Министерство образования и наук России.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮТЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«САРАТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

УТВЕРЖДАЮ

Зав.кафедрой,

д. ф.-м.н.

\_\_\_\_\_\_\_\_\_\_\_\_ Д.К. Андрейченко

**ОТЧЕТ ПО ПРАКТИКЕ**

Студента 1 курса 141 группы факультета КНиИТ

Голикова Артема Олеговича

вид практики : учебная

кафедра : математического обеспечения выч. комплексов и информационных систем

курс : 1

семестр : 2

продолжительность : 2 недели, с 01.07.2013 г. по 15.07.2013 г.

Руководитель от университета

доцент, к.ф.-м. н. \_\_\_\_\_\_\_\_\_\_\_\_ Д.К. Андрейченко

Руководитель практики от организации (EPAM SYSTEMS)

Старший преподаватель \_\_\_\_\_\_\_\_\_\_\_\_ П.В. Агуров

**ОГЛАВЛЕНИЕ**

ВВЕДЕНИЕ: 3

ГЛАВА 1: 5

1.1 Перегрузка операторов: 5

1.2 Типы контейнеров: 20

1.3 Итераторы 8

1.4 Функциональные объекты (функторы) и лямбда-выражения (лямбды): 9

1.5 Конструктор копирования и конструктор переноса 10

1.6 Наследование классов, виртуальные методы, абстрактные классы: 11

1.7 Динамичекие объекты, проблема утечки памяти, умные указатели: 222

ГЛАВА 2 13

2.1 Практическое задание 1 13

2.2 Практическое задание 2 16

2.3 Практическое задание 3 18

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ 24

**ВВЕДЕНИЕ**

Стандартная Библиотека Шаблонов (Standard Template Library) предоставляет набор хорошо сконструированных и согласованно работающих вместе обобщённых компонентов C++. Особая забота была проявлена для обеспечения того, чтобы все шаблонные алгоритмы работали не только со структурами данных в библиотеке, но также и с встроенными структурами данных C++. Например, все алгоритмы работают с обычными указателями. Проект библиотеки позволяет программистам использовать библиотечные структуры данных со своими собственными алгоритмами, а библиотечные алгоритмы - со своими собственными структурами данных. Хорошо определённые семантические требования и требования сложности гарантируют, что компонент пользователя будет работать с библиотекой и он будет работать эффективно. Эта гибкость обеспечивает широкую применимость библиотеки.

Другое важное соображение - эффективность. C++ успешен, потому что он объединяет выразительную мощность с эффективностью. Много усилий было потрачено, чтобы проверить, что каждый шаблонный компонент в библиотеке имеет обобщённую реализацию, которая имеет эффективность выполнения с разницей в пределах нескольких процентов от эффективности соответствующей программы ручной кодировки. Основным достоинством библиотеки C++ является то, что она действительно стандартная, т.е. входит в стандарт и обязательна для реализации любым компилятором, удовлетворяющего стандарту C++. Эту библиотеку вы найдёте и в легендарном UNIX-компиляторе GCC, и в Visual Studio C++ .NET, и в C++ Builder, и в Dev-C++, в практически любом компиляторе.

Библиотека стандартных шаблонов до включения в стандарт C++ была сторонней разработкой, вначале — фирмы HP, а затем SGI. Стандарт языка не называет её «STL», так как эта библиотека стала неотъемлемой частью языка, однако многие люди до сих пор используют это название, чтобы отличать её от остальной части стандартной библиотеки (потоки ввода/вывода (iostream), подраздел Си и др.).

Проект под названием STLPort, основанный на SGI STL, осуществляет постоянное обновление STL, iostream и строковых классов. Некоторые другие проекты также занимаются разработкой частных применений стандартной библиотеки для различных конструкторских задач. Каждый производитель компиляторов C++ обязательно поставляет какую-либо реализацию этой библиотеки, так как она является очень важной частью стандарта и широко используется. Архитектура STL была разработана Александром Степановым и Менг Ли.

