```
/*
 * Copyright 1988, 1989 Hans-J. Boehm, Alan J. Demers
 * Copyright (c) 1991-1995 by Xerox Corporation.  All rights reserved.
 * Copyright 1996 by Silicon Graphics.  All rights reserved.
 *
 * THIS MATERIAL IS PROVIDED AS IS, WITH ABSOLUTELY NO WARRANTY EXPRESSED
 * OR IMPLIED.  ANY USE IS AT YOUR OWN RISK.
 *
 * Permission is hereby granted to use or copy this program
 * for any purpose,  provided the above notices are retained on all copies.
 * Permission to modify the code and to distribute modified code is granted,
 * provided the above notices are retained, and a notice that the code was
 * modified is included with the above copyright notice.
 */

/*
 * Note that this defines a large number of tuning hooks, which can
 * safely be ignored in nearly all cases.  For normal use it suffices
 * to call only GC_MALLOC and perhaps GC_REALLOC.
 * For better performance, also look at GC_MALLOC_ATOMIC, and
 * GC_enable_incremental.  If you need an action to be performed
 * immediately before an object is collected, look at GC_register_finalizer.
 * If you are using Solaris threads, look at the end of this file.
 * Everything else is best ignored unless you encounter performance
 * problems.
 */

#ifndef _GC_H

# define _GC_H
# define __GC
# include <stddef.h>

#if defined(__CYGWIN32__) && defined(GC_USE_DLL)
#include "libgc_globals.h"
#endif

#if defined(_MSC_VER) && defined(_DLL)
#ifdef GC_BUILD
#define GC_API __declspec(dllexport)
#else
#define GC_API __declspec(dllimport)
```

```
#endif
#endif

#ifndef GC_API
#define GC_API extern
#endif

# if defined(__STDC__) || defined(__cplusplus)
#   define GC_PROTO(args) args
    typedef void * GC_PTR;
# else
#   define GC_PROTO(args) ()
    typedef char * GC_PTR;
#  endif

# ifdef __cplusplus
    extern "C" {
# endif


/* Define word and signed_word to be unsigned and signed types of the   */
/* size as char * or void *.  There seems to be no way to do this       */
/* even semi-portably.  The following is probably no better/worse       */
/* than almost anything else.                                           */
/* The ANSI standard suggests that size_t and ptr_diff_t might be       */
/* better choices.  But those appear to have incorrect definitions      */
/* on may systems.  Notably "typedef int size_t" seems to be both       */
/* frequent and WRONG.                                                  */
typedef unsigned long GC_word;
typedef long GC_signed_word;

/* Public read-only variables */

GC_API GC_word GC_gc_no;/* Counter incremented per collection.          */
                        /* Includes empty GCs at startup.               */


/* Public R/W variables */

GC_API GC_PTR (*GC_oom_fn) GC_PROTO((size_t bytes_requested));
                        /* When there is insufficient memory to satisfy */
                        /* an allocation request, we return             */
```

```
                        /* (*GC_oom_fn)().  By default this just        */
                        /* returns 0.                                    */
                        /* If it returns, it must return 0 or a valid   */
                        /* pointer to a previously allocated heap       */
                        /* object.                                       */

GC_API int GC_quiet;    /* Disable statistics output.  Only matters if  */
                        /* collector has been compiled with statistics  */
                        /* enabled.  This involves a performance cost,  */
                        /* and is thus not the default.                 */

GC_API int GC_dont_gc;  /* Dont collect unless explicitly requested, e.g. */
                        /* because it's not safe.                         */

GC_API int GC_dont_expand;
                        /* Dont expand heap unless explicitly requested */
                        /* or forced to.                                 */

GC_API int GC_full_freq;    /* Number of partial collections between    */
                            /* full collections.  Matters only if       */
                            /* GC_incremental is set.                   */

GC_API GC_word GC_non_gc_bytes;
                        /* Bytes not considered candidates for collection. */
                        /* Used only to control scheduling of collections. */

GC_API GC_word GC_free_space_divisor;
                        /* We try to make sure that we allocate at      */
                        /* least N/GC_free_space_divisor bytes between  */
                        /* collections, where N is the heap size plus   */
                        /* a rough estimate of the root set size.       */
                        /* Initially, GC_free_space_divisor = 4.        */
                        /* Increasing its value will use less space     */
                        /* but more collection time.  Decreasing it     */
                        /* will appreciably decrease collection time    */
                        /* at the expense of space.                     */
                        /* GC_free_space_divisor = 1 will effectively   */
                        /* disable collections.                         */

GC_API GC_word GC_max_retries;
                        /* The maximum number of GCs attempted before   */
                        /* reporting out of memory after heap           */
```

```
                        /* expansion fails.  Initially 0.              */


/* Public procedures */
/*
 * general purpose allocation routines, with roughly malloc calling conv.
