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

```
/*
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
 * Copyright (c) 1996
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

/* NOTE: This is an internal header file, included by other STL headers.
 *   You should not attempt to use it directly.
 */

#ifndef __SGI_STL_INTERNAL_FUNCTION_H
#define __SGI_STL_INTERNAL_FUNCTION_H

__STL_BEGIN_NAMESPACE

// C++ Standard 規定，每一個 Adaptable Unary Function 都必須繼承此類別
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};

// C++ Standard 規定，每一個 Adaptable Binary Function 都必須繼承此類別
template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
```

```
};

// 以下 6 個為算術類（Arithmetic）仿函式
template <class T>
struct plus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x + y; }
};

template <class T>
struct minus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x - y; }
};

template <class T>
struct multiplies : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x * y; }
};

template <class T>
struct divides : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x / y; }
};

// 所謂運算op的證同元素（identity element）是指，數值A與此元素做op運算，
// 會得到A自己。
// 加法的證同元素（identity element）為 0。
template <class T> inline T identity_element(plus<T>) { return T(0); }

// 乘法的證同元素（identity element）為 1
// 應用於 <stl_numerics.h> 的 power().
template <class T> inline T identity_element(multiplies<T>) { return T(1); }

template <class T>
struct modulus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x % y; }
};

template <class T>
struct negate : public unary_function<T, T> {
    T operator()(const T& x) const { return -x; }
};

// 以下 6 個為相對關係類（Relational）仿函式
template <class T>
struct equal_to : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x == y; }
};

template <class T>
```

```
struct not_equal_to : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x != y; }
};

template <class T>
struct greater : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x > y; }
};

template <class T>
struct less : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x < y; }
};

template <class T>
struct greater_equal : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x >= y; }
};

template <class T>
struct less_equal : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x <= y; }
};

// 以下 3 個為邏輯運算類（Logical）仿函式
template <class T>
struct logical_and : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x && y; }
};

template <class T>
struct logical_or : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x || y; }
};

template <class T>
struct logical_not : public unary_function<T, bool> {
    bool operator()(const T& x) const { return !x; }
};

// ---- 以上都是 functor，以下都是 function adapter

// 以下配接器用來表示某個 Adaptable Predicate 的邏輯負值（logical negation）
template <class Predicate>
class unary_negate
  : public unary_function<typename Predicate::argument_type, bool> {
protected:
  Predicate pred;
public:
```

```
  explicit unary_negate(const Predicate& x) : pred(x) {}
  bool operator()(const typename Predicate::argument_type& x) const {
    return !pred(x);
  }
};

// 輔助函式，使我們得以方便使用 unary_negate<Pred>
template <class Predicate>
inline unary_negate<Predicate> not1(const Predicate& pred) {
  return unary_negate<Predicate>(pred);
}

// 以下配接器用來表示某個 Adaptable Binary Predicate 的邏輯負值
template <class Predicate>
class binary_negate
  : public binary_function<typename Predicate::first_argument_type,
                           typename Predicate::second_argument_type,
                           bool> {
protected:
  Predicate pred;
public:
  explicit binary_negate(const Predicate& x) : pred(x) {}
  bool operator()(const typename Predicate::first_argument_type& x,
                  const typename Predicate::second_argument_type& y) const {
    return !pred(x, y);
  }
};

// 輔助函式，使我們得以方便使用 bineary_negate<Pred>
template <class Predicate>
inline binary_negate<Predicate> not2(const Predicate& pred) {
  return binary_negate<Predicate>(pred);
}

// 以下配接器用來將某個 Adaptable Binary function 轉換為 Unary Function
template <class Operation>
class binder1st
  : public unary_function<typename Operation::second_argument_type,
                          typename Operation::result_type> {
protected:
  Operation op;
  typename Operation::first_argument_type value;
public:
  // 以下 ctor 建立 op 和 value。
  binder1st(const Operation& x,
            const typename Operation::first_argument_type& y)
      : op(x), value(y) {}
  typename Operation::result_type
  operator()(const typename Operation::second_argument_type& x) const {
```

```
    return op(value, x);     // 將 value 繫結（binding）為第一引數
