Основы программирования. Конспект лекций

Притчин И.С.

Июль 2021 – Февраль 2022

Оглавление

0.1 Лабораторная работа №6а «Потоки. Ссылки»

Цель работы: получение навыком работы с потоками, ссылками, управляющими конструкциями; осознание необходимости появления данных языковых средств.

Содержание отчета:

- Тема лабораторной работы.
- Цель лабораторной работы.
- Тексты задания с набранными фрагментами кода **к каждому из пунктов** и ответами на вопросы по ходу выполнения работы.

Задания к лабораторной работе:

Задания к данной лабораторной работе представлены множеством небольших пунктов, выполнение которых позволит лучше понять аспекты языка. Настоятельно рекомендуется делать отчёт на этапе выполнения лабораторной.

Библиотека `<iostream>` предоставляет множество классов для работы с вводом / выводом:

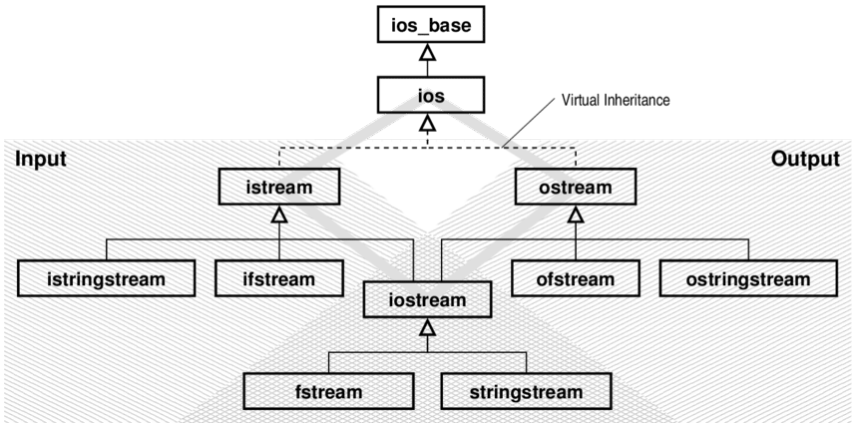


Рис. 1 – Иерархия классов ввода / вывода

Одна из целей лабораторной заключается в том, чтобы на практике поработать с данными средствами языка.

0.1.1 Буферизация в выходных потоках

1. Создайте функцию `infinitivePause`, которая в своём теле будет воспроизводить бесконечный цикл.
2. Отправьте в объект `cout`, связанный со стандартным потоком вывода, произвольное сообщение без использования `endl`. После чего сделайте вызов `infinitivePause`. Запустите программу. Опишите[1] наблюдаемое поведение.
3. Выполните аналогичные пункту 2 действия только с использованием `endl`. Опишите наблюдаемое поведение.

[1] Когда вас просят описать поведение, напишите то, что увидели, в процессе запуска приложения.

4. Для случая 2, после вывода сообщения вставьте инструкцию ввода какой-нибудь переменной. Запустите приложение. Опишите наблюдаемое поведение.

5. Выполните создание строки, состоящей из 10000 символов 'a':

```
std::string s(10000, 'a');
```

   Осуществите её вывод перед вызовом `infinitivePause`. Опишите наблюдаемое поведение.

6. Таким образом, выделите 3 случая, когда очищается буфер вывода.

7. Говорят, что частый сброс буфера способен повлиять на производительность приложения. Проведите эксперимент, доказывающий это. Например, он мог выглядеть так: выполним создание файла, в который будем записывать данные. В одном случае будем сбрасывать буфер, в другом не будем:

```
#include <iostream>
#include <fstream> // для работы с файлами
#include <ctime>

// вывод осуществляется в поток, связанный с логами
#define TIME_TEST(testCode, message) { \
    clock_t start_time = clock () ; \
    testCode \
    clock_t end_time = clock () ;\
    clock_t sort_time = end_time - start_time ; \
    std::clog << message << ": "  \
              << (double) sort_time/CLOCKS_PER_SEC << std::endl; \
}

int main() {
    const char *filename = "tmp.txt";
    std::ofstream file(filename); // выполняем создание файла

    TIME_TEST({
                  for (int i = 0; i < 1000000; i++)
                      file << 'a' << std::endl;
              }, "Often buffer reset");

    TIME_TEST({
                  for (int i = 0; i < 1000000; i++)
                      file << 'a' << '\n';
              }, "Buffer opt");

    file.close();           // закрываем файл
    std::remove(filename); // удаляем временный файл

    return 0;
}
```

   Вставьте код вашего эксперимента, а также выводимые программой значения.

