[33] B. Stroustrup, "Possible directions for C++," in *Usenix C++ Workshop Proceedings* [39], pp. 399–416.

[34] B. Stroustrup, "What is "object-oriented programming"," in *Usenix C++ Workshop Proceedings* [39], pp. 159–180.

[35] B. Stroustrup, *C++ Reference Manual.* AT&T, May 1989.

[36] D. Ungar, "Generation Scavenging: A non–disruptive high performance storage reclamation algorithm," in *ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments*, (Pittsburgh, PA), pp. 157–167, Association for Computing Machinery, Apr. 1984.

[37] D. Ungar and F. Jackson, "Tenuring policies for generation-based storage reclamation," in *OOPSLA '88 Conference Proceedings*, pp. 1–17, Association for Computing Machinery, ACM Press, Sept. 1988.

[38] D. M. Ungar, *The Design and Evaluation of a High Performance Smalltalk System.* Cambridge, MA: The MIT Press, 1986.

[39] Usenix Association, *Usenix C++ Workshop Proceedings*, (Santa Fe, NM), Nov. 1987.

[40] C. B. Weinstock, *Dynamic Storage Allocation.* PhD thesis, Carnegie-Mellon University, 1976.

[41] A. Wikstrom, *Functional programming using standard ML.* Prentice Hall, 1987.

[16] D. Edelson and I. Pohl, "Solving C's shortcomings: Use C++," *Computer Languages*, vol. 14, no. 3, 1989.

[17] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, Feb. 1990.

[18] R. Fenichel and J. Yochelson, "A LISP garbage-collector for virtual-memory systems," *Communications of the ACM*, vol. 12, pp. 611–612, Nov. 1969.

[19] R. Gabriel, *Performance and Evaluation of LISP Systems.* MIT Press, 1985.

[20] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Reading, MA: Addison-Wesley Publishing Company, 1983.

[21] B. Kernighan and D. Ritchie, *The C Programming Language.* Englewood-Cliffs, N.J.: Prentice-Hall, 1978.

[22] D. E. Knuth, *The Art of Computer Programming*, vol. 1. Reading, Mass.: Addison, Wesley, 1973. Second ed.

[23] A. Koenig, "An example of dynamic binding in C++," *The Journal of Object-Oriented Programming*, Aug. 1988.

[24] H. Lieberman and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Communications of the ACM*, vol. 26, pp. 419–429, June 1983.

[25] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. Levin, "Lisp 1.5 programmers manual," in *Programming Languages: A Grand Tour* (E. Horowitz, ed.), Computer Science Press, second edition ed., 1985.

[26] B. Meyer, *Object-Oriented Software Construction.* Prentice Hall, 1988.

[27] M. L. Minsky, "A LISP garbage collector algorithm using serial secondary storage," Tech. Rep. Memo 58 (rev.), Project Mac, MIT, Cambridge, MA, Dec. 1963.

[28] D. Moon, "Garbage collection in a large LISP system," in *SIGPLAN Symposium on Lisp and Functional Programming*, pp. 235–246, Association for Computing Machinery, 1984.

[29] I. Pohl and D. Edelson, "A–Z: C language shortcomings," *Computer Languages*, vol. 13, no. 2, 1988.

[30] P. Rovner, R. Levin, and J. Wick, "On extending Modula-2 for building large, integrated systems," Tech. Rep. Research Report 3, Digital Equipment Corporation Systems Research Center, 1985.

[31] T. A. Standish, *Data Structure Techniques.* Addison-Wesley, 1980.

[32] B. Stroustrup, "The evolution of C++ 1985 to 1987," in *Usenix C++ Workshop Proceedings* [39], pp. 1–22.

References

[1] "ANSI C Standard," 1989. American National Standard X3.159-1989.

[2] A. W. Appel, "Garbage collection can be faster than stack allocation," *Information Processing Letters*, vol. 25, pp. 275–279, June 1987.

