**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ФКТИ**

отчет

**по лабораторной работе №2**

**по дисциплине «Программирование»**

Тема: Работа со списком и вставкой элемента после заданного элемента

|  |  |  |
| --- | --- | --- |
| Студент гр. 4335 |  | Иванов Г.Д. |
| Преподаватель |  | Калмычков В.А. |

Санкт-Петербург

2025

**Исходная формулировка задания**

Дана список строк, разделенных символом пробела. После каждого вхождения заданного слова вставить в строку другое заданное слово. Результат записать в файл.

**Структура представления данных.**

Для хранения строковых типов используется класс marker\_string:

class marker\_string{

char \* \_data; -- указатель на начало хранимой строки

char \_marker; -- символ, обозначающий конец строки

};

Для хранения списков используется шаблонный класс nx::forward\_list

namespace nx{

template <typename T>

struct forward\_list\_node {

forward\_list\_node<T> \* \_next;

T \_data;

};

template<typename T>

class forward\_list {

forward\_list\_node<T> \_root;

uint64\_t \_size;

};

}

**Формат входного и выходного файлов.**

1. Входной файл: lab2.in:

Первая строка – строка, элемент списка, после которого будет производиться вставка

Вторая строка – строка, элемент списка, который будет вставлен в исходный список

Третья - N строка – список строк, над которым будет проводится работа

1. Выходной файл: lab1.out:

Первая строка – «Element: » + элемент списка с заданным значением

Вторая строка – «Inserted: » + элемент, вставленный после заданного

Третья - N строка – «Source: » + изначальный список

N+1 - M строка – «Res: » + результат обработки

Пример входного файла:

a

great

This is

a

list for insertion

of

a

word

Пример выходного файла:

Element: a

Inserted: great

Source:

This is

a

list for insertion

of

a

word

Res:

This is

a

great

list for insertion of

a

great

word

**Реализация типов данный и функций:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Имя модуля (файла)** | **Имя структуры, класса или функции/метода** | **Назначение** | **Входные параметры** | **Возвращаемое значение** |
| marker\_string.h | marker\_string() | Конструктор по умолчанию. Инициализирует указатель \_data как nullptr и маркер \_marker символом '$' | - | - |
| marker\_string.h | marker\_string(const char\* cstr) | Конструктор из C-строки. Выделяет память, копирует строку, добавляет маркер в конец. | const char \* - указатель на  с-style строку | - |
| marker\_string.h | marker\_string(const std::string& str) | Конструктор из std::string. Аналогичен предыдущему, но принимает объект std::string. | const std::stirng & - строка по ссылке | - |
| marker\_string.h | marker\_string(const marker\_string& other) | Конструктор копирования. Создаёт глубокую копию другого объекта marker\_string. | Const marker\_string & - объект для копирования | - |
| marker\_string.h | ~marker\_string() | Деструктор. Освобождает динамически выделенную память. | - | - |
| marker\_string.h | marker\_string::size() const | Возвращает длину строки (до маркера). | - | int |
| marker\_string.h | marker\_string::clear() | Освобождает память, выделенную под \_data. | - | void |
| marker\_string.h | marker\_string::data() | Возвращает указатель на данные. | - | char\* |
| marker\_string.h | marker\_string::marker() | Возвращает символ-маркер окончания строки. | - | char |
| marker\_string.h | operator << | Заносит строку в структуре marker\_string в поток вывода | std::basic\_ostream<char>&,  const marker\_string & | std::basic\_ostream<char> & |
| main.cpp | main | Точка входа в программу | - | int |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| marker\_string.h | marker\_string::split | Разделить строку на список строк по символу разделителю | char sep, bool keep\_empty\_parts | forward\_list<marker\_string> |
| nx\_forward\_list.h | forward\_list | конструктор | - | - |
| nx\_forward\_list.h | forward\_list::begin() | Итератор на начале списка | - | forward\_list::iterator |
| nx\_forward\_list.h | forward\_list::end() | Итератор на конец списка (пустой итератор) | - | forward\_list::iterator |
| nx\_forward\_list.h | forward\_list::true\_end() | Итератор на конец списка (настоящий) | - | forward\_list::iterator |
| nx\_forward\_list.h | forward\_list::push\_front | Вставка элемента в начало списка | T value | - |
| nx\_forward\_list.h | forward\_list::insert() | Вставка элемента в середину списка по итератору | iterator pos, T value | forward\_list::iterator |
| nx\_forward\_list.h | forward\_list::remove | Удаление элемента из середины списка по итератору | iterator pos | forward\_list::iterator |

