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

```
/*
 * Copyright (c) 1996-1997
 * 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_ALLOC_H
#define __SGI_STL_INTERNAL_ALLOC_H

#ifdef __SUNPRO_CC
#  define __PRIVATE public
   // Extra access restrictions prevent us from really making some things
   // private.
#else
#  define __PRIVATE private
#endif

#ifdef __STL_STATIC_TEMPLATE_MEMBER_BUG
#  define __USE_MALLOC
#endif


// This implements some standard node allocators.  These are
// NOT the same as the allocators in the C++ draft standard or in
// in the original STL.  They do not encapsulate different pointer
// types; indeed we assume that there is only one pointer type.
// The allocation primitives are intended to allocate individual objects,
// not larger arenas as with the original STL allocators.

#if 0
#   include <new>
#   define __THROW_BAD_ALLOC throw bad_alloc
#elif !defined(__THROW_BAD_ALLOC)
#   include <iostream.h>
#   define __THROW_BAD_ALLOC cerr << "out of memory" << endl; exit(1)
#endif
```

```
#ifndef __ALLOC
#   define __ALLOC alloc
#endif
#ifdef __STL_WIN32THREADS
#   include <windows.h>
#endif

#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#ifndef __RESTRICT
#  define __RESTRICT
#endif

#if !defined(__STL_PTHREADS) && !defined(_NOTHREADS) \
 && !defined(__STL_SGI_THREADS) && !defined(__STL_WIN32THREADS)
#   define _NOTHREADS
#endif

# ifdef __STL_PTHREADS
    // POSIX Threads
    // This is dubious, since this is likely to be a high contention
    // lock.   Performance may not be adequate.
#   include <pthread.h>
#   define __NODE_ALLOCATOR_LOCK \
       if (threads) pthread_mutex_lock(&__node_allocator_lock)
#   define __NODE_ALLOCATOR_UNLOCK \
       if (threads) pthread_mutex_unlock(&__node_allocator_lock)
#   define __NODE_ALLOCATOR_THREADS true
#   define __VOLATILE volatile  // Needed at -O3 on SGI
# endif
# ifdef __STL_WIN32THREADS
    // The lock needs to be initialized by constructing an allocator
    // objects of the right type.  We do that here explicitly for alloc.
#   define __NODE_ALLOCATOR_LOCK \
       EnterCriticalSection(&__node_allocator_lock)
#   define __NODE_ALLOCATOR_UNLOCK \
       LeaveCriticalSection(&__node_allocator_lock)
#   define __NODE_ALLOCATOR_THREADS true
#   define __VOLATILE volatile  // may not be needed
# endif /* WIN32THREADS */
# ifdef __STL_SGI_THREADS
    // This should work without threads, with sproc threads, or with
    // pthreads.  It is suboptimal in all cases.
    // It is unlikely to even compile on nonSGI machines.

    extern "C" {
      extern int __us_rsthread_malloc;
```

```
   }
      // The above is copied from malloc.h.  Including <malloc.h>
      // would be cleaner but fails with certain levels of standard
      // conformance.
#   define __NODE_ALLOCATOR_LOCK if (threads && __us_rsthread_malloc) \
            { __lock(&__node_allocator_lock); }
#   define __NODE_ALLOCATOR_UNLOCK if (threads && __us_rsthread_malloc) \
            { __unlock(&__node_allocator_lock); }
#   define __NODE_ALLOCATOR_THREADS true
#   define __VOLATILE volatile  // Needed at -O3 on SGI
# endif
# ifdef _NOTHREADS
//  Thread-unsafe
#   define __NODE_ALLOCATOR_LOCK
#   define __NODE_ALLOCATOR_UNLOCK
#   define __NODE_ALLOCATOR_THREADS false
#   define __VOLATILE
# endif

__STL_BEGIN_NAMESPACE

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM != _MIPS_SIM_ABI32)
#pragma set woff 1174
#endif

// Malloc-based allocator.  Typically slower than default alloc below.
// Typically thread-safe and more storage efficient.
#ifdef __STL_STATIC_TEMPLATE_MEMBER_BUG
# ifdef __DECLARE_GLOBALS_HERE
   void (* __malloc_alloc_oom_handler)() = 0;
   // g++ 2.7.2 does not handle static template data members.
