ООП 2021
Лекция 8

Строки.
std::basic_string

oopCpp@yandex.ru

Работа с символами.

cout << "Hello, World!\n"

Строка - это последовательность символов, заключенная в двойные кавычки. Два символа: обратной дробной черты \ и непосредственно следующий за ним - обозначают некоторый специальный символ. В данном случае \n является символом конца строки (или перевода строки), поэтому он выдается после символов Hello, world!

Каждая строка содержит на один символ больше, чем явно задано: все строки оканчиваются нулевым символом ('\0'), имеющим значение 0. Поэтому

sizeof("asdf")==5;

Типом строки считается "массив из соответствующего числа символов", поэтому тип "asdf" есть char[5]. Пустая строка записывается как "" и имеет тип char[1]. Отметим, что для любой строки s выполняется strlen(s)==sizeof(s)-1, поскольку функция strlen() не учитывает завершающий символ '\0'.

Внутри строки можно использовать для представления невидимых символов специальные комбинации с \. В частности, в строке можно задать сам символ двойной кавычки " или символ \. Чаще всего из таких символов оказывается нужным символ конца строки '\n‘.

Для большей наглядности программы длинные строки можно разбивать пробелами, например:

```
char alpha[ ] = "abcdefghijklmnopqrstuvwxyz"
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Подобные, подряд идущие, строки будут объединяться в одну, поэтому массив alpha можно эквивалентным образом инициализировать с помощью одной строки:

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

В строке можно задавать символ '\0', но большинство программ не ожидает после него встречи с какими-либо еще символами. Например, строку "asdf\000hjkl" стандартные функции strcpy() и strlen() будут рассматривать как строку "asdf".

Строки в STL

Класс **string** из стандартной библиотеки представляет собой специализацию общего шаблонного класса **basic_string** для символьного типа char; иначе говоря, объект string это последовательность переменных типа char.

```
typedef basic_string<char, char_traits<char>, allocator<char> >    string;
typedef basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t> >   wstring;

template<> struct char_traits<char>
template<> struct char_traits<wchar_t>

template <class _Elem>
struct char_traits : public _Char_traits<_Elem, long>
{      // properties of a string or stream unknown element
};
template < class _Elem,  class _Int_type >
struct   _Char_traits
// properties of a string or stream element

template<class _Ty>
class  allocator : public _Allocator_base<_Ty>
// generic allocator for objects of class _Ty

template<class _Ty>
struct _Allocator_base
{
typedef _Ty value_type;
};
```

Характеристические классы

Характеристический шаблонный класс определяет протокол описания типа, а специализации этого класса описывают конкретный тип. Частичные специализации описывают множество типов, обладающих общими характеристиками.

Некоторые характеристические классы предназначены для распознавания свойств типов, на основе которых они определяют собственные переменные-члены и типы-члены.

Большинство характеристических классов предоставляют информацию о типе или значении и специализированную функциональность.

```
static int   compare(
_In_reads_(_Count) const _Elem *_First1,
_In_reads_(_Count) const _Elem *_First2, size_t _Count)

static size_t   length(_In_z_ const _Elem *_First)
 // find length of null-terminated sequence

 static _Elem *copy (…)
```

```
template<class _Elem,  class _Traits, class _Alloc>
class basic_string : public _String_alloc
< !is_empty<_Alloc>::value,  _String_base_types<_Elem, _Alloc> >
// null-terminated transparent array of elements

template< bool _Al_has_storage, class  _Alloc_types>
class _String_alloc : public _String_val
 < typename  _Alloc_types::_Val_types>
// base class for basic_string  to hold allocator with storage