**Алгоритм работы обработки списка**

1. Инициализация:
   1. Попытка открытия входного файла. При неудаче – выход с кодом 1.
   2. Чтение строки, составляющей список
   3. Чтение строки, составляющей заданный элемент
   4. Чтение строки, составляющей элемент для вставки
   5. Создание списка, путем разделения первой строки по символу пробела
   6. При возникновении ошибки при чтении файла – выход с кодом 3.
2. Цикл по элементам списка (с помощью итераторов)
   1. Сравнение элемента списка с заданным элементом
   2. В случае равенства
      1. Инкрементировать итератор
      2. Вставить элемент методом insert
3. Завершение
   1. Попытка открытия файла для записи. При неудаче – выход с кодом 2.
   2. Запись исходных данных
   3. Запись результата
   4. Закрытие файла
   5. Выход с кодом 0

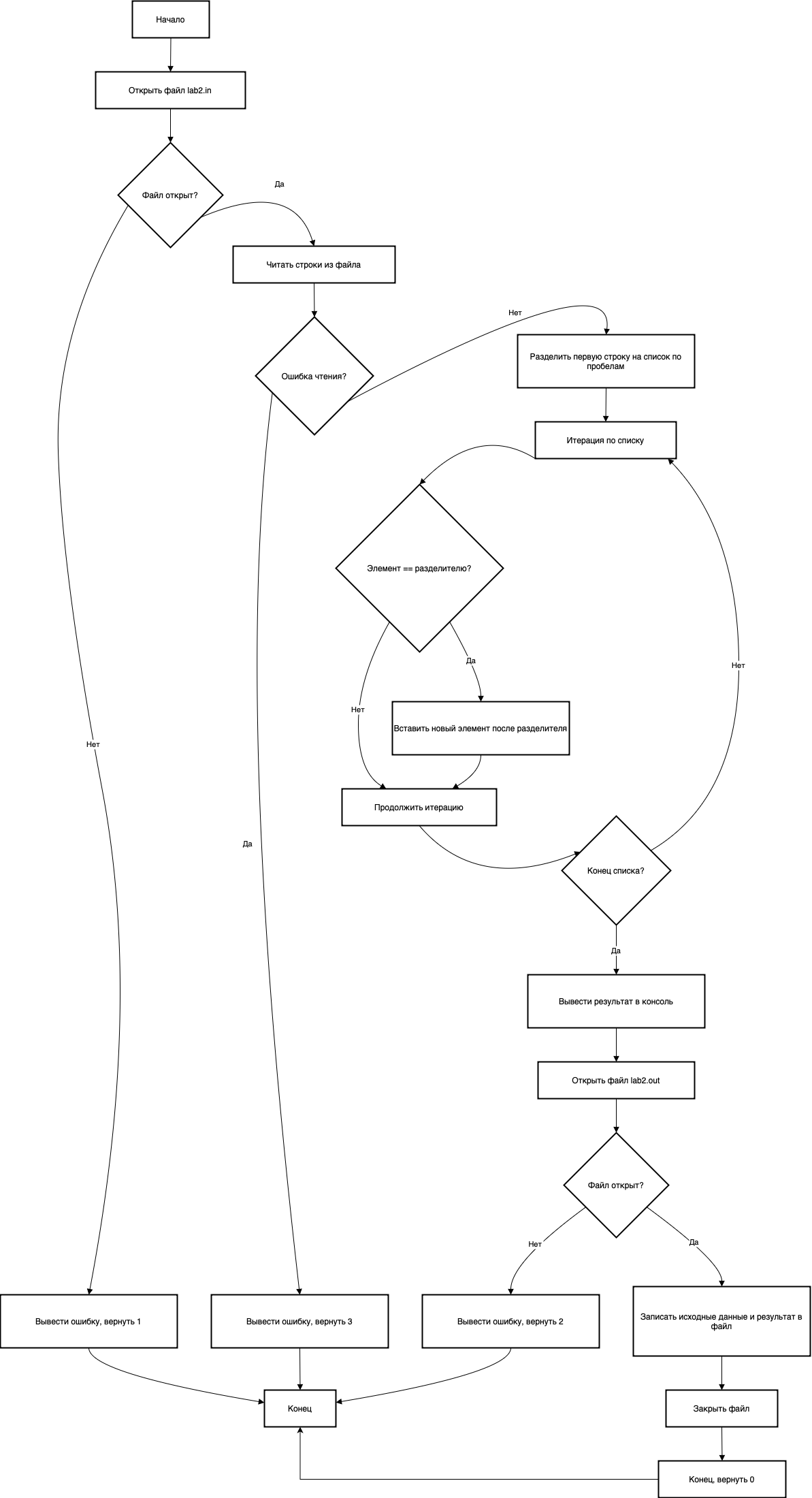


Диаграмма 1. Алгоритм работы функции main

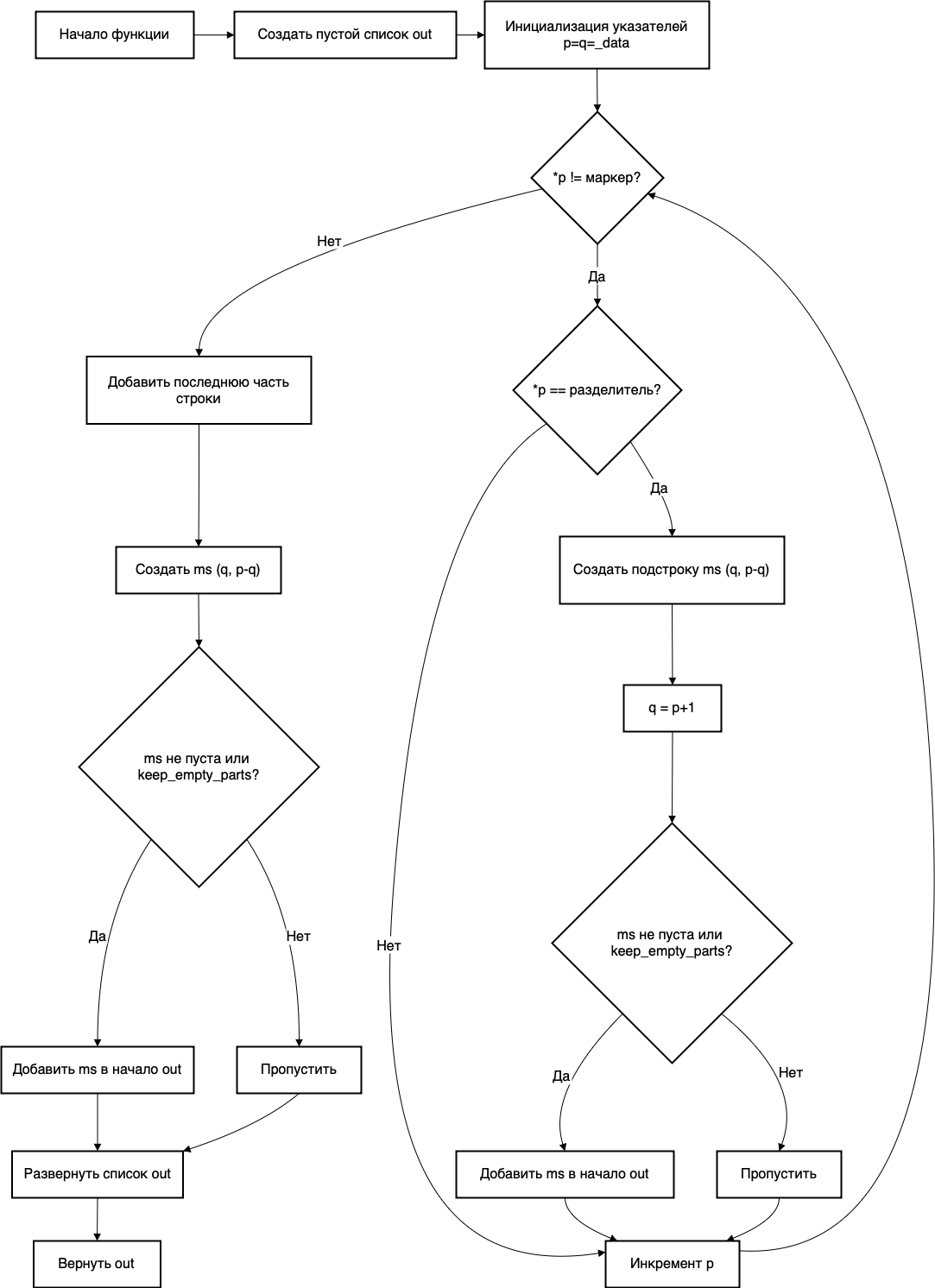


Диаграмма 2. Алгоритм работы метода marker\_string::split

Изображение выглядит как чек, диаграмма, текст

Содержимое, созданное искусственным интеллектом, может быть неверным.

Диаграмма 3. Алгоритм работы функции nx::forward\_list::insert

