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

```
// The template and inlines for the -*- C++ -*- complex number 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 in the 27 May 1994
// C++ working paper, ANSI document X3J16/94-0098.

#ifndef __COMPLEXT__
#define __COMPLEXT__

#ifdef __GNUG__
#pragma interface
#endif

#include <cmath>

#if ! defined (__GNUG__) && ! defined (__attribute__)
#define __attribute__(foo) /* Ignore.  */
#endif

class istream;    // 用於 operator>>
class ostream;    // 用於 operator<<

extern "C++" {
template <class _FLT> class complex;
// 以下四個是全域函式，負責複數的四則運算。詳見稍後的函式定義。
template <class _FLT> complex<_FLT>&
  __doapl (complex<_FLT>* ths, const complex<_FLT>& r);     // plus
template <class _FLT> complex<_FLT>&
```

```
  __doami (complex<_FLT>* ths, const complex<_FLT>& r);     // minus
template <class _FLT> complex<_FLT>&
  __doaml (complex<_FLT>* ths, const complex<_FLT>& r);     // multiplies
template <class _FLT> complex<_FLT>&
  __doadv (complex<_FLT>* ths, const complex<_FLT>& r);     // division

template <class _FLT>
class complex
{
public:
  complex (_FLT r = 0, _FLT i = 0): re (r), im (i) { }
  complex& operator += (const complex&);
  complex& operator -= (const complex&);
  complex& operator *= (const complex&);
  complex& operator /= (const complex&);
  _FLT real () const { return re; }  // 取出實部
  _FLT imag () const { return im; }  // 取出虛部
private:
  _FLT re, im;    // 實部 real, 虛部 imaginary

  // 以下奇特語法 <>，見C++ Primer p834 “bound friend function template”
  friend complex& __doapl<> (complex *, const complex&);
  friend complex& __doami<> (complex *, const complex&);
  friend complex& __doaml<> (complex *, const complex&);
  friend complex& __doadv<> (complex *, const complex&);
};

// 宣告特化類別（specializations）
class complex<float>;
class complex<double>;
class complex<long double>;

// 複數加法 plus
template <class _FLT>
inline complex<_FLT>&
__doapl (complex<_FLT>* ths, const complex<_FLT>& r)
{
  ths->re += r.re;
  ths->im += r.im;
  return *ths;
}
template <class _FLT>
inline complex<_FLT>&
complex<_FLT>::operator += (const complex<_FLT>& r)
{
  return __doapl (this, r);
}

// 複數減法 minus
```

```
template <class _FLT>
inline complex<_FLT>&
__doami (complex<_FLT>* ths, const complex<_FLT>& r)
{
  ths->re -= r.re;
  ths->im -= r.im;
  return *ths;
}
template <class _FLT>
inline complex<_FLT>&
complex<_FLT>::operator -= (const complex<_FLT>& r)
{
  return __doami (this, r);
}

// 複數乘法 multiplies
template <class _FLT>
inline complex<_FLT>&
__doaml (complex<_FLT>* ths, const complex<_FLT>& r)
{
  _FLT f = ths->re * r.re - ths->im * r.im;
  ths->im = ths->re * r.im + ths->im * r.re;
  ths->re = f;
  return *ths;
}
template <class _FLT>
inline complex<_FLT>&
complex<_FLT>::operator *= (const complex<_FLT>& r)
{
  return __doaml (this, r);
}

// 複數除法 division
template <class _FLT>
inline complex<_FLT>&
complex<_FLT>::operator /= (const complex<_FLT>& r)
{
  return __doadv (this, r);     // 複數除法定義於 complext.cc
}

// 以下的 imag() 和 real() 是全域函式，方便取得複數的虛部和實部
// 都是inline 函式，效率不比 member function imag(), real() 差
template <class _FLT> inline _FLT
imag (const complex<_FLT>& x) __attribute__ ((const));

template <class _FLT> inline _FLT
imag (const complex<_FLT>& x)
{
  return x.imag ();
```

```
}

template <class _FLT> inline _FLT
real (const complex<_FLT>& x) __attribute__ ((const));

template <class _FLT> inline _FLT
real (const complex<_FLT>& x)
{
  return x.real ();
}

// 全域函式，兩複數相加
template <class _FLT> inline complex<_FLT>
operator + (const complex<_FLT>& x, const complex<_FLT>& y) __attribute__
((const));

template <class _FLT> inline complex<_FLT>
operator + (const complex<_FLT>& x, const complex<_FLT>& y)
{
  return complex<_FLT> (real (x) + real (y), imag (x) + imag (y));
}

// 全域函式，複數 + 實數
template <class _FLT> inline complex<_FLT>
operator + (const complex<_FLT>& x, _FLT y) __attribute__ ((const));

template <class _FLT> inline complex<_FLT>
operator + (const complex<_FLT>& x, _FLT y)
{
  return complex<_FLT> (real (x) + y, imag (x));
}

// 全域函式，實數 + 複數
template <class _FLT> inline complex<_FLT>
operator + (_FLT x, const complex<_FLT>& y) __attribute__ ((const));

template <class _FLT> inline complex<_FLT>
operator + (_FLT x, const complex<_FLT>& y)
{