8. Рассмотрим случай, когда частый сброс буфера вывода из-за чередования ввода / вывода оказывает влияние на производительность. Выполните решение задачи Минимальная OR сумма (1635A) на *codeforces* и приложите вердикт тестирующей системы.

9. В условиях задачи 4 добавьте

```
cin.tie(nullptr);
```

Опишите наблюдаемое поведение.

10. В решение вашей задачи для пункта 8 добавьте строки:

```
cin.tie(nullptr);                       // разрывает связь с потоком вывода
ios_base::sync_with_stdio(false);       // устанавливает, синхронизируются ли
                                        // стандартные потоки C++
                                        // со стандартными потоками C
                                        // после каждой операции ввода/вывода.
// последняя команда обязана быть выполнена до первой операции
// ввода / вывода
```

И снова приложите вердикт тестирующей системы.

11. Существуют объекты, связанные с потоками вывода ошибок (`cerr`) и логов (`clog`). Приведите код, который может проверить, является ли вывод в них буферизированным. Как вы считаете, почему было принято именно такое решение по буферизации данных потоков? Ответ обоснуйте.

0.1.2 Некоторые особенности вывода

1. При помощи `cout` можно выводить в поток вывода значения переменных и адреса:

```
int i = 10;
cout << i << ' ' << &i;
```

Создайте переменную `s` типа `const char*` и попробуйте вывести её в поток вывода. Получившееся поведение опишите.

2. Несмотря на то, что переменная типа `const char*` и так является адресом, вы не должны были его увидеть. Чтобы увидеть адрес, необходимо такую переменную надо привести к типу `void*`[2]. Однако одним приведением типа вы не отделаетесь, так как мешает `const`. Цепочка преобразований будет такова:

$$const\,char* \to char* \to void*$$

```
cout << static_cast<void*>(const_cast<char*>(s));
```

3. Выведите литералы `true` и `false` в поток вывода. Какие значения были отображены на экране?

4. Добавьте манипулятор `std::boolalpha`.

```
#include <iostream>

int main() {
    std::cout << std::boolalpha << true << ' ' << false;

    return 0;
}
```

Опишите отображаемый вывод. Проверьте, распространяется ли действие манипулятора на последующие выводы.

5. Чтобы переключиться на вывод логических значений 1 и 0 используйте манипулятор `std::noboolalpha`.

[2] На самом деле, вы могли бы использовать приведение в стиле C, но оно не рекомендуется.

0.1.3 Ввод

1. Создайте переменную целочисленного типа, но при осуществлении ввода введите строку. Опишите наблюдаемое поведение.

2. Переменная, связанная с потоком ввода будет возвращать значение "истина", если поток ввода не находится в состоянии ошибки. Попробуйте запустить пример из прошлого пункта, добавив обработку:

```
int a;
std::cin >> a;

if (std::cin)
    std::cout << "Success";
else
    std::cout << "Error";
```

   Поэксперементируйте с разными вариантами ввода. Приведите пример такого ввода, который выдаёт "Success", но содержит символы, отличные от цифр.

3. Создайте переменную типа `char` и попробуйте выполнить ввод пробельного символа. Опишите наблюдаемое поведение.

4. Проделайте то же самое но при считывании осуществляйте ввод через `cin.get()`:

```
#include <iostream>

int main() {
    char c;
    std::cin.get(c);

    std::cout << c;
}
```

5. Манипуллятор `std::noskipws` заставит выполнить ввод и пробельного символа. Запустите пример:

```
#include <iostream>

int main() {
    char c;
    std::cin >> std::noskipws >> c;

    std::cout << c;

    return 0;
}
```

   Проверьте, распространяется ли действие манипулятора на последующие вводы.

