SGI STL 3.3 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) 1996-1998
 * 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

template <class _Arg, class _Result>
struct unary_function {
  typedef _Arg argument_type;
  typedef _Result result_type;
};

template <class _Arg1, class _Arg2, class _Result>
struct binary_function {
  typedef _Arg1 first_argument_type;
  typedef _Arg2 second_argument_type;
  typedef _Result result_type;
};
```

```
template <class _Tp>
struct plus : public binary_function<_Tp,_Tp,_Tp> {
  _Tp operator()(const _Tp& __x, const _Tp& __y) const { return __x + __y; }
};

template <class _Tp>
struct minus : public binary_function<_Tp,_Tp,_Tp> {
  _Tp operator()(const _Tp& __x, const _Tp& __y) const { return __x - __y; }
};

template <class _Tp>
struct multiplies : public binary_function<_Tp,_Tp,_Tp> {
  _Tp operator()(const _Tp& __x, const _Tp& __y) const { return __x * __y; }
};

template <class _Tp>
struct divides : public binary_function<_Tp,_Tp,_Tp> {
  _Tp operator()(const _Tp& __x, const _Tp& __y) const { return __x / __y; }
};

// identity_element (not part of the C++ standard).

template <class _Tp> inline _Tp identity_element(plus<_Tp>) {
  return _Tp(0);
}
template <class _Tp> inline _Tp identity_element(multiplies<_Tp>) {
  return _Tp(1);
}

template <class _Tp>
struct modulus : public binary_function<_Tp,_Tp,_Tp>
{
  _Tp operator()(const _Tp& __x, const _Tp& __y) const { return __x % __y; }
};

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

template <class _Tp>
struct equal_to : public binary_function<_Tp,_Tp,bool>
{
  bool operator()(const _Tp& __x, const _Tp& __y) const { return __x == __y; }
};

template <class _Tp>
struct not_equal_to : public binary_function<_Tp,_Tp,bool>
```

```
{
  bool operator()(const _Tp& __x, const _Tp& __y) const { return __x != __y; }
};

template <class _Tp>
struct greater : public binary_function<_Tp,_Tp,bool>
{
  bool operator()(const _Tp& __x, const _Tp& __y) const { return __x > __y; }
};

template <class _Tp>
struct less : public binary_function<_Tp,_Tp,bool>
{
  bool operator()(const _Tp& __x, const _Tp& __y) const { return __x < __y; }
};

template <class _Tp>
struct greater_equal : public binary_function<_Tp,_Tp,bool>
{
  bool operator()(const _Tp& __x, const _Tp& __y) const { return __x >= __y; }
};

template <class _Tp>
struct less_equal : public binary_function<_Tp,_Tp,bool>
{
  bool operator()(const _Tp& __x, const _Tp& __y) const { return __x <= __y; }
};

template <class _Tp>
struct logical_and : public binary_function<_Tp,_Tp,bool>
{
  bool operator()(const _Tp& __x, const _Tp& __y) const { return __x && __y; }
};

template <class _Tp>
struct logical_or : public binary_function<_Tp,_Tp,bool>
{
  bool operator()(const _Tp& __x, const _Tp& __y) const { return __x || __y; }
};

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

template <class _Predicate>
class unary_negate
  : public unary_function<typename _Predicate::argument_type, bool> {
```

```
protected:
  _Predicate _M_pred;
public:
  explicit unary_negate(const _Predicate& __x) : _M_pred(__x) {}
  bool operator()(const typename _Predicate::argument_type& __x) const {
    return !_M_pred(__x);
  }
};

template <class _Predicate>
inline unary_negate<_Predicate>
not1(const _Predicate& __pred)
{
  return unary_negate<_Predicate>(__pred);
}

template <class _Predicate>
class binary_negate
  : public binary_function<typename _Predicate::first_argument_type,
                           typename _Predicate::second_argument_type,
                           bool> {
protected:
  _Predicate _M_pred;
public:
  explicit binary_negate(const _Predicate& __x) : _M_pred(__x) {}
  bool operator()(const typename _Predicate::first_argument_type& __x,
                  const typename _Predicate::second_argument_type& __y) const
  {
    return !_M_pred(__x, __y);
  }
};

template <class _Predicate>
inline binary_negate<_Predicate>
not2(const _Predicate& __pred)
{
  return binary_negate<_Predicate>(__pred);
}