  return complex<_FLT> (x + real (y), imag (y));
}

// 全域函式，兩複數相減
template <class _FLT> inline complex<_FLT>
operator - (const complex<_FLT>& x, const complex<_FLT>& y) __attribute__
((const));

template <class _FLT> inline complex<_FLT>
operator - (const complex<_FLT>& x, const complex<_FLT>& y)
```

```
{
  return complex<_FLT> (real (x) - real (y), imag (x) - imag (y));
}

// 全域函式，複數 - 實數
template <class _FLT> inline complex<_FLT>
operator - (const complex<_FLT>& x, _FLT y) __attribute__ ((const));

template <class _FLT> inline complex<_FLT>
operator - (const complex<_FLT>& x, _FLT y)
{
  return complex<_FLT> (real (x) - y, imag (x));
}

// 全域函式，實數 - 複數
template <class _FLT> inline complex<_FLT>
operator - (_FLT x, const complex<_FLT>& y) __attribute__ ((const));

template <class _FLT> inline complex<_FLT>
operator - (_FLT x, const complex<_FLT>& y)
{
  return complex<_FLT> (x - real (y), - imag (y));
}

// 全域函式，兩複數相乘
template <class _FLT> inline complex<_FLT>
operator * (const complex<_FLT>& x, const complex<_FLT>& y) __attribute__
((const));

template <class _FLT> inline complex<_FLT>
operator * (const complex<_FLT>& x, const complex<_FLT>& y)
{
  return complex<_FLT> (real (x) * real (y) - imag (x) * imag (y),
                real (x) * imag (y) + imag (x) * real (y));
}

// 全域函式，複數 * 實數
template <class _FLT> inline complex<_FLT>
operator * (const complex<_FLT>& x, _FLT y) __attribute__ ((const));

template <class _FLT> inline complex<_FLT>
operator * (const complex<_FLT>& x, _FLT y)
{
  return complex<_FLT> (real (x) * y, imag (x) * y);
}

// 全域函式，實數 * 複數
template <class _FLT> inline complex<_FLT>
operator * (_FLT x, const complex<_FLT>& y) __attribute__ ((const));
```

```
template <class _FLT> inline complex<_FLT>
operator * (_FLT x, const complex<_FLT>& y)
{
  return complex<_FLT> (x * real (y), x * imag (y));
}

// 全域函式，複數 / 實數
template <class _FLT> complex<_FLT>
operator / (const complex<_FLT>& x, _FLT y) __attribute__ ((const));

template <class _FLT> complex<_FLT>
operator / (const complex<_FLT>& x, _FLT y)
{
  return complex<_FLT> (real (x) / y, imag (x) / y);
}

// 以下是一元運算子 positive，不是加法
template <class _FLT> inline complex<_FLT>
operator + (const complex<_FLT>& x) __attribute__ ((const));

template <class _FLT> inline complex<_FLT>
operator + (const complex<_FLT>& x)
{
  return x;
}

// 以下是一元運算子 negative，不是減法
template <class _FLT> inline complex<_FLT>
operator - (const complex<_FLT>& x) __attribute__ ((const));

template <class _FLT> inline complex<_FLT>
operator - (const complex<_FLT>& x)
{
  return complex<_FLT> (-real (x), -imag (x));
}

// 全域函式，比較兩複數是否相等
template <class _FLT> inline bool
operator == (const complex<_FLT>& x, const complex<_FLT>& y) __attribute__
((const));

template <class _FLT> inline bool
operator == (const complex<_FLT>& x, const complex<_FLT>& y)
{
  return real (x) == real (y) && imag (x) == imag (y);
}

// 全域函式，比較某複數是否等於某實數
```

```
template <class _FLT> inline bool
operator == (const complex<_FLT>& x, _FLT y) __attribute__ ((const));

template <class _FLT> inline bool
operator == (const complex<_FLT>& x, _FLT y)
{
  return real (x) == y && imag (x) == 0;
}

// 全域函式，比較某實數是否等於某複數
template <class _FLT> inline bool
operator == (_FLT x, const complex<_FLT>& y) __attribute__ ((const));

template <class _FLT> inline bool
operator == (_FLT x, const complex<_FLT>& y)
{
  return x == real (y) && imag (y) == 0;
}

// 全域函式，比較兩複數是否不相等
template <class _FLT> inline bool
operator != (const complex<_FLT>& x, const complex<_FLT>& y) __attribute__
((const));

template <class _FLT> inline bool
operator != (const complex<_FLT>& x, const complex<_FLT>& y)
{
  return real (x) != real (y) || imag (x) != imag (y);
}

// 全域函式，比較某複數是否不等於某實數
template <class _FLT> inline bool
operator != (const complex<_FLT>& x, _FLT y) __attribute__ ((const));

template <class _FLT> inline bool
operator != (const complex<_FLT>& x, _FLT y)
{
  return real (x) != y || imag (x) != 0;
}

// 全域函式，比較某實數是否不等於某複數
template <class _FLT> inline bool
operator != (_FLT x, const complex<_FLT>& y) __attribute__ ((const));

template <class _FLT> inline bool
operator != (_FLT x, const complex<_FLT>& y)
{
  return x != real (y) || imag (y) != 0;
}
```

```
// 各種複數運算。

// Some targets don't provide a prototype for hypot when -ansi.