[3] A. W. Appel, "Simple generational garbage collection and fast allocation," *Software—Practice and Experience*, vol. 19, pp. 171–183, Feb. 1989.

[4] A. W. Appel, J. R. Ellis, and K. Li, "Real-time concurrent collection on stock multiprocessors," in *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 11–20, Association for Computing Machinery, ACM Press, July 1988.

[5] H. G. Baker, "List processing in real time on a serial computer," *Communications of the ACM*, vol. 21, pp. 280–294, Apr. 1978.

[6] J. F. Bartlett, "Compacting garbage collection with ambiguous roots," tech. rep., Digital Equipment Corporation, Western Research Laboratory, Palo Alto, California, Feb. 1988.

[7] J. F. Bartlett, "Mostly copying garbage collection picks up generations and C++," Tech. Rep. TN–12, DEC WRL, Oct. 1989.

[8] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software—Practice and Experience*, vol. 18, pp. 807–820, Sept. 1988.

[9] A. Burgess, Apr. 1990. Personal Communication.

[10] J. A. Campbell, "A note on an optimal-fit method for dynamic allocation of storage," *Computer Journal*, vol. 14, pp. 7–9, Feb. 1971.

[11] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson, "Modula-3 report," tech. rep., Digital Systems Research Center and Olivetti Research Center, Palo Alto, CA, 1988.

[12] J. Cohen, "Garbage collection of linked data structures," *ACM Computing Surveys*, vol. 13, pp. 341–367, Sept. 1981.

[13] A. Demers, Dec. 1989. private communication.

[14] A. Demers, M. Weiser, B. Hayes, H. Bohem, D. Bobrow, and S. Shenker, "Combining generational and conservative garbage collection: Framework and implementations," in *popl90*, pp. 261–269, 1980.