# else
   extern void (* __malloc_alloc_oom_handler)();
# endif
#endif

template <int inst>
class __malloc_alloc_template {

private:

static void *oom_malloc(size_t);

static void *oom_realloc(void *, size_t);

#ifndef __STL_STATIC_TEMPLATE_MEMBER_BUG
   static void (* __malloc_alloc_oom_handler)();
#endif
```

```
public:

static void * allocate(size_t n)
{
   void *result = malloc(n);
   if (0 == result) result = oom_malloc(n);
   return result;
}

static void deallocate(void *p, size_t /* n */)
{
   free(p);
}

static void * reallocate(void *p, size_t /* old_sz */, size_t new_sz)
{
   void * result = realloc(p, new_sz);
   if (0 == result) result = oom_realloc(p, new_sz);
   return result;
}

static void (* set_malloc_handler(void (*f)()))()
{
   void (* old)() = __malloc_alloc_oom_handler;
   __malloc_alloc_oom_handler = f;
   return(old);
}

};

// malloc_alloc out-of-memory handling

#ifndef __STL_STATIC_TEMPLATE_MEMBER_BUG
template <int inst>
void (* __malloc_alloc_template<inst>::__malloc_alloc_oom_handler)() = 0;
#endif

template <int inst>
void * __malloc_alloc_template<inst>::oom_malloc(size_t n)
{
   void (* my_malloc_handler)();
   void *result;

   for (;;) {
      my_malloc_handler = __malloc_alloc_oom_handler;
      if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
      (*my_malloc_handler)();
      result = malloc(n);
      if (result) return(result);
```

```
    }
}

template <int inst>
void * __malloc_alloc_template<inst>::oom_realloc(void *p, size_t n)
{
    void (* my_malloc_handler)();
    void *result;

    for (;;) {
        my_malloc_handler = __malloc_alloc_oom_handler;
        if (0 == my_malloc_handler) { __THROW_BAD_ALLOC; }
        (*my_malloc_handler)();
        result = realloc(p, n);
        if (result) return(result);
    }
}

typedef __malloc_alloc_template<0> malloc_alloc;

template<class T, class Alloc>
class simple_alloc {

public:
    static T *allocate(size_t n)
              { return 0 == n? 0 : (T*) Alloc::allocate(n * sizeof (T)); }
    static T *allocate(void)
              { return (T*) Alloc::allocate(sizeof (T)); }
    static void deallocate(T *p, size_t n)
              { if (0 != n) Alloc::deallocate(p, n * sizeof (T)); }
    static void deallocate(T *p)
              { Alloc::deallocate(p, sizeof (T)); }
};

// Allocator adaptor to check size arguments for debugging.
// Reports errors using assert.  Checking can be disabled with
// NDEBUG, but it's far better to just use the underlying allocator
// instead when no checking is desired.
// There is some evidence that this can confuse Purify.
template <class Alloc>
class debug_alloc {

private:

enum {extra = 8};       // Size of space used to store size.  Note
                    // that this must be large enough to preserve
                    // alignment.

public:
```

```
static void * allocate(size_t n)
{
    char *result = (char *)Alloc::allocate(n + extra);
    *(size_t *)result = n;
    return result + extra;
}

static void deallocate(void *p, size_t n)
{
    char * real_p = (char *)p - extra;
    assert(*(size_t *)real_p == n);
    Alloc::deallocate(real_p, n + extra);
}

static void * reallocate(void *p, size_t old_sz, size_t new_sz)
{
    char * real_p = (char *)p - extra;
    assert(*(size_t *)real_p == old_sz);
    char * result = (char *)
                Alloc::reallocate(real_p, old_sz + extra, new_sz + extra);
    *(size_t *)result = new_sz;
    return result + extra;
}


};


# ifdef __USE_MALLOC

typedef malloc_alloc alloc;
typedef malloc_alloc single_client_alloc;

# else


// Default node allocator.
// With a reasonable compiler, this should be roughly as fast as the
// original STL class-specific allocators, but with less fragmentation.
// Default_alloc_template parameters are experimental and MAY
// DISAPPEAR in the future.  Clients should just use alloc for now.
//
// Important implementation properties:
// 1. If the client request an object of size > __MAX_BYTES, the resulting
//    object will be obtained directly from malloc.
// 2. In all other cases, we allocate an object of size exactly
//    ROUND_UP(requested_size).  Thus the client has enough size
//    information that we can return the object to the proper free list
```

```
//   without permanently losing part of the object.