                             // operator() 被呼叫時的引數成為op的第二引數
  }
};

// 輔助函式，讓我們得以方便使用 binder1st<Op>
// 用法：例如
template <class Operation, class T>
inline binder1st<Operation> bind1st(const Operation& op, const T& x)
{
  typedef typename Operation::first_argument_type arg1_type;
  return binder1st<Operation>(op, arg1_type(x));
         // 以上把x當做op的第一引數型別
         // 語法分析：binder1st<T>() 是產生一個暫時物件，() 之內是ctor參數。
         // arg1_type() 是強制轉型動作。
}

// 以下配接器用來將某個 Adaptable Binary function 轉換為 Unary Function
template <class Operation>
class binder2nd
  : public unary_function<typename Operation::first_argument_type,
                          typename Operation::result_type> {
protected:
  Operation op;
  typename Operation::second_argument_type value;
public:
  // 以下 ctor 建立 op 和 value。
  binder2nd(const Operation& x,
            const typename Operation::second_argument_type& y)
      : op(x), value(y) {}
  typename Operation::result_type
  operator()(const typename Operation::first_argument_type& x) const {
    return op(x, value);     // 將 value 繫結（binding）為第二引數
                             // operator() 被呼叫時的引數，將成為op的第一引數
  }
};

// 輔助函式，讓我們得以方便使用 binder2nd<Op>
// 用法：例如 bind2nd(less<int>(), 5)
template <class Operation, class T>
inline binder2nd<Operation> bind2nd(const Operation& op, const T& x)
{
  typedef typename Operation::second_argument_type arg2_type;
  return binder2nd<Operation>(op, arg2_type(x));
         // 以上把x當做op的第二引數型別
         // 語法分析：binder2nd<T>() 是產生一個暫時物件，() 之內是ctor參數。
         // arg2_type() 是強制轉型動作。
}
```

```
// 已知兩個 Adaptable Unary Functions f,g，以下配接器用來產生一個 h，
// 使 h(x) = f(g(x))
template <class Operation1, class Operation2>
class unary_compose
  : public unary_function<typename Operation2::argument_type,
                          typename Operation1::result_type> {
protected:
  Operation1 op1;
  Operation2 op2;
public:
  unary_compose(const Operation1& x, const Operation2& y) : op1(x), op2(y) {}
  typename Operation1::result_type
  operator()(const typename Operation2::argument_type& x) const {
    return op1(op2(x));
  }
};

// 輔助函式，讓我們得以方便使用 unary_compose<Op1,Op2>
template <class Operation1, class Operation2>
inline unary_compose<Operation1, Operation2>
compose1(const Operation1& op1, const Operation2& op2) {
  return unary_compose<Operation1, Operation2>(op1, op2);
}

// 已知一個 Adaptable Binary Function f 和兩個Adaptable Unary Functions g1,g2，
// 以下配接器用來產生一個 h，使 h(x) = f(g1(x),g2(x))
template <class Operation1, class Operation2, class Operation3>
class binary_compose
  : public unary_function<typename Operation2::argument_type,
                          typename Operation1::result_type> {
protected:
  Operation1 op1;
  Operation2 op2;
  Operation3 op3;
public:
  binary_compose(const Operation1& x, const Operation2& y,
                 const Operation3& z) : op1(x), op2(y), op3(z) { }
  typename Operation1::result_type
  operator()(const typename Operation2::argument_type& x) const {
    return op1(op2(x), op3(x));
  }
};

// 輔助函式，讓我們得以方便使用 binary_compose<Op1,Op2,Op3>
template <class Operation1, class Operation2, class Operation3>
inline binary_compose<Operation1, Operation2, Operation3>
compose2(const Operation1& op1, const Operation2& op2, const Operation3& op3)
{
```

```
  return binary_compose<Operation1, Operation2, Operation3>(op1, op2, op3);
}

// 以下配接器其實就是把一個一元函式指標包起來；當仿函式被使用時，就喚起該函式指標
template <class Arg, class Result>
class pointer_to_unary_function : public unary_function<Arg, Result>
{
protected:
  Result (*ptr)(Arg); // 函式指標
public:
  pointer_to_unary_function() {}
  explicit pointer_to_unary_function(Result (*x)(Arg)) : ptr(x) {}
  Result operator()(Arg x) const { return ptr(x); }
};

// 輔助函式，讓我們得以方便使用 pointer_to_unary_function
template <class Arg, class Result>
inline pointer_to_unary_function<Arg, Result>
ptr_fun(Result (*x)(Arg)) {
  return pointer_to_unary_function<Arg, Result>(x);
}

// 以下配接器其實就是把一個二元函式指標包起來；當仿函式被使用時，就喚起該函式指標
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function : public binary_function<Arg1, Arg2, Result>
{
protected:
    Result (*ptr)(Arg1, Arg2);
public:
    pointer_to_binary_function() {}
    explicit pointer_to_binary_function(Result (*x)(Arg1, Arg2)) : ptr(x) {}
    Result operator()(Arg1 x, Arg2 y) const { return ptr(x, y); }
};

// 輔助函式，讓我們得以方便使用 pointer_to_binary_function
template <class Arg1, class Arg2, class Result>
inline pointer_to_binary_function<Arg1, Arg2, Result>
ptr_fun(Result (*x)(Arg1, Arg2)) {
  return pointer_to_binary_function<Arg1, Arg2, Result>(x);
}

// 證同函式（identity function）。任何數值通過此函式後，不會有任何改變。
// 此式運用於 <stl_set.h>，用來指定 RB-tree 所需的 KeyOfValue op.