Цель данной работы:

* ознакомится с библиотекой STL
* научиться пользоваться структурами и алгоритмами библиотеки STL на практике

**ГЛАВА 1**

* 1. **Перегрузка операторов**

Перегрузка операторов — в программировании — один из способов реализации полиморфизма, заключающийся в возможности одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

Зачастую, когда одинаковые по смыслу операции применяются к операндам различных типов, их вынужденно приходится называть по-разному. Невозможность применять для разных типов функции с одним именем приводит к необходимости выдумывать различные имена для одного и того же, что создаёт путаницу, а может и приводить к ошибкам. А вот если оператор прописывать как функцию (<операция>(<операнд 1>,<операнд 2>)), то можно вызывать перегрузку функций, что даст нам один метод для различных типов данных.

* 1. **Типы контейнеров**

Контейнер - это хранилище объектов (как встроенных, так и определённых пользователем типов). Как правило, контейнеры реализуются в виде шаблонов классов. Простейшие виды контейнеров (статические и динамические массивы) встроены непосредственно в язык C++. Кроме того, стандартная библиотека включает в себя реализации таких контейнеров, как вектор (vector), список (list), очередь (deque), ассоциативный массив (map), множество (set) и некоторых других.

Каждый контейнер предоставляет строго определённый интерфейс, через который с ним будут взаимодействовать алгоритмы. Этот интерфейс обеспечивают соответствующие контейнеру итераторы. Важно подчеркнуть, что никакие дополнительные функции-члены для взаимодействия алгоритмов и контейнеров не используются. Это сделано потому, что стандартные алгоритмы должны работать, в том числе со встроенными контейнерами языка C++, у которых есть итераторы (указатели), но нет ничего, кроме них. Таким образом, при создании собственного контейнера реализация итератора - необходимый минимум. Каждый контейнер реализует определённый тип итераторов. При этом выбирается наиболее функциональный тип итератора, который может быть эффективно реализован для данного контейнера. "Эффективно" означает, что скорость выполнения операций над итератором не должна зависеть от количества элементов в контейнере. Например, для вектора реализуется итератор с произвольным доступом, а для списка - двунаправленный. Поскольку скорость выполнения операции [] для списка линейно зависит от его длины, итератор с произвольным доступом для списка не реализуется вне зависимости от фактической организации контейнера (вектор, список, дерево) хранящиеся в нём элементы можно рассматривать как последовательность. Итератор первого элемента в этой последовательности возвращает функция begin(), а итератор элемента, следующего за последним, - функция end(). Это очень важно, так как все алгоритмы в STL работают именно с последовательностями, заданными итераторами начала и конца.

Контейнеры делят на три типа:

* контейнеры последовательностей
* ассоциативные контейнеры
* адаптеры контейнеров

Контейнеры последовательностей:

Вектор (vector) напоминает нам массив, только он способен расти до произвольного размера, поддерживает информацию о размере. Как и массив к вектору можно обратиться воспользовавшись операцией индексирования []. Характеристики:

* Доступ к данным с одинаковой скоростью
* Вставка приводит к перемещению элементов
* При расширении данные копируются в другой блок

Вектор оптимален для получения информации, но при большом количестве вставок лучше воспользоваться другими контейнерами, например, списками. Проблема в том, что физически вектор располагается в непрерывной памяти.

Двунаправленная очередь (deque) - вид последовательности, которая, подобно вектору, поддерживает итераторы произвольного доступа. Кроме того, она поддерживает операции вставки и стирания в начале или в конце за постоянное время; вставка и стирание в середине занимают линейное время.

Двунаправленный линейный список (list) - вид последовательности, которая поддерживает двунаправленные итераторы и позволяет операции вставки и стирания с постоянным временем в любом месте последовательности, с управлением памятью, обрабатываемым автоматически. В отличие от векторов и двусторонних очередей, быстрый произвольный доступ к элементам списка не поддерживается, но многим алгоритмам, во всяком случае, только и нужен последовательный доступ.

