Строки
Конструкторы строк

Выражение	Действие
`string s`	Создает пустую строку *s*
`string s(str)`	Копирующий конструктор; создает строку, являющуюся копией существующей строки *str*
`string s(rvStr)`	Перемещающий конструктор; создает строку и перемещает в нее содержимое строки *rvStr* (строка *rvStr* должна иметь корректное состояние, а после перемещения ее значение не определено)
`string s(str,stridx)`	Создает строку *s*, инициализированную символами строки *str*, начинающимися с индекса *stridx*
`string s(str,stridx,strlen)`	Создает строку *s*, инициализированную не более чем *strlen* символами строки *str*, начинающихся с индекса *stridx*
`string s(cstr)`	Создает строку *s*, инициализированную С-строкой *cstr*
`string s(chars,charslen)`	Создает строку *s*, инициализированную *charslen* символами массива символов *chars*
`string s(num,c)`	Создает строку, имеющую *num* вхождения символа *c*
`string s(beg,end)`	Создает строку, инициализированную всеми символами диапазона [*beg*,*end*)
`string s(initlist)`	Создает строку, инициализированную всеми символами из списка инициализации (начиная со стандарта C++11)

Строки
Доступ к элементам

```cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string str = "abcde";
    char symb = str[1];
    cout << symb << endl; // выведет b
    char symb2 = str.at(4);
    cout << symb2 << endl; // выведет e
    //char symb3 = str.at(15); // сгенерирует исключение
    char symb4 = str.front(); // a
    char symb5 = str.back(); // e
    char symb6 = str[str.length()]; // символ конца строки \0
    char &r = str[3]; // ссылка на элемент 'd'
    char *p = &str[3]; // указатель на элемент 'd'

    return 0;
}
```

Строки
Присваивание

```cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string str = "abcdefg";
    string s;

    s = str; // присваиваем "abcdefg"
    s = "two\nlines"; /* two и lines будут на разных строках, потому что
                         \n - символ перехода на следующую строку */
    s.assign(str); // присваиваем "abcdefg"
    s.assign(str, // исходная строка
             2,   // начиная с символа с индексом 2, т.е. 'c'
             3);  // взять 3 символа
                  // получим "cde"
    s.assign(str,            // исходная строка
             2,              // начиная с символа с индексом 2, т.е. 'c'
             string::npos);  // до конца строки
                             // получим "cdefg"
    s.assign(5, 'j'); // получим "jjjjj"
    cout << s << endl;

    return 0;
}
```

Строки. Вставка и удаление символов

```cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string str = "abcdefg";
    string s; // изначально пустая строка

    s += str; // получим "abcdef"
    s += "two\nlines"; // получим "abcdeftwo\nlines"
    s += {'o', 'k'}; // получим "abcdeftwo\nlinesok"

    s.clear(); // удаление всех символов строки
    s.append(str); // получим "abcdefg"
    s.append(str,  // исходная строка
             2,    // начиная с символа с индексом 2, т.е. 'c'
             3);   // взять 3 символа
                   // получим "abcdefgcde"
    s.append(str,            // исходная строка
             2,              // начиная с символа с индексом 2, т.е. 'c'
             string::npos);  // до конца строки
                             // получим "abcdefcdecdefg"
    s.append(5, 'j'); // получим "abcdefcdecdefgjjjjj"
    s.push_back('x'); // получим "abcdefcdecdefgjjjjjx"
    cout << s << endl;

    return 0;
}
```

Строки. Вставка и удаление символов

```cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string str = "age";
    string s("p");
    s.insert(1, str); // "page"
    s.insert(2, "ssa"); // "passage"
    s.replace(0, 5, "ran"); // "range"
    s.erase(3); // удаление всех символов, начиная с 3-го, получим "ran"
    s = "international";
    s.erase(7, 5); // удалить пять символов, начиная с 7-го, получим "internal"

    cout << s << endl;

    return 0;
}
```

Строки
Извлечение подстрок

```cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string str = "interchangeability";

    cout << str.substr() << endl; // вся строка str
    cout << str.substr(11) << endl; // строка, начиная с 11-го символа, "ability"
    cout << str.substr(5, 6) << endl; // 6 символов, начиная с 5-го, "change"
    cout << str.substr(str.find('c')); // все символы, начиная с 'c'
                                       // получим "changeability"

    return 0;
}
```