 * The atomic versions promise that no relevant pointers are contained
 * in the object.  The nonatomic versions guarantee that the new object
 * is cleared.  GC_malloc_stubborn promises that no changes to the object
 * will occur after GC_end_stubborn_change has been called on the
 * result of GC_malloc_stubborn. GC_malloc_uncollectable allocates an object
 * that is scanned for pointers to collectable objects, but is not itself
 * collectable.  GC_malloc_uncollectable and GC_free called on the resulting
 * object implicitly update GC_non_gc_bytes appropriately.
 */
GC_API GC_PTR GC_malloc GC_PROTO((size_t size_in_bytes));
GC_API GC_PTR GC_malloc_atomic GC_PROTO((size_t size_in_bytes));
GC_API GC_PTR GC_malloc_uncollectable GC_PROTO((size_t size_in_bytes));
GC_API GC_PTR GC_malloc_stubborn GC_PROTO((size_t size_in_bytes));

/* The following is only defined if the library has been suitably       */
/* compiled:                                                            */
GC_API GC_PTR GC_malloc_atomic_uncollectable GC_PROTO((size_t size_in_bytes));

/* Explicitly deallocate an object.  Dangerous if used incorrectly.     */
/* Requires a pointer to the base of an object.                         */
/* If the argument is stubborn, it should not be changeable when freed. */
/* An object should not be enable for finalization when it is           */
/* explicitly deallocated.                                              */
/* GC_free(0) is a no-op, as required by ANSI C for free.               */
GC_API void GC_free GC_PROTO((GC_PTR object_addr));

/*
 * Stubborn objects may be changed only if the collector is explicitly informed.
 * The collector is implicitly informed of coming change when such
 * an object is first allocated.  The following routines inform the
 * collector that an object will no longer be changed, or that it will
 * once again be changed.  Only nonNIL pointer stores into the object
 * are considered to be changes.  The argument to GC_end_stubborn_change
 * must be exacly the value returned by GC_malloc_stubborn or passed to
 * GC_change_stubborn.  (In the second case it may be an interior pointer
 * within 512 bytes of the beginning of the objects.)
```

```
 * There is a performance penalty for allowing more than
 * one stubborn object to be changed at once, but it is acceptable to
 * do so.  The same applies to dropping stubborn objects that are still
 * changeable.
 */
GC_API void GC_change_stubborn GC_PROTO((GC_PTR));
GC_API void GC_end_stubborn_change GC_PROTO((GC_PTR));

/* Return a pointer to the base (lowest address) of an object given     */
/* a pointer to a location within the object.                           */
/* Return 0 if displaced_pointer doesn't point to within a valid        */
/* object.                                                              */
GC_API GC_PTR GC_base GC_PROTO((GC_PTR displaced_pointer));

/* Given a pointer to the base of an object, return its size in bytes.  */
/* The returned size may be slightly larger than what was originally    */
/* requested.                                                           */
GC_API size_t GC_size GC_PROTO((GC_PTR object_addr));

/* For compatibility with C library.  This is occasionally faster than  */
/* a malloc followed by a bcopy.  But if you rely on that, either here  */
/* or with the standard C library, your code is broken.  In my          */
/* opinion, it shouldn't have been invented, but now we're stuck. -HB   */
/* The resulting object has the same kind as the original.              */
/* If the argument is stubborn, the result will have changes enabled.   */
/* It is an error to have changes enabled for the original object.      */
/* Follows ANSI comventions for NULL old_object.                        */
GC_API GC_PTR GC_realloc GC_PROTO((GC_PTR old_object,
                                   size_t new_size_in_bytes));

/* Explicitly increase the heap size.   */
/* Returns 0 on failure, 1 on success.  */
GC_API int GC_expand_hp GC_PROTO((size_t number_of_bytes));

/* Limit the heap size to n bytes.  Useful when you're debugging,       */
/* especially on systems that don't handle running out of memory well.  */