Ассоциативные контейнеры:

(set) - множество, представляет собой объект, контролирующий произвольной длины последовательность уникальных элементов какого-либо типа. Используется для хранения неповторяющихся (уникальных) ключей, либо для проверки, есть ли элемент в наборе данных. Все элементы, помещенные во множество, оказываются рассортированы в порядке возрастания.

(multiset) - то же что и (set), но позволяет хранить повторяющиеся элементы.

(map) - ассоциативный контейнер, который поддерживает уникальные ключи (не содержит ключи с одинаковыми значениями) и обеспечивает быстрый поиск значений другого типа, связанных с ключами. Этот класс имеет недостаток - скорость вставки новой пары обратно пропорциональна количеству элементов, сохраненных в классе. Еще один важный момент - ключ должен быть уникальным.

(multimap) - то же что и map, но позволяет хранить несколько одинаковых ключей.

Адаптеры контейнеров:

Стандартный стек (stack) - контейнер, в котором добавление и удаление элементов осуществляется с одного конца.

Стандартная очередь (queue) - контейнер, с одного конца которого можно добавлять элементы, а с другого - вынимать.

Очередь с приоритетами (priority\_queue) – контейнер, организованный так, что самый большой элемент всегда стоит на первом месте.

* 1. **Итераторы**

Итератор - это абстракция указателя, то есть объект, который может ссылаться на другие объекты, содержащиеся в контейнере. Основные функции итератора - обеспечение доступа к объекту, на который он ссылается (разыменование), и переход от одного элемента контейнера к другому (итерация, отсюда и название итератора). Для встроенных контейнеров в качестве итераторов используются обычные указатели. В случае с более сложными контейнерами итераторы реализуются в виде классов с набором перегруженных операторов. Итераторы бывают нескольких типов – входные, выходные, прямые, двунаправленные и произвольного доступа. Тип итератора определяет подмножество операторов, применимых к нему.

Итераторы ввода (input iterator) поддерживают операции равенства, разыменования и инкремента.

==, !=, \*i, ++i, i++, \*i++

Итераторы вывода (output iterator) поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента.

++i, i++, \*i = t, \*i++ = t

Однонаправленные итераторы (forward iterator) поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без ограничения применять присваивание.

==, !=, =, \*i, ++i, i++, \*i

Двунаправленные итераторы (bidirectional iterator) обладают всеми свойствами forward-итераторов, а также имеют дополнительную операцию декремента (--i, i--, \*i--), что позволяет им проходить контейнер в обоих направлениях.

Итераторы произвольного доступа (random access iterator) обладают всеми свойствами bidirectional-итераторов, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ по индексу.

i += n, i + n, i -= n, i - n, i1 - i2, i[n], i1 < i2, i1 <= i2, i1 > i2, i1 >= i2

В общем случае итератор, который имеет большие возможности доступа, можно использовать вместо итератора с меньшими возможностями. Например, однонаправленным итератором можно заменить входной итератор.

Итераторы обрабатываются аналогично указателям. Их можно инкрементировать и декрементировать. К ним можно применять оператор разыменования адреса \*. Итераторы объявляются с помощью типа iterator, определяемого различными контейнерами.

Библиотека STL поддерживает реверсивные итераторы, которые являются либо двунаправленными, либо итераторами случайного доступа, т.е. позволяют перемещаться по последовательности в обратном направлении. Следовательно, если реверсивный итератор указывает на конец последовательности, то после инкрементирования он будет указывать на элемент, расположенный перед концом последовательности.

* 1. **Функциональные объекты (функторы) и лямбда-выражения (лямбды)**

Функтор - конструкция, которая предоставляет возможность использовать объект как функцию. Это может быть структура или класс, перегружающие оператор(). В языке С используется указатель на функцию. Конечно, подобная вещь оставлена для совместимости, но в реальности теряет смысл в использовании в С++. Они очень важны для эффективного использования. В тех местах, где предполагается передача указателей на функцию, создается интерфейс, принимающий объект с реализованным перегруженным оператором вызова функции. Все операторы обычно пишутся как inline, что дает дополнительный выигрыш в скорости.

