь к объекту файловой системы.

Связанные темы

1. Функции. Объявление и определение функций.
2. Пространства имен.
3. Организация ввода/вывода при помощи потоковых классов.
4. Обработка исключительных ситуаций.
5. Многофайловые проекты.
6. Работа с файловой системой.

Общие рекомендации

1. Вспомните материал по связанным темам.

2. Изучите теоретические сведения, касающиеся основ работы с библиотекой nlohmann/json. Используйте официальную документацию: https://json.nlohmann.me.

Порядок выполнения

1. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void CheckArgumentsAmount(int arguments_amount).
```

Данная функция принимает на вход количество аргументов командной строки и проверяет его на корректность. В случае некорректного значения необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Invalid command line arguments amount: current – указать текущее количество аргументов, required – указать требуемое!`».

2. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void CheckInputPath
(const std::filesystem::path& path_to_filesystem_object).
```

Данная функция принимает на вход путь к объекту файловой системы по константной ссылке и проверяет его корректность: существует ли такой объект и является ли он каталогом либо регулярным файлом. В случае некорректности пути необходимо сгенерировать исключение `std::invalid_argument`, содержащее соответствующее сообщение: «`Filesystem object by path указать путь is not exists!`», либо «`Filesystem object by path указать путь has invalid type!`». Использовать:

- функцию std::filesystem::exists;
- функцию std::filesystem::is_regular_file;
- функцию std::filesystem::is_directory.

3. В глобальной области видимости реализовать функцию со следующим прототипом:

```
std::size_t Size
(const std::filesystem::path& path_to_filesystem_object).
```

Данная функция принимает на вход путь к объекту файловой системы по константной ссылке и возвращает его размер в байтах. Использовать:

– функцию std::filesystem::is_directory;
– функцию std::filesystem::file_size;
– класс std::filesystem::recursive_directory_iterator.

4. В глобальной области видимости реализовать функцию со следующим прототипом:

```
nlohmann::json GetRegularFileInfo
(const std::filesystem::path& path_to_file).
```

Данная функция принимает на вход путь к файлу по константной ссылке и возвращает информацию о данном файле в качестве объекта класса `nlohmann::json`. Информация о файле должна быть представлена в следующем виде:

```
{
    "type" : "reguar_file",
    "full_name" : имя файла с расширением (строка),
    "name_without_extension" : имя файла без расширения (строка),
    "extension" : расширение (строка либо null),
    "size" : размер файла в байтах (число)
}
```

Использовать:

– функционал класса `nlohmann::json`, описанный в теоретических сведениях;
– метод filename класса std::filesystem::path;
– метод has_extension класса std::filesystem::path;
– метод extension класса std::filesysytem::path;
– метод stem класса std::filesystem::path;
– функцию `Size`.

5. В глобальной области видимости реализовать функцию со следующим прототипом:

```
nlohmann::json GetDirectoryInfo
(const std::filesystem::path& path_to_directory).
```

Данная функция принимает на вход путь к каталогу по константной ссылке и возвращает информацию об этом каталоге в качестве объекта класса `nlohmann::json`. Информация о каталоге должна быть представлена в следующем виде:

```
{
    "type" : "directory",
    "name" : имя каталога (строка),
    "size" : размер каталога в байтах (число),
    "files_amount" : количество файлов на первом уровне вложенности (число),
    "directories_amount" : количество каталогов на первом уровне вложенности
(число)
}
```

Использовать:

– функционал класса `nlohmann::json`, описанный в теоретических сведениях;
– метод stem класса std::filesystem::path;
– функцию `Size`;
– функционал класса std::filesystem::directory_iterator;

– функцию std::filesystem::is_directory;
– функцию std::filesystem::is_regular_file.

6. В глобальной области видимости реализовать функцию следующим прототипом:

```
nlohmann::json GetFsObjectInfo
(const std::filesystem::path& path_to_filesystem_object).
```

Данная функция принимает на вход путь к объекту файловой системы (регулярному файлу или каталогу) по константной ссылке и возвращает информацию об этом объекте в качестве объекта класса `nlohmann::json`. Использовать функции:

– std::filesystem::is_directory;
– std::filesystem::is_regular_file;
– GetRegularFileInfo;
– GetDirectoryInfo.

7. При помощи реализованных функций `CheckArgumentsAmount` и `ChekInputPath` проверить корректность входных данных приложения.

