G++ 2.91.57，cygnus\cygwin-b20\include\g++**std\bastring.h** 完整列表

```
// Main templates for the -*- C++ -*- string classes.
// Copyright (C) 1994, 1995 Free Software Foundation

// This file is part of the GNU ANSI C++ Library.  This library is free
// software; you can redistribute it and/or modify it under the
// terms of the GNU General Public License as published by the
// Free Software Foundation; either version 2, or (at your option)
// any later version.

// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.

// You should have received a copy of the GNU General Public License
// along with this library; see the file COPYING.  If not, write to the Free
// Software Foundation, 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.

// As a special exception, if you link this library with files
// compiled with a GNU compiler to produce an executable, this does not cause
// the resulting executable to be covered by the GNU General Public License.
// This exception does not however invalidate any other reasons why
// the executable file might be covered by the GNU General Public License.

// Written by Jason Merrill based upon the specification by Takanori Adachi
// in ANSI X3J16/94-0013R2.

#ifndef __BASTRING__
#define __BASTRING__

#ifdef __GNUG__
#pragma interface
#endif

#include <cstddef>
#include <std/straits.h>

// NOTE : This does NOT conform to the draft standard and is likely to change
#include <alloc.h>

extern "C++" {
class istream; class ostream;

#include <iterator>

#ifdef __STL_USE_EXCEPTIONS

extern void __out_of_range (const char *);
```

```
extern void __length_error (const char *);

#define OUTOFRANGE(cond) \
  do { if (cond) __out_of_range (#cond); } while (0)
#define LENGTHERROR(cond) \
  do { if (cond) __length_error (#cond); } while (0)

#else

#include <cassert>
#define OUTOFRANGE(cond) assert (!(cond))
#define LENGTHERROR(cond) assert (!(cond))

#endif

// 以下是 VC6 PJ STL 的 basic_string<> 定義
// template<class _E, class _Tr = char_traits<_E>,
//     class _A = allocator<_E> >
// class basic_string { ... }
//
// PJ STL 使用 char_traits 符號，定義於 vc6\include\iosfwd
// SGI STL 使用 string_char_traits 符號，定義於 std\straits.h
template <class charT, class traits = string_char_traits<charT>,
     class Allocator = alloc >
class basic_string
{
private:
  struct Rep {
    size_t len, res, ref;
    bool selfish;

    charT* data () { return reinterpret_cast<charT *>(this + 1); }
    charT& operator[] (size_t s) { return data () [s]; }
    charT* grab () { if (selfish) return clone (); ++ref; return data (); }
    void release () { if (--ref == 0) delete this; }

    inline static void * operator new (size_t, size_t);
    inline static void operator delete (void *);
    inline static Rep* create (size_t);
    charT* clone ();

    inline void copy (size_t, const charT *, size_t);
    inline void move (size_t, const charT *, size_t);
    inline void set  (size_t, const charT,   size_t);

    inline static bool excess_slop (size_t, size_t);
    inline static size_t frob_size (size_t);

  private:
```

```
    Rep &operator= (const Rep &);
  };

public:
// types:
  typedef      traits     traits_type;
  typedef typename traits::char_type value_type;
  typedef      Allocator       allocator_type;

  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
  typedef charT& reference;
  typedef const charT& const_reference;
  typedef charT* pointer;
  typedef const charT* const_pointer;
  typedef pointer iterator;
  typedef const_pointer const_iterator;
  typedef ::reverse_iterator<iterator> reverse_iterator;
  typedef ::reverse_iterator<const_iterator> const_reverse_iterator;
  static const size_type npos = static_cast<size_type>(-1);

private:
  Rep *rep () const { return reinterpret_cast<Rep *>(dat) - 1; }
  void repup (Rep *p) { rep ()->release (); dat = p->data (); }

public:
  const charT* data () const
    { return rep ()->data(); }
  size_type length () const
    { return rep ()->len; }
  size_type size () const
    { return rep ()->len; }
  size_type capacity () const
    { return rep ()->res; }
  size_type max_size () const
    { return (npos - 1)/sizeof (charT); }      // XXX
  bool empty () const
    { return size () == 0; }

// _lib.string.cons_ construct/copy/destroy:
  basic_string& operator= (const basic_string& str)
    {
      if (&str != this) { rep ()->release (); dat = str.rep ()->grab (); }
      return *this;
    }

  explicit basic_string (): dat (nilRep.grab ()) { }
  basic_string (const basic_string& str): dat (str.rep ()->grab ()) { }