extern "C" double hypot (double, double) __attribute__ ((const));

// 複數的絕對值（absolute value, or modulus），亦即其大小（magnitude）
template <class _FLT> inline _FLT
abs (const complex<_FLT>& x) __attribute__ ((const));

template <class _FLT> inline _FLT
abs (const complex<_FLT>& x)
{
  return hypot (real (x), imag (x));
}

// 複數在極座標中的幅角（argument, or amplitude）
template <class _FLT> inline _FLT
arg (const complex<_FLT>& x) __attribute__ ((const));

template <class _FLT> inline _FLT
arg (const complex<_FLT>& x)
{
  return atan2 (imag (x), real (x));
}

// 根據極座標（極值和幅角）產生一個複數
template <class _FLT> inline complex<_FLT>
polar (_FLT r, _FLT t) __attribute__ ((const));

template <class _FLT> inline complex<_FLT>
polar (_FLT r, _FLT t)
{
  return complex<_FLT> (r * cos (t), r * sin (t));
}

// 共軛複數
template <class _FLT> inline complex<_FLT>
conj (const complex<_FLT>& x)  __attribute__ ((const));

template <class _FLT> inline complex<_FLT>
conj (const complex<_FLT>& x)
{
  return complex<_FLT> (real (x), -imag (x));
}

// 複數絕對值的平方（squared absolute value）
template <class _FLT> inline _FLT
norm (const complex<_FLT>& x) __attribute__ ((const));
```

```
template <class _FLT> inline _FLT
norm (const complex<_FLT>& x)
{
  return real (x) * real (x) + imag (x) * imag (x);
}

// 各種複數運算。實際定義於complext.cc
template <class _FLT> complex<_FLT>
  operator / (const complex<_FLT>&, const complex<_FLT>&) __attribute__ ((const));
template <class _FLT> complex<_FLT>
  operator / (_FLT, const complex<_FLT>&) __attribute__ ((const));
template <class _FLT> complex<_FLT>
  cos (const complex<_FLT>&) __attribute__ ((const));
template <class _FLT> complex<_FLT>
  cosh (const complex<_FLT>&) __attribute__ ((const));
template <class _FLT> complex<_FLT>
  exp (const complex<_FLT>&) __attribute__ ((const));
template <class _FLT> complex<_FLT>
  log (const complex<_FLT>&) __attribute__ ((const));
template <class _FLT> complex<_FLT>
  pow (const complex<_FLT>&, const complex<_FLT>&) __attribute__ ((const));
template <class _FLT> complex<_FLT>
  pow (const complex<_FLT>&, _FLT) __attribute__ ((const));
template <class _FLT> complex<_FLT>
  pow (const complex<_FLT>&, int) __attribute__ ((const));
template <class _FLT> complex<_FLT>
  pow (_FLT, const complex<_FLT>&) __attribute__ ((const));
template <class _FLT> complex<_FLT>
  sin (const complex<_FLT>&) __attribute__ ((const));
template <class _FLT> complex<_FLT>
  sinh (const complex<_FLT>&) __attribute__ ((const));
template <class _FLT> complex<_FLT>
  sqrt (const complex<_FLT>&) __attribute__ ((const));

template <class _FLT> istream& operator >> (istream&, complex<_FLT>&);
template <class _FLT> ostream& operator << (ostream&, const complex<_FLT>&);
} // extern "C++"

// Specializations and such

#include <std/fcomplex.h>
#include <std/dcomplex.h>
#include <std/ldcomplex.h>

#endif
```