[15] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: An excercise in cooperation," *Communications of the ACM*, vol. 21, pp. 966–974, Nov. 1978.

```
        } \
    } \
    void update() { } \
}; \
                                                                    \
inline \
droot_base_/**/T::droot_base_/**/T() \
    : prev(&dummy_/**/T::dummy), next(dummy_/**/T::dummy.next) \
{ \
    prev→next = this; \
    next→prev = this; \
} \
                                                                    \
inline DR_/**/T::DR_/**/T(T * p) : ptr(p) { } \
inline DR_/**/T::DR_/**/T() : ptr(NULL) { } \
                                                                    \

#define SUB_DROOT(D,B) \
                                                                        \
struct DR_/**/D : public droot_base_/**/B { \
private: \
    D * ptr; \
public: \
    virtual void update() { if (ptr) ptr→copy(ptr); } \
    D & operator*() { return *ptr; } \
    D * operator→() { return ptr; } \
    void operator=(D * p) { ptr = p; } \
                                                                        \
    DR_/**/D(D * p) : ptr(p) { } \
    DR_/**/D() : ptr(NULL) { } \
}; \
```

```
/* \
 * This class is for non-stack-allocated roots to the \
 * data structure. \
 */ \
                                                                \
struct droot_base_/**/T { \
friend void T::gc(); \
private: \
    droot_base_/**/T * prev; \
    droot_base_/**/T * next; \
protected: \
    droot_base_/**/T(); \
    ~droot_base_/**/T() \
        { next→prev = prev; prev→next = next; } \
public: \
    virtual void update() = 0; \
    static ostream & print(ostream &); /* for debugging */\
}; \
                                                                \
class T; \
                                                                \
struct DR_/**/T : public droot_base_/**/T { \
private: \
    T * ptr; \
public: \
    virtual void update() { if (ptr) ptr→copy(ptr); } \
    T & operator*() { return *ptr; } \
    T * operator→() { return ptr; } \
    void operator=(T * p) { ptr = p; } \
            operator T * () { return ptr; } \
                                                                \
    DR_/**/T(T * p); \
    DR_/**/T(); \
}; \
                                                                \
struct dummy_/**/T : public droot_base_/**/T { \
friend class droot_base_/**/T; \
friend void T::gc(); \
private: \
    static DR_/**/T dummy; \
public: \
    dummy_/**/T() : droot_base_/**/T(*this) { \
        if ((droot_base_/**/T *) this ≠ &dummy_/**/T::dummy) { \
            error("Fatal error:  dummy node as wrong object.\n"); \
```

```
    R_/**/T(T * p) : ptr(p), link(head.root_list) \
        { void(head.root_list = this); } \
    R_/**/T() : ptr(NULL), link(head.root_list) \
        { void(head.root_list = this); } \
    ~R_/**/T() \
        { void(head.root_list = link); } \
};

#define SUB_ROOT(D,B) \
                                                    \
struct R_/**/D; \
struct D; \
                                                    \
struct root_list_head_/**/D : root_list_head { \
    R_/**/D * root_list; \
    virtual void copy(); \
    root_list_head_/**/D() { \
        next = root_list_head_/**/B::liststart; \
        root_list_head_/**/B::liststart = this; \
        root_list = NULL; \
    } \
}; \
                                                    \
struct R_/**/D { \
    static root_list_head_/**/D head; \
    D * ptr; \
    R_/**/D * link; \
public: \
    D * & get_ptr() { return ptr; } \
    D & operator*() { return *ptr; } \
    D * operator->() { return ptr; } \
    void operator=(D * p) { ptr = p; } \
            operator D * () { return ptr; } \
                                                    \
    R_/**/D(D * p) : ptr(p), link(head.root_list) \
        { head.root_list = this; } \
    R_/**/D() : ptr(NULL), link(head.root_list) \
        { head.root_list = this; } \
    ~R_/**/D() \
        { head.root_list = link; } \
};

#define DROOT(T) \
                                                    \
```

```
 * parameter indicates the type of object the root points at.
 *
 * Stacked and global roots are allocated off a stack. The stack is
 * a "protected stack." Such a stack will automatically grow itself
 * when its last page is touched. This does not have the overhead of
 * having to check bounds on every push.
 *
 */

extern void error(char *);

struct root_list_head {
    virtual void copy() = 0;
    root_list_head * next;
};

#define ROOT(T) \
                                                                \
struct R_/**/T; \
struct T; \
                                                                \
struct root_list_head_/**/T : root_list_head { \
    static root_list_head * liststart; \
    R_/**/T * root_list; \
    virtual void copy(); \
    root_list_head_/**/T() { \
        next = liststart; \
        liststart = this; \
        root_list = NULL; \
    } \
}; \
                                                                \
struct R_/**/T { \
friend int main(); \
    static root_list_head_/**/T head; \
    T * ptr; \
    R_/**/T * link; \
public: \
    T * & get_ptr() { return ptr; } \
    T & operator*() { return *ptr; } \
    T * operator->() { return ptr; } \
    void operator=(T * p) { ptr = p; } \
            operator T * () { return ptr; } \
                                                                \
```

```
        * Number of R/W pages in stack.
        * Add 1 to get total pages because of the R/O page.
        */
       int page_count;

       /*
        * For the linked list of protected page addresses
        */
       protected_item link; // my node in the list


public:

       /*
        * Constructors and destructors maintain the list of
        * protected addresses.
        */
       protected_obj(client * cp, int npages);
       ~protected_obj();

       char * getbase() { return firstpage; }
       char * getlimit();
};
```