8. При помощи функции `GetFsObjectInfo` получить объект класса `nlohmann::json`, содержащий информацию о входном объекте файловой системы. Вывести полученную информацию на экран, а также записать в файл с именем «`fs_object_info.json`» в каталог, в котором находится входной объект файловой системы. Использовать:

– функционал класса `nlohmann::json`, описанный в теоретических сведениях;
– метод parent_path класса std::filesystem::path;
– функционал класса `std::ofstream`.

Требования

1. Объявления и реализации функций должны находиться в разных файлах.

2. Исключения, генерируемые реализованными функциями, а также функциями и методами классов библиотеки nlohmann/json, должны быть обработаны.

Лабораторная работа №5. Извлечение информации об объектах файловой системы

Цель работы

Сформировать умения по использованию стандартной библиотеки языка программирования C++ для получения информации об объектах файловой системы.

Задание

Разработать консольное приложение, выводящее на экран информацию о каталоге как об объекте файловой системы (обобщенную информацию о каталоге, обобщенную информацию о каждом вложенном в каталог объекте

файловой системы). Входные данные приложения: полный путь к объекту файловой системы.

Связанные темы

1. Функции. Объявление и определение функций.
2. Пространства имен.
3. Основы объектно-ориентированного программирования.
4. Перегрузка операторов.
5. Класс `std::string` и его методы.
6. Форматированный вывод.
7. Обработка исключительных ситуаций.
8. Многофайловые проекты.

Общие рекомендации

1. Вспомните материал по связанным темам.

2. Изучите теоретические сведения, касающиеся возможностей стандартной библиотеки языка программирования C++ для работы с файловой системой. Используйте официальную документацию: https://en.cppreference.com/w/cpp/filesystem.

Порядок выполнения

1. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void CheckArgumentsAmount(int arguments_amount).
```

Данная функция принимает на вход количество аргументов командной строки и проверяет его на корректность. В случае некорректного значения необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Invalid command line arguments amount: current – указать текущее количество аргументов, required – указать требуемое!`».

2. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void ChekInputPath
(const std::filesystem::path& path_to_filesystem_object).
```

Данная функция принимает на вход путь к объекту файловой системы по константной ссылке и проверяет его корректность: существует ли такой объект и является ли он каталогом. В случае некорректности пути необходимо сгенерировать исключение `std::invalid_argument`, содержащее соответствующее сообщение: «`Filesystem object by path указать путь is not exists!`» либо «`Filesystem object by path указать путь is not a directory!`». Использовать следующие функции:

- std::filesystem::exists;
- std::filesystem::is_directory.

3. В пространстве имен `filesystem_object` реализовать функцию со следующим прототипом:

```
std::size_t Size
(const std::filesystem::path& path_to_filesystem_object).
```

Данная функция принимает на вход путь к объекту файловой системы по константной ссылке и возвращает его размер в байтах. Использовать:

- функцию std::filesystem::is_directory;
- функцию std::filesystem::file_size;
- класс std::filesystem::recursive_directory_iterator.

4. В пространстве имен `filesystem_object` объявить структуру `Info`, содержащую обобщенную информацию об объекте файловой системы в следующих полях:

- `name` типа `std::string`, содержащее имя объекта файловой системы (без расширения);
- `type` типа `std::string` с тремя возможными значениями: `directory` (объект файловой системы является каталогом), `file without extension` (объект файловой системы является файлом без расширения) или непосредственно *`расширение файла;`*
- `size` типа `std::size_t`, содержащее размер объекта файловой системы в байтах.

5. Для структуры `filesystem_object::Info` реализовать оператор вывода. Значения всех полей вывести в одну строку. Ширину вывода значения поля `path_to_directory` сделать равной 50 символам, ширину вывода значений остальных полей – 20 символам. Символом заполнения сделать пробельный символ. Обеспечить выравнивание отображаемых имен и значений по левому краю. Использовать следующие манипуляторы потоков ввода-вывода:

- std::setw
- std::setfill;
- std::left.

6. В пространстве имен `filesystem_object` реализовать функцию со следующим прототипом:

```
Info GetInfo
(const std::filesystem::path& path_to_filesystem_object).
```

Данная функция принимает на вход путь к объекту файловой системы по константной ссылке и возвращает экземпляр структуры `filesystem_object::Info` с соответствующей информацией об указанном объекте файловой системы. Использовать следующие методы класса `std::filesystem::path`:

- has_extension;
- extension;
- stem;
- string.

