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

```
// Member templates for the -*- C++ -*- string classes.
// Copyright (C) 1994 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.

extern "C++" {
template <class charT, class traits, class Allocator>
inline void * basic_string <charT, traits, Allocator>::Rep::
operator new (size_t s, size_t extra)
{
  return Allocator::allocate(s + extra * sizeof (charT));
}

template <class charT, class traits, class Allocator>
inline void basic_string <charT, traits, Allocator>::Rep::
operator delete (void * ptr)
{
  Allocator::deallocate(ptr, sizeof(Rep) +
             reinterpret_cast<Rep *>(ptr)->res *
             sizeof (charT));
}

template <class charT, class traits, class Allocator>
inline size_t basic_string <charT, traits, Allocator>::Rep::
frob_size (size_t s)
{
```

```
  size_t i = 16;
  while (i < s) i *= 2;
  return i;
}

template <class charT, class traits, class Allocator>
inline basic_string <charT, traits, Allocator>::Rep *
basic_string <charT, traits, Allocator>::Rep::
create (size_t extra)
{
  extra = frob_size (extra + 1);
  Rep *p = new (extra) Rep;
  p->res = extra;
  p->ref = 1;
  p->selfish = false;
  return p;
}

template <class charT, class traits, class Allocator>
charT * basic_string <charT, traits, Allocator>::Rep::
clone ()
{
  Rep *p = Rep::create (len);
  p->copy (0, data (), len);
  p->len = len;
  return p->data ();
}

template <class charT, class traits, class Allocator>
inline bool basic_string <charT, traits, Allocator>::Rep::
excess_slop (size_t s, size_t r)
{
  return 2 * (s <= 16 ? 16 : s) < r;
}

template <class charT, class traits, class Allocator>
inline bool basic_string <charT, traits, Allocator>::
check_realloc (basic_string::size_type s) const
{
  s += sizeof (charT);
  rep ()->selfish = false;
  return (rep ()->ref > 1
     || s > capacity ()
     || Rep::excess_slop (s, capacity ()));
}

template <class charT, class traits, class Allocator>
void basic_string <charT, traits, Allocator>::
alloc (basic_string::size_type size, bool save)
```

```
{
  if (! check_realloc (size))
    return;

  Rep *p = Rep::create (size);

  if (save)
    {
      p->copy (0, data (), length ());
      p->len = length ();
    }
  else
    p->len = 0;

  repup (p);
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>&
basic_string <charT, traits, Allocator>::
replace (size_type pos1, size_type n1,
    const basic_string& str, size_type pos2, size_type n2)
{
  const size_t len2 = str.length ();

  if (pos1 == 0 && n1 >= length () && pos2 == 0 && n2 >= len2)
    return operator= (str);

  OUTOFRANGE (pos2 > len2);

  if (n2 > len2 - pos2)
    n2 = len2 - pos2;

  return replace (pos1, n1, str.data () + pos2, n2);
}

template <class charT, class traits, class Allocator>
inline void basic_string <charT, traits, Allocator>::Rep::
copy (size_t pos, const charT *s, size_t n)
{
  if (n)
    traits::copy (data () + pos, s, n);
}

template <class charT, class traits, class Allocator>
inline void basic_string <charT, traits, Allocator>::Rep::
move (size_t pos, const charT *s, size_t n)
{
  if (n)
```

```
    traits::move (data () + pos, s, n);
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>&
basic_string <charT, traits, Allocator>::
replace (size_type pos, size_type n1, const charT* s, size_type n2)
{
  const size_type len = length ();
  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));
      p->copy (pos, s, n2);
      repup (p);
    }
  else
    {
      rep ()->move (pos + n2, data () + pos + n1, len - (pos + n1));
      rep ()->copy (pos, s, n2);
    }
  rep ()->len = newlen;

  return *this;
}

template <class charT, class traits, class Allocator>
inline void basic_string <charT, traits, Allocator>::Rep::
set (size_t pos, const charT c, size_t n)
{
  traits::set  (data () + pos, c, n);
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>& basic_string <charT, traits, Allocator>::
replace (size_type pos, size_type n1, size_type n2, charT c)
{
  const size_t len = length ();
  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));
      p->set  (pos, c, n2);
      repup (p);
    }
  else
    {
      rep ()->move (pos + n2, data () + pos + n1, len - (pos + n1));
      rep ()->set  (pos, c, n2);
    }
  rep ()->len = newlen;

  return *this;
}

template <class charT, class traits, class Allocator>
void basic_string <charT, traits, Allocator>::
resize (size_type n, charT c)
{
  LENGTHERROR (n > max_size ());

  if (n > length ())
    append (n - length (), c);
  else
    erase (n);
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::
copy (charT* s, size_type n, size_type pos) const
{
  OUTOFRANGE (pos > length ());

  if (n > length () - pos)
    n = length () - pos;

  traits::copy (s, data () + pos, n);
  return n;
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::
```

```
find (const charT* s, size_type pos, size_type n) const
{
  size_t xpos = pos;
  for (; xpos + n <= length (); ++xpos)
    if (traits::eq (data () [xpos], *s)
    && traits::compare (data () + xpos, s, n) == 0)
      return xpos;
  return npos;
}

template <class charT, class traits, class Allocator>
inline basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::
_find (const charT* ptr, charT c, size_type xpos, size_type len)
{
  for (; xpos < len; ++xpos)
    if (traits::eq (ptr [xpos], c))
      return xpos;
  return npos;
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::
find (charT c, size_type pos) const