A.4.6 `root` Macro Definitions

```
/*
 *
 * ROOT.H
 *
 *
 * In order to perform garbage collection the collector must be able
 * to locate the roots of the data structure. Conservative GC
 * (e.g., Bartlett) does not identify the roots, instead it searches
 * the entire address space for pointer-sized quantities that might
 * be pointers. It assumes they are. In this collector we explicitly
 * identify the roots of the data structure to permit copying
 * collection.
 *
 * Roots are of two kinds: stack-allocated (stacked) and heap-
 * allocated (dynamic). Dynamic roots are normally members of a more
 * complex therefore simple overloading of root::new is not possible.
 *
 * Roots are parameterized with a "node type" of some kind. The
```

```
};

typedef protection_client client;

struct protected_item {
        char * address; // Lowest addr on the R/O page

        client * faultee; // Object to invoke handler on

        protected_item * next; // Next low addr in chain


        void setbase(char * base) { address = base; }

        // The linked list of protected addresses

        static protected_item * list_head;

        protected_item(client * cp) : address(0), faultee(cp), next(list_head)
                { list_head = this; }

        ~protected_item();

        enum curr_protection { RO, RW } status;

#if DEBUG
        ostream & print(ostream & o);
        static ostream & print_list(ostream &);
#endif

};

/*
 * CLASS PROTECTED
 */

class protected_obj {
private:
        /*
         * Data for identifying myself
         * Pointer to first page of chunk
         */
        char * firstpage;
        /*
```

```
 *
 * pointer to first allocated (R/W) page
 *
 * int page_count;
 *
 * Number of R/W pages! One less than total pages allocated!
 *
 * static protected_item * listhead;
 *
 * The first link of the chain of protected page addresses
 * This static variable is global to all protected_stacks.
 *
 * protected * link;
 *
 * This object's link in the chain of addresses
 *
 * HANDLER
 *
 * The handler cannot be a member function. It must be called
 * by the operating system when a fault comes it. It needs to
 * detemine which protected_stack to grow. It knows the fault
 * address.
 *
 * The protected_stacks maintain a linked list that indicates
 * all their last_page addresses. The one whose last_page
 * matches the fault address is the one that needs to grow.
 *
 */

/*
 * Nodes of type protected_address are used for the linked list
 * through which the fault-handler locates the proper
 * protected_stack object.
 */


struct protection_client {
# ifndef ultrix
                friend int sighandler(int, int, struct sigcontext *, char * addr);
# else
                friend int sighandler(int, char * addr, struct sigcontext *);
# endif
protected:
        virtual void fault_handler() = 0;
```

```
*
* Other functions
*
* set_signal(int mode = 0)
*
* mode == 0: cause SIGSEGV to be handled by our routine
* mode == 1: cause SIGSEGV to be handled by double_segv()
*
* grow()
* Grow the region by one page and
* copy all the existing data.
*
* NONMEBER OPERATIONS
*
* friend void sighandler(int, int, struct sigcontext *, char *)
*
* The routine to catch the signal.
* Temporary redirect the signal to a double-fault handler.
* Figure out which protected object got the signal,
* call its handler.
* Then reset the signal handler.
*
* alloc_protected(int pages)
*
* allocate the number of pages specified plus one more page
* that is read-only. Return a pointer to the start address.
* Performs the memory protection.
*
* double_segv(int sig, int code, struct sigcontext * scp, char * addr)
*
* The emergency signal handler.
* If a really bad thing happens, i.e., we get a SIGSEGV while
* in our SIGSEGV handler, this routine is called.
*
* void set_signal(int mode)
*
* Cause SIGSEGV to be trapped. The mode argument determines
* which handler catchs it, the normal one or the double-fault
* handler. While we are handling one we set the handler to
* be the double-fault handler, then we restore it.
*
* STATE:
*
* int * firstpage;
```

```
}

/*
 * mprotect.h
 *
 * Header file for the memory protection objects.
 *
 * Protected memory manager.
 *
 * Clients use this class to obtain a set of memory pages.
 * The last of the pages is protected read-only.