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:
  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);
  }
};

template <class _Operation, class _Tp>
inline binder1st<_Operation>
bind1st(const _Operation& __fn, const _Tp& __x)
{
  typedef typename _Operation::first_argument_type _Arg1_type;
  return binder1st<_Operation>(__fn, _Arg1_type(__x));
}

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:
  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);
  }
};

template <class _Operation, class _Tp>
inline binder2nd<_Operation>
bind2nd(const _Operation& __fn, const _Tp& __x)
{
  typedef typename _Operation::second_argument_type _Arg2_type;
  return binder2nd<_Operation>(__fn, _Arg2_type(__x));
}

// unary_compose and binary_compose (extensions, not part of the standard).

template <class _Operation1, class _Operation2>
class unary_compose
  : public unary_function<typename _Operation2::argument_type,
                          typename _Operation1::result_type>
{
protected:
  _Operation1 _M_fn1;
```

```
  _Operation2 _M_fn2;
public:
  unary_compose(const _Operation1& __x, const _Operation2& __y)
    : _M_fn1(__x), _M_fn2(__y) {}
  typename _Operation1::result_type
  operator()(const typename _Operation2::argument_type& __x) const {
    return _M_fn1(_M_fn2(__x));
  }
};

template <class _Operation1, class _Operation2>
inline unary_compose<_Operation1,_Operation2>
compose1(const _Operation1& __fn1, const _Operation2& __fn2)
{
  return unary_compose<_Operation1,_Operation2>(__fn1, __fn2);
}

template <class _Operation1, class _Operation2, class _Operation3>
class binary_compose
  : public unary_function<typename _Operation2::argument_type,
                          typename _Operation1::result_type> {
protected:
  _Operation1 _M_fn1;
  _Operation2 _M_fn2;
  _Operation3 _M_fn3;
public:
  binary_compose(const _Operation1& __x, const _Operation2& __y,
                 const _Operation3& __z)
    : _M_fn1(__x), _M_fn2(__y), _M_fn3(__z) { }
  typename _Operation1::result_type
  operator()(const typename _Operation2::argument_type& __x) const {
    return _M_fn1(_M_fn2(__x), _M_fn3(__x));
  }
};

template <class _Operation1, class _Operation2, class _Operation3>
inline binary_compose<_Operation1, _Operation2, _Operation3>
compose2(const _Operation1& __fn1, const _Operation2& __fn2,
         const _Operation3& __fn3)
{
  return binary_compose<_Operation1,_Operation2,_Operation3>
    (__fn1, __fn2, __fn3);
}

template <class _Arg, class _Result>
class pointer_to_unary_function : public unary_function<_Arg, _Result> {
protected:
  _Result (*_M_ptr)(_Arg);
public:
```

```
  pointer_to_unary_function() {}
  explicit pointer_to_unary_function(_Result (*__x)(_Arg)) : _M_ptr(__x) {}
  _Result operator()(_Arg __x) const { return _M_ptr(__x); }
};

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);
}

template <class _Arg1, class _Arg2, class _Result>
class pointer_to_binary_function :
  public binary_function<_Arg1,_Arg2,_Result> {
protected:
    _Result (*_M_ptr)(_Arg1, _Arg2);
public:
    pointer_to_binary_function() {}
    explicit pointer_to_binary_function(_Result (*__x)(_Arg1, _Arg2))
      : _M_ptr(__x) {}
    _Result operator()(_Arg1 __x, _Arg2 __y) const {
      return _M_ptr(__x, __y);
    }
};