template <class T>
struct identity : public unary_function<T, T> {
  const T& operator()(const T& x) const { return x; }
};

// 選擇函式：接受一個pair，傳回其第一元素
```

```
template <class Pair>
struct select1st : public unary_function<Pair, typename Pair::first_type>
{
  const typename Pair::first_type& operator()(const Pair& x) const
  {
    return x.first;
  }
};

// 選擇函式：接受一個pair，傳回其第二元素
template <class Pair>
struct select2nd : public unary_function<Pair, typename Pair::second_type>
{
  const typename Pair::second_type& operator()(const Pair& x) const
  {
    return x.second;
  }
};

// 投射函式：傳回第一引數，忽略第二引數
template <class Arg1, class Arg2>
struct project1st : public binary_function<Arg1, Arg2, Arg1> {
  Arg1 operator()(const Arg1& x, const Arg2&) const { return x; }
};

// 投射函式：傳回第二引數，忽略第一引數
template <class Arg1, class Arg2>
struct project2nd : public binary_function<Arg1, Arg2, Arg2> {
  Arg2 operator()(const Arg1&, const Arg2& y) const { return y; }
};

template <class Result>
struct constant_void_fun
{
  typedef Result result_type;
  result_type val;
  constant_void_fun(const result_type& v) : val(v) {}
  const result_type& operator()() const { return val; }
};

#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class Result, class Argument = Result>
#else
template <class Result, class Argument>
#endif
struct constant_unary_fun : public unary_function<Argument, Result> {
  Result val;
  constant_unary_fun(const Result& v) : val(v) {}
  const Result& operator()(const Argument&) const { return val; }
```

```
};

#ifndef __STL_LIMITED_DEFAULT_TEMPLATES
template <class Result, class Arg1 = Result, class Arg2 = Arg1>
#else
template <class Result, class Arg1, class Arg2>
#endif
struct constant_binary_fun : public binary_function<Arg1, Arg2, Result>
{
  Result val;
  constant_binary_fun(const Result& v) : val(v) {}
  const Result& operator()(const Arg1&, const Arg2&) const {
    return val;
  }
};

template <class Result>
inline constant_void_fun<Result> constant0(const Result& val)
{
  return constant_void_fun<Result>(val);
}

template <class Result>
inline constant_unary_fun<Result,Result> constant1(const Result& val)
{
  return constant_unary_fun<Result,Result>(val);
}

template <class Result>
inline constant_binary_fun<Result,Result,Result> constant2(const Result& val)
{
  return constant_binary_fun<Result,Result,Result>(val);
}

// Note: this code assumes that int is 32 bits.
class subtractive_rng : public unary_function<unsigned int, unsigned int> {
private:
  unsigned int table[55];
  size_t index1;
  size_t index2;
public:
  unsigned int operator()(unsigned int limit) {
    index1 = (index1 + 1) % 55;
    index2 = (index2 + 1) % 55;
    table[index1] = table[index1] - table[index2];
    return table[index1] % limit;
  }

  void initialize(unsigned int seed)
```

```
  {
    unsigned int k = 1;
    table[54] = seed;
    size_t i;
    for (i = 0; i < 54; i++) {
        size_t ii = (21 * (i + 1) % 55) - 1;
        table[ii] = k;
        k = seed - k;
        seed = table[ii];
    }
    for (int loop = 0; loop < 4; loop++) {
        for (i = 0; i < 55; i++)
            table[i] = table[i] - table[(1 + i + 30) % 55];
    }
    index1 = 0;
    index2 = 31;
  }

  subtractive_rng(unsigned int seed) { initialize(seed); }
  subtractive_rng() { initialize(161803398u); }
};


// Adapter function objects: pointers to member functions.

