Модуль «Библиотека строковых алгоритмов Boost»

Комплексная цель

Получить понятие о проблемах представления строк на разных платформах. Познакомиться с подходами к обобщенному дизайну алгоритмов на строках. Изучить основные группы алгоритмов на строках.

Операции на строках — одна из наиболее распространенных категорий алгоритмов, которая является замечательным кандидатом для включения в стандартную библиотеку языка программирования. Это подтверждают примеры современных объектно-ориентированных языков, таких как Java и C#, не говоря уже о чрезвычайно популярных на сегодняшний день скриптовых языках, таких как Perl, Python, PHP, Ruby, где задачи обработки текста являются приоритетными. Стандартная библиотека C++, однако, располагает лишь инструментами, доставшимися ей в наследство от языка C (см. заголовочные файлы <cctype> и <cstring>), которые, очевидно, не могут полностью использовать все сильные стороны языка C++. Более того, большая их часть просто дублирует возможности стандартного класса C++ для представления строк `std::string`, оставшиеся же не способны эффективно взаимодействовать с ним для решения распространенных задач. Остается, конечно, возможность использовать мощнейшую библиотеку стандартных обобщенных алгоритмов C++ (см. заголовочный файл <algorithm>), рассматривая строку как обыкновенный STL-совместимый контейнер (здесь и далее под «STL» понимается часть стандартной библиотеки C++, основанная на библиотеке STL, созданной Алексом Степановым и его коллегами в Hewlett-Packard, и состоящая из набора обобщенных алгоритмов, контейнеров и итераторов; вообще говоря, эта часть на совпадает с исходной версией Степанова). Однако, как известно, специализированные инструменты для решения конкретной задачи всегда позволяют решать ее — в широком смысле — более эффективно.

Библиотека Boost String Algo предлагает набор обобщенных алгоритмов для решения основных задач обработки строк: обрезка (trimming), изменение регистра, проверка условий относительно содержимого строк: общих («содержит», «начинается с» и т. п.) и классифицирующих («пробельная строка», «числовая строка» и т. п.), функции поиска и заме-

ны, разбиения и слияния. Обобщенность здесь понимается в том смысле, как ее описывал Алекс Степанов: библиотека имеет большой потенциал к повторному использованию, так как не привязана к какому-либо конкретному представлению строк (используется подход, опирающийся на «неявные интерфейсы» в шаблонах C++), но в то же время потери производительности практически не происходит (в частности, за счет использования такой шаблонной техники как «свойства типов», type traits; подробнее с ней можно познакомиться в [C++Templates]).

О допустимых кодировках

При использовании библиотек, работающих с текстом, неизбежно встает вопрос о поддерживаемых кодировках символов: это определяет, какие символы использовать в тексте можно, а какие нельзя, чтобы библиотека работала корректно. С одной стороны, здесь не место для всестороннего обсуждения кодировок и связанных с ними трудностей, с другой, оставить читателя наедине с не слишком дружественным в вопросах обработки текста языком C++ авторы чувствуют себя не в праве. Здесь будет дан необходимый минимум знаний и указаний для работы со строковой библиотекой Boost в случае текста, содержащего кириллические символы.

Конкретная кодировка определяет набор символов и их байтовое представление в памяти компьютера. Все кодировки можно разделить на две большие группы: однобайтовые и многобайтовые. Многобайтовые кодировки раньше использовались только для языков с чрезвычайно большими наборами символов, обычно, иероглифическими. Сегодня, с появлением стандарта Unicode дело обстоит иначе. Unicode это не кодировка, а набор занумерованных символов (он не описывает байтовое представление символов), очень большой набор; он включает и кириллицу. Две наиболее распространенные кодировки, реализующие Unicode это UTF-8 и UTF-16. Наиболее приемлемая однобайтовая кодировка для отображения кирилли-

цы — cp1251 (еще ее называют windows-1251).

Кодировка это еще не все. Очевидно, программе обработки текста нужна некоторая метаинформация. Пример: мы знаем, что символы «ы» и «Ы» некоторым образом связаны между собой (это одна и та же буква русского языка, только в двух вариантах: строчном и прописном). Такого рода информация относится к так называемым «локалям». Локаль это набор настроек, характерных для данного географического региона и данного языка, они включают форматы даты и времени, денежную единицу и языковые особенности. Информация о локалях хранится в операционной системе (говорят, что в ОС установлен некоторый набор локалей). Программа на языке C++, которая должна обрабатывать текст за пределами латинского алфавита, обычно должна запросить информацию о соответствующей локали у ОС. И в этом месте начинаются проблемы, потому что на разных ОС, хотя могут быть установлены одни и те же локали, их названия могут различаться.