template<class _Val_types>
class _String_val
: public _Container_base
 // base class for basic_string to hold data
```

Конструкторы

explicit basic_string (1) (const allocator_type& alloc = allocator_type());
Конструирует пустую строку.

copy (2) Создает копию
basic_string (const basic_string& str);

substring (3) конструктор подстроки: Копирует часть строки str, которая начинается в позиции символа pos
basic_string (const basic_string& str, size_type pos, size_type len = npos, const allocator_type& alloc = allocator_type());

from c-string (4) Копирует символьную последовательность (C-строку), указанную символом s
basic_string (const charT* s, const allocator_type& alloc = allocator_type());

from sequence (5) Копирует первые n символов из массива символов, на которые указывает s
basic_string (const charT* s, size_type n, const allocator_type& alloc = allocator_type());

Конструкторы

fill (6) Заполняет строку n последовательными копиями символа c

basic_string (size_type n, charT c, const allocator_type& alloc = allocator_type());

range (7) Копирует последовательность символов в диапазоне [первый, последний) в том же порядке.

template <class InputIterator>

basic_string (InputIterator first, InputIterator last, const allocator_type& alloc = allocator_type());

(8) Копирует каждый из символов в il, в том же порядке.

basic_string (initializer_list<charT> il, const allocator_type& alloc = allocator_type());

(9) перемещающие конструкторы

basic_string (basic_string&& str) noexcept;

basic_string (basic_string&& str, const allocator_type& alloc);

Примеры

```
#include <iostream>
#include <string>
int main () {
  std::string s0 ("Initial string");
       // конструкторы используются в том же порядке, как описано выше:
  std::string s1;
  std::string s2 (s0);
  std::string s3 (s0, 8, 3);
  std::string s4 ("A character sequence", 6);
  std::string s5 ("Another character sequence");
  std::string s6 (10, 'x');
  std::string s7a (10, 42);
  std::string s7b (s0.begin(), s0.begin()+7);

  std::cout << "s1: " << s1 << "\ns2: " << s2 << "\ns3: " << s3;
  std::cout << "\ns4: " << s4 << "\ns5: " << s5 << "\ns6: " << s6;
  std::cout << "\ns7a: " << s7a << "\ns7b: " << s7b << '\n';
  return 0;
}
```