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);
}

// identity is an extensions: it is not part of the standard.
template <class _Tp>
struct _Identity : public unary_function<_Tp,_Tp> {
  const _Tp& operator()(const _Tp& __x) const { return __x; }
};

template <class _Tp> struct identity : public _Identity<_Tp> {};

// select1st and select2nd are extensions: they are not part of the standard.
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;
  }
};

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;
  }
};

template <class _Pair> struct select1st : public _Select1st<_Pair> {};
template <class _Pair> struct select2nd : public _Select2nd<_Pair> {};

// project1st and project2nd are extensions: they are not part of the standard
template <class _Arg1, class _Arg2>
struct _Project1st : public binary_function<_Arg1, _Arg2, _Arg1> {
  _Arg1 operator()(const _Arg1& __x, const _Arg2&) const { return __x; }
};

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 _Arg1, class _Arg2>
struct project1st : public _Project1st<_Arg1, _Arg2> {};

template <class _Arg1, class _Arg2>
struct project2nd : public _Project2nd<_Arg1, _Arg2> {};

// constant_void_fun, constant_unary_fun, and constant_binary_fun are
// extensions: they are not part of the standard.  (The same, of course,
// is true of the helper functions constant0, constant1, and constant2.)

template <class _Result>
struct _Constant_void_fun {
  typedef _Result result_type;
  result_type _M_val;

  _Constant_void_fun(const result_type& __v) : _M_val(__v) {}
  const result_type& operator()() const { return _M_val; }
};

template <class _Result, class _Argument>
struct _Constant_unary_fun {
  typedef _Argument argument_type;
  typedef  _Result  result_type;
  result_type _M_val;

  _Constant_unary_fun(const result_type& __v) : _M_val(__v) {}
  const result_type& operator()(const _Argument&) const { return _M_val; }
};
```

```
template <class _Result, class _Arg1, class _Arg2>
struct _Constant_binary_fun {
  typedef  _Arg1   first_argument_type;
  typedef  _Arg2   second_argument_type;
  typedef  _Result result_type;
  _Result _M_val;

  _Constant_binary_fun(const _Result& __v) : _M_val(__v) {}
  const result_type& operator()(const _Arg1&, const _Arg2&) const {
    return _M_val;
  }
};

template <class _Result>
struct constant_void_fun : public _Constant_void_fun<_Result> {
  constant_void_fun(const _Result& __v) : _Constant_void_fun<_Result>(__v) {}
};


template <class _Result,
          class _Argument __STL_DEPENDENT_DEFAULT_TMPL(_Result)>
struct constant_unary_fun : public _Constant_unary_fun<_Result, _Argument>
{
  constant_unary_fun(const _Result& __v)
    : _Constant_unary_fun<_Result, _Argument>(__v) {}
};


template <class _Result,
          class _Arg1 __STL_DEPENDENT_DEFAULT_TMPL(_Result),
          class _Arg2 __STL_DEPENDENT_DEFAULT_TMPL(_Arg1)>
struct constant_binary_fun
  : public _Constant_binary_fun<_Result, _Arg1, _Arg2>
{
  constant_binary_fun(const _Result& __v)
    : _Constant_binary_fun<_Result, _Arg1, _Arg2>(__v) {}
};

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);
}

// subtractive_rng is an extension: it is not part of the standard.
// Note: this code assumes that int is 32 bits.
class subtractive_rng : public unary_function<unsigned int, unsigned int> {
private:
  unsigned int _M_table[55];
  size_t _M_index1;
  size_t _M_index2;
public:
  unsigned int operator()(unsigned int __limit) {
    _M_index1 = (_M_index1 + 1) % 55;
    _M_index2 = (_M_index2 + 1) % 55;
    _M_table[_M_index1] = _M_table[_M_index1] - _M_table[_M_index2];
    return _M_table[_M_index1] % __limit;
  }