// 這個族群一共有 16 = 2^4 個function objects。
//  (1) 「無任何引數」vs 「有一個引數」
//  (2) 「透過 pointer 呼叫」vs「透過 reference 呼叫」
//  (3) 「void 回返型別」vs「non-void 回返型別」
//  (4) 「const成員函式」vs「non-const成員函式」

// 注意，(4) 並未出現於 8/97 C++ 標準草案中。草案只允許這些配接器用於
//  non-const 函式身上。這很可能會在C++ 標準定案之前被修正。
// 注意，(3) 其實只是個workaround: 就草案之規範而言，編譯器應該以
//  相同的方式處理void 和 non-void，不過此一性質尚未能夠被廣為實現。
//  如果你的編譯器支援partial specialization，那麼你可以只使用
//  那些傳回值為void 的 member functions。

// 所有的複雜都只存在於function objects 內部。你可以忽略它們，只使用
//  輔助函式 mem_fun, mem_fun_ref, mem_fun1 和 mem_fun1_ref，
//  它們會產生適當的配接器。

// 「無任何引數」、「透過 pointer 呼叫」、「non-const成員函式」
template <class S, class T>
class mem_fun_t : public unary_function<T*, S> {
public:
  explicit mem_fun_t(S (T::*pf)()) : f(pf) {}
  S operator()(T* p) const { return (p->*f)(); }
private:
```

```
  S (T::*f)();
};

// 「無任何引數」、「透過 pointer 呼叫」、「const成員函式」
template <class S, class T>
class const_mem_fun_t : public unary_function<const T*, S> {
public:
  explicit const_mem_fun_t(S (T::*pf)() const) : f(pf) {}
  S operator()(const T* p) const { return (p->*f)(); }
private:
  S (T::*f)() const;
};

// 「無任何引數」、「透過 reference 呼叫」、「non-const成員函式」
template <class S, class T>
class mem_fun_ref_t : public unary_function<T, S> {
public:
  explicit mem_fun_ref_t(S (T::*pf)()) : f(pf) {}
  S operator()(T& r) const { return (r.*f)(); }
private:
  S (T::*f)();
};

// 「無任何引數」、「透過 reference 呼叫」、「const成員函式」
template <class S, class T>
class const_mem_fun_ref_t : public unary_function<T, S> {
public:
  explicit const_mem_fun_ref_t(S (T::*pf)() const) : f(pf) {}
  S operator()(const T& r) const { return (r.*f)(); }
private:
  S (T::*f)() const;
};

// 「有一個引數」、「透過 pointer 呼叫」、「non-const成員函式」
template <class S, class T, class A>
class mem_fun1_t : public binary_function<T*, A, S> {
public:
  explicit mem_fun1_t(S (T::*pf)(A)) : f(pf) {}
  S operator()(T* p, A x) const { return (p->*f)(x); }
private:
  S (T::*f)(A);
};

// 「有一個引數」、「透過 pointer 呼叫」、「const成員函式」
template <class S, class T, class A>
class const_mem_fun1_t : public binary_function<const T*, A, S> {
public:
  explicit const_mem_fun1_t(S (T::*pf)(A) const) : f(pf) {}
  S operator()(const T* p, A x) const { return (p->*f)(x); }
```

```
private:
  S (T::*f)(A) const;
};

// 「有一個引數」、「透過 reference 呼叫」、「non-const成員函式」
template <class S, class T, class A>
class mem_fun1_ref_t : public binary_function<T, A, S> {
public:
  explicit mem_fun1_ref_t(S (T::*pf)(A)) : f(pf) {}
  S operator()(T& r, A x) const { return (r.*f)(x); }
private:
  S (T::*f)(A);
};

// 「有一個引數」、「透過 reference 呼叫」、「const成員函式」
template <class S, class T, class A>
class const_mem_fun1_ref_t : public binary_function<T, A, S> {
public:
  explicit const_mem_fun1_ref_t(S (T::*pf)(A) const) : f(pf) {}
  S operator()(const T& r, A x) const { return (r.*f)(x); }
private:
  S (T::*f)(A) const;
};

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class T>
class mem_fun_t<void, T> : public unary_function<T*, void> {
public:
  explicit mem_fun_t(void (T::*pf)()) : f(pf) {}
  void operator()(T* p) const { (p->*f)(); }
private:
  void (T::*f)();
};

template <class T>
class const_mem_fun_t<void, T> : public unary_function<const T*, void>