//

// The first template parameter specifies whether more than one thread
// may use this allocator.  It is safe to allocate an object from
// one instance of a default_alloc and deallocate it with another
// one.  This effectively transfers its ownership to the second one.
// This may have undesirable effects on reference locality.
// The second parameter is unreferenced and serves only to allow the
// creation of multiple default_alloc instances.
// Node that containers built on different allocator instances have
// different types, limiting the utility of this approach.
#ifdef __SUNPRO_CC
// breaks if we make these template class members:
  enum {__ALIGN = 8};
  enum {__MAX_BYTES = 128};
  enum {__NFREELISTS = __MAX_BYTES/__ALIGN};
#endif

template <bool threads, int inst>
class __default_alloc_template {

private:
  // Really we should use static const int x = N
  // instead of enum { x = N }, but few compilers accept the former.
# ifndef __SUNPRO_CC
    enum {__ALIGN = 8};
    enum {__MAX_BYTES = 128};
    enum {__NFREELISTS = __MAX_BYTES/__ALIGN};
# endif
  static size_t ROUND_UP(size_t bytes) {
        return (((bytes) + __ALIGN-1) & ~(__ALIGN - 1));
  }
__PRIVATE:
  union obj {
        union obj * free_list_link;
        char client_data[1];    /* The client sees this.       */
  };
private:
# ifdef __SUNPRO_CC
    static obj * __VOLATILE free_list[];
        // Specifying a size results in duplicate def for 4.1
# else
    static obj * __VOLATILE free_list[__NFREELISTS];
# endif
  static  size_t FREELIST_INDEX(size_t bytes) {
        return (((bytes) + __ALIGN-1)/__ALIGN - 1);
  }
```

```
  // Returns an object of size n, and optionally adds to size n free list.
  static void *refill(size_t n);
  // Allocates a chunk for nobjs of size "size".  nobjs may be reduced
  // if it is inconvenient to allocate the requested number.
  static char *chunk_alloc(size_t size, int &nobjs);

  // Chunk allocation state.
  static char *start_free;
  static char *end_free;
  static size_t heap_size;

# ifdef __STL_SGI_THREADS
    static volatile unsigned long __node_allocator_lock;
    static void __lock(volatile unsigned long *);
    static inline void __unlock(volatile unsigned long *);
# endif

# ifdef __STL_PTHREADS
    static pthread_mutex_t __node_allocator_lock;
# endif

# ifdef __STL_WIN32THREADS
    static CRITICAL_SECTION __node_allocator_lock;
    static bool __node_allocator_lock_initialized;

  public:
    __default_alloc_template() {
       // This assumes the first constructor is called before threads
       // are started.
        if (!__node_allocator_lock_initialized) {
            InitializeCriticalSection(&__node_allocator_lock);
            __node_allocator_lock_initialized = true;
        }
    }
  private:
# endif

    class lock {
        public:
            lock() { __NODE_ALLOCATOR_LOCK; }
            ~lock() { __NODE_ALLOCATOR_UNLOCK; }
    };
    friend class lock;

public:

  /* n must be > 0      */
  static void * allocate(size_t n)
  {
```

```
    obj * __VOLATILE * my_free_list;
    obj * __RESTRICT result;

    if (n > (size_t) __MAX_BYTES) {
        return(malloc_alloc::allocate(n));
    }
    my_free_list = free_list + FREELIST_INDEX(n);
    // Acquire the lock here with a constructor call.
    // This ensures that it is released in exit or during stack
    // unwinding.