**Примеры работы программы.**

1. Обычный случай:  
   Входной файл:

this is

not

Hello world

this is

a test  
Выходной файл:

Element: is

Inserted: not

Source:

Hello world

this is

a test

Res:

Hello world

this is

not

a test

1. Случай, когда разделителя нет в строке:

missing

insertion

This string

has no

target word  
Выходной файл:

Element: missing

Inserted: insertion

Source:

This string

has no

target word

Res:

This string

has no

target word

1. Случай с пустой входной строкой:

Входной файл:

separator

new  
Выходной файл:

Element: separator

Inserted: new

Source:

Res:

**Визуализация внутреннего представления**

Структура класса marker\_string в памяти:



Структура класса forward\_list<T> в памяти:

Изображение выглядит как текст, чек, Шрифт, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

Исходный текст программы.

marker\_string.h

#ifndef MARKER\_STRING\_H  
#define MARKER\_STRING\_H  
  
#include <string>  
#include <iostream>  
#include "nx\_forward\_list.h"  
  
class marker\_string {  
 char \* \_data;  
 char \_marker;  
  
 void m\_init() {  
 \_data = nullptr;  
 \_marker = '$';  
 }  
  
public:  
 marker\_string() : \_data{ nullptr }, \_marker{ '$' } {}  
 marker\_string(const char \* cstr, int l = -1) : marker\_string() {  
 auto len = (l == -1) ? strlen(cstr) : l;  
 \_data = new char [len + 1];  
 memcpy(\_data, cstr, len);  
 \_data[len] = \_marker;  
 }  
 marker\_string(const std::string & str) : marker\_string() {  
 auto len = str.size();  
 \_data = new char [len + 1];  
 memcpy(\_data, str.c\_str(), len);  
 \_data[len] = \_marker;  
 }  
 marker\_string(const marker\_string & other) :  
 \_marker{other.\_marker}  
 {  
 int len = other.size();  
 \_data = new char [len + 1];  
 \_data[len] = \_marker;  
 for (int i = 0; i < len; i++)  
 \_data[i] = other.\_data[i];  
 }  
 ~marker\_string() {  
 clear();  
 }  
  
 int size () const {  
 if (\_data == nullptr) return 0;  
 int out = 0;  
 auto ch = data();  
 while (\*ch++ != \_marker) out += 1;  
 return out;  
 }  
 void clear () { if (\_data) delete [] \_data; }  
 char \* data () const { return \_data; }  
 char marker () const { return \_marker; }  
 bool empty () const { return (\_data == nullptr) || (\_data[0] == \_marker); }  
  