  basic_string (const basic_string& str, size_type pos, size_type n = npos)
```

```
    : dat (nilRep.grab ()) { assign (str, pos, n); }
  basic_string (const charT* s, size_type n)
    : dat (nilRep.grab ()) { assign (s, n); }
  basic_string (const charT* s)
    : dat (nilRep.grab ()) { assign (s); }
  basic_string (size_type n, charT c)
    : dat (nilRep.grab ()) { assign (n, c); }
#ifdef __STL_MEMBER_TEMPLATES
  template<class InputIterator>
    basic_string(InputIterator begin, InputIterator end)
#else
  basic_string(const_iterator begin, const_iterator end)
#endif
    : dat (nilRep.grab ()) { assign (begin, end); }

  ~basic_string ()
    { rep ()->release (); }

  void swap (basic_string &s) { charT *d = dat; dat = s.dat; s.dat = d; }

  basic_string& append (const basic_string& str, size_type pos = 0,
            size_type n = npos)
    { return replace (length (), 0, str, pos, n); }
  basic_string& append (const charT* s, size_type n)
    { return replace (length (), 0, s, n); }
  basic_string& append (const charT* s)
    { return append (s, traits::length (s)); }
  basic_string& append (size_type n, charT c)
    { return replace (length (), 0, n, c); }
#ifdef __STL_MEMBER_TEMPLATES
  template<class InputIterator>
    basic_string& append(InputIterator first, InputIterator last)
#else
  basic_string& append(const_iterator first, const_iterator last)
#endif
    { return replace (iend (), iend (), first, last); }

  basic_string& assign (const basic_string& str, size_type pos = 0,
            size_type n = npos)
    { return replace (0, npos, str, pos, n); }
  basic_string& assign (const charT* s, size_type n)
    { return replace (0, npos, s, n); }
  basic_string& assign (const charT* s)
    { return assign (s, traits::length (s)); }
  basic_string& assign (size_type n, charT c)
    { return replace (0, npos, n, c); }
#ifdef __STL_MEMBER_TEMPLATES
  template<class InputIterator>
    basic_string& assign(InputIterator first, InputIterator last)
```

```
#else
  basic_string& assign(const_iterator first, const_iterator last)
#endif
    { return replace (ibegin (), iend (), first, last); }

  basic_string& operator= (const charT* s)
    { return assign (s); }
  basic_string& operator= (charT c)
    { return assign (1, c); }

  basic_string& operator+= (const basic_string& rhs)
    { return append (rhs); }
  basic_string& operator+= (const charT* s)
    { return append (s); }
  basic_string& operator+= (charT c)
    { return append (1, c); }

  basic_string& insert (size_type pos1, const basic_string& str,
             size_type pos2 = 0, size_type n = npos)
    { return replace (pos1, 0, str, pos2, n); }
  basic_string& insert (size_type pos, const charT* s, size_type n)
    { return replace (pos, 0, s, n); }
  basic_string& insert (size_type pos, const charT* s)
    { return insert (pos, s, traits::length (s)); }
  basic_string& insert (size_type pos, size_type n, charT c)
    { return replace (pos, 0, n, c); }
  iterator insert(iterator p, charT c)
    { size_type __o = p - ibegin ();
      insert (p - ibegin (), 1, c); selfish ();
      return ibegin () + __o; }
  iterator insert(iterator p, size_type n, charT c)
    { size_type __o = p - ibegin ();
      insert (p - ibegin (), n, c); selfish ();
      return ibegin () + __o; }
#ifdef __STL_MEMBER_TEMPLATES
  template<class InputIterator>
    void insert(iterator p, InputIterator first, InputIterator last)
#else
  void insert(iterator p, const_iterator first, const_iterator last)
#endif
    { replace (p, p, first, last); }

  basic_string& erase (size_type pos = 0, size_type n = npos)
    { return replace (pos, n, (size_type)0, (charT)0); }
  iterator erase(iterator p)
    { size_type __o = p - begin();
      replace (__o, 1, (size_type)0, (charT)0); selfish ();
      return ibegin() + __o; }
  iterator erase(iterator f, iterator l)
```

```
    { size_type __o = f - ibegin();
      replace (__o, l-f, (size_type)0, (charT)0);selfish ();
      return ibegin() + __o; }

  basic_string& replace (size_type pos1, size_type n1, const basic_string& str,
              size_type pos2 = 0, size_type n2 = npos);
  basic_string& replace (size_type pos, size_type n1, const charT* s,
              size_type n2);
  basic_string& replace (size_type pos, size_type n1, const charT* s)
    { return replace (pos, n1, s, traits::length (s)); }
  basic_string& replace (size_type pos, size_type n1, size_type n2, charT c);
  basic_string& replace (size_type pos, size_type n, charT c)
    { return replace (pos, n, 1, c); }