Есть два специальных переносимых между разными ОС имени локали: «C» и «» — «классическая» и «системная». Первая устанавливается автоматически, когда начинает работать программа на C++ и содержит минимальную информацию о языках: она достаточна для обработки текстов только в латинском алфавите без учета региональных особенностей стран его использующих. Вторая строка является псевдонимом текущей системной локали, которой должно хватить для обработки кириллического текста, если у вас установлена локализованная для России операционная система. В случае ОС Windows на данный момент такая локаль предусматривает поддержку текстов в кодировке cp1251. На компьютере одного из авторов установлена локализованная ОС GNU/Linux, которая по умолчанию предоставляет возможность обработки текстов в UTF-8. В обоих случаях первая строка вашей программы, использующей возможности рассматриваемой

библиотеки должна содержать строку:

```
locale::global(locale("")); // выставляем системную локаль
                    //как локаль по умолчанию во всей программе
```

Для использования класса `locale` нужно включить заголовок `<locale>`; квалификация имен из пространства имен `std` далее не производится для краткости. После такого вызова метода `global()` все объекты `locale`, которые будут создаваться конструктором без аргументов, будут соответствовать системной локали, а ее реальное имя имя можно узнать так:

```
cout << locale().name();
```

На одной из современных локализованных систем Windows было получено «Russian_Russia.CP1251», а в GNU/Linux «ru_RU.UTF-8». Теперь ваша программа готова к обработке текстов в кодировке, соответствующей системной локали.

Стоит упомянуть теперь о различии при обработке символов в однобайтовых и многобайтовых кодировках в программах C++. В первом случае в программе нужно использовать встроенный тип `char` и специализации шаблонов стандартной библиотеки для него: `string`, `fstream`, `cout`. Во втором — встроенный тип `wchar_t` и специализации для него: `wstring`, `wfstream`, `wcout`; кроме того, если исходный текст программы представлен в многобайтовой кодировке, то строковые литералы, чтобы они были восприняты компилятором как массивы `wchar_t`, должны предваряться символом «L».

Первый пример. Особенности дизайна библиотеки

```
#include <cassert>
#include <boost/algorithm/string.hpp>
using namespace boost::algorithm;
```

```
// ...
wstring str1(L" hello world! ");
to_upper(str1); // к верхнему регистру
assert(str1 == L" HELLO WORLD! ");
trim(str1); // обрезка лидирующих и завершающих пробелов
assert(str1 == L"HELLO WORLD!");

wstring str2=
   to_lower_copy( // приведение к нижнему регистру;
      ireplace_first_copy( // нечувствительная к регистру
         str1,L"hello",L"goodbye"));//замена первого вхождения
assert(str2 == L"goodbye world!");
```

Макросы `assert` стандартной библиотеки здесь и далее поясняют действие вызываемых функций: условия, заключенные в них, всегда истинны (обратите также внимание на неявное преобразование строкового литерала в правой части оператора сравнения на равенство с типу `wstring`). `str2` получается из `str1` последовательно нечувствительной к регистру заменой первого вхождения «hello» на «goodbye» и приведением к нижнему регистру, результаты обеих операций возвращаются как результаты функций, исходная строка остается неизменной. Этот пример демонстрирует некоторые особенности дизайна всей библиотеки:

- *контейнеры как параметры*: один из ключевых принципов дизайна стандартной библиотеки C++ состоит в отделении алгоритмов от контейнеров посредством итераторов, таким образом вы должны писать `for_each(c.begin(), c.end(), ...)` вместо `for_each(c, ...)` (под `c` понимается некоторый контейнер); этим

достигается:

- более высокий уровень обобщенности (например, пара итераторов может описывать как контейнер, так и его часть),
- поддержка встроенных массивов аналогично библиотечным классам,
- привязка к аскетичному интерфейсу итератора (++, *, !=) минимизирует зацепление интерфейсов разных частей библиотеки друг на друга,
- избавление от одного уровня косвенности; влечет за собой потенциально более высокую производительность (например, уменьшаются возможности производить виртуальные вызовы).

[Evolving, 4.1.3]. Стандартная библиотека, таким образом, предпочитает обобщенность перед простотой использования. Однако создатели строковой библиотеки Boost сделали другой выбор. Одним из мотивов такого решения стала возможность формировать стек вызова функций обработки сток, как это показано во второй части примера (в случае двух параметров-итераторов такой подход был бы не реализуем). Однако зачастую возникает ситуация, когда у вас есть только пара итераторов (например, при использовании строковой библиотеки в своей библиотечной функции, которая следует конвенции STL). В этом случае нужно использовать класс `boost::iterator_range`, который упаковывает пару итераторов в структуру, эмитирующую контейнер STL (в частности, подходящую под требования строковой библиотеки):

```
#include <boost/range.hpp>
//...
template<typename ForwardIt>
void f(ForwardIt begin, ForwardIt end) {
```

```
    if (!all(boost::make_iterator_range(begin, end),
                                          is_alpha()))
    throw std::runtime_error("Incorrect range, "
         "f works only with alphabetic characters");
   //...
}
```

`make_iterator_range` создает экземпляр упомянутого класса `iterator_range` (аналогично паре `std::pair` и `std::make_pair`).