/* n == 0 ==> unbounded.  This is the default.                          */
GC_API void GC_set_max_heap_size GC_PROTO((GC_word n));

/* Inform the collector that a certain section of statically allocated  */
/* memory contains no pointers to garbage collected memory.  Thus it    */
/* need not be scanned.  This is sometimes important if the application */
```

```
/* maps large read/write files into the address space, which could be   */
/* mistaken for dynamic library data segments on some systems.          */
GC_API void GC_exclude_static_roots GC_PROTO((GC_PTR start, GC_PTR finish));

/* Clear the set of root segments.  Wizards only. */
GC_API void GC_clear_roots GC_PROTO((void));

/* Add a root segment.  Wizards only. */
GC_API void GC_add_roots GC_PROTO((char * low_address,
                                   char * high_address_plus_1));

/* Add a displacement to the set of those considered valid by the       */
/* collector.  GC_register_displacement(n) means that if p was returned */
/* by GC_malloc, then (char *)p + n will be considered to be a valid    */
/* pointer to n.  N must be small and less than the size of p.          */
/* (All pointers to the interior of objects from the stack are          */
/* considered valid in any case.  This applies to heap objects and      */
/* static data.)                                                        */
/* Preferably, this should be called before any other GC procedures.    */
/* Calling it later adds to the probability of excess memory            */
/* retention.                                                           */
/* This is a no-op if the collector was compiled with recognition of    */
/* arbitrary interior pointers enabled, which is now the default.       */
GC_API void GC_register_displacement GC_PROTO((GC_word n));

/* The following version should be used if any debugging allocation is  */
/* being done.                                                          */
GC_API void GC_debug_register_displacement GC_PROTO((GC_word n));

/* Explicitly trigger a full, world-stop collection.    */
GC_API void GC_gcollect GC_PROTO((void));

/* Trigger a full world-stopped collection.  Abort the collection if    */
/* and when stop_func returns a nonzero value.  Stop_func will be       */
/* called frequently, and should be reasonably fast.  This works even   */
/* if virtual dirty bits, and hence incremental collection is not       */
/* available for this architecture.  Collections can be aborted faster  */
/* than normal pause times for incremental collection.  However,        */
/* aborted collections do no useful work; the next collection needs     */
/* to start from the beginning.                                         */
typedef int (* GC_stop_func) GC_PROTO((void));
GC_API int GC_try_to_collect GC_PROTO((GC_stop_func stop_func));
```

```
/* Return the number of bytes in the heap.  Excludes collector private  */
/* data structures.  Includes empty blocks and fragmentation loss.      */
/* Includes some pages that were allocated but never written.           */
GC_API size_t GC_get_heap_size GC_PROTO((void));

/* Return the number of bytes allocated since the last collection.      */
GC_API size_t GC_get_bytes_since_gc GC_PROTO((void));

/* Enable incremental/generational collection.  */
/* Not advisable unless dirty bits are          */
/* available or most heap objects are           */
/* pointerfree(atomic) or immutable.            */
/* Don't use in leak finding mode.              */
/* Ignored if GC_dont_gc is true.               */
GC_API void GC_enable_incremental GC_PROTO((void));

/* Perform some garbage collection work, if appropriate.        */
/* Return 0 if there is no more work to be done.                */
/* Typically performs an amount of work corresponding roughly   */
/* to marking from one page.  May do more work if further       */
/* progress requires it, e.g. if incremental collection is      */
/* disabled.  It is reasonable to call this in a wait loop      */
/* until it returns 0.                                          */
GC_API int GC_collect_a_little GC_PROTO((void));

/* Allocate an object of size lb bytes.  The client guarantees that     */