6. Иногда возникает потребность пропустить некоторые символы в потоке ввода. Например, вводится дата в формате "DD.MM.YYYY"(например 19.02.2003). И стоит цель вычленить отдельно день, месяц и год в переменные типа `int`. Это можно сделать так:

```
#include <iostream>

int main() {
```

```
    int day;
    std::cin >> day;
    std::cin.ignore(1);

    int month;
    std::cin >> month;
    std::cin.ignore(1);

    int year;
    std::cin >> year;

    // при вводе 19.02.2003 выдаст 19/02/2003
    std::cout << day << '/' << month << '/' << year;

    return 0;
}
```

Выполните чтение числа до точки и после точки. Например "14.341"должно быть разбито на 14 и 341.

7. Обсудим строковый ввод. Несмотря на возможность осуществлять ввод как строк в стиле C, так и за счёт класса `std::string` в C++, вам следует остановиться на последнем, так как происходит автоматическое управление памятью. Идеальный ввод на C++ строк в стиле C выглядел бы так:

```
#include <iostream>
#include <cstring>

int main() {
    char buf[1000];
    std::cin >> buf;

    char *s = new char[strlen(buf) + 1]; // выделение памяти под строку
    strcpy(s, buf);                      // копирование из буфера
                                         // в строку

    //что-то делаем со строкой, например, выводим
    std::cout << s;

    delete[] s;                          // освобождение памяти

    return 0;
}
```

Внедрение работы с памятью в алгоритм вовсе не улучшает его читаемость. Тот же фрагмент на C++:

```
#include <iostream>

int main() {
    std::string s;
    std::cin >> s;

    std::cout << s;

    return 0;
}
```

Просто откажитесь от соблазна использовать строки в стиле C, и с этого пункта достаточно.

0.1.4 Файловый ввод / вывод

Библиотека *iostream* поддерживает чтение и запись в файлы. Для этого предназначены следующие классы:

- $ifstream$ – связывает ввод программы с файлом
- $ofstream$ – связывает вывод программы с файлом
- $fstream$ – связывает как ввод, так и вывод программы с файлом

1. Очень часто приходится оперировать работой с **файлами** – именованной областью данных на носителе информации. В рамках данной лабораторной разберёмся, как осуществляется работа с текстовыми файлами. Создайте файл `input.txt` содержащий несколько строк, по одному числу в каждой.

2. Убедитесь, что можете использовать оператор `»` для считывания строк из файла:

```
#include <fstream>

int main() {
    // чтение из файла
    std::ifstream inputFile("input.txt");

    std::string s;
    while (inputFile >> s)
        std::cout << s << '\n';

    return 0;
}
```

   Отметьте, что если файл будет считан полностью, в результате выполнения выражения `inputFile » s` будет возвращено значение 'ложь'.

3. Код описанный выше несколько небезопасен. Файл с таким именем мог и не существовать. Попробуйте запустить прошлый фрагмент для файла, который не был создан до этого и опишите наблюдаемое поведение.

4. В прошлом примере для проверки на то, открыт ли файл, мог использоваться функция-член `.is_open()`:

```
std::ifstream inputFile("notExists.txt");

if (inputFile.is_open())
    std::cout << "Exists";
else
    std::cout << "Not exists";
```

   однако и сама переменная, ассоциированная с файлом может быть использована для проверки на то, прошла ли операция открытия успешно:

```
std::ifstream inputFile("notExists.txt");

if (inputFile)
    std::cout << "Exists";
else
    std::cout << "Not exists";
```

5. В примере выше мы осуществляли ввод в строку, но с таким же успехом вы можете считывать данные в переменные, например, целочисленного типа.

```
int x;
while (inputFile >> x) {

}
```

Имея данные сведения, реализуйте функцию `long long getSum(const std::string &filename)` для вычисления суммы чисел, записанных в файл с именем `filename`. Если файла с таким именем нет, функция должна возвращать -1.

6. История с возвратом значения -1, если файл не был найден, смотрится несколько нелепо. В C++ появились другие средства для сигнализирования об ошибках, называемых исключениями. Мы не будем вдаваться в подробности сейчас. Если вы хотите сигнализировать, что возникла некоторая ошибка времени исполнения, можно использовать такой подход:

```
#include <stdexcept>

long long getSum(const std::string &filename) {
    std::ifstream inputFile(filename);

    if (!inputFile)
        throw std::runtime_error("File doesn't exist");

    // работа с файлом

}
```

Модифицируйте программу из пункта 5, запустите её для несуществующего файла. Вставьте в отчёт полученное сообщение.