```
s1:
s2: Initial string
s3: str
s4: A char
s5: Another character sequence
s6: xxxxxxxxxx
s7a: **********
s7b: Initial
```

Операции со строками

s1 = s2 Присвоение строки s2 строке s1; строка s2 может быть объектом класса string или строкой к стиле языка C

s += s1 Добавление объекта s1 в конец строки; объект s1 может быть символом, объектом класса string или строкой в стиле языка C

s[i] Индексация (как у обычного массива)

s = s1+s2 Конкатенация; символы в целевом объекте класса string будут копиями символов их строки s1, за которыми следуют копии символов из строки s2

s1==s2 Сравнение объектов класса string; либо s1, либо s2, но не оба объекта могут быть строкой в стиле языка C.

s1 != s2 Проверка неравенства объектов s1 и s2

s1 < s2 Лексикографическое сравнение объектов класса string; либо s1, либо s2, но не оба объекта могут быть строкой в стиле языка C.

. <=, > и >= Аналогично.

s.size () Количество символов в строке s

s.length () Количество символов в строке s

s.c_str () Преобразование в строку в стиле языка C

s.begin () Итератор на первый символ

s.end() Итератор ячейки, следующей за концом строки s

s.insert(pos, x) Вставка объекта x перед строкой s [pos]; объект x может быть символом, объектом класса string или строкой в стиле языка C. Строка s увеличивается, чтобы поместить символы из объекта x

s.append(x) Вставка объекта x в конец строки s ; Строка s увеличивается, чтобы поместить символы из объекта x

s.erase(pos) Удаление символа из позиции s [pos]. Размер строки s уменьшается на единицу

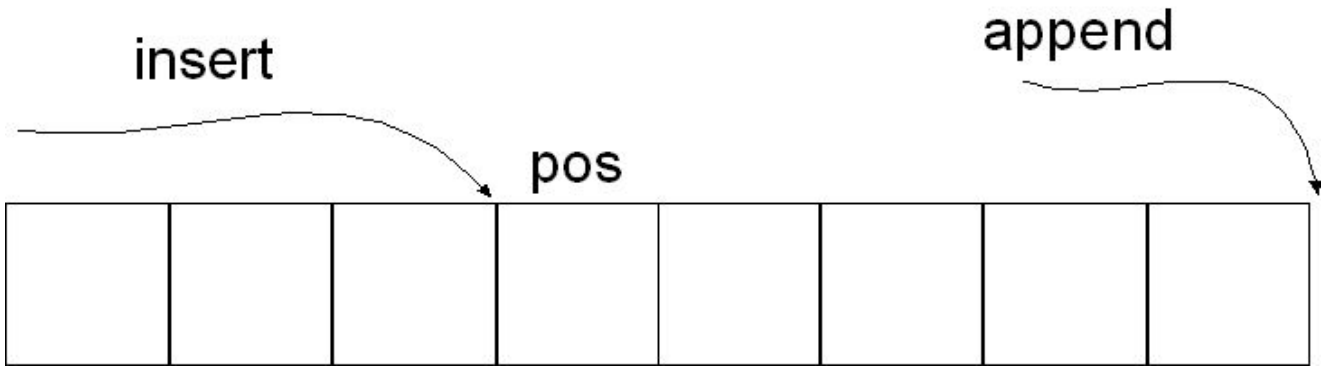
pos = s.find(x) Поиск объекта x в строке s; переменная pos — это индекс первого найденного символа или значение npos (позиция ячейки, следующей за концом строки s

In >> s Считывание слова, отделенного пробелами из потока in в объект s

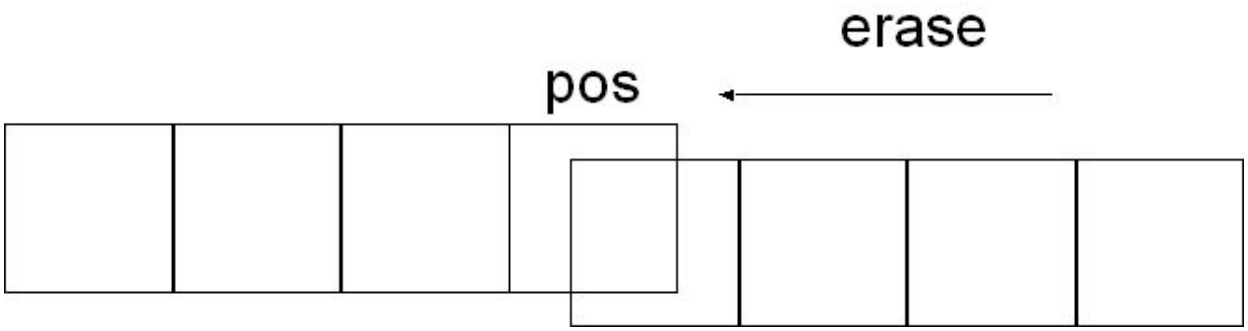
getline(in, s) Считывание строки текста из потока in в объект s

out << s Запись данных из объекта s в поток out

Операции **insert** и **append** перемещают символы, чтобы освободить место для новых.



Операция **erase** (pos) сдвигает символы влево, чтобы заполнить пробел, оставшийся после удаления символа



Примеры: append

```
#include <iostream>
#include <string>
using namespace std;
int main () {
    string str;
   string str2="Writing ";
   string str3="print 10 and then 5 more";
      // used in the same order as described above:
  str.append(str2);                       // "Writing "
  str.append(str3,6,3);                    // "10 "
  str.append("dots are cool",5);    // "dots "
  str.append("here: ");                    // "here: "
  str.append(10u,'.');                    // ".........."
  str.append(str3.begin()+8,str3.end());  // " and then 5 more"
  str.append (5,0x2E);        // "....."
     cout << str << '\n';
}
```

Output: Writing 10 dots here: ..... and then 5 more.....

Примеры: insert