/* as long as the object is live, it will be referenced by a pointer    */
/* that points to somewhere within the first 256 bytes of the object.   */
/* (This should normally be declared volatile to prevent the compiler   */
/* from invalidating this assertion.)  This routine is only useful      */
/* if a large array is being allocated.  It reduces the chance of       */
/* accidentally retaining such an array as a result of scanning an      */
/* integer that happens to be an address inside the array.  (Actually, */
/* it reduces the chance of the allocator not finding space for such    */
/* an array, since it will try hard to avoid introducing such a false   */
/* reference.)  On a SunOS 4.X or MS Windows system this is recommended */
/* for arrays likely to be larger than 100K or so.  For other systems,  */
/* or if the collector is not configured to recognize all interior      */
/* pointers, the threshold is normally much higher.                     */
GC_API GC_PTR GC_malloc_ignore_off_page GC_PROTO((size_t lb));
GC_API GC_PTR GC_malloc_atomic_ignore_off_page GC_PROTO((size_t lb));
```

```
/* Debugging (annotated) allocation.  GC_gcollect will check            */
/* objects allocated in this way for overwrites, etc.                   */
GC_API GC_PTR GC_debug_malloc
        GC_PROTO((size_t size_in_bytes, char * descr_string, int descr_int));
GC_API GC_PTR GC_debug_malloc_atomic
        GC_PROTO((size_t size_in_bytes, char * descr_string, int descr_int));
GC_API GC_PTR GC_debug_malloc_uncollectable
        GC_PROTO((size_t size_in_bytes, char * descr_string, int descr_int));
GC_API GC_PTR GC_debug_malloc_stubborn
        GC_PROTO((size_t size_in_bytes, char * descr_string, int descr_int));
GC_API void GC_debug_free GC_PROTO((GC_PTR object_addr));
GC_API GC_PTR GC_debug_realloc
        GC_PROTO((GC_PTR old_object, size_t new_size_in_bytes,
                  char * descr_string, int descr_int));

GC_API void GC_debug_change_stubborn GC_PROTO((GC_PTR));
GC_API void GC_debug_end_stubborn_change GC_PROTO((GC_PTR));
# ifdef GC_DEBUG
#   define GC_MALLOC(sz) GC_debug_malloc(sz, __FILE__, __LINE__)
#   define GC_MALLOC_ATOMIC(sz) GC_debug_malloc_atomic(sz, __FILE__, __LINE__)
#   define GC_MALLOC_UNCOLLECTABLE(sz) GC_debug_malloc_uncollectable(sz, \
                                                        __FILE__, __LINE__)
#   define GC_REALLOC(old, sz) GC_debug_realloc(old, sz, __FILE__, \
                                                               __LINE__)
#   define GC_FREE(p) GC_debug_free(p)
#   define GC_REGISTER_FINALIZER(p, f, d, of, od) \
        GC_debug_register_finalizer(p, f, d, of, od)
#   define GC_REGISTER_FINALIZER_IGNORE_SELF(p, f, d, of, od) \
        GC_debug_register_finalizer_ignore_self(p, f, d, of, od)
#   define GC_MALLOC_STUBBORN(sz) GC_debug_malloc_stubborn(sz, __FILE__, \
                                                               __LINE__)
#   define GC_CHANGE_STUBBORN(p) GC_debug_change_stubborn(p)
#   define GC_END_STUBBORN_CHANGE(p) GC_debug_end_stubborn_change(p)
#   define GC_GENERAL_REGISTER_DISAPPEARING_LINK(link, obj) \
        GC_general_register_disappearing_link(link, GC_base(obj))
#   define GC_REGISTER_DISPLACEMENT(n) GC_debug_register_displacement(n)
# else
#   define GC_MALLOC(sz) GC_malloc(sz)
#   define GC_MALLOC_ATOMIC(sz) GC_malloc_atomic(sz)
#   define GC_MALLOC_UNCOLLECTABLE(sz) GC_malloc_uncollectable(sz)
#   define GC_REALLOC(old, sz) GC_realloc(old, sz)
```

```
#   define GC_FREE(p) GC_free(p)
#   define GC_REGISTER_FINALIZER(p, f, d, of, od) \
        GC_register_finalizer(p, f, d, of, od)
#   define GC_REGISTER_FINALIZER_IGNORE_SELF(p, f, d, of, od) \
        GC_register_finalizer_ignore_self(p, f, d, of, od)
#   define GC_MALLOC_STUBBORN(sz) GC_malloc_stubborn(sz)
#   define GC_CHANGE_STUBBORN(p) GC_change_stubborn(p)
#   define GC_END_STUBBORN_CHANGE(p) GC_end_stubborn_change(p)
#   define GC_GENERAL_REGISTER_DISAPPEARING_LINK(link, obj) \
        GC_general_register_disappearing_link(link, obj)
#   define GC_REGISTER_DISPLACEMENT(n) GC_register_displacement(n)
# endif
/* The following are included because they are often convenient, and    */
/* reduce the chance for a misspecifed size argument.  But calls may     */