  void _M_initialize(unsigned int __seed)
  {
    unsigned int __k = 1;
    _M_table[54] = __seed;
    size_t __i;
    for (__i = 0; __i < 54; __i++) {
        size_t __ii = (21 * (__i + 1) % 55) - 1;
        _M_table[__ii] = __k;
        __k = __seed - __k;
        __seed = _M_table[__ii];
    }
    for (int __loop = 0; __loop < 4; __loop++) {
        for (__i = 0; __i < 55; __i++)
            _M_table[__i] = _M_table[__i] - _M_table[(1 + __i + 30) % 55];
    }
    _M_index1 = 0;
    _M_index2 = 31;
  }

  subtractive_rng(unsigned int __seed) { _M_initialize(__seed); }
  subtractive_rng() { _M_initialize(161803398u); }
};


// Adaptor function objects: pointers to member functions.
```

```
// There are a total of 16 = 2^4 function objects in this family.
//  (1) Member functions taking no arguments vs member functions taking
//      one argument.
//  (2) Call through pointer vs call through reference.
//  (3) Member function with void return type vs member function with
//     non-void return type.
//  (4) Const vs non-const member function.

// Note that choice (3) is nothing more than a workaround: according
//  to the draft, compilers should handle void and non-void the same way.
//  This feature is not yet widely implemented, though.  You can only use
//  member functions returning void if your compiler supports partial
//  specialization.

// All of this complexity is in the function objects themselves.  You can
//  ignore it by using the helper function mem_fun and mem_fun_ref,
//  which create whichever type of adaptor is appropriate.
//  (mem_fun1 and mem_fun1_ref are no longer part of the C++ standard,
//  but they are provided for backward compatibility.)


template <class _Ret, class _Tp>
class mem_fun_t : public unary_function<_Tp*,_Ret> {
public:
  explicit mem_fun_t(_Ret (_Tp::*__pf)()) : _M_f(__pf) {}
  _Ret operator()(_Tp* __p) const { return (__p->*_M_f)(); }
private:
  _Ret (_Tp::*_M_f)();
};

template <class _Ret, class _Tp>
class const_mem_fun_t : public unary_function<const _Tp*,_Ret> {
public:
  explicit const_mem_fun_t(_Ret (_Tp::*__pf)() const) : _M_f(__pf) {}
  _Ret operator()(const _Tp* __p) const { return (__p->*_M_f)(); }
private:
  _Ret (_Tp::*_M_f)() const;
};


template <class _Ret, class _Tp>
class mem_fun_ref_t : public unary_function<_Tp,_Ret> {
public:
  explicit mem_fun_ref_t(_Ret (_Tp::*__pf)()) : _M_f(__pf) {}
  _Ret operator()(_Tp& __r) const { return (__r.*_M_f)(); }
private:
  _Ret (_Tp::*_M_f)();
};
```

```
template <class _Ret, class _Tp>
class const_mem_fun_ref_t : public unary_function<_Tp,_Ret> {
public:
  explicit const_mem_fun_ref_t(_Ret (_Tp::*__pf)() const) : _M_f(__pf) {}
  _Ret operator()(const _Tp& __r) const { return (__r.*_M_f)(); }
private:
  _Ret (_Tp::*_M_f)() const;
};

template <class _Ret, class _Tp, class _Arg>
class mem_fun1_t : public binary_function<_Tp*,_Arg,_Ret> {
public:
  explicit mem_fun1_t(_Ret (_Tp::*__pf)(_Arg)) : _M_f(__pf) {}
  _Ret operator()(_Tp* __p, _Arg __x) const { return (__p->*_M_f)(__x); }
private:
  _Ret (_Tp::*_M_f)(_Arg);
};

template <class _Ret, class _Tp, class _Arg>
class const_mem_fun1_t : public binary_function<const _Tp*,_Arg,_Ret> {
public:
  explicit const_mem_fun1_t(_Ret (_Tp::*__pf)(_Arg) const) : _M_f(__pf) {}
  _Ret operator()(const _Tp* __p, _Arg __x) const
    { return (__p->*_M_f)(__x); }
private:
  _Ret (_Tp::*_M_f)(_Arg) const;
};