Строки
Поиск подстрок

Строка	Действие
`find()`	Выполняет поиск первого вхождения *value*
`rfind()`	Выполняет поиск последнего вхождения *value* (поиск в обратном направлении)
`find_first_of()`	Выполняет поиск первого символа, который является частью *value*
`find_last_of()`	Выполняет поиск последнего символа, который является частью *value*
`find_first_not_of()`	Выполняет поиск первого символа, не являющегося частью *value*
`find_last_not_of()`	Выполняет поиск последнего символа, не являющегося частью *value*

Строки
Поиск подстрок

```cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string str = "Hi Bill, I'm ill, so please pay the bill";

    str.find("il"); // возвращает 4 (первая подстрока "il")
    str.find("il", 10); // возвращает 13 (первая подстрока "il", начиная ч 10-го символа)
    str.rfind("il"); // возвращает 37 (последняя подстрока "il")
    str.find_first_of("il"); // возвращает 1 (первый символ 'i' или 'l')
    str.find_last_of("il"); // возвращает 39 (последний символ 'i' или 'l')
    str.find_first_not_of("il"); // возвращает 0 (первый символ, не равный ни 'i', ни 'l')
    str.find_last_not_of("il"); // возвращает 36 (последний символ, не равный ни 'i', ни 'l')
    str.find("hi"); // возвращает npos

    return 0;
}
```

Числовые преобразования строк

Строковая функция	Действие
`stoi(`*str*`, `*idxRet*`=nullptr, `*base*`=10)`	Преобразует *str* в тип `int`
`stol(`*str*`, `*idxRet*`=nullptr, `*base*`=10)`	Преобразует *str* в тип `long`
`stoul(`*str*`, `*idxRet*`=nullptr, `*base*`=10)`	Преобразует *str* в тип `unsigned long`
`stoll(`*str*`, `*idxRet*`=nullptr, `*base*`=10)`	Преобразует *str* в тип `long long`
`stoull(`*str*`, `*idxRet*`=nullptr, `*base*`=10)`	Преобразует *str* в тип `unsigned long long`
`stof(`*str*`, `*idxRet*`=nullptr)`	Преобразует *str* в тип `float`
`stod(`*str*`, `*idxRet*`=nullptr)`	Преобразует *str* в тип `double`
`stold(`*str*`, `*idxRet*`=nullptr)`	Преобразует *str* в тип `long double`
`to_string(`*val*`)`	Преобразует *val* в тип `string`
`to_wstring(`*val*`)`	Преобразует *val* в тип `wstring`

Строковые операции над итераторами

Выражение	Действие
`s.begin(), s.cbegin()`	Возвращает итератор произвольного доступа, установленный на первый символ
`s.end(), s.cend()`	Возвращает итератор произвольного доступа, установленный на последний символ
`s.rbegin(), s.crbegin()`	Возвращает обратный итератор, установленный на первый символ при обратном обходе (т.е. последний символ при прямом обходе)
`s.rend(), crend()`	Возвращает обратный итератор, установленный на позицию, следующую за последним символом при обратном обходе (т.е. на позицию, предшествующую первому символу при прямом обходе)
`string s(`*beg*`,`*end*`)`	Создает строку, инициализированную всеми символами диапазона [*beg*,*end*)
`s.append(`*beg*`,`*end*`)`	Добавляет все символы диапазона [*beg*,*end*)
`s.assign(`*beg*`,`*end*`)`	Присваивает все символы диапазона [*beg*,*end*)
`s.insert(`*pos*`,`*c*`)`	Вставляет символ *c* в позицию итератора *pos* и возвращает позицию итератора, установленного на новый символ
`s.insert(`*pos*`,`*num*`,`*c*`)`	Вставляет *num* вхождений символа *c* в позицию итератора *pos* и возвращает позицию итератора, установленного на первый новый символ
`s.insert(`*pos*`,`*beg*`,`*end*`)`	Вставляет все символы диапазона [*beg*,*end*) в позицию итератора *pos*
`s.insert(`*pos*`,`*initlist*`)`	Вставляет все символы списка инициализации *initlist* в позицию итератора *pos* (начиная со стандарта C++11)