Лямбда-выражение — это специальный синтаксис для объявления анонимных функторов по месту их использования. Используя лямбда-выражения, можно объявлять функции в любом месте кода. Используя лямбда-выражения, вы можете кратко определять функциональные объекты для использования в любое время. Лямбда-выражения связывают эти два действия – создание и подключение – в один выразительный оператор кода. Вдобавок вы можете легко ассоциировать среду с функциональными объектами. Например, функционал может принимать две функции, одна из которых выполняет одну математическую операцию, а другая – другую математическую операцию, и возвращать третью функцию, представляющую комбинацию первых двух. Лямбда-выражения предоставляют более естественный способ создания и вызова функционалов.

В простых синтаксических терминах лямбда-выражение – это синтаксис, посредством которого вы можете объявлять анонимные функции (делегаты) более выразительным способом.

* 1. **Конструктор копирования и конструктор переноса**

Конструктором копирования) называется специальный конструктор в языке программирования C++, применяемый для создания нового объекта как копии уже существующего. Такой конструктор принимает как минимум один аргумент: ссылку на копируемый объект.

Обычно компилятор автоматически создает конструктор копирования для каждого класса (известные как неявные конструкторы копирования, т.е. конструкторы копирования, заданные неявным образом), но в некоторых случаях программист создает конструктор копирования, называемый в таком случае явным конструктором копирования (или "конструктором копирования, заданным явным образом"). В подобных случаях компилятор не создает неявные конструкторы.

Конструктор копирования в основном необходим, когда объект имеет указатель или неразделяемую ссылку, как например, на файл, в этом случае вам обычно также потребуется деструктор и оператор присваивания.

* 1. **Наследование классов, виртуальные методы, абстрактные классы**

Наследование — механизм объектно-ориентированного программирования (наряду с инкапсуляцией, полиморфизмом и абстракцией), позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом.

Другими словами, класс-наследник реализует спецификацию уже существующего класса (базовый класс). Это позволяет обращаться с объектами класса-наследника точно так же, как с объектами базового класса.

Виртуальный метод (виртуальная функция) — в объектно-ориентированном программировании метод (функция) класса, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.

Таким образом, программисту необязательно знать точный тип объекта для работы с ним через виртуальные методы: достаточно лишь знать, что объект принадлежит классу или наследнику класса, в котором метод объявлен.

Абстрактный класс является классом, который может использоваться только в качестве базового для других классов. Абстрактный класс содержит одну или несколько чистых виртуальных функций. Чистая виртуальная функция может рассматриваться как встроенная функция, тело которой определено как =0 (чистый спецификатор). Для чистой виртуальной функции не нужно приводить действительное определение; предполагается, что она переопределяется в производных классах.

К абстрактным классам применимы следующие правила:

• абстрактный класс не может использоваться в качестве типа аргу-мента функции или типа возвращаемого значения;

• абстрактный класс нельзя использовать в явном преобразовании;

• нельзя определить представитель абстрактного класса (локальную/глобальную переменную или элемент данных);

• можно определять указатель или ссылку на абстрактный класс;

• если класс, производный от абстрактного, не определяет все чистые виртуальные функции абстрактного класса, он также является абстрактным.

**1.7 Динамические объекты, проблема утечек памяти, умные указатели**

Утечка памяти — процесс неконтролируемого уменьшения объёма свободной оперативной или виртуальной памяти компьютера, связанный с ошибками в работающих программах, вовремя не освобождающих не нужные уже участки памяти, или с ошибками системных служб контроля памяти.

Динамическая память является ограниченным ресурсом. Управление динамической памятью программы обычно осуществляется библиотекой языка программирования, которая сама работает поверх динамической памяти, предоставляемой операционной системой. Утечки памяти приводят к тому, что потребление памяти программой неконтролируемо возрастает, в результате рано или поздно вступают в действие архитектурные ограничения среды исполнения (операционной системы, виртуальной машины, ЭВМ), и тогда новое выделение памяти становится невозможным. В этой ситуации в программе, которая запрашивает память, обычно происходит аварийная остановка. Это может по стечению обстоятельств произойти и совсем с другой программой после того, как программа, подверженная утечкам, потребит всю память ЭВМ.