#       ifndef _NOTHREADS
        /*REFERENCED*/
        lock lock_instance;
#       endif
    result = *my_free_list;
    if (result == 0) {
        void *r = refill(ROUND_UP(n));
        return r;
    }
    *my_free_list = result -> free_list_link;
    return (result);
  };

  /* p may not be 0 */
  static void deallocate(void *p, size_t n)
  {
    obj *q = (obj *)p;
    obj * __VOLATILE * my_free_list;

    if (n > (size_t) __MAX_BYTES) {
        malloc_alloc::deallocate(p, n);
        return;
    }
    my_free_list = free_list + FREELIST_INDEX(n);
    // acquire lock
#       ifndef _NOTHREADS
        /*REFERENCED*/
        lock lock_instance;
#       endif /* _NOTHREADS */
    q -> free_list_link = *my_free_list;
    *my_free_list = q;
    // lock is released here
  }

  static void * reallocate(void *p, size_t old_sz, size_t new_sz);

} ;

typedef __default_alloc_template<__NODE_ALLOCATOR_THREADS, 0> alloc;
```

```
typedef __default_alloc_template<false, 0> single_client_alloc;


/* We allocate memory in large chunks in order to avoid fragmenting     */
/* the malloc heap too much.                                            */
/* We assume that size is properly aligned.                             */
/* We hold the allocation lock.                                         */
template <bool threads, int inst>
char*
__default_alloc_template<threads, inst>::chunk_alloc(size_t size, int& nobjs)
{
    char * result;
    size_t total_bytes = size * nobjs;
    size_t bytes_left = end_free - start_free;

    if (bytes_left >= total_bytes) {
        result = start_free;
        start_free += total_bytes;
        return(result);
    } else if (bytes_left >= size) {
        nobjs = bytes_left/size;
        total_bytes = size * nobjs;
        result = start_free;
        start_free += total_bytes;
        return(result);
    } else {
        size_t bytes_to_get = 2 * total_bytes + ROUND_UP(heap_size >> 4);
        // Try to make use of the left-over piece.
        if (bytes_left > 0) {
            obj * __VOLATILE * my_free_list =
                        free_list + FREELIST_INDEX(bytes_left);

            ((obj *)start_free) -> free_list_link = *my_free_list;
            *my_free_list = (obj *)start_free;
        }
        start_free = (char *)malloc(bytes_to_get);
        if (0 == start_free) {
            int i;
            obj * __VOLATILE * my_free_list, *p;
            // Try to make do with what we have.  That can't
            // hurt.  We do not try smaller requests, since that tends
            // to result in disaster on multi-process machines.
            for (i = size; i <= __MAX_BYTES; i += __ALIGN) {
                my_free_list = free_list + FREELIST_INDEX(i);
                p = *my_free_list;
                if (0 != p) {
                    *my_free_list = p -> free_list_link;
                    start_free = (char *)p;
```

```
                end_free = start_free + i;
                return(chunk_alloc(size, nobjs));
                // Any leftover piece will eventually make it to the
                // right free list.
            }
          }
          end_free = 0;   // In case of exception.
           start_free = (char *)malloc_alloc::allocate(bytes_to_get);
           // This should either throw an exception
           // or remedy the situation.  Thus we assume it
           // succeeded.
       }
       heap_size += bytes_to_get;
       end_free = start_free + bytes_to_get;
       return(chunk_alloc(size, nobjs));
   }
}


/* Returns an object of size n, and optionally adds to size n free list.*/
/* We assume that n is properly aligned.                                */
/* We hold the allocation lock.                                         */
template <bool threads, int inst>
void* __default_alloc_template<threads, inst>::refill(size_t n)
{
    int nobjs = 20;
    char * chunk = chunk_alloc(n, nobjs);
    obj * __VOLATILE * my_free_list;
    obj * result;
    obj * current_obj, * next_obj;
    int i;

    if (1 == nobjs) return(chunk);
    my_free_list = free_list + FREELIST_INDEX(n);

    /* Build free list in chunk */
      result = (obj *)chunk;
      *my_free_list = next_obj = (obj *)(chunk + n);
      for (i = 1; ; i++) {
        current_obj = next_obj;
        next_obj = (obj *)((char *)next_obj + n);
        if (nobjs - 1 == i) {
            current_obj -> free_list_link = 0;
            break;
        } else {
            current_obj -> free_list_link = next_obj;
        }
      }
    return(result);
```

```
}

template <bool threads, int inst>
void*
__default_alloc_template<threads, inst>::reallocate(void *p,
                                            size_t old_sz,
                                            size_t new_sz)
{
    void * result;
    size_t copy_sz;

    if (old_sz > (size_t) __MAX_BYTES && new_sz > (size_t) __MAX_BYTES) {
        return(realloc(p, new_sz));
    }
    if (ROUND_UP(old_sz) == ROUND_UP(new_sz)) return(p);
    result = allocate(new_sz);
    copy_sz = new_sz > old_sz? old_sz : new_sz;
    memcpy(result, p, copy_sz);
    deallocate(p, old_sz);
    return(result);
}

#ifdef __STL_PTHREADS
    template <bool threads, int inst>
    pthread_mutex_t
    __default_alloc_template<threads, inst>::__node_allocator_lock
        = PTHREAD_MUTEX_INITIALIZER;
#endif

#ifdef __STL_WIN32THREADS
    template <bool threads, int inst> CRITICAL_SECTION
    __default_alloc_template<threads, inst>::__node_allocator_lock;

    template <bool threads, int inst> bool
    __default_alloc_template<threads, inst>::__node_allocator_lock_initialized
       = false;
#endif

#ifdef __STL_SGI_THREADS
__STL_END_NAMESPACE
#include <mutex.h>
#include <time.h>
__STL_BEGIN_NAMESPACE
// Somewhat generic lock implementations.  We need only test-and-set
// and some way to sleep.  These should work with both SGI pthreads
// and sproc threads.  They may be useful on other systems.