```
void ins(){
 string str="to be question";
   string str2="the ";
   string str3="or not to be";
    string::iterator it;
  str.insert(6,str2);                 // to be (the )question
  str.insert(6,str3,3,4);              // to be (not )the question
  str.insert(10,"that is cool",8);    // to be not (that is )the question
  str.insert(10,"to be ");             // to be not (to be )that is the question
  str.insert(15,1,':');               // to be not to be(:) that is the question
  it = str.insert(str.begin()+5,','); // to be(,) not to be: that is the question
  str.insert (str.end(),3,'.');       // to be, not to be: that is the question(...)
  str.insert (it+2,str3.begin(),str3.begin()+3); // (or )
  std::cout << str << '\n';
}
```

Output: to be, or not to be: that is the question...

Преобразования

stoi - Преобразовать строку в целое число
stol - Конвертировать строку в long int
stoul - Преобразовать строку в целое число без знака
stoll - Конвертировать строку в long long
stoull - Convert string to unsigned long long
stof - Преобразование строки в float
stod - Преобразование строки в double

stold - Преобразование строки в long double

Итераторы basic_string

begin - Возвратить итератор к началу
end - Возвратить итератор к концу
rbegin - Возвратить обратный итератор к обратному началу
rend - Возвратить обратный итератор к обратному концу
cbegin
    Return const_iterator to beginning
cend
    Return const_iterator to end
crbegin
    Return const_reverse_iterator to reverse beginning
crend
    Return const_reverse_iterator to reverse end

Пример: **Итераторы basic_string**

```
// string::begin/end
#include <iostream>
#include <string>

int main ()
{
  std::string str ("Test string");
  for ( std::string::iterator it=str.begin(); it!=str.end(); ++it)
    std::cout << *it;
  std::cout << '\n';

  return 0;
}}
```

Output:
Test string

Пример: **Итераторы basic_string**

```
// string::rbegin/rend
#include <iostream>
#include <string>

int main ()
{
  std::string str ("now step live...");
  for (std::string::reverse_iterator rit=str.rbegin(); rit!=str.rend(); ++rit)
    std::cout << *rit;
  return 0;
}
```

Output:
...evil pets won

basic_string::operator=

string (1)
basic_string& operator= (const basic_string& str);
c-string (2)
basic_string& operator= (const charT* s);
character (3)
basic_string& operator= (charT c);
initializer list (4)
basic_string& operator= (initializer_list<charT> il);
move (5)
basic_string& operator= (basic_string&& str) noexcept;

Пример: **operator=**

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

int main ()
{
  std::string str1, str2, str3;
  str1 = "Test string: ";   // c-string
  str2 = 'x';               // single character
  str3 = str1 + str2;       // string

  std::cout << str3  << '\n';
  return 0;
}
```

Output:

```
Test string: x
```

basic_string::operator[ ]

reference operator[ ] (size_type pos);

const_reference operator[ ] (size_type pos) const;

```
#include <iostream>
#include <string>
int main () {
    std::string str ("Test string");
  for (int i=0; i<str.length(); ++i) {
    std::cout << str[i];
  }
  return 0;
}
```