Умный указатель — класс (обычно шаблонный), имитирующий интерфейс обычного указателя и добавляющий некую новую функциональность, например проверку границ при доступе или очистку памяти.

Умные указатели призваны для борьбы с утечками памяти, которые сложно избежать в больших проектах. Они особенно удобны в местах, где возникают исключения, так как при последних происходит процесс раскрутки стека и уничтожаются локальные объекты. В случае обычного указателя — уничтожится переменная-указатель, при этом ресурс останется не освобожденным. В случае умного указателя — вызовется деструктор, который и освободит выделенный ресурс.

**ГЛАВА 2**

**2.1 Практическое задание 1**

Реализовать класс-контейнер вектор и класс (Iterator) итераторов для него, а так же различные методы для работы с классом вектора.

Класс шаблонный, что позволяет нам работать с различными типами данных. Внутри private находятся begin и end, которые отвечают за начало и конец данных соответственно. Внутри public находятся три конструктора: по умолчанию, копирования и переноса. Первый обнуляет начало и конец данных. Второй получает vector и копирует его данные, не изменяя исходные. Третий получает только указатели на начало и конец заданного вектора, а затем обнуляет их. Перенос намного эффективней копирования, т.к. не копирует сами данные, что дает значительный выигрыш во времени работы. Так же присутствуют два метода push, которые добавляют данные в вектор. Первый копированием – второй переносом.

template <class A>

class Vec

{

private:

A \*begin,\*end;

public:

typedef Iterator<A> It;

typedef A Vt;

Vec (const Vec& val)

{

It temp=val.begin,temp2;

int size=val.end-val.begin;

begin=new A[size];

end=begin+size;

temp2=begin;

for(;temp!=val.end;++temp,++temp2)

{

\*temp2=\*temp;

}

temp2=temp;

}

Vec (Vec && val)

{

begin=val.begin;

end=val.end;

val.begin=NULL;

val.end=NULL;

}

Vec():begin(NULL),end(NULL)

{

}

~Vec()

{

if (begin!=NULL)

{

delete [] begin;

}

}

void push(const A& x)

{

int count=end-begin;

A \*p=new A[count+1];

for(int i=0;i<count;++i)

p[i]=begin[i];

p[count]=x;

if(begin!=NULL)

delete [] begin;

begin=p;

end=p+count+1;

}

void push(A && x)

{

int count=end-begin;

A \*p=new A[count+1];

for(int i=0;i<count;++i)

p[i]=begin[i];

p[count]=std::move(x);

if(begin!=NULL)

delete [] begin;

begin=p;

end=p+count+1;

}

Iterator<A> Begin()

{

return Iterator<A>(begin);

}

Iterator<A> End()

{

return Iterator<A>(end);

}

};

Класс итераторов (Iterator) для вектора (vector):

template <class A>

class Iterator

{

private:

A \*el;

public:

Iterator():el(0)

{

}

Iterator(A \*e):el(e)

{

}

Iterator& operator++()

{

++el;

return (\*this);

}

Iterator& operator--()

{

--el;

return (\*this);

}

Iterator& operator+=(int x)

{

el+=x;

return(\*this);

}

bool operator==(const Iterator& i)

{

return el==i.el;

}

bool operator!=(const Iterator& i)

{

return el!=i.el;

}

A& operator\*()

{

return \*el;

}

};

Шаблонный метод, который считает сумму чисел в нашем векторе:

template <class Vt, class It>

Vt sum(It begin, It end)

{

Vt result = 0;

for (;begin!=end;++begin)

{

result+=\*begin;

}

return result;

}

Для реализации того же самого, но через функтор нужно создать класс, который будет выполнять необходимые операции с данными из нашего вектора:

template <class A>

class S

{

public:

A sum;

S():sum(0)

{

}

void operator()(A v)

{

sum+=v;

}

};

Теперь с помощью функторного класса мы можем посчитать сумму элементов:

template <class It,class M>

void for\_each(It begin, It end, M& m)

{

for(;begin!=end;++begin)

{

m(\*begin);

}

}

**2.2 Практическое задание 2**

Задан текстовый файл. Нужно составить словарь всех использовавшихся слов и посчитать, сколько каждое встречается в тексте.