/* expand to something syntactically incorrect if t is a complicated     */
/* type expression.                                                      */
# define GC_NEW(t) (t *)GC_MALLOC(sizeof (t))
# define GC_NEW_ATOMIC(t) (t *)GC_MALLOC_ATOMIC(sizeof (t))
# define GC_NEW_STUBBORN(t) (t *)GC_MALLOC_STUBBORN(sizeof (t))
# define GC_NEW_UNCOLLECTABLE(t) (t *)GC_MALLOC_UNCOLLECTABLE(sizeof (t))

/* Finalization.  Some of these primitives are grossly unsafe.           */
/* The idea is to make them both cheap, and sufficient to build          */
/* a safer layer, closer to PCedar finalization.                         */
/* The interface represents my conclusions from a long discussion        */
/* with Alan Demers, Dan Greene, Carl Hauser, Barry Hayes,               */
/* Christian Jacobi, and Russ Atkinson.  It's not perfect, and           */
/* probably nobody else agrees with it.     Hans-J. Boehm  3/13/92       */
typedef void (*GC_finalization_proc)
        GC_PROTO((GC_PTR obj, GC_PTR client_data));

GC_API void GC_register_finalizer
        GC_PROTO((GC_PTR obj, GC_finalization_proc fn, GC_PTR cd,
                  GC_finalization_proc *ofn, GC_PTR *ocd));
GC_API void GC_debug_register_finalizer
        GC_PROTO((GC_PTR obj, GC_finalization_proc fn, GC_PTR cd,
                  GC_finalization_proc *ofn, GC_PTR *ocd));
        /* When obj is no longer accessible, invoke              */
        /* (*fn)(obj, cd).  If a and b are inaccessible, and     */
        /* a points to b (after disappearing links have been     */
        /* made to disappear), then only a will be               */
        /* finalized.  (If this does not create any new          */
```

```
        /* pointers to b, then b will be finalized after the    */
        /* next collection.)  Any finalizable object that       */
        /* is reachable from itself by following one or more    */
        /* pointers will not be finalized (or collected).       */
        /* Thus cycles involving finalizable objects should     */
        /* be avoided, or broken by disappearing links.         */
        /* All but the last finalizer registered for an object  */
        /* is ignored.                                          */
        /* Finalization may be removed by passing 0 as fn.      */
        /* Finalizers are implicitly unregistered just before   */
        /* they are invoked.                                    */
        /* The old finalizer and client data are stored in      */
        /* *ofn and *ocd.                                       */
        /* Fn is never invoked on an accessible object,         */
        /* provided hidden pointers are converted to real       */
        /* pointers only if the allocation lock is held, and    */
        /* such conversions are not performed by finalization   */
        /* routines.                                            */
        /* If GC_register_finalizer is aborted as a result of   */
        /* a signal, the object may be left with no             */
        /* finalization, even if neither the old nor new        */
        /* finalizer were NULL.                                 */
        /* Obj should be the nonNULL starting address of an     */
        /* object allocated by GC_malloc or friends.            */
        /* Note that any garbage collectable object referenced  */
        /* by cd will be considered accessible until the        */
        /* finalizer is invoked.                                */

/* Another versions of the above follow.  It ignores            */
/* self-cycles, i.e. pointers from a finalizable object to      */
/* itself.  There is a stylistic argument that this is wrong,   */
/* but it's unavoidable for C++, since the compiler may         */
/* silently introduce these.  It's also benign in that specific */
/* case.                                                        */
GC_API void GC_register_finalizer_ignore_self
        GC_PROTO((GC_PTR obj, GC_finalization_proc fn, GC_PTR cd,
                  GC_finalization_proc *ofn, GC_PTR *ocd));
GC_API void GC_debug_register_finalizer_ignore_self
        GC_PROTO((GC_PTR obj, GC_finalization_proc fn, GC_PTR cd,