- *Копирование или изменение*: многие алгоритмы библиотеки производят преобразования входных строк, которые могут выполняться «на месте», изменяя аргумент, или создавая копию для проведения преобразований, не затрагивая вход. Оба подхода могут быть полезны в разных ситуациях, потому библиотека предоставляет в таких случаях пару функций.

- *Стек вызовов* объединяет возможности первых двух пунктов, позволяя использовать цепочки вызовов функций, передавая возвращаемое значение одной функции на вход другой. Если вы решаете пользоваться возвращаемым значением, то необходимо вызывать копирующую версию функции, поскольку изменяющие вход варианты возвращают `void`, что подчеркивает указанное выше разделение и позволяет повысить эффективность обработки строк, когда операции производятся на месте.

- *Соглашения по именованию.* В целом соглашения по именованию классов и функций следуют стандартной библиотеке C++. Копирующие версии алгоритмов отмечены суффиксом `_copy`, в то время как неизменяющие версии специальных суффиксов не имеют. Многие алгоритмы имеют версии с префиксом `i`, показывающим, что он может работать в не чувствительном к регистру режиме.

- *Алгоритмы и объекты-функции.* Несмотря на отказ от повсеместного использования итераторов, в остальном дизайн библиотеки напоминает STL: выбор контейнеров ложится на пользователя, а алгоритмы предоставляет библиотека. Таким образом, здесь имеется большой набор шаблонов функций (которые во-многом являются обертками над алгоритмами стандартной библиотеки C++). В дополнение к алгоритмам предоставляется ряд объектов-функций, которые параметризуют их (алгоритмов) поведение — снова прямая аналогия с STL.

- *Высокоуровневые и низкоуровневые средства.* Для решения почти каждого класса задач (классификация, поиск и др.) библиотека предоставляет как узкоспециализированные средства, которые просты в использовании, так и довольно общие, способные к тонкой настройке. Первые обыкновенно можно реализовать через вторые, передав им соответствующим образом сформированные параметры, которые инкапсулируют различные фазы работы соответствующих алгоритмов.

Все основные средства библиотеки строковых алгоритмов собраны в заголовочном файле `<boost/algorithm/string.hpp>`, дополнительные средства, поддерживающие регулярные выражения, — в файле `<boost/algorithm/string_regex.hpp>`. Последние зависят от другой библиотеки Boost, посвященной непосредственно регулярным выражениям, и рассматриваться не будут.

Составляющие библиотеки с примерами использования

При перечислении функций библиотеки будут применяться БНФ-подобные обозначения, чтобы учитывать опциональные суффиксы (квадратные скобки обозначают повторение ноль или один раз) или возможность

альтернативы (круглые скобки с вариантами через вертикальную черту). В разделах, посвященных предикатам, перечисляются не библиотечные классы предикатов, а функции, предназначенные для их создания, которые и следует использовать вместо явного вызова конструкторов классов (что потребовало бы указания шаблонных параметров, в то время как в случае функций они выводятся). Возвращаемый ими объект предиката может использоваться далее внутри библиотечной функции или непосредственно клиентским кодом для проверки тех или иных условий при помощи вызова `operator()`.

Изменение регистра. `to_(lower | upper)[_copy]`.

Это функции с наиболее очевидной семантикой, пример использования уже был приведен выше.

Предикаты для символов.

`is_classified;`

`is_alnum, is_alpha, is_cntrl, is_digit, is_graph, is_lower, is_print, is_punct, is_upper, is_xdigit;`

`is_any_of, is_from_range;`

`operator&&, operator||, operator!`

Первый предикат не даром занимает свое место: он проверяет соответствие переданного символа указанной в параметре маске типа `std::ctype_base::mask` (стандартный тип для классификации символов), следующий блок предикатов (вплоть до `is_xdigit`), в свою очередь оборачивают вызов `is_classified` с нужными масками, так что маски стандартной библиотеки C++ для разных классов символов использовать явно необязательно. `is_any_of` получает в качестве параметра контейнер символов, и полученный предикат будет выдерживаться, если проверяемый символ содержится в этом контейнере. Аналогично действует `is_from_range`, только вместо контейнера передается пара символов, задающих диапазон

(напомним, что каждая фиксированная кодировка определяет нумерацию символов, так что на множестве символов есть отношение порядка, индуцированное этими номерами), принадлежность которому будет проверять предикат. Логические операции позволяют конструировать новые предикаты на основе уже имеющихся, как будет показано ниже.

Предикаты для строк.