 nx::forward\_list<marker\_string> split (char sep, bool keep\_empty\_parts = false) const {  
 nx::forward\_list<marker\_string> out;  
 char \* p = \_data;  
 char \* q = \_data;  
 while (\*p != \_marker) {  
 if (\*p == sep) {  
 marker\_string ms {q, (int)(p - q)};  
 q = p + 1;  
 if (!ms.empty() || keep\_empty\_parts) out.push\_front(ms);  
 }  
 p++;  
 }  
 marker\_string ms {q, (int)(p - q)};  
 if (!ms.empty() || keep\_empty\_parts) out.push\_front(ms);  
 out.reverse();  
 return out;  
 }  
  
 bool operator == (const marker\_string & other) const {  
 int this\_size = size();  
 int other\_size = other.size();  
 if (this\_size != other\_size) return false;  
  
 char \* this\_p = data();  
 char \* other\_p = other.data();  
 while (\*this\_p != \_marker) {  
 if (\*this\_p != \*other\_p) return false;  
 this\_p++;  
 other\_p++;  
 }  
 return true;  
 }  
};  
  
inline std::basic\_ostream<char> & operator << (std::basic\_ostream<char> & stream, const marker\_string & str) {  
 auto ch = str.data();  
 while (\*ch != str.marker())  
 stream << \*ch++;  
 return stream;  
}  
  
marker\_string remove\_special\_symbols\_from\_word\_n(const marker\_string & in, int n);  
  
#endif //MARKER\_STRING\_H

marker\_string.cpp

#include "marker\_string.h"  
  
marker\_string remove\_special\_symbols\_from\_word\_n(const marker\_string & in, int n) {  
 marker\_string out {in};  
 int i = 0;  
 bool current\_is\_word = (\*in.data() != ' ');  
 int word\_count = current\_is\_word ? 1 : 0;  
 for (char \* ch = in.data(); \*ch != in.marker(); ch++) {  
 if (\*ch != ' ' && (current\_is\_word == false)) {  
 word\_count++;  
 current\_is\_word = true;  
 }  
 else if (\*ch == ' ') current\_is\_word = false;  
 else current\_is\_word = true;  
 if (\*ch == ' ') {  
 out.data()[i++] = \*ch;  
 continue;  
 }  
 if (word\_count == n) {  
 if ((\*ch >= 'a' && \*ch < 'z') || (\*ch >= 'A' && \*ch <= 'Z'))  
 out.data()[i++] = \*ch;  
 }  
 else  
 out.data()[i++] = \*ch;  
 }  
 out.data()[i] = out.marker();  
 return out;  
}

nx\_forward\_list.h

#ifndef NX\_FORWARD\_LIST\_H  
#define NX\_FORWARD\_LIST\_H  
  
#include <ranges>  
  
namespace nx {  
  
 namespace \_detail  
 {  
 struct \_forward\_list\_node\_base {  
 \_forward\_list\_node\_base \* \_next;  
 explicit \_forward\_list\_node\_base (\_forward\_list\_node\_base \* p\_next = nullptr) : \_next{p\_next} {};  
 };  
 template <typename \_Tp>  
 struct \_forward\_list\_node : public \_forward\_list\_node\_base  
 {  
 using base = \_forward\_list\_node\_base;  
 using value\_type = \_Tp;  
 value\_type \_data;  
  
 explicit \_forward\_list\_node (const value\_type & p\_data) :  
 \_data {p\_data},  
 \_forward\_list\_node\_base{nullptr}  
 {};  
 explicit \_forward\_list\_node (const \_Tp & p\_data, base \* p\_next) :  
 \_data {p\_data},  
 \_forward\_list\_node\_base{p\_next}  
 {};  
  
 base \* m\_base () const { return static\_cast<base \*> (this); }  
 value\_type & m\_data () { return \_data; }  
 value\_type const & m\_data () const { return \_data; }  
 value\_type \* m\_data\_ptr () { return &\_data; }  
 value\_type \* m\_data\_ptr () const { return &\_data; }  
 };  
  