template <bool threads, int inst>
volatile unsigned long
__default_alloc_template<threads, inst>::__node_allocator_lock = 0;
```

```
#if __mips < 3 || !(defined (_ABIN32) || defined(_ABI64)) || defined(__GNUC__)
#   define __test_and_set(l,v) test_and_set(l,v)
#endif

template <bool threads, int inst>
void
__default_alloc_template<threads, inst>::__lock(volatile unsigned long *lock)
{
    const unsigned low_spin_max = 30;  // spin cycles if we suspect uniprocessor
    const unsigned high_spin_max = 1000; // spin cycles for multiprocessor
    static unsigned spin_max = low_spin_max;
    unsigned my_spin_max;
    static unsigned last_spins = 0;
    unsigned my_last_spins;
    static struct timespec ts = {0, 1000};
    unsigned junk;
#   define __ALLOC_PAUSE junk *= junk; junk *= junk; junk *= junk; junk *= junk
    int i;

    if (!__test_and_set((unsigned long *)lock, 1)) {
        return;
    }
    my_spin_max = spin_max;
    my_last_spins = last_spins;
    for (i = 0; i < my_spin_max; i++) {
        if (i < my_last_spins/2 || *lock) {
            __ALLOC_PAUSE;
            continue;
        }
        if (!__test_and_set((unsigned long *)lock, 1)) {
            // got it!
            // Spinning worked.  Thus we're probably not being scheduled
            // against the other process with which we were contending.
            // Thus it makes sense to spin longer the next time.
            last_spins = i;
            spin_max = high_spin_max;
            return;
        }
    }
    // We are probably being scheduled against the other process.  Sleep.
    spin_max = low_spin_max;
    for (;;) {
        if (!__test_and_set((unsigned long *)lock, 1)) {
            return;
        }
        nanosleep(&ts, 0);
    }
}
```

```
template <bool threads, int inst>
inline void
__default_alloc_template<threads, inst>::__unlock(volatile unsigned long *lock)
{
#   if defined(__GNUC__) && __mips >= 3
       asm("sync");
       *lock = 0;
#   elif __mips >= 3 && (defined (_ABIN32) || defined(_ABI64))
       __lock_release(lock);
#   else
       *lock = 0;
       // This is not sufficient on many multiprocessors, since
       // writes to protected variables and the lock may be reordered.
#   endif
}
#endif

template <bool threads, int inst>
char *__default_alloc_template<threads, inst>::start_free = 0;

template <bool threads, int inst>
char *__default_alloc_template<threads, inst>::end_free = 0;

template <bool threads, int inst>
size_t __default_alloc_template<threads, inst>::heap_size = 0;

template <bool threads, int inst>
__default_alloc_template<threads, inst>::obj * __VOLATILE
__default_alloc_template<threads, inst> ::free_list[
# ifdef __SUNPRO_CC
    __NFREELISTS
# else
    __default_alloc_template<threads, inst>::__NFREELISTS
# endif
] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, };
// The 16 zeros are necessary to make version 4.1 of the SunPro
// compiler happy.  Otherwise it appears to allocate too little
// space for the array.

# ifdef __STL_WIN32THREADS
  // Create one to get critical section initialized.
  // We do this onece per file, but only the first constructor
  // does anything.
  static alloc __node_allocator_dummy_instance;
# endif

#endif /* ! __USE_MALLOC */
```

```
#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM != _MIPS_SIM_ABI32)
#pragma reset woff 1174
#endif

__STL_END_NAMESPACE

#undef __PRIVATE

#endif /* __SGI_STL_INTERNAL_ALLOC_H */

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