                  GC_finalization_proc *ofn, GC_PTR *ocd));

/* The following routine may be used to break cycles between    */
```

```
/* finalizable objects, thus causing cyclic finalizable         */
/* objects to be finalized in the correct order.  Standard      */
/* use involves calling GC_register_disappearing_link(&p),      */
/* where p is a pointer that is not followed by finalization    */
/* code, and should not be considered in determining            */
/* finalization order.                                          */
GC_API int GC_register_disappearing_link GC_PROTO((GC_PTR * /* link */));
        /* Link should point to a field of a heap allocated     */
        /* object obj.  *link will be cleared when obj is       */
        /* found to be inaccessible.  This happens BEFORE any   */
        /* finalization code is invoked, and BEFORE any         */
        /* decisions about finalization order are made.         */
        /* This is useful in telling the finalizer that         */
        /* some pointers are not essential for proper           */
        /* finalization.  This may avoid finalization cycles.   */
        /* Note that obj may be resurrected by another          */
        /* finalizer, and thus the clearing of *link may        */
        /* be visible to non-finalization code.                 */
        /* There's an argument that an arbitrary action should  */
        /* be allowed here, instead of just clearing a pointer. */
        /* But this causes problems if that action alters, or   */
        /* examines connectivity.                               */
        /* Returns 1 if link was already registered, 0          */
        /* otherwise.                                           */
        /* Only exists for backward compatibility.  See below:  */

GC_API int GC_general_register_disappearing_link
        GC_PROTO((GC_PTR * /* link */, GC_PTR obj));
        /* A slight generalization of the above. *link is       */
        /* cleared when obj first becomes inaccessible.  This   */
        /* can be used to implement weak pointers easily and    */
        /* safely. Typically link will point to a location      */
        /* holding a disguised pointer to obj.  (A pointer      */
        /* inside an "atomic" object is effectively             */
        /* disguised.)   In this way soft                       */
        /* pointers are broken before any object                */
        /* reachable from them are finalized.  Each link        */
        /* May be registered only once, i.e. with one obj       */
        /* value.  This was added after a long email discussion */
        /* with John Ellis.                                     */
        /* Obj must be a pointer to the first word of an object */
        /* we allocated.  It is unsafe to explicitly deallocate */
```

```
        /* the object containing link.  Explicitly deallocating */
        /* obj may or may not cause link to eventually be       */
        /* cleared.                                             */
GC_API int GC_unregister_disappearing_link GC_PROTO((GC_PTR * /* link */));
        /* Returns 0 if link was not actually registered.       */
        /* Undoes a registration by either of the above two     */
        /* routines.                                            */

/* Auxiliary fns to make finalization work correctly with displaced     */
/* pointers introduced by the debugging allocators.                     */
GC_API GC_PTR GC_make_closure GC_PROTO((GC_finalization_proc fn, GC_PTR data));
GC_API void GC_debug_invoke_finalizer GC_PROTO((GC_PTR obj, GC_PTR data));

/* GC_set_warn_proc can be used to redirect or filter warning messages. */
/* p may not be a NULL pointer.                                         */
typedef void (*GC_warn_proc) GC_PROTO((char *msg, GC_word arg));
GC_API GC_warn_proc GC_set_warn_proc GC_PROTO((GC_warn_proc p));
    /* Returns old warning procedure.   */

/* The following is intended to be used by a higher level       */
/* (e.g. cedar-like) finalization facility.  It is expected     */
/* that finalization code will arrange for hidden pointers to   */