{
  return _find (data (), c, pos, length ());
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::
rfind (const charT* s, size_type pos, size_type n) const
{
  if (n > length ())
    return npos;

  size_t xpos = length () - n;
  if (xpos > pos)
    xpos = pos;

  for (++xpos; xpos-- > 0; )
    if (traits::eq (data () [xpos], *s)
    && traits::compare (data () + xpos, s, n) == 0)
      return xpos;
  return npos;
}

template <class charT, class traits, class Allocator>
```

```
basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::
rfind (charT c, size_type pos) const
{
  if (1 > length ())
    return npos;

  size_t xpos = length () - 1;
  if (xpos > pos)
    xpos = pos;

  for (++xpos; xpos-- > 0; )
    if (traits::eq (data () [xpos], c))
      return xpos;
  return npos;
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::
find_first_of (const charT* s, size_type pos, size_type n) const
{
  size_t xpos = pos;
  for (; xpos < length (); ++xpos)
    if (_find (s, data () [xpos], 0, n) != npos)
      return xpos;
  return npos;
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::
find_last_of (const charT* s, size_type pos, size_type n) const
{
  if (length() == 0)
    return npos;
  size_t xpos = length () - 1;
  if (xpos > pos)
    xpos = pos;
  for (++xpos; xpos-- > 0;)
    if (_find (s, data () [xpos], 0, n) != npos)
      return xpos;
  return npos;
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::
find_first_not_of (const charT* s, size_type pos, size_type n) const
```

```
{
  size_t xpos = pos;
  for (; xpos < length (); ++xpos)
    if (_find (s, data () [xpos], 0, n) == npos)
      return xpos;
  return npos;
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::
find_first_not_of (charT c, size_type pos) const
{
  size_t xpos = pos;
  for (; xpos < length (); ++xpos)
    if (traits::ne (data () [xpos], c))
      return xpos;
  return npos;
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::
find_last_not_of (const charT* s, size_type pos, size_type n) const
{
  if (length() == 0)
    return npos;
  size_t xpos = length () - 1;
  if (xpos > pos)
    xpos = pos;
  for (++xpos; xpos-- > 0;)
    if (_find (s, data () [xpos], 0, n) == npos)
      return xpos;
  return npos;
}

template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::
find_last_not_of (charT c, size_type pos) const
{
  if (length() == 0)
    return npos;
  size_t xpos = length () - 1;
  if (xpos > pos)
    xpos = pos;
  for (++xpos; xpos-- > 0;)
    if (traits::ne (data () [xpos], c))
      return xpos;
```

```
  return npos;
}

template <class charT, class traits, class Allocator>
int basic_string <charT, traits, Allocator>::
compare (const basic_string& str, size_type pos, size_type n) const
{
  OUTOFRANGE (pos > length ());

  size_t rlen = length () - pos;
  if (rlen > n)
    rlen = n;
  if (rlen > str.length ())
    rlen = str.length ();
  int r = traits::compare (data () + pos, str.data (), rlen);
  if (r != 0)
    return r;
  if (rlen == n)
    return 0;
  return (length () - pos) - str.length ();
}

template <class charT, class traits, class Allocator>
int basic_string <charT, traits, Allocator>::
compare (const charT* s, size_type pos, size_type n) const
{
  OUTOFRANGE (pos > length ());

  size_t rlen = length () - pos;
  if (rlen > n)
    rlen = n;
  int r = traits::compare (data () + pos, s, rlen);
  if (r != 0)
    return r;
  return (length () - pos) - n;
}

#include <iostream.h>

template <class charT, class traits, class Allocator>
istream &
operator>> (istream &is, basic_string <charT, traits, Allocator> &s)
{
  int w = is.width (0);
  if (is.ipfx0 ())
    {
      register streambuf *sb = is.rdbuf ();
      s.resize (0);
      while (1)
```

```
    {
      int ch = sb->sbumpc ();
      if (ch == EOF)
        {
          is.setstate (ios::eofbit);
          break;
        }
      else if (traits::is_del (ch))
        {
          sb->sungetc ();
          break;
        }
      s += ch;
      if (--w == 1)
        break;
    }
    }

  is.isfx ();
  if (s.length () == 0)
    is.setstate (ios::failbit);

  return is;
}

template <class charT, class traits, class Allocator>
ostream &
operator<< (ostream &o, const basic_string <charT, traits, Allocator>& s)
{
  return o.write (s.data (), s.length ());
}

template <class charT, class traits, class Allocator>
istream&
getline (istream &is, basic_string <charT, traits, Allocator>& s, charT delim)
{
  if (is.ipfx1 ())
    {
      _IO_size_t count = 0;
      streambuf *sb = is.rdbuf ();
      s.resize (0);

      while (1)
    {
      int ch = sb->sbumpc ();
      if (ch == EOF)
        {
          is.setstate (count == 0
                 ? (ios::failbit|ios::eofbit)
```

```
                : ios::eofbit);
          break;
        }

      ++count;

      if (ch == delim)
        break;

      s += ch;

      if (s.length () == s.npos - 1)
        {
          is.setstate (ios::failbit);
          break;
        }
    }
    }

  // We need to be friends with istream to do this.
  // is._gcount = count;
  is.isfx ();

  return is;
}

// static data member of class basic_string<>
template <class charT, class traits, class Allocator>
basic_string <charT, traits, Allocator>::Rep
basic_string<charT, traits, Allocator>::nilRep = { 0, 0, 1, false };

template <class charT, class traits, class Allocator>
const basic_string <charT, traits, Allocator>::size_type
basic_string <charT, traits, Allocator>::npos;

} // extern "C++"
```