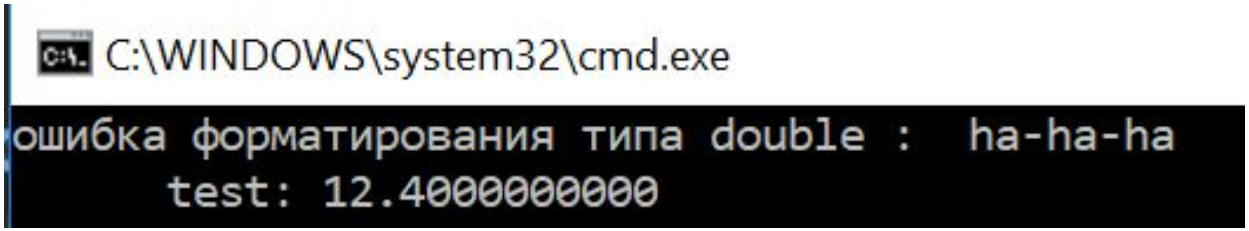
Потоки строк

Объект класса string можно использовать в качестве источника ввода для потока istream или цели вывода для потока ostream. Поток istream, считывающий данные из объекта класса string, называется **istringstream**, а поток ostream, записывающий символы в объект класса string, называется **ostringstream**. Например, поток istringstream полезен для извлечения числовых значений из строк.

```
stringstream ss;
   ss << "22.84";
    float k = 0;
   ss >> k;
```

```
#include<iostream>
#include <sstream>
#include <iomanip>
using namespace std;

void error( char* e, const char* s){
  cerr<<e<<":  "<<s<<endl;
}

double str_to_double(string s){
 istringstream  is (s); // создаем поток для ввода из строки s
  double d;

  is >> d;

  if (! is) error("ошибка форматирования типа double ",s.c_str());

return d;
}
```

```
void func_out( string label, double d) {
  ostringstream os;     // поток для составления сообщения
  os << setw(8) << label << ": "
    << fixed << setprecision (10)
    <<  d  ;
  cout<<os.str ()<<endl;
}
int _tmain(int argc, _TCHAR* argv[])
{
 setlocale(LC_ALL, "Russian");
string s;
double d1 = str_to_double("12.4");
double d2 = str_to_double("1.34e-3");
double d3 = str_to_double("ha-ha-ha"); // вызывается  error()

  func_out("test",   d1) ;
return 0;
}
```



Содержание строко- потоковых классов

typedef **basic_istringstream**<char, char_traits<char>,
allocator<char> > **istringstream**;

template<class _Elem,
class _Traits,
class _Alloc>
class **basic_istringstream**
: public **basic_istream**<_Elem, _Traits>
// input stream associated with a character array

template<class _Elem,
class _Traits>
class **basic_istream**
: virtual public basic_ios<_Elem, _Traits>
// control extractions from a stream buffer

```
    template<class _Elem,
class _Traits>
class basic_ios
    : public ios_base
// base class for basic_istream/basic_ostream

class   ios_base
    : public _losb <int>
// base class for ios
    template<class _Dummy> class _losb
// define templatized bitmask / enumerated types, instantiate on
    demand
```

Реализация операций string

```
string s="test";
const char* cc= s.c_str();
////////////////////////////////////
const _Elem  *c_str() const
{// return pointer to null-terminated nonmutable array
return (this-> _Myptr());
}
const value_type * _Myptr() const
{// determine current pointer to buffer for nonmutable string
return (this->_BUF_SIZE <= this->_Myres
? _STD addressof(*this->_Bx._Ptr)
: this->_Bx._Buf);
}
union _Bxty
{// storage for small buffer or pointer to larger one
value_type _Buf[_BUF_SIZE];
pointer _Ptr;
char _Alias[_BUF_SIZE];// to permit aliasing
} _Bx;
```

Примеры: substr и find

```
void sub(){
  std::string str="We think in generalities, but we live in details.";
                                    // (quoting Alfred N. Whitehead)

  std::string str2 = str.substr (3,5);     // "think"

  std::size_t pos = str.find("live");      // position of "live" in str

  std::string str3 = str.substr (pos);     // get from "live" to the end

  std::cout << str2 << ' ' << str3 << '\n';

}
```

Output: think live in details.

Примеры: replace

```
void repl(){
   string base="this is a test string.";
   string str2="n example";
   string str3="sample phrase";
   string str4="useful.";
   string str=base;                    // "this is a test string."
  str.replace(9,5,str2);               // "this is an example string." (1)
  str.replace(19,6,str3,7,6);          // "this is an example phrase." (2)
  str.replace(8,10,"just a");          // "this is just a phrase."     (3)
  str.replace(8,6,"a shorty",7);  // "this is a short phrase."    (4)
  str.replace(22,1,3,'!');             // "this is a short phrase!!!"  (5)
  str.replace(str.begin(),str.end()-3,str3);     // "sample phrase!!!"      (1)
  str.replace(str.begin(),str.begin()+6,"replace");  // "replace phrase!!!"     (3)
  str.replace(str.begin()+8,str.begin()+14,"is coolness",7);  // "replace is coo"  (4)
  str.replace(str.begin()+12,str.end()-4,4,'o');  // "replace is cooool!!!"  (5)
  str.replace(str.begin()+11,str.end(),str4.begin(),str4.end());// "replace is useful."  (6)
        cout << str << '\n';
}
```

Output: replace is useful.