7. В пространстве имен `directory_content` объявить структуру `Info`, содержащую обобщенную информацию о содержимом каталога в следующих полях:

- path_to_directory типа std::filesystem::path, содержащее путь к каталогу;

– `size` типа `std::size_t`, содержащее размер каталога в байтах;
– `files_amount` типа `uint32_t`, содержащее количество файлов в каталоге на первом уровне вложенности;
– `directories_amount` типа `uint32_t`, содержащее количество вложенных каталогов на первом уровне вложенности.

8. Для структуры `directory_content::Info` реализовать оператор вывода. Для каждого поля в отдельной строке вывести имя этого поля и его значение. Ширину вывода имени поля сделать равной 50 символам, ширину вывода значения – 20 символам. Символом заполнения сделать пробельный символ. Обеспечить выравнивание всех значений по левому краю. Использовать следующие манипуляторы потоков ввода-вывода:
– std::setw;
– std::setfill;
– std::left.

9. В пространстве имен `directory_content` реализовать функцию со следующим прототипом:

```
Info GetInfo(const std::filesystem::path&
path_to_directory).
```

Данная функция принимает на вход путь к каталогу и возвращает экземпляр структуры `directory_content::Info` с соответствующей информацией о содержимом указанного каталога. Использовать:
– функцию filesystem_object::Size;
– функцию std::filesystem::is_directory;
– класс std::filesystem::directory_iterator.

10. При помощи реализованных функций `CheckArgumentsAmount` и `ChekInputPath` проверить корректность входных данных приложения.

11. Получить обобщенную информацию о содержимом каталога: при помощи функции `directory_content::GetInfo` получить экземпляр структуры `directory_content::Info` и вывести его содержимое на экран.

12. Получить обобщенную информацию о каждом вложенном в каталог объекте файловой системы: для каждого вложенного объекта при помощи функции `filesystem_object::GetInfo` получить экземпляр структуры `filesystem_object::Info` и вывести его содержимое на экран в отдельной строке. Вывод предыдущего пункта отделить двумя пустыми строками.

Требования

1. Объявления и реализации функций должны находиться в разных файлах.

2. Исключения, генерируемые реализованными функциями, должны быть обработаны.

Лабораторная работа №6. Копирование файлов. Сравнение файлов по содержимому

Цель работы

Сформировать умения по использованию стандартной библиотеки языка программирования C++ для управления объектами файловой системы: копирования файлов, выявления дубликатов файлов по содержимому.

Задание

Разработать консольное приложение, копирующее файлы из одного каталога в другой. Копирование файла должно происходить при условии отсутствия в каталоге назначения файла с таким же содержимым. При совпадении имени копируемого файла с именем некоторого файла в каталоге назначения последний должен быть перезаписан. Входные данные приложения: два полных пути к объектам файловой системы:

- каталог, из которого будут копироваться файлы (исходный каталог);
- каталог, в который будут копироваться файлы (каталог назначения).

Связанные темы

1. Функции. Объявление и определение функций.
2. Бинарные файлы.
3. Организация ввода/вывода при помощи потоковых классов.
4. Класс `std::string` и его методы.
5. Класс `std::set` и его методы.
6. Обработка исключительных ситуаций.
7. Многофайловые проекты.

Общие рекомендации

1. Вспомните материал по связанным темам.
2. Изучите теоретические сведения, касающиеся возможностей стандартной библиотеки языка программирования C++ для работы с файловой системой. Используйте официальную документацию: https://en.cppreference.com/w/cpp/filesystem.

Порядок выполнения

1. В глобальной области видимости реализовать функцию со следующим прототипом:

`void CheckArgumentsAmount(int arguments_amount).`

Данная функция принимает на вход количество аргументов командной строки
и проверяет его на корректность. В случае некорректного значения необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Invalid command line arguments amount: current – `*`указать текущее количество аргументов`*`, required – `*`указать требуемое!`*».

2. В глобальной области видимости реализуйте функцию со следующим прототипом:

```
void CheckDirectoryPath (const std::filesystem::path& path_to_directory)
```

Данная функция принимает на вход путь к каталогу по константной ссылке и проверяет его на корректность: существование объекта файловой системы по указанному пути и факт того, что указанный объект файловой системы является каталогом. В случае некорректного значения сгенерировать исключение `std::runtime_error`, содержащее соответствующее сообщение: «`Filesystem object by path` *`указать путь`* `is not exists!`» либо «`Filesystem object by path` *`указать путь`* `is not a directory!`». Использовать следующие функции:

- std::filesystem::exists;
- std::filesystem::is_directory.