Строковые операции над итераторами

`s.erase(`*pos*`)`	Удаляет символ, на который ссылается итератор *pos*, и возвращает позицию следующего символа
`s.erase(`*beg*`,`*end*`)`	Удаляет все символы диапазона [*beg*,*end*) и возвращает позицию следующего символа
`s.replace(`*beg*`,`*end*`,`*str*`)`	Заменяет все символы диапазона [*beg*,*end*) символами строки *str*
`s.replace(`*beg*`,`*end*`,`*cstr*`)`	Заменяет все символы диапазона [*beg*,*end*) символами С-строки *cstr*
`s.replace(`*beg*`,`*end*`,`*cstr*`,`*len*`)`	Заменяет все символы диапазона [*beg*,*end*) *len* символами массива символов *cstr*
`s.replace(`*beg*`,`*end*`,`*num*`,`*c*`)`	Заменяет все символы диапазона [*beg*,*end*) *num* вхождений символа *c*
`s.replace(`*beg*`,`*end*`,`*newBeg*`,`*newEnd*`)`	Заменяет все символы диапазона [*beg*,*end*) всеми символами диапазона [*newBeg*,*newEnd*)
`s.replace(`*beg*`,`*end*`,`*initlist*`)`	Заменяет все символы диапазона [*beg*,*end*) значениями списка инициализации *initlist* (начиная со стандарта C++11)

Пример использования строковых итераторов

```cpp
#include <string>
#include <iostream>
#include <algorithm>
#include <cctype>
#include <regex>
using namespace std;

int main()
{
    string s("The zip code of Braunschweig in Germany is 38100");
    cout << "original: " << s << endl;
    // переводим все символы в нижний регистр
    transform (s.begin(), s.end(),                // источник
               s.begin(),                           // получатель
               [] (char c) {return tolower(c);});   // операция
    cout << "lowered:  " << s << endl;
    // переводим все символы в верхний регистр
    transform (s.begin(), s.end(),                // источник
               s.begin(),                           // получатель
               [] (char c) {return toupper(c);});   // операция
    cout << "uppered:  " << s << endl;
    // выполняем поиск подстроки "Germany" без учета регистра
    string g("Germany");
    string::const_iterator pos;
    pos = search (s.begin(),s.end(),        // строка, в которой идет поиск
                  g.begin(),g.end(),        // строка, которую ищем
                  [] (char c1, char c2) {   // критерий сравнения символов
                      return toupper(c1) == toupper(c2);});
    if (pos != s.cend()) {
        cout << "substring \"" << g << "\" found at index "
             << pos - s.begin() << endl; }
}
```

Регулярные выражения
Набор символов

Предположим, мы хотим найти в тексте все междометия, обозначающие смех. Просто Хаха нам не подойдёт — ведь под него не попадут “Хехе”, “Хохо” и “Хихи”. Да и проблему с регистром первой буквы нужно как-то решить.

Здесь нам на помощь придут наборы — вместо указания конкретного символа, мы можем записать целый список, и если в исследуемой строке на указанном месте будет стоять любой из перечисленных символов, строка будет считаться подходящей. Наборы записываются в квадратных скобках — паттерну **[abcd]** будет соответствовать любой из символов **“a”, “b”, “c” или “d”**.

Если сразу после **[** записать символ **^**, то набор приобретёт обратный смысл — подходящим будет считаться любой символ кроме указанных. Так, паттерну **[^xyz]** соответствует любой символ, кроме, собственно, **“x”, “y” или “z”**.

Итак, применяя данный инструмент к нашему случаю, если мы напишем **[Xx][аоие]х[аоие]**, то каждая из строк **“Хаха”, “хехе”, “хихи”** и даже **“Хохо”** будут соответствовать шаблону.

Регулярные выражения
Предопределенные классы символов

Для некоторых наборов, которые используются достаточно часто, существуют специальные шаблоны. Так, для описания любого **пробельного символа** (пробел, табуляция, перенос строки) используется **\s**, для **цифр** — **\d**, для **символов латиницы, цифр и подчёркивания “_”** — **\w**.

Если необходимо описать **вообще любой символ**, для этого используется **точка — .**. Если указанные классы написать **с заглавной буквы** (\S, \D, \W) то они **поменяют свой смысл на противоположный** — любой непробельный символ, любой символ, который не является цифрой, и любой символ кроме латиницы, цифр или подчёркивания соответственно.