7. Пусть в файле `inputFile` записаны размеры матрицы, а далее записаны непосредственно элементы матрицы, например:

```
2 4
1 2 3 4
5 3 4 2
```

Функция `long long getSumOfMaxesInRows(const std::string &filename)` должна выполнять поиск суммы максимальных элементов строк. Для примера выше сумма равняется 9. Реализуйте её.

0.1.5 *sstream*

Библиотека *iostream* поддерживает также ввод / вывод в область памяти, при этом поток связывается со строкой в памяти программы. С помощью потоковых операторов ввода / вывода мы можем записывать данные в эту строку и считывать их оттуда. Определены следующие классы:

- *istringstream* - чтение из строки
- *ostringstream* - запись в строку
- *stringstream* - чтение в строку и запись из строки

Например, можно крайне комфортно сконструировать сообщения об ошибках:

```
#include <iostream>
#include <sstream>

using namespace std;

int main() {
    cout << "Input x > 0." << '\n';

    int x;
    cin >> x;

    if (x <= 0) {
        ostringstream errorMessage;
        errorMessage << "Expected x < 0, "
                     << "Got: " << x << '\n';

        throw runtime_error(errorMessage.str());
    } else {
        cout << "Success";
    }

    return 0;
}
```

Решить задачу о выводе слов из двух строк с чередованием можно и так:

```
#include <iostream>
#include <sstream>

using namespace std;

int main() {
    string s1, s2;
    getline(cin, s1);
    getline(cin, s2);

    istringstream ss1(s1);
    istringstream ss2(s2);

    string word;
    // ss1.eof() возвращает значение истина, если буфер пуст
    while (!ss1.eof() || !ss2.eof()) {
        if (!ss1.eof()) {
            ss1 >> word;
            cout << word << ' ';
        }
        if (!ss2.eof()) {
            ss2 >> word;
            cout << word << ' ';
        }
    }


    return 0;
}
```

0.1.6 Определение операций ввода / вывода для произвольных типов

1. Опишите произвольную (но отличную от примера) структуру. Например `Person` с полями `name` и `age`:

```
struct Person {
    std::string name;
    int age;
};
```

2. Реализуйте функции ввода и вывода[3] структуры:

```
void inputPerson(Person &p);
void outputPerson(const Person &p)
```

3. В `main` создайте переменную данного типа и выполните ввод и вывод структуры на экран:

```
int main() {
    Person p;

    inputPerson(p);
    outputPerson(p);

    return 0;
}
```

   Такой подход немного неудобен. Ведь когда мы вводим переменные, нам бы хотелось использовать `cin`, а для вывода – `cout`. Для этого придётся для нашей структуры определить операцию ввода и вывода:

```
void operator>>(std::istream &in, Person &p) {
    in >> p.name >> p.age;
}

void operator<<(std::ostream &out, Person &p) {
    out << p.name << " is " << p.age << " years old";
}
```

   Переменная `in` выше имеет тип `std::istream&`. Описание механизм работы для `std::istream&` позволит осуществлять ввод структуры для произвольного объекта, ассоциированных с вводом: хоть с клавиатуры, хоть с файла.

4. Создайте файл `intput.txt`, и выполните чтение структуры и с клавиатуры, и из файла:

```
Person p;

// ввод с клавиатуры
std::cin >> p;
// вывод в консоль
std::cout << p;

// ввод с файла
std::ifstream inputFile("input.txt");
// вывод в файл
```

[3] Операция вывода никогда не должна выводить символа переноса на новую строку. Оставьте возможность определить данный момент пользователю библиотеки.

```
11 std::ofstream outputFile("output.txt");
12
13 // ввод с файла
14 inputFile >> p;
15 outputFile << p;
```

5. Попробуйте написать цепочку вида:

```
1 Person p1, p2;
2 std::cin >> p1 >> p2;
```

   Опишите наблюдаемое поведение.