3. В глобальной области видимости реализовать функцию со следующим прототипом:

```
std::string ReadFileContent (const std::filesystem::path& path_to_file).
```

Данная функция принимает на вход путь к файлу по константной ссылке и возвращает его содержимое в качестве объекта класса `std::string`. В случае, когда объекта файловой системы по указанному пути не существует, необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Filesystem object by path` *`указать путь`* `is not exists!`». В случае ошибки при открытии файла необходимо сгенерировать исключение `std::runtime_error`, содержащее следующее сообщение: «`File by path` *`указать путь`* `hasn't been opened!`». Использовать:

- метод is_open класса std::ifstream;
- метод read класса std::ifstream;
- метод `data` класса `std::string`;
- метод length класса std::string.

4. В глобальной области видимости реализовать функцию со следующим прототипом:

```
std::set<std::string> GetFilesContentFromDirectory (const std::filesystem::path& path_to_directory).
```

Данная функция принимает на вход путь к каталогу по константной ссылке и возвращает множество, состоящее из объектов класса `std::string`. Каждый такой объект соответствует содержимому некоторого файла из указанного каталога. Использовать:

- функцию ReadFileContent;
- метод insert класса std::set<std::string>.

5. При помощи реализованных функций `CheckArgumentsAmount` и `CheckDirectoryPath` проверить корректность входных данных приложения.

6. При помощи функции `GetFilesContentFromDirectory` получить множество содержимых файлов из каталога назначения.

7. Для каждого файла из исходного каталога получить его содержимое при помощи функции `ReadFileContent`. Вставить полученное содержимое во множество, полученное в предыдущем пункте. Вставка должна происходить при условии отсутствия во множестве такого же элемента. В случае успешной вставки скопировать файл в каталог назначения и вывести соответствующее сообщение на экран в отдельной строке: «`File by path` *`указать путь к файлу`* `has been copied to directory by path` *`указать путь к каталогу назначения!`*». Использовать:

- функцию std::filesystem::copy_file;
- класс std::filesystem::directory_iterator.

Требования

1. Объявления и реализации функций должны находиться в разных файлах.

2. Исключения, генерируемые реализованными функциями, должны быть обработаны.

Лабораторная работа №7 Копирование файлов. Сравнение файлов по содержимому

Цель работы

Сформировать умения по использованию стандартной библиотеки языка программирования C++ для управления объектами файловой системы: копирования файлов, выявления дубликатов файлов по содержимому.

Задание

Разработать консольное приложение, копирующее файлы из одного каталога в другой. Копирование файла должно происходить при условии отсутствия в каталоге назначения файла с таким же содержимым. При совпадении имени копируемого файла с именем некоторого файла в каталоге назначения последний должен быть перезаписан. Приложение принимает на вход два полных пути к объектам файловой системы:

- каталог, из которого будут копироваться файлы (исходный каталог);
- каталог, в который будут копироваться файлы (каталог назначения).

Связанные темы

1. Функции. Объявление и определение функций.
2. Основы объектно-ориентированного программирования.
3. Бинарные файлы.
4. Организация ввода/вывода при помощи потоковых классов.
5. Класс `std::string` и его методы.
6. Класс `std::set` и его методы.
7. Обработка исключительных ситуаций.
8. Многофайловые проекты.

Общие рекомендации

1. Вспомните материал по связанным темам.
2. Изучите теоретические сведения, касающиеся возможностей стандартной библиотеки языка программирования C++ для работы с файловой системой. Используйте официальную документацию: https://en.cppreference.com/w/cpp/filesystem.

Порядок выполнения

1. В глобальной области видимости реализовать функцию со следующим прототипом:

`void CheckArgumentsAmount(int arguments_amount).`

Данная функция принимает на вход количество аргументов командной строки
и проверяет его на корректность. В случае некорректного значения необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Invalid command line arguments amount: current – `*`указать текущее количество аргументов`*`, required – `*`указать требуемое!`*`»`.

2. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void CheckDirectoryPath
(const std::filesystem::path& path_to_directory)
```

Данная функция принимает на вход путь к каталогу по константной ссылке и проверяет его на корректность: существование объекта файловой системы по указанному пути и факт того, что указанный объект файловой системы является каталогом. В случае некорректного значения сгенерировать исключение `std::invalid_argument`, содержащее соответствующее сообщение: «`Filesystem object by path `*`указать путь`*` is not exists!`» либо «`Filesystem object by path `*`указать путь`*` is not a directory!`». Использовать следующие функции:

- std::filesystem::exists;
- std::filesystem::is_directory.

3. В глобальной области видимости реализовать функцию со следующим прототипом:

```
std::string ReadFileContent
(const std::filesystem::path& path_to_file).
```

Данная функция принимает на вход путь к файлу по константной ссылке и возвращает его содержимое в качестве объекта класса `std::string`. В случае, когда объекта файловой системы по указанному пути не существует, необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Filesystem object by path `*`указать путь`*` is not exists!`». В случае ошибки при открытии файла необходимо сгенерировать исключение `std::runtime_error`, содержащее следующее сообщение: «`File by path `*`указать путь`*` hasn't been opened!`». Использовать:

– метод is_open класса std::ifstream;
– метод read класса std::ifstream;
– метод `data` класса `std::string`;
– метод length класса std::string.

4. В глобальной области видимости объявить класс `FilesStorage` со следующими полями:
– std::filesystem::path path_to_directory_;
– std::set<std::string> files_content_storage_.

5. В классе `FilesStorage` реализовать следующий конструктор:
`FilesStorage(const std::filesystem::path& path_to_directory).`

Данный конструктор инициализирует поле `path_to_directory_` значением `path_to_directoty`.

6. В классе `FilesStorage` реализовать следующий метод:
`void InitStorage().`

Данный метод должен добавить во множество `files_content_storage_` содержимое каждого файла, вложенного в каталог по пути `path_to_directory_`. Использовать:
– функцию ReadFileContent;
– метод insert класса std::set<std::string>;
– класс std::filesystem::directory_iterator.

7. В классе `FilesStorage` реализовать следующий метод:
`void CopyFile(const std::filesystem::path& path_to_file).`

Данный метод добавляет содержимое файла по пути `path_to_file` во множество `files_contents_storage_` при условии, что данное множество его не содержит. В случае успешного добавления, необходимо скопировать файл в каталог по пути `path_to_directory_` и вывести соответствующе сообщение на экран: «`File by path` *`указать путь к файлу`* `has been copied to directory by path` *`указать путь к каталогу!`*».

8. При помощи реализованных функций `CheckArgumentsAmount` и `ChekDirectoryPath` проверить корректность входных данных приложения.

9. Создать экземпляр класса `FilesStorage` при помощи конструктора, описанного выше. В качестве аргумента использовать путь к каталогу назначения.

10. У созданного экземпляра класса `FilesStorage` вызвать метод `InitStorage`.

11. Для каждого файла из исходного каталога у созданного экземпляра класса `FilesStorage` вызвать метод `CopyFile`. Использовать класс `std::filesystem::directory_iterator`.

Требования

1. Объявления и реализации функций и методов классов должны находиться в разных файлах.

2. Исключения, генерируемые реализованными функциями и методами классов, должны быть обработаны.

Лабораторная работа №8. Удаление файлов. Механизм контрольных сумм

Цель работы

Сформировать умения по использованию стандартной библиотеки языка программирования C++ для управления объектами файловой системы: удаления файлов, вычисления контрольных сумм (хэш-значений) от содержимого.

Задание

Разработать консольное приложение, удаляющие дубликаты файлов по содержимому из каталога. Входные данные приложения: полный путь к объекту файловой системы (каталог, из которого необходимо удалить дубликаты).

Замечание: в данной работе файлы считаются равными (отличающимися) по содержимому в случае равенства (неравенства) хэш-значений (контрольных сумм) от их содержимых. Т.е., вероятность возникновения коллизии при вычислении хэш-значений считается равной нулю.

Связанные темы

1. Функции. Объявление и определение функций.
2. Бинарные файлы.
3. Организация ввода/вывода при помощи потоковых классов.
4. Класс `std::string` и его методы.
5. Класс `std::set` и его методы.
6. Обработка исключительных ситуаций.
7. Многофайловые проекты.

Общие рекомендации

1. Вспомните материал по связанным темам.
2. Изучите теоретические сведения, касающиеся возможностей стандартной библиотеки языка программирования C++ для работы с файловой системой. Используйте официальную документацию: https://en.cppreference.com/w/cpp/filesystem.

Порядок выполнения

1. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void CheckArgumentsAmount(int arguments_amount).
```

Данная функция принимает на вход количество аргументов командной строки
и проверяет его на корректность. В случае некорректного значения необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Invalid command line arguments amount: current – `*`указать текущее количество аргументов`*`, required – `*`указать требуемое!`*».