Также с помощью регулярных выражений есть возможность проверить **положение строки относительно остального текста**. Выражение **\b** обозначает **границу слова**, **\B** — **не границу слова**, ^ — **начало текста**, а **$** — **конец**. Так, по паттерну **\bJava\b** в строке **“Java and JavaScript”** найдутся **первые 4 символа**, а по паттерну **\bJava\B** — символы **с 10-го по 13-й** (в составе слова “JavaScript”).

Регулярные выражения
Диапазоны

У вас может возникнуть необходимость обозначить набор, в который входят буквы, например, **от “б” до “ф”**. Вместо того, чтобы писать [бвгдежзиклмнопрстуф] можно воспользоваться механизмом диапазонов и написать **[б-ф]**. Так, паттерну **x[0-8A-F][0-8A-F]** соответствует строка **“xA6”**, но **не** соответствует **“xb9”** (во-первых, из-за того, что в диапазоне указаны только заглавные буквы, во-вторых, из-за того, что 9 не входит в промежуток 0-8).

Чтобы обозначить **все буквы русского алфавита**, можно использовать паттерн **[а-яА-ЯёЁ]**. Обратите внимание, что буква “ё” не включается в общий диапазон букв, и её нужно указывать отдельно.

Регулярные выражения
Квантификаторы (указание количества повторений)

Квантификатор	Число повторений	Пример	Подходящие строки
{n}	Ровно n раз	Xa{3}xa	Xaaaxa
{m,n}	От m до n включительно	Xa{2,4}xa	Xaa, Xaaa, Xaaaaxa
{m,}	Не менее m	Xa{2,}xa	Xaaxa, Xaaaxa, Xaaaaxa и т. д.
{,n}	Не более n	Xa{,3}xa	Xxa, Xaxa, Xaaxa, Xaaaxa

Квантификатор	Аналог	Значение
?	{0,1}	Ноль или одно вхождение
*	{0,}	Ноль или более
+	{1,}	Одно или более

Регулярные выражения
Экранирование специальных символов

Регулярные выражения имеют спецсимволы, которые нужно экранировать. Вот их список:

. ^ $ * + ? { } [ ] \ | ( )

Экранирование осуществляется обычным способом — добавлением \ перед спецсимволом.

Регулярные выражения
regex

```cpp
#include <iostream>
#include <regex>

using namespace std;

int main()
{
    cout << boolalpha;

    regex reg1("<.*>.*</.*>");
    bool found = regex_match("<tag>value</tag>", reg1); // true
    cout << found;

    return 0;
}
```

заголовочный файл для работы с регулярными выражениями
#include <regex>

regex reg1("<.*>.*</.*>");
Регулярное выражение проверяет строку на соответствие шаблону **"<символы>символы</символы>"**

bool found = regex_match("<tag>value</tag>", reg1); - true

Регулярные выражения
regex

```cpp
#include <iostream>
#include <regex>

using namespace std;

int main()
{
    cout << boolalpha;

    regex reg1("<(.*)>.*</\\1>");
    bool found = regex_match("<tag>value</tag>", reg1); // true
    cout << found;

    return 0;
}
```

Можно указать, чтобы начальный и конечный дескрипторы должны быть одинаковыми символьными последовательностями.
regex reg1("<(.*)>.*</\\1>");
Здесь используется конструкция **(…)** для определения «группы захвата», на которую впоследствии ссылаемся с помощью регулярного выражения «**\1**».

bool found = regex_match("<tag>value</tag>", reg1); - true

Регулярные выражения
regex_search()

regex_match() проверяет, соответствует ли строка регулярному выражению полностью.

regex_search() проверяет, соответствует ли строка регулярному выражению частично, то есть оператор
regex_search(data, regex(pattern))
Эквивалентен оператору
regex_match(data, regex("(.|\n)*"+pattern+"(.|\n)*")),
где символы "(.|\n)*" означает любое количество любых символов или символ новой строки.