/* disappear.  Otherwise objects can be accessed after they     */
/* have been collected.                                         */
/* Note that putting pointers in atomic objects or in           */
/* nonpointer slots of "typed" objects is equivalent to         */
/* disguising them in this way, and may have other advantages.  */
# if defined(I_HIDE_POINTERS) || defined(GC_I_HIDE_POINTERS)
    typedef GC_word GC_hidden_pointer;
#   define HIDE_POINTER(p) (~(GC_hidden_pointer)(p))
#   define REVEAL_POINTER(p) ((GC_PTR)(HIDE_POINTER(p)))
    /* Converting a hidden pointer to a real pointer requires verifying */
    /* that the object still exists.  This involves acquiring the       */
    /* allocator lock to avoid a race with the collector.               */
# endif /* I_HIDE_POINTERS */

typedef GC_PTR (*GC_fn_type) GC_PROTO((GC_PTR client_data));
GC_API GC_PTR GC_call_with_alloc_lock
                GC_PROTO((GC_fn_type fn, GC_PTR client_data));

/* Check that p and q point to the same object.                 */
/* Fail conspicuously if they don't.                            */
```

```
/* Returns the first argument.                                  */
/* Succeeds if neither p nor q points to the heap.              */
/* May succeed if both p and q point to between heap objects.   */
GC_API GC_PTR GC_same_obj GC_PROTO((GC_PTR p, GC_PTR q));

/* Checked pointer pre- and post- increment operations.  Note that      */
/* the second argument is in units of bytes, not multiples of the       */
/* object size.  This should either be invoked from a macro, or the     */
/* call should be automatically generated.                              */
GC_API GC_PTR GC_pre_incr GC_PROTO((GC_PTR *p, size_t how_much));
GC_API GC_PTR GC_post_incr GC_PROTO((GC_PTR *p, size_t how_much));

/* Check that p is visible                                              */
/* to the collector as a possibly pointer containing location.          */
/* If it isn't fail conspicuously.                                      */
/* Returns the argument in all cases.  May erroneously succeed          */
/* in hard cases.  (This is intended for debugging use with             */
/* untyped allocations.  The idea is that it should be possible, though */
/* slow, to add such a call to all indirect pointer stores.)            */
/* Currently useless for multithreaded worlds.                          */
GC_API GC_PTR GC_is_visible GC_PROTO((GC_PTR p));

/* Check that if p is a pointer to a heap page, then it points to       */
/* a valid displacement within a heap object.                           */
/* Fail conspicuously if this property does not hold.                   */
/* Uninteresting with ALL_INTERIOR_POINTERS.                            */
/* Always returns its argument.                                         */
GC_API GC_PTR GC_is_valid_displacement GC_PROTO((GC_PTR p));

/* Safer, but slow, pointer addition.  Probably useful mainly with      */
/* a preprocessor.  Useful only for heap pointers.                      */
#ifdef GC_DEBUG
#   define GC_PTR_ADD3(x, n, type_of_result) \
        ((type_of_result)GC_same_obj((x)+(n), (x)))
#   define GC_PRE_INCR3(x, n, type_of_result) \
        ((type_of_result)GC_pre_incr(&(x), (n)*sizeof(*x))
#   define GC_POST_INCR2(x, type_of_result) \
        ((type_of_result)GC_post_incr(&(x), sizeof(*x))
#   ifdef __GNUC__
#       define GC_PTR_ADD(x, n) \
            GC_PTR_ADD3(x, n, typeof(x))
#   define GC_PRE_INCR(x, n) \
```

```
            GC_PRE_INCR3(x, n, typeof(x))
#   define GC_POST_INCR(x, n) \
            GC_POST_INCR3(x, typeof(x))
#   else
        /* We can't do this right without typeof, which ANSI    */
        /* decided was not sufficiently useful.  Repeatedly     */
        /* mentioning the arguments seems too dangerous to be   */
        /* useful.  So does not casting the result.             */
#       define GC_PTR_ADD(x, n) ((x)+(n))
#   endif
#else   /* !GC_DEBUG */
#   define GC_PTR_ADD3(x, n, type_of_result) ((x)+(n))
#   define GC_PTR_ADD(x, n) ((x)+(n))
#   define GC_PRE_INCR3(x, n, type_of_result) ((x) += (n))
#   define GC_PRE_INCR(x, n) ((x) += (n))
#   define GC_POST_INCR2(x, n, type_of_result) ((x)++)
#   define GC_POST_INCR(x, n) ((x)++)
#endif