Для решения этого задания использовался (map) и вектор (vector) строк библиотеки STL. Реализована функция, которая разбивает текст на слова и записывает их в вектор строк. Затем записывает из вектора в map слова, где ключи – сами слова, а их значение – количество.

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#include <map>

#include <algorithm>

#include <locale>

using namespace std;

std :: ifstream file("C:\\1.txt");

std :: ofstream out("C:\\voc.txt");

void mew(const std :: string & line, std :: vector <std :: string> & result)

{

std::string::size\_type beginPos = 0;

while (beginPos != std::string::npos)

{

std::string::size\_type pos = line.find(" ", beginPos);

if (pos != std::string::npos)

{

std :: string s = line.substr(beginPos,pos - beginPos);

std :: locale loc;

if (ispunct(\*s.begin(),loc)) s.erase(0,1);

if (ispunct(\*(s.end() -1 ), loc)) s.erase(s.length() - 1,1);

result.push\_back(s);

beginPos = pos + 1;

}

else

{

std :: string s = line.substr(beginPos);

result.push\_back(s);

break;

}

}

}

int main()

{

if (! file.is\_open() ) return 0;

std :: vector < std :: string> a;

while (! file.eof())

{

std :: string line;

std :: getline(file,line);

mew(line,a);

}

std :: map <std :: string, int> ans;

for (std :: vector <std :: string> :: iterator i = a.begin(); i != a.end(); ++i)

{

if (ans.find(\*i) == ans.end() )

{

ans.insert(std::pair<std :: string,int>(\*i,1));

}

else ans.find(\*i)->second += 1;

}

out << ans.size();

for\_each(ans.begin(),ans.end(),[](const std :: map <std :: string, int>::value\_type & v){out << v.first << " " << v.second << std :: endl;});

system("pause");

return 0;

}

**2.3 Практическое задание 3**

Реализовать алгоритм «Ляна-Кнута».

При работе с текстом часто возникает потребность корректно расставить переносы. Задача на первый взгляд не такая уж очевидная, нужно учитывать особенности каждого языка, чтобы решить, в каком месте разорвать слово. Как правильно формализовать такие требования, и как потом применить их в алгоритме? Одно из самых распространенных на сей день решений предложил Франклин Марк Лян, студент известного профессора Дональда Кнута. Алгоритм так и называется – «Алгоритм Ляна-Кнута».

Алгоритм основан на сравнении исходного слова с набором правил (шаблонов). Чем больше правил и чем качественнее они составлены, тем лучше будут расставляться переносы.

Пример правил:

при1

при3в

2и1ве

.по3ж2

Каждое правило состоит из букв и цифр между ними, а также цифр в начале и в конце. Цифру 0 обычно опускают. Например, первое правило должно пониматься как 0п0р0и1. Последовательность букв – это часть слова, для которой определяется перенос, т.е. эта последовательность должна присутствовать в слове. Цифры называют «уровнем», они задают приоритет между правилами и возможность переноса в соответствующей позиции. Четные цифры, включая 0, запрещают перенос. Нечетные – разрешают. Точка в начале правила означает, что правило применяется, только если последовательность находится в начале слова. Аналогично с точкой в конце – слово должно заканчиваться этой последовательностью. Если точка есть и в начале и в конце, то правило содержит слово целиком.