Примеры: erase

```
void erase(){
   string str ("This is an example sentence.");
    cout << str << '\n';                          // "This is an example sentence."
  str.erase (10,8);                               //          ^^^^^^^^
   cout << str << '\n';                          // "This is an sentence."
  str.erase (str.begin()+9);                      //          ^
   cout << str << '\n';                          // "This is a  sentence."
  str.erase (str.begin()+5, str.end()-9);         //       ^^^^^
    cout << str << '\n';
}
```

Output:

```
This is an example sentence.
This is an sentence.
This is a sentence.
This sentence..
```

Широкие строки

typedef basic_string < **wchar_t** > **wstring**;
wstring ws = L"Super";

Символы, которые могут содержаться в строке, не ограничены типом **wchar_t** . В действительности символы могут быть практически любым типом **без конструктора**, для которого можно определить свойства символов.

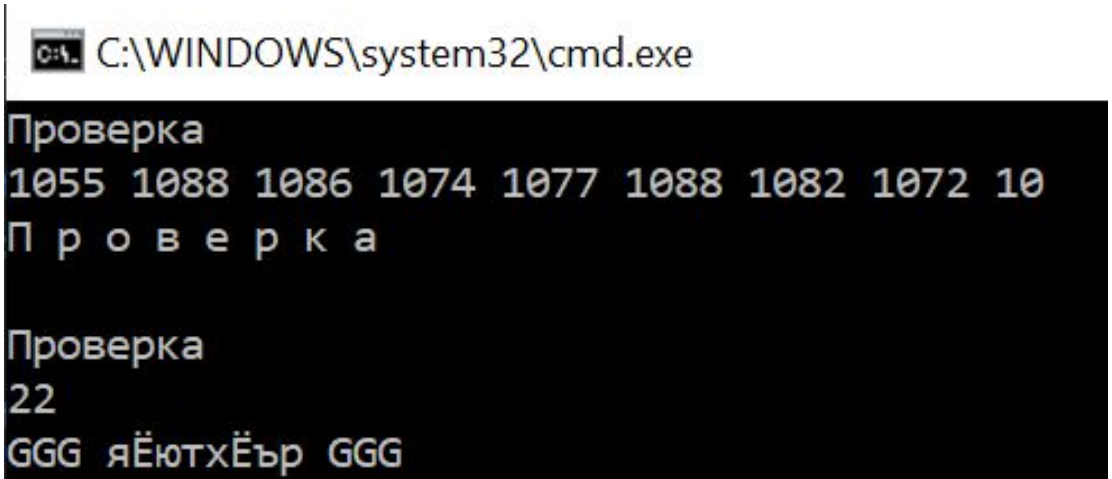
Свойства символов определяют различные характеристики символов, включая их **порядок для сравнения**, **метод завершения последовательностей**, а также как **копировать и перемещать** диапазоны символов.

Это обеспечивает высокую гибкость представления символов, включая типы многобайтных символов, использующиеся в международных наборах символов.

Пример

```
#include <iostream>
#include <vector>
#include <string>
#include <io.h>
#include <fcntl.h>
using namespace std;
 int main() {
  _setmode(_fileno(stdout), _O_U16TEXT); // переключаем на юникод
   std::wcout.clear();                                  // очистка потока
    std::wstring s(L"Проверка\n");
 wcout << s;
     std::vector<int> v(s.length());
    for (size_t idx = 0; idx < s.length(); idx++)
        v[idx] = s[idx];                              // посимвольно копируем в вектор целых

std::basic_string <wchar_t>ss  (v.begin(), v.end()); // создаем новую строку
                              // юникод-символов, передавая диапазон вектора
```

```
    for (int   i : v)
        wcout << i << " ";  //  вывод значений вектора целых
      wcout << endl;
    for ( wchar_t   i : ss)
        wcout << i << " "; //  вывод юникод-строки посимвольно
    wcout << endl;


 wcout << ss;               //  вывод юникод-строки
   _setmode(_fileno(stdout), _O_TEXT);     //  переключение на обычный текст
     cout<< 22<< endl;
     cout<< "GGG проверка GGG"<< endl;;
    return 0;
}
```