template <class _Ret, class _Tp, class _Arg>
class mem_fun1_ref_t : public binary_function<_Tp,_Arg,_Ret> {
public:
  explicit mem_fun1_ref_t(_Ret (_Tp::*__pf)(_Arg)) : _M_f(__pf) {}
  _Ret operator()(_Tp& __r, _Arg __x) const { return (__r.*_M_f)(__x); }
private:
  _Ret (_Tp::*_M_f)(_Arg);
};

template <class _Ret, class _Tp, class _Arg>
class const_mem_fun1_ref_t : public binary_function<_Tp,_Arg,_Ret> {
public:
  explicit const_mem_fun1_ref_t(_Ret (_Tp::*__pf)(_Arg) const) : _M_f(__pf) {}
  _Ret operator()(const _Tp& __r, _Arg __x) const { return (__r.*_M_f)(__x); }
private:
  _Ret (_Tp::*_M_f)(_Arg) const;
};

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class _Tp>
```

```
class mem_fun_t<void, _Tp> : public unary_function<_Tp*,void> {
public:
  explicit mem_fun_t(void (_Tp::*__pf)()) : _M_f(__pf) {}
  void operator()(_Tp* __p) const { (__p->*_M_f)(); }
private:
  void (_Tp::*_M_f)();
};

template <class _Tp>
class const_mem_fun_t<void, _Tp> : public unary_function<const _Tp*,void> {
public:
  explicit const_mem_fun_t(void (_Tp::*__pf)() const) : _M_f(__pf) {}
  void operator()(const _Tp* __p) const { (__p->*_M_f)(); }
private:
  void (_Tp::*_M_f)() const;
};

template <class _Tp>
class mem_fun_ref_t<void, _Tp> : public unary_function<_Tp,void> {
public:
  explicit mem_fun_ref_t(void (_Tp::*__pf)()) : _M_f(__pf) {}
  void operator()(_Tp& __r) const { (__r.*_M_f)(); }
private:
  void (_Tp::*_M_f)();
};

template <class _Tp>
class const_mem_fun_ref_t<void, _Tp> : public unary_function<_Tp,void> {
public:
  explicit const_mem_fun_ref_t(void (_Tp::*__pf)() const) : _M_f(__pf) {}
  void operator()(const _Tp& __r) const { (__r.*_M_f)(); }
private:
  void (_Tp::*_M_f)() const;
};

template <class _Tp, class _Arg>
class mem_fun1_t<void, _Tp, _Arg> : public binary_function<_Tp*,_Arg,void> {
public:
  explicit mem_fun1_t(void (_Tp::*__pf)(_Arg)) : _M_f(__pf) {}
  void operator()(_Tp* __p, _Arg __x) const { (__p->*_M_f)(__x); }
private:
  void (_Tp::*_M_f)(_Arg);
};

template <class _Tp, class _Arg>
class const_mem_fun1_t<void, _Tp, _Arg>
  : public binary_function<const _Tp*,_Arg,void> {
public:
  explicit const_mem_fun1_t(void (_Tp::*__pf)(_Arg) const) : _M_f(__pf) {}
```

```
  void operator()(const _Tp* __p, _Arg __x) const { (__p->*_M_f)(__x); }
private:
  void (_Tp::*_M_f)(_Arg) const;
};

template <class _Tp, class _Arg>
class mem_fun1_ref_t<void, _Tp, _Arg>
  : public binary_function<_Tp,_Arg,void> {
public:
  explicit mem_fun1_ref_t(void (_Tp::*__pf)(_Arg)) : _M_f(__pf) {}
  void operator()(_Tp& __r, _Arg __x) const { (__r.*_M_f)(__x); }
private:
  void (_Tp::*_M_f)(_Arg);
};

template <class _Tp, class _Arg>
class const_mem_fun1_ref_t<void, _Tp, _Arg>
  : public binary_function<_Tp,_Arg,void> {
public:
  explicit const_mem_fun1_ref_t(void (_Tp::*__pf)(_Arg) const) : _M_f(__pf) {}
  void operator()(const _Tp& __r, _Arg __x) const { (__r.*_M_f)(__x); }
private:
  void (_Tp::*_M_f)(_Arg) const;
};