Основные этапы работы алгоритма:

* выбрать все правила, подходящие к выбранному слову и для каждой позиции в слове получить набор уровней (сколько правил пришлось на одну позицию, столько и уровней получим)
* в каждой позиции выбрать максимальный уровень. Если он четный – здесь переносить нельзя, если нечетный – допустимое место переноса
* отсечь недопустимые переносы (например, одна буква в начале или в конце)

В этой реализации задействовано проектирование приложения, т.е приложение состоит из 4-ех классов: Node, Tree, Builder, Hyphenate.

Файл Node.cpp (Header):

#pragma once;

#include <map>

#include <iostream>

#include <vector>

#include <algorithm>

#include <memory>

class Node

{

public:

typedef std :: shared\_ptr <Node> Ptr;

typedef std :: map <char, Ptr> NodeMap;

typedef std :: vector <int> IntVec;

NodeMap nodeMap\_;

IntVec nodePos\_;

};

Файл Tree.cpp (Header):

#pragma once;

#include "Node.h"

#include <map>

#include <iostream>

#include <vector>

#include <algorithm>

#include <memory>

class Tree

{

public:

typedef std :: shared\_ptr <Tree> Ptr;

Node :: Ptr roodNode\_;

};

Файл Builder.cpp:

#pragma once;

#include "Tree.h"

#include "Node.h"

#include <map>

#include <iostream>

#include <fstream>

#include <vector>

#include <algorithm>

#include <memory>

#include <string>

#include <cassert>

class Builder

{

public:

Tree :: Ptr tree ;

Builder();

Tree :: Ptr LoadFromFILE(const std :: string & Name);

Tree :: Ptr GetTree();

};

Файл Builed.cpp:

#include "Builder.h"

Builder :: Builder(): tree(new Tree())

{

tree->roodNode\_.reset(new Node());

}

void HyphRuleTrans(std :: string wordIn, std :: string & wordOut, std :: vector <int> & digits)

{

assert(! wordIn.empty());

bool flag = true;

for (std :: string :: iterator ruleIterator = wordIn.begin(); ruleIterator != wordIn.end(); ++ruleIterator)

{

if (isdigit(\*ruleIterator))

{

assert( flag );

digits.push\_back(int(\*ruleIterator - '0'));

flag = false;

}

else

{

if (flag) digits.push\_back(0);

else flag = true;

wordOut += \*ruleIterator;

}

}

if (flag) digits.push\_back(0);

}

Tree :: Ptr Builder :: LoadFromFILE(const std :: string & Name)

{

std :: ifstream file(Name);

while (! file.eof())

{

std :: string line;

std :: getline(file,line);

if (line.empty()) continue;

std :: vector <int> Vec;

std :: string str;

HyphRuleTrans(line,str,Vec);

Node :: Ptr node = tree->roodNode\_;

for (std :: string :: iterator i = str.begin(); i != str.end(); ++i)

{

Node :: NodeMap :: iterator NodeIt = node->nodeMap\_.find(\*i);

if (NodeIt == node->nodeMap\_.end())

NodeIt = node->nodeMap\_.insert(Node :: NodeMap :: value\_type (\*i, Node :: Ptr (new Node()))).first;

node = NodeIt->second;

}

node->nodePos\_ = Vec;

}

return tree;

}

Tree ::Ptr Builder :: GetTree()

{

return tree;

}

Файл Hyphenate.cpp (Header):

#pragma once;

#include "Tree.h"

#include "Node.h"

#include <map>

#include <iostream>

#include <vector>

#include <algorithm>

#include <memory>

#include "Builder.h"

#include <string>

class Hyphenate

{

public:

Tree :: Ptr tree;

Hyphenate();

std :: vector <int> HyphenateWord(std :: string & word);

};

Файл Hyphenate.cpp:

#include "Hyphenate.h"

Hyphenate :: Hyphenate(): tree(new Tree())

{

}

std :: vector <int> Hyphenate :: HyphenateWord (std :: string & word)

{

if (word.length() < 4) return std :: vector <int>();

word.insert(0,".");

word.insert(word.length(),".");

std :: vector <int> pos(word.length() + 1, 0);

std :: vector <int> :: iterator posIt = pos.begin();

for (std :: string :: iterator i = word.begin(); i != word.end(); ++i, ++posIt)