`[i](starts | ends)_with, [i]contains, [i]equals, [i]lexicographical_compare, all.`

Позволяют проверять наличие префиксов и суффиксов строк, содержание подстроки, сравнивать на равенство и лексикографически, проверять удовлетворение некоторого условия (переданного в виде предиката для символа) для каждого символа строки. Все функции без префикса `i` есть дополнительная версия, получающая наряду с двумя строками предикат для сравнения двух символов.

```
bool is_executable( wstring const & filename ) {
    return iends_with(filename, L".exe") ||
         iends_with(filename, L".com");
}
/* ... */ #include <sstream> /* ... */
wstring comcom(L"command.com");
wostringstream oss_com; // выходной строковый поток
oss_com << comcom
     << (is_executable(comcom)? L" is": L" is not")
     << L" an executable";
assert( wstring(oss_com.str()) // содержимое oss_com
      == L"command.com is an executable" );
wstring hw(L"Hello, world");
wostringstream oss_hw;
```

```
oss_hw << hw << L" is written"
     << ( all( hw, !is_digit() ) ? L" without" : L" with" )
     << L" a digit";
assert( wstring(oss_hw.str())
     == L"Hello, world is written without a digit");
```

Обрезка. `trim[_(left | right)][_copy][_if]`

Простейший пример использования одного из алгоритмов этого семейства был приведен: `trim` удаляет пробелы на обоих концах строки. Как видно, можно указать только один конец, а также передать предикат для символов, которые будут удалены (в версии с `if`).

```
assert(
     trim_left_copy_if(wstring(L"0012300"),is_any_of(L"0"))
     == L"12300");
```

Поиск.

`[i]find_(fist | last | nth); find_(head | tail); find_token; find.`

Начать с того, что результат всех алгоритмов данной категории это экземпляр шаблонного класса `boost::iterator_range`, с которым мы уже встречались: он оборачивает пару итераторов, отмечающих вхождение найденного промежутка в исходном контейнере и также может быть неявно приведен к `bool` или явно проверен на пустоту с помощью метода `empty()` — результат двух этих операций одинаков и в случае значения `false` говорит, что поиск ничего не дал.

Первая группа функций просто ищет подстроку, задаваемую вторым параметром, в первом: соответственно, первое, последнее или n-ое ее вхождение, где n начинается с 0 (то есть вызов `find_nth(s1, s2, 0)` эквивалентен `find_first(s1, s2)`). Следующая группа позволяет получить

диапазон, соответствующий первым (head) или последним (tail) n символам строки (здесь n действительно обозначает число символов, то есть вызов функции с параметром 0 вернет пустой диапазон). Особенность этой функции в том, что ее работа определена для n, больших длины строки (тогда возвращается диапазон, содержащий всю строку) и для отрицательных (тогда строка индексируется с конца, то есть для строки s длиной 6 символов find_head(s, -2) вернет диапазон, указывающий на первые четыре символа).

find_token кроме строки, в которой осуществляется поиск получает предикат для символа, который позволяет выделять часть строки — токен; особо важен третий параметр, который определяет, нужно ли собирать подряд идущие символы, которые удовлетворяют предикату, в один токен или считать их разными токенами: его значение должно быть одной из предопределенных констант: token_compress_on или token_compress_off, соответственно, по умолчанию передается второе значение.

Наконец, find это низкоуровневое средство, которое является минимальной оберткой для своего единственного кроме обрабатываемой строки параметра объекта-функции, operator() которого принимает пару итераторов, а возвращает результат поиска. Этот объект-функция в рамках библиотеки обозначается именем Finder (его можно встретить читая справочное руководство — как имя шаблонного параметра). Все перечисленные выше функции для поиска скрывают вызов find с объектом одного из библиотечных классов; их при желании можно использовать и непосредственно: имя каждого из этих классов образуется из имени соответствующей функции, к которому добавляется окончание er. Это решение позволяет библиотеке выдерживать ряд своих решений в отношении дизайна (в частности, перечисленных выше). Пример:

```
wstring text(L"Search with Google!");
boost::iterator_range<wstring::iterator> result =
     find_first(text, L"oo");
to_upper(result);
assert(text == L"Search with GOOgle!");
wostringstream oss;
oss << L"GOOgle"
     << (find_first(text, L"GOOgle") ? L"" : L" not")
     << L" detected!"; //iterator range is convertible to bool
assert( wstring(oss.str()) == L"GOOgle detected!");
```

```
find_iterator, make_find_iterator;
split_iterator, make_split_iterator.
```

Часто возникает необходимость последовательно обрабатывать все вхождения искомой строки. Или же все промежуточные области строки, располагающиеся между вхождениями искомой строки. Для этого введены два указанных итераторах вместе с функциями их создания, которые принимают целевую строку и объект класса типа `Finder`, о котором упоминалось выше. Пример:

```
wstring str(L"abc-*-ABC-*-aBc");
assert( true == std::inner_product(
          make_find_iterator(str,
               first_finder(L"abc", is_iequal())),
          find_iterator<wstring::iterator>(), //'end-iterator'
          make_split_iterator(str,first_finder(L"-*-")),
          true, //init value for grouping &&
          std::logical_and<bool>(), // grouping operator -- &&
          is_equal() ) // by component operator -- ==
```

```
);
```

В этом примере создаются оба рассматриваемых типа итераторов. Итератор поиска проходит по частям строки, совпадающим без учета регистра с «abc»: конструктор объекта `Finder`, определяющего логику поиска, принимает искомую строку и, вообще говоря (см. пример далее), опциональный предикат сравнения символов (из предоставляемых библиотекой действительно представляет интерес лишь использованный в примере: сравнение без учета регистра). Разрезающий итератор (split-итератор) проходит по частям строки, находящимся между «-*-». Таким образом, оба итератора проходят по одним и тем же частям строки. Этот факт выявляется с помощью применения стандартного алгоритма `inner_product`, соответствующему традиционному скалярному произведению (inner product или «внутреннему произведению» в западной терминологии), где вместо произведения («группирующей» операции) используется сравнение на равенство, а сложения («покомпонентной» операции) — конъюнкция.