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// Mem_fun adaptor helper functions.  There are only two:
//  mem_fun and mem_fun_ref.  (mem_fun1 and mem_fun1_ref
//  are provided for backward compatibility, but they are no longer
//  part of the C++ standard.)

template <class _Ret, class _Tp>
inline mem_fun_t<_Ret,_Tp> mem_fun(_Ret (_Tp::*__f)())
  { return mem_fun_t<_Ret,_Tp>(__f); }

template <class _Ret, class _Tp>
inline const_mem_fun_t<_Ret,_Tp> mem_fun(_Ret (_Tp::*__f)() const)
  { return const_mem_fun_t<_Ret,_Tp>(__f); }

template <class _Ret, class _Tp>
inline mem_fun_ref_t<_Ret,_Tp> mem_fun_ref(_Ret (_Tp::*__f)())
  { return mem_fun_ref_t<_Ret,_Tp>(__f); }

template <class _Ret, class _Tp>
inline const_mem_fun_ref_t<_Ret,_Tp> mem_fun_ref(_Ret (_Tp::*__f)() const)
  { return const_mem_fun_ref_t<_Ret,_Tp>(__f); }

template <class _Ret, class _Tp, class _Arg>
```

```
inline mem_fun1_t<_Ret,_Tp,_Arg> mem_fun(_Ret (_Tp::*__f)(_Arg))
  { return mem_fun1_t<_Ret,_Tp,_Arg>(__f); }

template <class _Ret, class _Tp, class _Arg>
inline const_mem_fun1_t<_Ret,_Tp,_Arg> mem_fun(_Ret (_Tp::*__f)(_Arg) const)
  { return const_mem_fun1_t<_Ret,_Tp,_Arg>(__f); }

template <class _Ret, class _Tp, class _Arg>
inline mem_fun1_ref_t<_Ret,_Tp,_Arg> mem_fun_ref(_Ret (_Tp::*__f)(_Arg))
  { return mem_fun1_ref_t<_Ret,_Tp,_Arg>(__f); }

template <class _Ret, class _Tp, class _Arg>
inline const_mem_fun1_ref_t<_Ret,_Tp,_Arg>
mem_fun_ref(_Ret (_Tp::*__f)(_Arg) const)
  { return const_mem_fun1_ref_t<_Ret,_Tp,_Arg>(__f); }

template <class _Ret, class _Tp, class _Arg>
inline mem_fun1_t<_Ret,_Tp,_Arg> mem_fun1(_Ret (_Tp::*__f)(_Arg))
  { return mem_fun1_t<_Ret,_Tp,_Arg>(__f); }

template <class _Ret, class _Tp, class _Arg>
inline const_mem_fun1_t<_Ret,_Tp,_Arg> mem_fun1(_Ret (_Tp::*__f)(_Arg) const)
  { return const_mem_fun1_t<_Ret,_Tp,_Arg>(__f); }

template <class _Ret, class _Tp, class _Arg>
inline mem_fun1_ref_t<_Ret,_Tp,_Arg> mem_fun1_ref(_Ret (_Tp::*__f)(_Arg))
  { return mem_fun1_ref_t<_Ret,_Tp,_Arg>(__f); }

template <class _Ret, class _Tp, class _Arg>
inline const_mem_fun1_ref_t<_Ret,_Tp,_Arg>
mem_fun1_ref(_Ret (_Tp::*__f)(_Arg) const)
  { return const_mem_fun1_ref_t<_Ret,_Tp,_Arg>(__f); }

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_FUNCTION_H */

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