 template <typename \_Tp>  
 struct \_forward\_list\_iterator {  
 using value\_type = \_Tp;  
 using base\_type = \_forward\_list\_node\_base;  
 using node\_type = \_forward\_list\_node<value\_type>;  
 using self\_type = \_forward\_list\_iterator<value\_type>;  
 base\_type \* \_node;  
  
 explicit \_forward\_list\_iterator (base\_type \* p\_node) : \_node(p\_node) {}  
 const self\_type & m\_const\_cast () const { return \*this; }  
 value\_type \* operator -> () { return static\_cast<node\_type \*>(\_node->\_next)->m\_data\_ptr(); }  
 value\_type & operator \* () { return static\_cast<node\_type \*>(\_node->\_next)->m\_data(); }  
 self\_type & operator ++ () { \_node = \_node->\_next; return \*this; }  
 self\_type operator ++ (int) { auto tmp = \*this; \_node = \_node->\_next; return tmp; }  
 bool operator == (const self\_type & p\_other) { return \_node == p\_other.\_node || (at\_end() && p\_other.at\_end()); }  
 bool operator != (const self\_type & other) { return !operator == (other); }  
 bool at\_end () const { return !valid() || (\_node->\_next == nullptr); } // || (\_node->\_next->\_next == nullptr); }  
 bool valid () const { return \_node != nullptr; }  
 };  
  
 template <typename \_Tp>  
 struct \_const\_forward\_list\_iterator {  
 using value\_type = \_Tp;  
 using base\_type = \_forward\_list\_node\_base;  
 using node\_type = \_forward\_list\_node<value\_type>;  
 using self\_type = \_const\_forward\_list\_iterator<value\_type>;  
 using iterator\_type = \_forward\_list\_iterator<value\_type>;  
 const base\_type \* \_node;  
  
 explicit \_const\_forward\_list\_iterator (const base\_type \* p\_node) : \_node(p\_node) {}  
 \_const\_forward\_list\_iterator (const iterator\_type & p\_other) : \_node(p\_other.\_node) {}  
 value\_type \* operator -> () const { return static\_cast<node\_type \*>(\_node->\_next)->m\_data\_ptr(); }  
 value\_type & operator \* () const { return static\_cast<node\_type \*>(\_node->\_next)->m\_data(); }  
 self\_type & operator ++ () { \_node = \_node->\_next; return \*this; }  
 self\_type operator ++ (int) { auto tmp = \*this; \_node = \_node->\_next; return tmp; }  
 bool operator == (const self\_type & p\_other) { return \_node == p\_other.\_node || (at\_end() && p\_other.at\_end());; }  
 bool operator != (const self\_type & other) { return !operator == (other); }  
 bool at\_end () const { return !valid() || (\_node->\_next == nullptr); }// || (\_node->\_next->\_next == nullptr); }  
 bool valid () const { return \_node != nullptr; }  
 };  
 };  
  
 template <typename \_Tp = void>  
 class forward\_list {  
 public:  
 using value\_type = \_Tp;  
 using reference = value\_type&;  
 using pointer = value\_type\*;  
 using const\_reference = const value\_type&;  
 using const\_pointer = const value\_type\*;  
 using iterator = \_detail::\_forward\_list\_iterator<value\_type>;  
 using const\_iterator = \_detail::\_const\_forward\_list\_iterator<value\_type>;  
 using list\_node = \_detail::\_forward\_list\_node<value\_type>;  
 using self\_type = forward\_list<value\_type>;  
#ifdef \_\_aarch64\_\_  
 using size\_type = uint64\_t;  
#else  
 using size\_type = size\_t;  
#endif  
 private:  
 \_detail::\_forward\_list\_node\_base \_root;  
 size\_type \_size;  
 void m\_init() {  
 \_root.\_next = nullptr;  
 \_size = 0;  
 }  
 const\_iterator m\_last\_node () {  
 auto it = begin();  
 for (; !it.at\_end(); ++it) {}  
 return it;  
 }  
 public:  
 forward\_list () { m\_init(); }  
 ~forward\_list () { clear(); }  
 forward\_list (::std::initializer\_list<value\_type> p\_list) {  
 m\_init();  
 auto back\_it = begin();  
 for (auto it = p\_list.begin(); it != p\_list.end(); ++it, ++back\_it)  
 insert(back\_it, \*it);  
 }  
 forward\_list (const self\_type & p\_other) {  
 m\_init();  
 splice\_back(p\_other);  
 }  
 forward\_list (self\_type && p\_other) {  
 \_root = p\_other.\_root;  
 p\_other.m\_init();  
 }  
 forward\_list & operator = (const self\_type & p\_other) {  
 clear();  
 m\_init();  
 splice\_back(p\_other);  
 return \*this;  
 }  
 forward\_list & operator = (self\_type && p\_other) {  
 clear();  
 m\_init();  
 \_root = p\_other.\_root;  
 \_size = p\_other.\_size;  
 p\_other.m\_init();  
 return \*this;  
 }  
  