В результате итераторы проходят по компонентам воображаемых векторов, которые (компоненты) являются одними и теми же частями исходной строки; эти части сравниваются на равенство, а результат булевски умножается на то, что получилось в результате предыдущего шага. В таком алгоритме необходимо начальное значение (оно используется в конъюнкции с результатом первого сравнения), которое и передается алгоритму вместе с двумя объектами-функций, определяющими указанные операции. Обратите внимание, что в качестве завершающего перебор найденных частей (как, впрочем, и разрезанных частей, но в примере этого не видно) строки итератора используется итератор, полученный вызовом конструктора без параметров соответствующего класса итератора. Это прямая аналогия с итераторами потоков стандартной библиотеки C++ — строковая библиотека и здесь следует ее канонам.

Знакомый со стандартной библиотекой C++ человек может спросить: нельзя ли было в качестве предиката сравнения использовать стандартный `std::equal_to`, который часто применяется в подобных случаях. Ответ положительный. Хотя `equal_to` более ограничен чем `is_equals` в следующем: сравниваемые операнды должны иметь один тип — в данном случае результаты разыменования двух рассматриваемых итераторов могут быть приведены к одному типу `boost::iterator_range<wstring::iterator>`, который и следовало бы указать в качестве шаблонного параметра при создании `equal_to`, но дизайн `is_equals` кроме того что в общем случае позволяет сравнивать операнды разных типов (при условии, что для них будет найден подходящий `operator==`), дает возможность не указывать никаких параметров при создании, так как сам класс предиката не шаблонный (шаблонным является его `operator()`, как уже обсуждалось ранее). Последнее делает `is_equals` более удобным в использовании и облегчает чтение полученного кода.

`[i]find_all, split.`

Если нужен не последовательный доступ с помощью итератора, а хочется получить все найденные части строки сразу, то можно использовать одну из этих функций. Первым параметром они получают STL-подобный контейнер, способный хранить либо копии найденных частей строк, то есть, к примеру, `std::string, std::wstring` или `iterator_range`. Обе версии `find` ищут точное вхождение одной строки (из третьего параметра) в другой (из второго параметра). `split` третьим параметром получает предикат для символа, который будет выделять разрезающие токены. Более гибкими аналогами этих алгоритмов, выполняющими задачами являются следующие.

`iter_split, iter_find.`

Они отличаются от трех предыдущих третьим параметром, который должен удовлетворять требованиям `Finder`, то есть выделять подстроки по

некоторому алгоритму: `iter_find` занесет их в полученный контейнер, а `iter_split` — промежутки между ними.

Замена (удаление) подстроки. Как правило, после поиска с результатами производятся некоторые действия. Наиболее типичными являются операции замены и удаления найденных фрагментов, и обсуждаемый в этом разделе набор алгоритмов позволяет автоматизировать этот шаг в задаче обработки строк. Следующие алгоритмы являются прямыми наследниками соответствующих версий `find_*`:

```
[i](replace | erase)_(fist | last | nth)[_copy];
(replace | erase)_(head | tail); (replace | erase)_all[_copy].
```

Функции `erase` эквивалентны замене на пустую строку. `replace` по сравнению с соответствующими версиями `find` имеют соответствующую семантику и получают на один (последний в списке) аргумент больше, он и определяет на что заменять найденный фрагмент. Для копирующих вариантов предусмотрены две версии одна из них получает также, как неизменяющая, три аргумента (где, что, на что менять) и возвращает копию переданной строки с внесенным изменениями; вторая версия получает на один аргумент больше (он идет первым в списке), он представляет собой итератор (`OutputIterator` в терминологии стандартной библиотеки), который будет использоваться для запаси результата замены. Эта вторая версия вернет итератор, указывающий на элемент, следующий за последним записанным символом.

При использовании алгоритмов поиска вы могли получить пару итераторов, обозначающих вхождение искомого фрагмента. Для замены этого, уже имеющегося фрагмента предназначены следующие функции:

```
(replace | erase)_range[_copy]
```

Вместо параметра, задающего искомую строку он получает экземпляр `iterator_range`. Для копирующих вариантов, как и выше, преду-

смотрены две версии.