  basic_string& replace (iterator i1, iterator i2, const basic_string& str)
    { return replace (i1 - ibegin (), i2 - i1, str); }
  basic_string& replace (iterator i1, iterator i2, const charT* s, size_type n)
    { return replace (i1 - ibegin (), i2 - i1, s, n); }
  basic_string& replace (iterator i1, iterator i2, const charT* s)
    { return replace (i1 - ibegin (), i2 - i1, s); }
  basic_string& replace (iterator i1, iterator i2, size_type n, charT c)
    { return replace (i1 - ibegin (), i2 - i1, n, c); }
#ifdef __STL_MEMBER_TEMPLATES
  template<class InputIterator>
    basic_string& replace(iterator i1, iterator i2,
                InputIterator j1, InputIterator j2);
#else
  basic_string& replace(iterator i1, iterator i2,
              const_iterator j1, const_iterator j2);
#endif

private:
  static charT eos () { return traits::eos (); }
  void unique () { if (rep ()->ref > 1) alloc (length (), true); }
  void selfish () { unique (); rep ()->selfish = true; }

public:
  charT operator[] (size_type pos) const
    {
      if (pos == length ())
    return eos ();
      return data ()[pos];
    }

  reference operator[] (size_type pos)
    { selfish (); return (*rep ())[pos]; }

  reference at (size_type pos)
    {
      OUTOFRANGE (pos >= length ());
```

```
      return (*this)[pos];
    }
  const_reference at (size_type pos) const
    {
      OUTOFRANGE (pos >= length ());
      return data ()[pos];
    }

private:
  void terminate () const
    { traits::assign ((*rep ())[length ()], eos ()); }

public:
  const charT* c_str () const
    { if (length () == 0) return ""; terminate (); return data (); }
  void resize (size_type n, charT c);
  void resize (size_type n)
    { resize (n, eos ()); }
  void reserve (size_type) { }

  size_type copy (charT* s, size_type n, size_type pos = 0) const;

  size_type find (const basic_string& str, size_type pos = 0) const
    { return find (str.data(), pos, str.length()); }
  size_type find (const charT* s, size_type pos, size_type n) const;
  size_type find (const charT* s, size_type pos = 0) const
    { return find (s, pos, traits::length (s)); }
  size_type find (charT c, size_type pos = 0) const;

  size_type rfind (const basic_string& str, size_type pos = npos) const
    { return rfind (str.data(), pos, str.length()); }
  size_type rfind (const charT* s, size_type pos, size_type n) const;
  size_type rfind (const charT* s, size_type pos = npos) const
    { return rfind (s, pos, traits::length (s)); }
  size_type rfind (charT c, size_type pos = npos) const;

  size_type find_first_of (const basic_string& str, size_type pos = 0) const
    { return find_first_of (str.data(), pos, str.length()); }
  size_type find_first_of (const charT* s, size_type pos, size_type n) const;
  size_type find_first_of (const charT* s, size_type pos = 0) const
    { return find_first_of (s, pos, traits::length (s)); }
  size_type find_first_of (charT c, size_type pos = 0) const
    { return find (c, pos); }

  size_type find_last_of (const basic_string& str, size_type pos = npos) const
    { return find_last_of (str.data(), pos, str.length()); }
  size_type find_last_of (const charT* s, size_type pos, size_type n) const;
  size_type find_last_of (const charT* s, size_type pos = npos) const
    { return find_last_of (s, pos, traits::length (s)); }
```

```
  size_type find_last_of (charT c, size_type pos = npos) const
    { return rfind (c, pos); }

  size_type find_first_not_of (const basic_string& str, size_type pos = 0) const
    { return find_first_not_of (str.data(), pos, str.length()); }
  size_type find_first_not_of (const charT* s, size_type pos, size_type n) const;
  size_type find_first_not_of (const charT* s, size_type pos = 0) const
    { return find_first_not_of (s, pos, traits::length (s)); }
  size_type find_first_not_of (charT c, size_type pos = 0) const;

  size_type find_last_not_of (const basic_string& str, size_type pos = npos) const
    { return find_last_not_of (str.data(), pos, str.length()); }
  size_type find_last_not_of (const charT* s, size_type pos, size_type n) const;
  size_type find_last_not_of (const charT* s, size_type pos = npos) const
    { return find_last_not_of (s, pos, traits::length (s)); }
  size_type find_last_not_of (charT c, size_type pos = npos) const;

  basic_string substr (size_type pos = 0, size_type n = npos) const
    { return basic_string (*this, pos, n); }

  int compare (const basic_string& str, size_type pos = 0, size_type n = npos) const;
  // There is no 'strncmp' equivalent for charT pointers.