Регулярные выражения
regex_search()

```cpp
#include <iostream>
#include <string>
#include <regex>

using namespace std;

int main()
{
    string data = "XML tag: <tag-name>the value</tag-name>.";
    smatch m; // для возвращаемой информации о соответствии
    bool found = regex_search(data, m, regex("<(.*)>(.*)</(\\1)>"));
    cout << boolalpha;
    cout << m.empty() << endl; // false
    cout << m.size() << endl; /* 4, так как в регулярном выражении три группы захвата,
    определенные в m[1], m[2] и m[3].
    В m[0] содержится вся подстрока <tag-name>the value</tag-name> соответствующая
    регулярному выражению. */
    cout << m.str() << endl; // выведет строку data
    cout << m.length() << endl; // 30 - длина <tag-name>the value</tag-name>
    cout << m.position() << endl; // 9 - позиция подстроки <tag-name>the value</tag-name>
    // в строке XML tag: <tag-name>the value</tag-name>.
    cout << m.prefix().str() << endl; // XML tag: - подстрока перед строкой, соотв. рег. выр.
    cout << m.suffix().str() << endl; // . подстрока после строки, соотв. рег. выр
    for (int i = 0; i<m.size(); i++)
        cout << m[i].str() << '\t' << m.position(i) << endl;
    /*  выведет:
        <tag-name>the value</tag-name>  9
        tag-name    10
        the value   19
        tag-name    30    */
    return 0;
}
```

Регулярные выражения
Итераторы

```cpp
#include <iostream>
#include <string>
#include <regex>

using namespace std;

int main()
{
    string data = "<person>\n"
                  " <first>Ivan</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";
    regex reg("<(.*)>(.*)</(\\1)>");
    sregex_iterator pos(data.cbegin(), data.cend(), reg);
    sregex_iterator end;
    for ( ; pos!=end; ++pos)
    {
        cout << pos->str() << '\t' << pos->str(1) << '\t' << pos->str(2) << '\t' << pos->str(3) << endl;
    }
    /*  выведет:
        <first>Ivan</first> first   Ivan    first
        <last>Josuttis</last>   last    Josuttis    last */

    return 0;
}
```

Регулярные выражения
Итераторы

```cpp
#include <iostream>
#include <string>
#include <regex>
#include <algorithm>

using namespace std;

int main()
{
    string data = "<person>\n"
                  " <first>Ivan</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";
    regex reg("<(.*)>(.*)</(\\1)>");
    sregex_iterator beg(data.cbegin(), data.cend(), reg);
    sregex_iterator end;
    for_each(beg, end, [](const smatch &m)
    {
        cout << m.str() << '\t' << m.str(1) << '\t' << m.str(2) << '\t' << m.str(3) << endl;
    });
    /*  выведет:
        <first>Ivan</first> first   Ivan    first
        <last>Josuttis</last>   last    Josuttis    last */

    return 0;
}
```

Регулярные выражения
Итераторы токенов регулярных выражений

```cpp
#include <string>
#include <regex>
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";
    regex reg("<(.*)>(.*)</(\\1)>");
    // перебираем все соответствия с помощью итератора
    sregex_token_iterator pos(data.begin(),data.end(),   // диапазон поиска
                              reg,                       // регулярное выражение для поиска
                              {0,2});      // 0: полное соответствие, 2: вторая подстрока
    sregex_token_iterator end;
    for ( ; pos!=end ; ++pos ) {
        cout << "match: " << pos->str() << endl;
    }
    /* Вывод: match: <first>Nico</first>
              match: Nico
              match: <last>Josuttis<last>
              match: Josuttis */
}
```

Регулярные выражения
Итераторы токенов регулярных выражений

```cpp
#include <string>
#include <regex>
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
    string names = "nico, jim, helmut, paul, tim, john paul, rita";
    regex sep("[ \t\n]*[,;.][ \t\n]*");  // регулярное выражение определяет разделители: ,:.
    sregex_token_iterator p(names.begin(),names.end(),  // диапазон
                            sep,                        // регулярное выражение
                            -1);        // -1: значения между разделителями
    sregex_token_iterator e;
    for ( ; p!=e ; ++p ) {
        cout << "name:  " << *p << endl;
    }
    /* Вывод: name: nico
              name: jim
              name: helmut
              name: paul
              name: tim
              name: john paul
              name: rita */
}
```

Замена регулярных выражений

```cpp
#include <string>
#include <regex>
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";
    regex reg("<(.*)>(.*)</(\\1)>");
    cout << regex_replace(data,                    // данные
                          reg,                     // регулярное выражение
                          "<$1 value=\"$2\"/>");   // замена

    /* Вывод: <person>
                <first value="Nico"/>
                <last value="Josuttis"/>
               <person> */
}
```