/* Safer assignment of a pointer to a nonstack location.        */
#ifdef GC_DEBUG
# ifdef __STDC__
#   define GC_PTR_STORE(p, q) \
        (*(void **)GC_is_visible(p) = GC_is_valid_displacement(q))
# else
#   define GC_PTR_STORE(p, q) \
        (*(char **)GC_is_visible(p) = GC_is_valid_displacement(q))
# endif
#else /* !GC_DEBUG */
#   define GC_PTR_STORE(p, q) *((p) = (q))
#endif

/* Fynctions called to report pointer checking errors */
GC_API void (*GC_same_obj_print_proc) GC_PROTO((GC_PTR p, GC_PTR q));

GC_API void (*GC_is_valid_displacement_print_proc)
        GC_PROTO((GC_PTR p));

GC_API void (*GC_is_visible_print_proc)
        GC_PROTO((GC_PTR p));

#ifdef SOLARIS_THREADS
```

```
/* We need to intercept calls to many of the threads primitives, so     */
/* that we can locate thread stacks and stop the world.                 */
/* Note also that the collector cannot see thread specific data.        */
/* Thread specific data should generally consist of pointers to         */
/* uncollectable objects, which are deallocated using the destructor    */
/* facility in thr_keycreate.                                           */
# include <thread.h>
# include <signal.h>
  int GC_thr_create(void *stack_base, size_t stack_size,
                    void *(*start_routine)(void *), void *arg, long flags,
                    thread_t *new_thread);
  int GC_thr_join(thread_t wait_for, thread_t *departed, void **status);
  int GC_thr_suspend(thread_t target_thread);
  int GC_thr_continue(thread_t target_thread);
  void * GC_dlopen(const char *path, int mode);

# ifdef _SOLARIS_PTHREADS
#   include <pthread.h>
    extern int GC_pthread_create(pthread_t *new_thread,
                                 const pthread_attr_t *attr,
                                 void * (*thread_execp)(void *), void *arg);
    extern int GC_pthread_join(pthread_t wait_for, void **status);

#   undef thread_t

#   define pthread_join GC_pthread_join
#   define pthread_create GC_pthread_create
#endif

# define thr_create GC_thr_create
# define thr_join GC_thr_join
# define thr_suspend GC_thr_suspend
# define thr_continue GC_thr_continue
# define dlopen GC_dlopen

# endif /* SOLARIS_THREADS */


#ifdef IRIX_THREADS
/* We treat these similarly. */
# include <pthread.h>
# include <signal.h>
```

```
  int GC_pthread_create(pthread_t *new_thread,
                        const pthread_attr_t *attr,
                        void *(*start_routine)(void *), void *arg);
  int GC_pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
  int GC_pthread_join(pthread_t thread, void **retval);

# define pthread_create GC_pthread_create
# define pthread_sigmask GC_pthread_sigmask
# define pthread_join GC_pthread_join

#endif /* IRIX_THREADS */

#if defined(SOLARIS_THREADS) || defined(IRIX_THREADS)
/* This returns a list of objects, linked through their first           */
/* word.  Its use can greatly reduce lock contention problems, since    */
/* the allocation lock can be acquired and released many fewer times.   */
GC_PTR GC_malloc_many(size_t lb);
#define GC_NEXT(p) (*(GC_PTR *)(p))     /* Retrieve the next element    */
                                        /* in returned list.            */
extern void GC_thr_init();      /* Needed for Solaris/X86       */

#endif /* SOLARIS_THREADS */

/*
 * If you are planning on putting
 * the collector in a SunOS 5 dynamic library, you need to call GC_INIT()
 * from the statically loaded program section.
 * This circumvents a Solaris 2.X (X<=4) linker bug.
 */
#if defined(sparc) || defined(__sparc)
#   define GC_INIT() { extern end, etext; \
                       GC_noop(&end, &etext); }
#else
# if defined(__CYGWIN32__) && defined(GC_USE_DLL)
    /*
     * Similarly gnu-win32 DLLs need explicit initialization
     */
#   define GC_INIT() { GC_add_roots(DATASTART, DATAEND); }
# else
#   define GC_INIT()
# endif
```

```
#endif

#ifdef __WATCOMC__
  /* Ivan Demakov: Programs compiled by Watcom C with -5r option
   * crash without this declaration
   * HB: Could this go into gc_priv.h?
   */
  void GC_noop(void*, ...);
#endif

#ifdef __cplusplus
    }  /* end of extern "C" */
#endif

#endif /* _GC_H */
```