  int compare (const charT* s, size_type pos, size_type n) const;
  int compare (const charT* s, size_type pos = 0) const
    { return compare (s, pos, traits::length (s)); }

  iterator begin () { selfish (); return &(*this)[0]; }
  iterator end () { selfish (); return &(*this)[length ()]; }

private:
  iterator ibegin () const { return &(*rep ())[0]; }
  iterator iend () const { return &(*rep ())[length ()]; }

public:
  const_iterator begin () const { return ibegin (); }
  const_iterator end () const { return iend (); }

  reverse_iterator      rbegin() { return reverse_iterator (end ()); }
  const_reverse_iterator rbegin() const
    { return const_reverse_iterator (end ()); }
  reverse_iterator      rend() { return reverse_iterator (begin ()); }
  const_reverse_iterator rend() const
    { return const_reverse_iterator (begin ()); }

private:
  void alloc (size_type size, bool save);
  static size_type _find (const charT* ptr, charT c, size_type xpos, size_type len);
  inline bool check_realloc (size_type s) const;
```

```
  static Rep nilRep;
  charT *dat;
};

#ifdef __STL_MEMBER_TEMPLATES
template <class charT, class traits, class Allocator> template <class InputIterator>
basic_string <charT, traits, Allocator>& basic_string <charT, traits, Allocator>::
replace (iterator i1, iterator i2, InputIterator j1, InputIterator j2)
#else
template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>& basic_string <charT, traits, Allocator>::
replace (iterator i1, iterator i2, const_iterator j1, const_iterator j2)
#endif
{
  const size_type len = length ();
  size_type pos = i1 - ibegin ();
  size_type n1 = i2 - i1;
  size_type n2 = j2 - j1;

  OUTOFRANGE (pos > len);
  if (n1 > len - pos)
    n1 = len - pos;
  LENGTHERROR (len - n1 > max_size () - n2);
  size_t newlen = len - n1 + n2;

  if (check_realloc (newlen))
    {
      Rep *p = Rep::create (newlen);
      p->copy (0, data (), pos);
      p->copy (pos + n2, data () + pos + n1, len - (pos + n1));
      for (; j1 != j2; ++j1, ++pos)
    traits::assign ((*p)[pos], *j1);
      repup (p);
    }
  else
    {
      rep ()->move (pos + n2, data () + pos + n1, len - (pos + n1));
      for (; j1 != j2; ++j1, ++pos)
    traits::assign ((*rep ())[pos], *j1);
    }
  rep ()->len = newlen;

  return *this;
}

template <class charT, class traits, class Allocator>
inline basic_string <charT, traits, Allocator>
operator+ (const basic_string <charT, traits, Allocator>& lhs,
      const basic_string <charT, traits, Allocator>& rhs)
```

```
{
  basic_string <charT, traits, Allocator> str (lhs);
  str.append (rhs);
  return str;
}

template <class charT, class traits, class Allocator>
inline basic_string <charT, traits, Allocator>
operator+ (const charT* lhs, const basic_string <charT, traits, Allocator>& rhs)
{
  basic_string <charT, traits, Allocator> str (lhs);
  str.append (rhs);
  return str;
}

template <class charT, class traits, class Allocator>
inline basic_string <charT, traits, Allocator>
operator+ (charT lhs, const basic_string <charT, traits, Allocator>& rhs)
{
  basic_string <charT, traits, Allocator> str (1, lhs);
  str.append (rhs);
  return str;
}

template <class charT, class traits, class Allocator>
inline basic_string <charT, traits, Allocator>
operator+ (const basic_string <charT, traits, Allocator>& lhs, const charT* rhs)
{
  basic_string <charT, traits, Allocator> str (lhs);
  str.append (rhs);
  return str;
}

template <class charT, class traits, class Allocator>
inline basic_string <charT, traits, Allocator>
operator+ (const basic_string <charT, traits, Allocator>& lhs, charT rhs)
{
  basic_string <charT, traits, Allocator> str (lhs);
  str.append (1, rhs);
  return str;
}

template <class charT, class traits, class Allocator>