Пример. Преобразование.

```
#include <iostream>
#include <string>
#include <codecvt>
#include <locale>
#include <vector>

std::wstring  wide_string ( std::string const& s, std::locale const& loc )
{          //  из string в wstring
   if (s.empty())         return std::wstring();

   std::ctype<wchar_t> const& facet = std::use_facet<std::ctype<wchar_t> >(loc);
   char const* first = s.c_str();
   char const* last = first + s.size();

      std::vector<wchar_t> result(s.size());

   facet.widen(first, last, &result[0]);

    return std::wstring(result.begin(), result.end());
}
```

```
std::string narrow_string(  std::wstring const& s,
                            std::locale const& loc,      char default_char = '?')
{
     //  из wstring   в   string
    if (s.empty())       return std::string();
    std::ctype<wchar_t> const& facet = std::use_facet<std::ctype<wchar_t> >(loc);
    wchar_t const* first = s.c_str();
    wchar_t const* last = first + s.size();

     std::vector<char> result(s.size());

    facet.narrow(first, last, default_char, &result[0]);

    return std::string(result.begin(), result.end());
}
```

```
char const *russian_locale_designator = "rus";

int main()  {
 std::locale loc(russian_locale_designator);

    std::wstring const s = L"русский текст";
    std::string const  sa = ::narrow_string(s, loc);
    std::wstring const sw = ::wide_string(sa, loc);

    std::locale::global ( loc );

    std::wcout << "original  wide: " << s << std::endl;
    std::cout << "wide -> narrow: " << sa << std::endl;
    std::wcout << "narrow -> wide: " << sw << std::endl;
return 1;
}
```

Вывод : original wide: русский текст
wide -> narrow: русский текст
narrow -> wide: русский текст

Еще один опыт.

```
using int_string = std::basic_string< int, std::char_traits<int>, std::allocator<int> >;

void add_to_int_str(int n) {
    int_string si;
    si.resize(n);
    for (int i = 0; i < n; i++) {
        si[i] = i;
   }
}
void add_to_vec(int n) {
    vector<int> v;
    v.resize(n);
    for (int i = 0; i < n; i++) {
        v[i] = i;
   }
}
```

Продолжение …

```
#include <ppl.h>
#include <random>
#include <windows.h>

using namespace concurrency;

template <class Function>
__int64  time_call(Function&& f) {
    __int64 begin = GetTickCount();
    f();
    return  GetTickCount() - begin;
}

#define Print   for (auto& i : si)      cout << i << "   ";       cout <<  endl;
```

Продолжение …

```
void main() {
   //  typedef  std::basic_string< int, std::char_traits<int>,
                                 // std::allocator<int> >          int_string  ;
     int_string si ;      si.append(2, 3); Print
     si.append({ 4,1 }); Print
     int sz = si.size(); Print
     si = si.substr(1, 2); Print
     si += 55; Print
     si.insert(1, 5, 10); Print
     si.insert(4, si);  Print
     si.erase(11);  Print
```

Вывод: 3 3
3 3 4 1
3 3 4 1
3 4
3 4 55
3 10 10 10 10 10 4 55
3 10 10 10 3 10 10 10 10 10 4 55 10 10 4 55
3 10 10 10 3 10 10 10 10 10 4

Завершение

```
int N = 15000000;
    wcout << L"int_string: " << time_call(
        [&] {  add_to_int_str(N);  }
    ) << endl;

    wcout << L"Vector: " << time_call(  [&] {  add_to_vec(N);  } ) << endl;
}
```

Вывод:

int_string: 1172

Vector: 266