2. В глобальной области видимости реализуйте функцию со следующим прототипом:

```
void CheckDirectoryPath
(const std::filesystem::path& path_to_directory)
```

Данная функция принимает на вход путь к каталогу по константной ссылке и проверяет его на корректность: существование объекта файловой системы по указанному пути, факт того, что указанный объект файловой системы является каталогом. В случае некорректного значения сгенерировать исключение `std::runtime_error`, содержащее соответствующее сообщение: «`Filesystem object by path` *`указать путь`* `is not exists!`» либо «`Filesystem object by path` *`указать путь`* `is not a directory!`». Использовать следующие функции:

- std::filesystem::exists;
- std::filesystem::is_directory.

3. В глобальной области видимости реализовать функцию со следующим прототипом:

```
std::string ReadFileContent
(const std::filesystem::path& path_to_file).
```

Данная функция принимает на вход путь к файлу по константной ссылке и возвращает его содержимое в качестве объекта класса `std::string`. В случае, когда объекта файловой системы по указанному пути не существует, необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Filesystem object by path` *`указать путь`* `is not exists!`». В случае ошибки при открытии файла необходимо сгенерировать исключение `std::runtime_error`, содержащее следующее сообщение: «`File by path` *`указать путь`* `hasn't been opened!`». Использовать:

- метод is_open класса std::ifstream;
- метод read класса std::ifstream;
- метод `data` класса `std::string`;
- метод length класса std::string.

4. В глобальной области видимости реализовать функцию со следующим прототипом:

```
std::size_t GetFileContentHash
(const std::filesystem::path& path_to_file).
```

Данная функция принимает на вход путь к файлу по константной ссылке и возвращает хэш-значение от содержимого данного файла. Использовать:

- функцию ReadFileContent;
- operator() структуры std::hash<std::string>.

5. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void RemoveDuplicatesFromDirectory
(const std::filesystem::path& path_to_directory).
```

Данная функция принимает на вход путь к каталогу и удаляет из него дубликаты файлов по содержимому. Использовать:

- класс std::filesystem::directory_iterator;
- контейнер std::set<std::size_t>;
- метод insert класса std::set<std::size_t>;
- функцию GetFileContentHash;
- функцию std::filesystem::remove.

В теле данной функции необходимо создать объект класса `std::set<std::size_t>` и последовательно помещать в него хэш-значения от содержимого каждого файла из каталога по пути `path_to_directory`. В случае неуспешной вставки, текущий файл необходимо удалить и вывести на экран в отдельной строке соответствующее сообщение: «`File by path` *`указать путь к файлу`* `has been removed`».

6. При помощи функций `CheckArgumentsAmount` и `CheckDirectoryPath` проверить корректность входных параметров приложения.

7. Вызвать функцию `RemoveDuplicatesFromDirectory`. В качестве параметра передать путь к каталогу, из которого необходимо удалить дубликаты файлов по содержимому.

Требования

1. Объявления и реализации функций должны находиться в разных файлах.
2. Исключения, генерируемые реализованными функциями, должны быть обработаны.

Лабораторная работа №9. Перемещение файлов

Цель работы

Сформировать умения по использованию стандартной библиотеки языка программирования C++ для управления объектами файловой системы: перемещения файлов, извлечения дополнительной информации из имен файлов.

Задание

Разработать консольное приложение, перемещающее файлы из указанного (исходного) каталога по определенному правилу. Входные данные приложения: полный путь к исходному каталогу, содержащему файлы, которые необходимо переместить.

Имена файлов соответствуют следующему формату: *год_месяц_число_имя.расширение*, (например, `2022_08_13_filename.pdf`). Файл с именем *год_месяц_число_имя.расширение* должен быть переименован в *имя.расширение* и перемещен в каталог *число*, содержащийся в каталоге *месяц*, который, в свою очередь, содержится в каталоге *год*. Каталог с именем *год* должен быть расположен в исходном каталоге.

Связанные темы

1. Функции. Объявление и определение функций.
2. Класс `std::string` и его методы.
3. Обработка исключительных ситуаций.
4. Многофайловые проекты.

Общие рекомендации

1. Вспомните материал по связанным темам.

2. Изучите теоретические сведения, касающиеся возможностей стандартной библиотеки языка программирования C++ для работы с файловой системой. Используйте официальную документацию: https://en.cppreference.com/w/cpp/filesystem.

Порядок выполнения

1. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void CheckArgumentsAmount(int arguments_amount).
```

Данная функция принимает на вход количество аргументов командной строки и проверяет его на корректность. В случае некорректного значения необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Invalid command line arguments amount: current – указать текущее количество аргументов, required – указать требуемое!`».