6. Чтобы двигаться дальше, необходимо понять, почему наблюдалось такое поведение. Когда вы пишите:

```
1 std::cin >> p;
```

   неявно происходит вызов функции:

```
1 operator>>(std::cin, p);
```

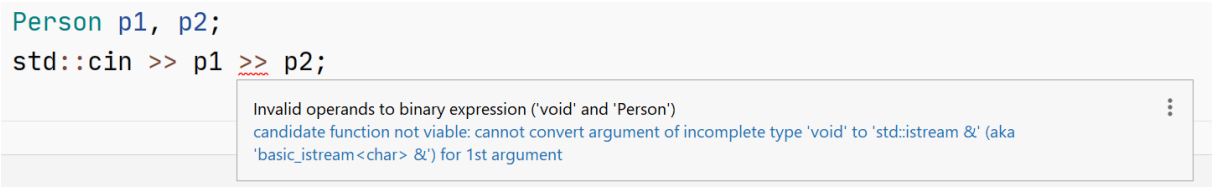
   Для ввода одного значения такой вызов хорошо работает. Но если будет цепочка:

```
1 cin >> p1 >> p2;
```

   Это заменяется на

```
1 operator>>(operator>>(std::cin, p1), p2);
```

   Но в силу того первый вызов вернёт `void` произойдет ошибка компиляции. Внимательно почитайте об ошибке, например, для CLion:



   что даст понимание того, что данное рассуждение верно.[4]

   Решается данная проблема просто, изменением сигнатуры функции:

[4] Читать ошибки, выдаваемые плюсами, довольно тяжело.

```
std::istream& operator>>(std::istream &in, Person &p) {
    in >> p.name >> p.age;
    return in;
}

std::ostream& operator<<(std::ostream &out, Person &p) {
    out << p.name << " is " << p.age << " years old\n";
    return out;
}
```

Выполните такую замену для своей структуры, убедитесь, что всё работает:

```
int main() {
    Person p1, p2;
    std::cin >> p1 >> p2;

    std::cout << p1 << p2;

    return 0;
}
```

0.1.7 Ссылки

Предпосылками к созданию ссылок было определенное неудобство при работе с указателями. Напомню, что в языке программирования C они решали следующие 5 задач:

- как способ передачи объектов в функции для их изменения;
- как способ передачи объектов без копирования;
- как необязательный аргумент функции, если функция имела несколько выходных параметров (когда было недостаточно одного возвращаемого значения);
- как половина пары указатель/длина, используемой для представления массивов;
- как средство управления памяти в куче.

Первые три задачи были отданы ссылкам.

1. Реализуйте функцию `sort2` с использованием указателей. Введите два значения в `main` и выполните вызов функции `sort2`.
2. Реализуйте функцию `sort2` с использованием ссылок. Введите два значения в `main` и выполните вызов функции `sort2`.
3. Создайте в функции `main` вектор из 1000000 целых значений:

   ```
   std::vector<int> v(1000000);
   ```

   Напишите функции которые в своём теле ничего не делают, но принимают вектор следующими способами:

   - По значению
   - По значению указателя
   - По ссылке

Замерьте время вызова каждой из функций и приложите к отчёту вместе с кодом эксперимента. Сделайте выводы.

4. Напишите следующие функции:

   - Ввод вектора целых чисел
   - Вывод вектора целых чисел
   - Поиск минимального значения среди элементов вектора
   - Вводится последовательность с клавиатуры, признак конца ввода – 0. Изменить порядок следования элементов в векторе на обратный[5]. Индексы в функции изменения порядка должны иметь тип `size_t`. Убедитесь, что функция корректно работает для пустого вектора (проще обработать данный случай до тела цикла).

     Убедитесь, что используете оптимальный способ передачи вектора в функцию (по ссылке или по ссылке на константу).

5. Решите следующую задачу, используя указатели: даны коэффициенты a, b, c квадратного уравнения. Гарантируется, что количество корней равняется двум. Вывести полученные корни. Корни должны выводиться в функции *main*, а их вычисление – происходить в `void getRoots(int a, int b, int c, double *x1, double *x2)`.

6. Для условия задачи 1, выполните переход на ссылки: `void getRoots(int a, int b, int c, double &x1, double &x2)`

7. При необходимости проведите эксперименты и заполните следующую таблицу:

Аспект	**Указатель**	**Ссылка**
Обязан быть инициализирован		
Особенности обращения к элементам		
Возможность перенацеливания		
Возможность получения адреса переменной (адреса указателя или адреса ссылки)		
Возможность непосредственной работы с динамической памятью		
Возможность создавать массивы		

[5]Для добавления элементов в конец вектора используется метод `.push_back()`. Для данной задачи весьма красиво будет смотреться использование бесконечного цикла.