Наиболее гибкими средствами, решающими задачи замены, являются следующие алгоритмы:

`find_format[_all][_copy]`

Как ясно, для замены в строке нужно три параметра: строка, часть для замены и содержимое замены. Если в первом блоке алгоритмов все три параметра задавались строками, во втором часть для замены задавалась двумя итераторами, то здесь строкой задается только первый компонент этой тройки, а следующие два — объектами-функций, обладающими семантикой `Finder` и `Formatter`, соответственно. С первым из их мы уже знакомы, а второй должен предоставлять `operator()` получающий заменяемую часть строки и возвращающий результат замены.

Как и в других случаях, касающихся конфигурируемых частей алгоритмов, инкапсулированных в объектах-функций (или, аналогичным образом, в масках, определяющих типы символов), изначально библиотека предоставляет лишь те классы, которые уже были задействованы для создания специализированных версий алгоритмов включенных в библиотеку. Таким образом, для получения нового функционала нужно написать новый объект-функции и передать его функции, предназначенной для этого (`find_fornat`, `find`).

Объединение строк. `join[_if]`

Двойственная к разбиению функциональность: первый аргумент должен содержать контейнер объединяемых строк, второй — строку-разделитель (будет вставляться между объединяемыми строками), `if`-версия третьим аргументом принимает предикат для строк, чтобы фильтровать объединяемые строки.

Проектное задание

Создайте объект-функции, удовлетворяющий требованиям `Finder`, который инкапсулирует один из более эффективных алгоритмов поиска строки в подстроке; описание таких алгоритмов можно найти в главе 34 [Cormen]. Примените его в комбинации с обобщенными алгоритмами строковой библиотеки, которые допускают это (`boost::find`, `boost::find_format`). Оцените практический прирост в быстродействии относительно ожидаемого теоретического прироста, учитывая, что в текущей версии библиотеки Boost (1.36) реализован самый примитивный алгоритм сложностью O(n*m).

Тест рубежного контроля

1. Какую кодировку нельзя использовать для представления кириллических символов:

- UTF-8
- cp1251
- UTF-16
- cp1250
- windows-1251

2. Какое понятие включает в себя особенности данного географического региона:

- кодировка
- локаль
- операционная система

3. Какое соображение повлияло на решение об использовании контейнеров в качестве параметров строковых алгоритмов:

- формирование стека вызовов алгоритмов
- обобщенность
- соответствие стандартной библиотеке

4. Какой предикат для символов является самым низкоуровневым из имеющихся:

- is_any_of
- is_classified
- operator!

5. Что произойдет, если алгоритму `find_head(s, n)` в качетве второго параметра передать отрицательное число:

- будут возвращены первые `s.size()-n` символов строки `s`
- будет сгенерировано исключение `boost::illegal_index`
- будут возвращены `std::abs(n)` символов строки `s`

Модуль «Умные указатели Boost (Boost.Smart_Ptr)»

Комплексная цель

Познакомиться с основными трудностями в управлении ресурсами программы. Изучить возможные стратегии управления ресурсами и их реализации в C++. Рассмотреть преимущества встроенных механизмов C++ для решения задачи управления ресурсами и их ограничения.

Одним из важнейших вопросов разработки программного обеспечения является управление ресурсами. *Ресурсом* называется все, что можно «запросить» и, при успешном получении, после использования, «вернуть». Ресурсы может предоставлять операционная система (дескрипторы файлов на диске, оперативная память, сетевые соединения, потоки выполнения), программное обеспечение промежуточного уровня (соединения с базой данных, транзакции); ресурсы могут существовать в программе независимо и самостоятельно распределять доступ к некоторым сервисам на основе какой-либо стратегии (мьютекс, обеспечивающий *экс*клюзивный доступ к стандартному потоку вывода для разных потоков выполнения).

Умные указатели Boost ориентированны на один вид ресурсов — оперативную память, хотя некоторые из них могут использоваться и в других случаях. Зачастую приходится создавать специализированные классы для управления другими видами ресурсов, однако некоторые проблемы и подходы к их решению, возникающие в этой области, являются довольно общими и будут рассмотрены ниже, а в качестве иллюстрации будут использованы классы Boost (часто под словом «класс» подразумевается шаблон класса — там, где это не может привести к путанице).

При использовании ресурсов возникают две основные задачи:

- корректное совместное использование ресурса в разных частях программы,
- корректное освобождение ресурса.

Совместное использование может осуществляться в соответствии с разными стратегиями, несколько основных реализованы в классах Boost.Smart_Ptr. Корректное освобождение реализуется при помощи идиомы *Resource Aquisition Is Initialization* (захват ресурса при инициализации, RAII)[TC++PL, 14.4], основанной на симметрии конструкторов и деструкторов C++. RAII использует тот факт, что для каждого локального объекта

программы, который был полностью сконструирован в данном блоке программы, при выходе из этого блока будет вызван деструктор. Важно то, что «выход из блока» в данном случае это одно из трех:

- достижение соответствующей «}»,
- выход по любому из имеющихся в блоке `return`,
- выход в результате возникновения исключительной ситуации.