inline bool
operator== (const basic_string <charT, traits, Allocator>& lhs,
       const basic_string <charT, traits, Allocator>& rhs)
{
  return (lhs.compare (rhs) == 0);
}
```

```
template <class charT, class traits, class Allocator>
inline bool
operator== (const charT* lhs, const basic_string <charT, traits, Allocator>& rhs)
{
  return (rhs.compare (lhs) == 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator== (const basic_string <charT, traits, Allocator>& lhs, const charT* rhs)
{
  return (lhs.compare (rhs) == 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator!= (const charT* lhs, const basic_string <charT, traits, Allocator>& rhs)
{
  return (rhs.compare (lhs) != 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator!= (const basic_string <charT, traits, Allocator>& lhs, const charT* rhs)
{
  return (lhs.compare (rhs) != 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator< (const basic_string <charT, traits, Allocator>& lhs,
       const basic_string <charT, traits, Allocator>& rhs)
{
  return (lhs.compare (rhs) < 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator< (const charT* lhs, const basic_string <charT, traits, Allocator>& rhs)
{
  return (rhs.compare (lhs) > 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator< (const basic_string <charT, traits, Allocator>& lhs, const charT* rhs)
{
  return (lhs.compare (rhs) < 0);
```

```
}

template <class charT, class traits, class Allocator>
inline bool
operator> (const charT* lhs, const basic_string <charT, traits, Allocator>& rhs)
{
  return (rhs.compare (lhs) < 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator> (const basic_string <charT, traits, Allocator>& lhs, const charT* rhs)
{
  return (lhs.compare (rhs) > 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator<= (const charT* lhs, const basic_string <charT, traits, Allocator>& rhs)
{
  return (rhs.compare (lhs) >= 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator<= (const basic_string <charT, traits, Allocator>& lhs, const charT* rhs)
{
  return (lhs.compare (rhs) <= 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator>= (const charT* lhs, const basic_string <charT, traits, Allocator>& rhs)
{
  return (rhs.compare (lhs) <= 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator>= (const basic_string <charT, traits, Allocator>& lhs, const charT* rhs)
{
  return (lhs.compare (rhs) >= 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator!= (const basic_string <charT, traits, Allocator>& lhs,
       const basic_string <charT, traits, Allocator>& rhs)
{
```

```
  return (lhs.compare (rhs) != 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator> (const basic_string <charT, traits, Allocator>& lhs,
      const basic_string <charT, traits, Allocator>& rhs)
{
  return (lhs.compare (rhs) > 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator<= (const basic_string <charT, traits, Allocator>& lhs,
       const basic_string <charT, traits, Allocator>& rhs)
{
  return (lhs.compare (rhs) <= 0);
}

template <class charT, class traits, class Allocator>
inline bool
operator>= (const basic_string <charT, traits, Allocator>& lhs,
       const basic_string <charT, traits, Allocator>& rhs)
{
  return (lhs.compare (rhs) >= 0);
}

class istream; class ostream;
template <class charT, class traits, class Allocator> istream&
operator>> (istream&, basic_string <charT, traits, Allocator>&);
template <class charT, class traits, class Allocator> ostream&
operator<< (ostream&, const basic_string <charT, traits, Allocator>&);
template <class charT, class traits, class Allocator> istream&
getline (istream&, basic_string <charT, traits, Allocator>&, charT delim = '\n');

} // extern "C++"

#include <std/bastring.cc>

#endif
```