 iterator begin () { return iterator(&\_root); }  
 const\_iterator begin () const { return const\_iterator(&\_root); }  
 iterator end () { return iterator(nullptr); }  
 const\_iterator end () const { return iterator(nullptr); }  
 iterator true\_end () {auto it = begin(); while (!it.at\_end()) {++it;} return it; }  
 const\_iterator true\_end () const {auto it = begin(); while (!it.at\_end()) {++it;} return it; }  
  
 void clear() {  
 auto cur\_ptr = &\_root;  
 auto next\_ptr = \_root.\_next;  
 while (next\_ptr && (cur\_ptr != next\_ptr)) {  
 cur\_ptr = next\_ptr;  
 next\_ptr = next\_ptr->\_next;  
 delete cur\_ptr;  
 }  
 m\_init();  
 }  
 size\_type size () const {  
 return \_size;  
 }  
 bool empty () const { return \_size == 0; }  
 iterator insert (const iterator & pos, const value\_type & p\_data) {  
 if (!pos.valid()) return iterator(nullptr);  
 auto new\_node = new list\_node(p\_data, pos.\_node->\_next);  
 pos.\_node->\_next = new\_node;  
 \_size += 1;  
 return iterator(new\_node);  
 }  
 iterator remove (const iterator & pos) {  
 if (!pos.valid()) { return iterator(nullptr); }  
 if (pos.at\_end()) { pop\_back(); return end(); }  
 auto node = pos.\_node->\_next;  
 pos.\_node->\_next = node->\_next;  
 delete node;  
 return iterator(pos.\_node);  
 }  
 void push\_front(const value\_type & p\_data) {  
 auto new\_node = new list\_node(p\_data, \_root.\_next);  
 \_root.\_next = new\_node;  
 \_size += 1;  
 }  
 void pop\_front () {  
 if (\_root.\_next == nullptr) return;  
 auto tmp = \_root.\_next;  
 \_root.\_next = \_root.\_next->\_next;  
 delete tmp;  
 \_size -= 1;  
 }  
 void push\_back(const value\_type & data) {  
 auto it = begin();  
 for (; !it.at\_end(); ++it) {}  
 insert(it, data);  
 }  
 void pop\_back() {  
 if (empty()) return;  
 auto tmp = &\_root;  
 while (tmp->\_next->\_next != nullptr) { tmp = tmp->\_next; }  
 delete tmp->\_next;  
 tmp->\_next = nullptr;  
 \_size -= 1;  
 }  
 void splice\_back (const self\_type & p\_other) {  
 auto back\_it = iterator(&\_root);  
 for (; !back\_it.at\_end(); ++back\_it) {}  
 auto other\_it = p\_other.begin();  
 while (other\_it != p\_other.end()) {  
 insert(back\_it, \*other\_it);  
 ++back\_it;  
 ++other\_it;  
 }  
 \_size += p\_other.\_size;  
 }  
  