{

Node :: Ptr node = tree->roodNode\_;

for (std :: string :: iterator j = i; j != word.end(); ++j)

{

Node :: NodeMap :: iterator NodeIt = node->nodeMap\_.find(\*j);

if (NodeIt == node->nodeMap\_.end()) break;

node = NodeIt->second;

if (! node->nodePos\_.empty())

{

Node :: IntVec &vec = node->nodePos\_;

std :: vector <int> :: iterator p = posIt;

for (Node :: IntVec :: iterator k = vec.begin(); k != vec.end(); ++k)

{

\*p = std :: max(\*p,\*k);

++p;

}

}

}

}

std :: vector <int> ans;

for (std :: vector <int> :: iterator i = pos.begin() + 3; i != pos.end() - 3; ++i)

{

if (\*i % 2 == 1) ans.push\_back(i - pos.begin() - 1);

}

return ans;

}

Файл Main.cpp:

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#include <map>

#include <algorithm>

#include <cassert>

#include "Builder.h"

#include "Hyphenate.h"

typedef std :: vector <int> MyVec;

std :: ifstream file("C:\\1.txt");

std :: ofstream out("C:\\voc.txt");

void vec(std :: string & s, std :: vector <std :: string> & a)

{

int pos = 0;

while (pos != std :: string :: npos)

{

int Np = s.find(" ",pos);

if (Np != std :: string :: npos)

{

std :: string ns = s.substr(pos,Np - pos);

if (! isalpha(ns[0])) ns.erase(0,1);

if (! isalpha(ns[ns.length() -1])) ns.erase(s.length(),pos);

a.push\_back(ns);

pos = Np + 1;

}

std :: string ns = s.substr(pos,s.length() - pos);

if (! isalpha(ns[0])) ns.erase(0,1);

if (! isalpha(ns[ns.length() -1])) ns.erase(s.length(),pos);

a.push\_back(ns);

}

}

int main()

{

std :: string w;

std :: ofstream out("C:\\voc.txt");

Builder t;

Hyphenate f;

f.tree = t.LoadFromFILE("C:\\En\_us.rules");

std :: vector <int> ans = f.HyphenateWord(w);

if (! file.is\_open() ) return 0;

std :: vector < std :: string> a;

while (! file.eof())

{

std :: string line;

std :: getline(file,line);

std :: string nw ="";

int pos = 0;

w.erase(0,1);

w.erase(w.length()-1,1);

for (std :: vector <int> :: iterator i = ans.begin(); i != ans.end(); ++i)

{

nw += w.substr(pos,\*i - pos);

nw += "-";

pos = \*i;

}

nw +=w.substr(pos,w.length() - pos);

vec(nw,a);

}

std :: map <std :: string, int> m;

for (std :: vector <std :: string> :: iterator i = a.begin(); i != a.end(); ++i)

{

if (m.find(\*i) == m.end() )

{

m.insert(std::pair<std :: string,int>(\*i,1));

}

else m.find(\*i)->second += 1;

}

out << m.size();

for\_each(ans.begin(),ans.end(),[](const std :: map <std :: string, int>::value\_type & v){out << v.first << " " << v.second << std :: endl;});

system("pause");

return 0;

};

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ**

1. Алгоритм Ляна-Кнута для расстановки мягких переносов: [Электронный ресурс]., URL: <http://habrahabr.ru/post/138088/> (Дата обращения: 20.10.13)
2. Руководство по стандартной библиотеке шаблонов (STL): [Электронный ресурс]., URL: <http://www.solarix.ru/for_developers/cpp/stl/stl.shtml> (Дата обращения: 20.10.13)
3. Библиотека STL (Standard Template Library): [Электронный ресурс]., URL: <http://www.itshop.ru/Biblioteka-STL-Standart-Template-Library/l9i22387> (Дата обращения: 20.10.13)