Таким образом, чтобы быть уверенным в освобождении ресурса, нужно связать его со специальным объектом, который выполняет захват ресурса в своем конструкторе (или, менее желательно, в специальном «инициализирующем» методе) , а освобождение ресурса — в деструкторе.

Рассмотрим два фрагмента исходного кода:

```
{ Matrix* mp = new
Matrix(300,200);

fillMatrix(mp);
cout << "Det: " << mp->det();
delete mp; }
```

```
{ scoped_ptr<Matrix> msp =
scoped_ptr<Matrix>(new
Matrix(300,200));
fillMatrix(msp.get());
cout <<"Det: "<< msp->det();
}
```

В левой части демонстрируется создание в динамической памяти и работа с объектом типа `Matrix` посредством обыкновенного указателя, в правой — аналогичный код, но с применением умного указателя Boost. В последнем случае сразу после того, как `new` закончит свою работу, вновь созданный на куче объект `Matrix` попадает во владение умного указателя, который по достижении конца блока (естественным образом или после выброса исключения, например, из `fillMatrix` — что не вызовет удивления, если он обрабатывает ввод пользователя) сам выполнит `delete`. В случае возникновения исключительной ситуации фрагмент в левой части вызовет утечку памяти.

Обращение с умным указателем во многих случаях не отличается от обычного, что видно в выражении с вызовом метода `Matrix::det`. Это достигается тем, что во всех классах умных указателей есть перегруженные операции `operator*` и `operator->`. Наличие интерфейсов, рассчитанных на обычные (их еще называют «сырые», raw) указатели не должно останавливать перед использованием умных указателей: в этой ситуации можно использовать метод умного указателя `get`, как в примере выше. Однако злоупотреблять этим методом не стоит, так как предоставление непосредственного доступа к ресурсу может затруднить общее управление его использованием, о чем пойдет речь дальше.

Еще три функции-члена, которыми обладают все классы умных указателей Boost: `reset()` с опциональным параметром-указателем, вызов которой вызывает удаление старого ресурса и, если передан аргумент, прием во владение нового; `swap()`, обеспечивающая более эффективный, чем стандартная реализация шаблона `std::swap`, обмен двух экземпляров умных указателей (кроме того, в пространстве имен `boost` объявлена перегрузка свободной функции `swap` для классов умных указателей); неявное преобразование к логическому типу, которое показывает, находится ли какой-либо ресурс во владении данного объекта или нет. Поясним, что ситуация «объект без ресурса» является вполне корректной при традиционном использовании умных указателей для управления памятью(RAII говорит, что не должно возникать противоположной ситуации «ресурс без объекта»(управляющего им)): в таком случае обычный указатель, который хранится внутри умного имеет значение нуль, в деструкторе вызовется оператор `delete` для нулевого указателя, что с точки зрения стандарта C++ не является ошибкой и не влечет за собой каких-либо отрицательных последствий.

Совместное использование ресурса. Типы умных указателей

Когда ресурс получен в данной точке программы, возникает вопрос, можно ли передавать его в другие части программы и если да, то кто в конечном итоге должен быть ответствен за его возврат. Под передачей ресурса подразумевается копирование управляющего объекта, а не копирование самого ресурса (что чаще всего невозможно: например, нельзя скопировать оперативную память, можно скопировать ее содержимое). Таким образом, вопрос о совместном владении ресурсом в идиоме RAII определяется семантикой копирующих операторов (конструктора копий и `operator=`) управляющего класса. Вот некоторые подходы к реализации этих операций с примерами реализующих их шаблонов классов умных указателей.

1. Запрет на копирование, `boost::scoped_ptr`. Ресурс не может быть передан в другие части программы (функции и методы классов), при попытке вызова копирующей операции возникнет ошибка компиляции.

2. Передача владения, `std::auto_ptr`. В момент вызова копирующей операции новый объект становится владельцем ресурса, в то время как старый теряет доступ к нему. В случае с умными указателями исходный указатель «зануляется». Такой тип умного указателя, как видно, определен в стандартной библиотеке C++ (заголовок `<memory>`). Такое поведение является неочевидным и потому легко вызывает ошибки. К тому же его **нельзя** использовать совместно с контейнерными классами стандартной библиотеки, которые зачастую для оптимизации делают дополнительные копии своих элементов.

3. Совместное владение ресурсом с подсчетом ссылок, `boost::shared_ptr`, `boost::intrusive_ptr`. Чаще всего это наиболее удобный в использовании подход, хотя он связан с некоторыми накладными ресурсами при подсчете копий объектов, «заведующих»

одним ресурсом. Операция освобождения ресурса будет автоматически вызвана, когда счетчик станет равным нулю. `boost::intrusive_ptr` предполагает, что механизм подсчета ссылок реализован в самом классе-параметре шаблона `T`: при вызове копирующих операций этот умный указатель производит неквалифицированный вызов функции `intrusive_ptr_add_ref(T*)` — компилятор должен найти соответствующую перегрузку (чаще всего ее следует определять в одном пространстве имен с каждым конкретным `T`). Аналогично в деструкторе выполняется вызов `intrusive_ptr_release(T*)`.