 reference at (size\_type p\_pos) {  
 // static\_assert(p\_pos < \_size && p\_pos > 0);  
 size\_type pos { 0 };  
 for (auto it = begin(); it != end(); ++it, pos++)  
 if (pos == p\_pos)  
 return \*it;  
 }  
 const\_reference at (size\_type p\_pos) const {  
 // static\_assert(p\_pos < \_size && p\_pos > 0);  
 size\_type pos { 0 };  
 for (auto it = begin(); it != end(); ++it, pos++)  
 if (pos == p\_pos)  
 return \*it;  
 }  
  
 template <typename compare\_func = std::less<\_Tp>>  
 void sort (compare\_func comp = std::less<\_Tp>())  
 {  
 if (size() <= 1) return;  
 value\_type pivot = \*begin();  
 pop\_front();  
  
 self\_type less;  
 self\_type greater;  
  
 for (const auto& item : \*this) {  
 if (comp(item, pivot)) {  
 less.push\_back(item);  
 } else {  
 greater.push\_back(item);  
 }  
 }  
  
 clear();  
 less.sort(comp);  
 greater.sort(comp);  
  
 splice\_back(less);  
 push\_back(pivot);  
 splice\_back(greater);  
 }  
  
 self\_type reversed () const {  
 forward\_list out;  
 for (auto it = begin(); it != end(); ++it)  
 out.push\_front(\*it);  
 return out;  
 }  
  
 self\_type & reverse () {  
 return \*this = reversed();  
 }  
 };  
  
} // nx  
  
#endif //NX\_FORWARD\_LIST\_H

main.cpp

#include "nx\_forward\_list.h"  
#include <iostream>  
#include "marker\_string.h"  
#include <fstream>  
  
int main (int argc, char \* argv []) {  
 std::ifstream fin ("lab2.in");  
 if (!fin) {  
 std::cerr << "Cannot open file lab2.in for read" << std::endl;  
 return 1;  
 }  
  
 std::string line;  
 // std::getline(fin, line);  
 // marker\_string source { line };  
 // nx::forward\_list<marker\_string> list = source.split(' ', false);  
 nx::forward\_list<marker\_string> list;  
  
 std::getline(fin, line);  
 marker\_string sep { line };  
 std::getline(fin, line);  
 marker\_string elem { line };  
  
 while (!fin.eof()) {  
 std::getline(fin, line);  
 list.push\_front(line);  
 }  
  
 list.reverse();  
 auto source\_list = list;  
  
  
 for (auto it = list.begin(); it != list.end(); ++it) {  
 if (\*it == sep)  
 list.insert(++it, elem);  
 }  
 std::cout << "Result: ";  
 for (auto i: list) std::cout << i << std::endl;  
  
 std::ofstream fout ("lab2.out");  
 if (!fout) {  
 std::cerr << "Cannot open file lab2.out for write" << std::endl;  
 return 2;  
 }  
  
 // fout << "Source: " << source << std::endl;  
 fout << "Element: " << sep << std::endl;  
 fout << "Inserted: " << elem << std::endl;  
 fout << "Source :" << std::endl;  
 for (auto it = source\_list.begin(); it != source\_list.end(); ++it)  
 fout << \*it << std::endl;  
 fout << "Res: " << std::endl;  
 for (auto it = list.begin(); it != list.end(); ++it)  
 fout << \*it << std::endl;  
  
 fout.close();  
  
 return 0;  
}

Вывод.

В ходе лабораторной работы была реализована программа для обработки строки из файла с последующей вставкой заданного элемента после каждого вхождения разделителя. Программа использует пользовательский контейнер nx::forward\_list и структуру marker\_string для эффективной работы со строками.

Реализация включала:

1. Чтение данных из файла с обработкой возможных ошибок ввода-вывода.
2. Разделение строки на элементы с помощью метода split().
3. Поиск разделителя и вставку нового элемента с использованием итераторов.
4. Вывод результатов как в консоль, так и в файл для дальнейшего анализа.

Программа успешно обрабатывает различные сценарии, включая отсутствие разделителя и пустую входную строку. Работа позволила закрепить навыки работы с файлами, односвязными списками, итераторами и пользовательскими типами данных в C++.