2. В глобальной области видимости реализуйте функцию со следующим прототипом:

```
void CheckDirectoryPath
(const std::filesystem::path& path_to_directory)
```

Данная функция принимает на вход путь к каталогу по константной ссылке и проверяет его на корректность: существование объекта файловой системы по указанному пути и факт того, что указанный объект файловой системы является каталогом. В случае некорректного значения сгенерировать исключение `std::runtime_error`, содержащее соответствующее сообщение: «`Filesystem object by path указать путь is not exists!`» либо «`Filesystem object by path указать путь is not a directory!`». Использовать следующие функции:

- std::filesystem::exists;
- std::filesystem::is_directory.

3. В глобальной области видимости реализовать функцию со следующим прототипом:

```
std::filesystem::path GetPathToMove
(const std::filesystem::path& path_to_file)
```

Входной параметр – полный путь к файлу, имя которого соответствует описанию из задания лабораторной работы. В случае некорректного имени файла необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Invalid filename. File path: указать путь к файлу!`». Выходной параметр – полный путь, указывающий расположение исходного файла после перемещения в соответствии с правилом,

описанным
в задании лабораторной работы. Использовать:

- метод parent_path класса std::filesystem::path;
- метод stem класса std::filesystem::path;
- метод string класса std::filesystem::path;
- метод append класса std::filesystem::path;
- метод `find` класса `std::string`;
- метод substr класса std::string.

4. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void Move(const std::filesystem::path& path_to_file)
```

Данная функция принимает на вход путь к файлу по константной ссылке и выполняет его перемещение по описанному в задании лабораторной работы правилу. После перемещения файла вывести на экран в отдельной строке сообщение следующего вида: «`File by path указать исходный путь к файлу has been moved to указать путь к файлу после перемещения`».

Использовать следующие функции:

- функцию std::filesystem::rename;
- функцию std::filesystem::create_directories;
- метод parent_path класса std::filesystem::path;
- функцию GetPathToMove.

Замечание: перемещать файлы при помощи функции `std::filesystem::rename` можно только в существующие каталоги.

5. При помощи функций `CheckArgumentsAmount` и `CheckDirectoryPath` проверить корректность входных параметров приложения.

6. Вызвать функцию `Move` от полного пути каждого файла из исходного каталога. Использовать:

- класс std::filesystem::directory_iterator;
- метод path класса std::filesystem::directory_entry.

Требования

1. Объявления и реализации функций должны находиться в разных файлах.

2. Исключения, генерируемые реализованными функциями, должны быть обработаны.

Лабораторная работа №10. Перемещение файлов

Цель работы

Сформировать умения по использованию стандартной библиотеки языка программирования C++ для управления объектами файловой системы: перемещения файлов, извлечения дополнительной информации из имен файлов.

Задание

Разработать консольное приложение, перемещающее файлы из указанного (исходного) каталога по определенному правилу. Входные данные приложения: полный путь к исходному каталогу, содержащему файлы, которые необходимо переместить.

Имена файлов соответствуют следующему формату: *год_месяц_число_имя.расширение*, (например, `2022_08_13_filename.pdf`). Файл с именем *год_месяц_число_имя.расширение* должен быть переименован в *имя.расширение* и перемещен в каталог *число*, содержащийся в каталоге *месяц*, который, в свою очередь, содержится в каталоге *год*. Каталог с именем *год* должен быть расположен в исходном каталоге. Файлы, в имени которых *год* и *число* кратны 5, должны быть удалены.

Связанные темы

1. Функции. Объявление и определение функций.
2. Основы объектно-ориентированного программирования.
3. Класс `std::string` и его методы.
4. Обработка исключительных ситуаций.
5. Многофайловые проекты.

Общие рекомендации

1. Вспомните материал по связанным темам.

2. Изучите теоретические сведения, касающиеся возможностей стандартной библиотеки языка программирования C++ для работы с файловой системой. Используйте официальную документацию: https://en.cppreference.com/w/cpp/filesystem.

Порядок выполнения

1. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void CheckArgumentsAmount(int arguments_amount).
```

Данная функция принимает на вход количество аргументов командной строки
и проверяет его на корректность. В случае некорректного значения необходимо сгенерировать исключение `std::invalid_argument`, содержащее следующее сообщение: «`Invalid command line arguments amount: current – `*`указать текущее количество аргументов`*`, required – `*`указать требуемое!`*».