{
public:
  explicit const_mem_fun_t(void (T::*pf)() const) : f(pf) {}
  void operator()(const T* p) const { (p->*f)(); }
private:
  void (T::*f)() const;
};

template <class T>
class mem_fun_ref_t<void, T> : public unary_function<T, void> {
public:
  explicit mem_fun_ref_t(void (T::*pf)()) : f(pf) {}
```

```
  void operator()(T& r) const { (r.*f)(); }
private:
  void (T::*f)();
};

template <class T>
class const_mem_fun_ref_t<void, T> : public unary_function<T, void> {
public:
  explicit const_mem_fun_ref_t(void (T::*pf)() const) : f(pf) {}
  void operator()(const T& r) const { (r.*f)(); }
private:
  void (T::*f)() const;
};

template <class T, class A>
class mem_fun1_t<void, T, A> : public binary_function<T*, A, void> {
public:
  explicit mem_fun1_t(void (T::*pf)(A)) : f(pf) {}
  void operator()(T* p, A x) const { (p->*f)(x); }
private:
  void (T::*f)(A);
};

template <class T, class A>
class const_mem_fun1_t<void, T, A> : public binary_function<const T*, A, void> {
public:
  explicit const_mem_fun1_t(void (T::*pf)(A) const) : f(pf) {}
  void operator()(const T* p, A x) const { (p->*f)(x); }
private:
  void (T::*f)(A) const;
};

template <class T, class A>
class mem_fun1_ref_t<void, T, A> : public binary_function<T, A, void>
{
public:
  explicit mem_fun1_ref_t(void (T::*pf)(A)) : f(pf) {}
  void operator()(T& r, A x) const { (r.*f)(x); }
private:
  void (T::*f)(A);
};

template <class T, class A>
class const_mem_fun1_ref_t<void, T, A> : public binary_function<T, A,
void> {
public:
  explicit const_mem_fun1_ref_t(void (T::*pf)(A) const) : f(pf) {}
  void operator()(const T& r, A x) const { (r.*f)(x); }
private:
```

```
  void (T::*f)(A) const;
};

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// Mem_fun adapter 的輔助函式。只有四個：
//  mem_fun, mem_fun_ref, mem_fun1, mem_fun1_ref.

template <class S, class T>
inline mem_fun_t<S,T> mem_fun(S (T::*f)()) {
  return mem_fun_t<S,T>(f);
}

template <class S, class T>
inline const_mem_fun_t<S,T> mem_fun(S (T::*f)() const) {
  return const_mem_fun_t<S,T>(f);
}

template <class S, class T>
inline mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)()) {
  return mem_fun_ref_t<S,T>(f);
}

template <class S, class T>
inline const_mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)() const) {
  return const_mem_fun_ref_t<S,T>(f);
}

// 注意：以下四個函式，其實可以採用和先前四個函式相同的名稱（函式多載化）。
// 事實上C++ 標準也是這麼做。我手上的G++ 2.91.57 並未遵循標準，不過只要
// 把 mem_fun1() 改名為 mem_fun()，把 mem_fun1_ref() 改名為 mem_fun()，
// 即可符合標準。
template <class S, class T, class A>
inline mem_fun1_t<S,T,A> mem_fun1(S (T::*f)(A)) {
  return mem_fun1_t<S,T,A>(f);
}

template <class S, class T, class A>
inline const_mem_fun1_t<S,T,A> mem_fun1(S (T::*f)(A) const) {
  return const_mem_fun1_t<S,T,A>(f);
}

template <class S, class T, class A>
inline mem_fun1_ref_t<S,T,A> mem_fun1_ref(S (T::*f)(A)) {
  return mem_fun1_ref_t<S,T,A>(f);
}

template <class S, class T, class A>
inline const_mem_fun1_ref_t<S,T,A> mem_fun1_ref(S (T::*f)(A) const) {
```

```
  return const_mem_fun1_ref_t<S,T,A>(f);
}

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_FUNCTION_H */

// Local Variables:
// mode:C++
// End:
```