`boost::shared_ptr` работает с любыми классами, удовлетворяющими базовым требованиям всех умных указателей (деструктор не должен выбрасывать исключений) и является самым распространенным инструментом обеспечения тех условий, которые требуются от умных указателей.

Есть еще три класса умных указателей: `scoped_array`, `shared_array`, `weak_ptr`. Назначение первых двух очевидно и происходит из того факта, что для освобождения выделенной памяти в C++ существует две формы оператора `delete`, употребляемых в зависимости от того, выделялась ли память под один объект или под массив объектов; политика управления ресурсом аналогична `scoped_ptr` и `shared_ptr`, соответственно. С третьим дело обстоит сложней: он выполняет роль обозревателя разделяемого ресурса. Слово «разделяемый» здесь указывает на связь с `shared_ptr`, конструируется `weak_ptr` только из него (или из другого `weak_ptr`). Особенность этого типа в том, что непосредственного доступа к ресурсу с его помощью получить нельзя (операции `*` и `->`, как и метод `get`, не определены), для доступа нужно использовать метод `lock`, возвращающий `shared_ptr` или конструктор `shared_ptr` с параметром `weak_ptr`. `weak_ptr` не увеличивает количества ссылок на разделяемый

объект, таким образом, когда разрушается последний `shared_ptr`, связанный с данным ресурсом, этот ресурс возвращается, а `weak_ptr` автоматически «зануляется» (таким образом, `weak_ptr` это Обозреватель в смысле паттерна [GoF]), это состояние можно проверить при помощи метода `expired`.

Когда нужно хранить `weak_ptr`, строя при необходимости с его помощью `shared_ptr` (как описано выше), вместо того, чтобы просто хранить `shared_ptr`? Этот вопрос не имеет общего ответа и должен решаться в каждом конкретном случае. Данный шаблон возник для того, чтобы компенсировать недостатки `shared_ptr`, которые не проявляются в простых случаях. `shared_ptr` — хорошее решение, когда время жизни управляемого ресурса довольно велико по отношению к коду программы, то есть используется во многих его местах: `shared_ptr` передается между кадрами стека и наверняка обеспечит освобождение ресурса, когда исчезнет последняя ссылка на него; такое поведение похоже на работу *сборщика мусора*, которыми оснащены некоторые языки в отличие от C++. Однако сборщик мусора обычно более умен, чем умные указатели C++. Например, он может разрывать так называемые *циклические ссылки*: находить случаи, когда два (или более) объекта указывают друг на друга, но уже не доступны программе, и удалять такие объекты. Так как счетчик ссылок каждого из этих объектов отличен от нуля, `shared_ptr` не станет их удалять. `weak_ptr`, в частности, помогает избежать возникновения циклических ссылок (обозреватель не увеличивает счетчик ссылок на ресурс и не препятствует его возврату); однако места программы, где они могут появиться, вам надо выявлять самостоятельно.

Проектное задание

Смоделируйте ситуацию циклических ссылок при использовании `boost::shared_ptr` и продемонстрируйте ее разрешение при помощи `boost::weak_prt`. Создайте контейнер, аналогичный по семантике std::vector, но безопасный относительно хранения std::auto_ptr.

Тест рубежного контроля

1. Что из перечисленного не является ресурсом:
   - дексрипторы файлов на диске
   - память на стеке программы
   - динамическая память программы(куча)
   - соединение с базой данных
2. Какую задачу решает идиома RAII:
   - задача получения ресурса
   - задача совместного использования ресурса
   - задача освобождения ресурса
3. Какая из перечисленных функций-членов является специфичной для указателя `boost::weak_ptr`:
   - `reset`
   - `swap`
   - `lock`
   - приведение к логическому типу
4. Какой из умных указателей реализует подсчет ссылок для произвольных классов:
   - `scoped_ptr`
   - `std::auto_ptr`
   - `shared_ptr`
   - `intrusive_ptr`
   - `weak_ptr`

Библиография

- [C++Templates] — Д. Вандевурд, Н. Джосаттис. Шаблоны C++: справочник разработчика. — М.: «Вильямс», 2003
- [Cormen] — Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы. Построение и анализ, 2-е изд., пер. с англ. — М.: «Вильямс», 2005.
- [Evolving] — Stroustrup, B. 2007. Evolving a language in and for the real world: C++ 1991-2006. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California, June 09 - 10, 2007). HOPL III. ACM, New York, NY, 4-1-4-59.
- [GoF] — Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования, пер. с англ. — СПб: «Питер», 2001.
- [TC++PL] — Б. Страуструп. Язык программирования C++, 3-е изд., пер. с англ. — СПб.; М.: «Невский диалект» — «Издательство БИНОМ», 1999.