2. В глобальной области видимости реализуйте функцию со следующим прототипом:

```
void CheckDirectoryPath
(const std::filesystem::path& path_to_directory)
```

Данная функция принимает на вход путь к каталогу по константной ссылке и проверяет его на корректность: существование объекта файловой системы по указанному пути и факт того, что указанный объект файловой системы является каталогом. В случае некорректного значения сгенерировать исключение `std::runtime_error`, содержащее соответствующее сообщение:

«Filesystem object by path *указать путь* is not exists!» либо «Filesystem object by path *указать путь* is not a directory!». Использовать следующие функции:

- std::filesystem::exists;
- std::filesystem::is_directory.

3. В глобальной области видимости объявить класс `FileName`. В секции `public` объявить следующие поля:

- std::filesystem::path path_to_current_file – полный путь к соответствующему файлу;
- `uint16_t year` – *год*, извлеченный из имени файла;
- `uint16_t month` – *месяц*, извлеченный из имени файла;
- `uint16_t day` – *день*, извлеченный из имени файла;
- `std::string only_name` – *имя* файла без части *год_месяц_число_*.

4. В классе `FileName` реализовать следующий конструктор:

```
FileName(std::filesystem::path path_to_file);
```

Данный конструктор инициализирует поле `path_to_current_file` значением `path_to_file`.

5. В классе `FileName` в секции `public` реализовать метод

```
void Parse();
```

Данный метод заполняет поля `year`, `month`, `day` и `only_name` значениями, извлеченными из имени файла, содержащемся в поле `path_to_current_file`. Если имя файла не соответствует формату, описанному в задании лабораторной работы, сгенерировать исключение `std::invalid_argument`, содержащее следующие сообщение «Invalid filename: *указать полный путь к файлу*!». Использовать следующие функции и методы классов:

- std::filesystem::path::string;
- std::string::find;
- std::string::substr;
- std::stoul.

6. В классе `FileName` в секции `public` реализовать метод

```
bool IsRemoveRequired() const;
```

Данный метод определяет необходимость удаления файла по пути `path_to_file` (*год* и *число* в имени должны быть кратны 5).

7. В глобальной области видимости реализовать функцию со следующим прототипом:

```
std::filesystem::path GetPathToMove
(const std::filesystem::path& path_to_file)
```

Входной параметр – полный путь к файлу, имя которого соответствует описанию из задания лабораторной работы. Выходной параметр – полный путь, указывающий расположение исходного файла после перемещения в соответствии с правилом, описанным в задании лабораторной работы. Использовать:

- метод parent_path класса std::filesystem::path;

– метод append класса std::filesystem::path;
– класс `FileName`.

8. В глобальной области видимости реализовать функцию со следующим прототипом:

```
void Move(const std::filesystem::path& path_to_file)
```

Данная функция принимает на вход путь к файлу по константной ссылке и выполняет его перемещение по описанному в задании лабораторной работы правилу. После перемещения файла вывести на экран в отдельной строке сообщение следующего вида: «`File by path указать исходный путь к файлу has been moved to указать путь к файлу после перемещения!`».

Использовать:

– функцию std::filesystem::rename;
– функцию std::filesystem::create_directories;
– метод parent_path класса std::filesystem::path;
– функцию GetPathToMove.

Замечание: перемещать файлы при помощи функции `std::filesystem::rename` можно только в существующие каталоги.

9. При помощи функций `CheckArgumentsAmount` и `CheckDirectoryPath` проверить корректность входных параметров приложения.

10. Выполнить перемещение и удаление файлов из исходного каталога по правилу, описанному в задании лабораторной работы. Использовать:

– класс `FileName`;
– функцию `Move`;
– класс std::filesystem::directory_iterator;
– метод path класса std::filesystem::directory_entry.

Требования

1. Объявления и реализации функций должны находиться в разных файлах.

2. Исключения, генерируемые реализованными функциями и методами классов, должны быть обработаны.