 * This module catches the segmentation violation signal.
 * It figures out which protected page the violation occured on.
 * Then it calls a handler for that page.
 *
 * Every memory object is constructed with a function pointer and
 * a void * object pointer. The object that the fault occured from
 * calls through the function pointer with the object pointer. That
 * allows object specific processing for the faulting page.
 *
 * All of the protected stack objects must be locatable by the signal
 * handler. This is so that it can figure out which caused the fault.
 * Therefore a linked list of all the existing objects plus their
 * respective fault-addresses is maintained and available to the
 * fault-handler. Included in the linked list is the handler and
 * handler-argument for the object.
 *
 * MEMBER OPERATIONS:
 *
 * Constructors and Destructors
 *
 * protected(handler, handler_argument, npages)
 *
 * Cause the object to have npages R/W pages allocated.
 * It has one more R/O page.
 * Initialize the object. Insert it into the linked list.
 * This is how the signal handler identifies the correct
 * object to handle a fault.
 *
 * ~protected()
 *
 * Remove me from the linked list of protected addresses.
 * Unprotect my protected page. Free my memory.
```

```
 * The last page must be read-only.
 * We first allocate the pages for read/write,
 * and then modify the flags on the last page.
 * Since we are doing memory protecting we must allocate
 * on page boundarys. That's why valloc is used.
 */
static char *
palloc(int page_count)
{
        /*
         * Allocate page_count pages on a page-aligned boundary.
         */
        char * addr = (char*) low_alloc(map_size((page_count) << PGLOG));
        char * last = addr + ((page_count-1) << PGLOG);

        /*
         * Mark last page as read-only, returns nonzero on failure
         */
        if ( mprotect(last, PGSIZE, PROT_READ) == -1 ) {
                cerr << "\nCan't write protect last page, errno " << errno <<
endl;
                exit(2);
        }

        return addr;
}

/*
 * Trap SIG_PROTECT with the specified signal handler.
 */

static void
set_signal()
{
        const int sig = SIG_PROTECT;
        struct sigvec vec;

        sigvec(SIG_PROTECT, NULL, &vec); // Get current signal mask

        vec.sv_mask &= ~sigmask(SIG_PROTECT); // Allow another SEGV signal

        vec.sv_handler = (int(*)())sighandler; // set to single or double

        sigvec(SIG_PROTECT, &vec, NULL); // Set the handler
```

```
 * Then restore the signal handler, me.
 */
#ifndef ultrix
int sighandler(int, int, struct sigcontext *, char * addr)
#else
int sighandler(int, char * addr, struct sigcontext *)
#endif
{
        /*
         * Locate the right protected object
         */
        register protected_item * p = protected_item::list_head;

        while (p && !samepage(p→address, addr))
                        p = p→next;

        if (!p) {
                cerr ≪ "Unexpected " ≪ sigtype ≪ " at " ≪ (void*) addr ≪
"\n";
                abort();
        }

        p→faultee→fault_handler(); // Handle the fault


        // Unprotect the page

        if (p→status == protected_item::RO) {
                if ( mprotect(p→address, PGSIZE, PROT_READ|PROT_WRITE)
== -1 ) {
                        cerr ≪ "Can't unprotect page at " ≪
(void*)p→address ≪ "\n";
                        exit(2);
                }
                p→status = protected_item::RW;
        }

        return 0; // ignored

}

/*
 * Allocate pages of memory from the system.
 * All but the last page must be flagged for read/write.
```

```
}

// Unprotect my page and free the memory

protected_obj::~protected_obj()
{
        if (link.status == protected_item::RO) {
                if (mprotect(link.address, PGSIZE, PROT_READ|PROT_WRITE)
== -1) {
                        cerr << "protected_obj::destructor() can't unprotect
old page,"
                                << " errno " << errno << endl;
                        exit(1);
                }
        }

        // Free the memory

        register int size_code = map_size((page_count + 1) << PGLOG);
        low_free(firstpage, size_code);
}

/*
 * Nonmember function implementations
 */

/*
 * Ultrix and SunOS use different signals for protection viloations
 */
#ifdef ultrix
#define SIG_PROTECT SIGBUS
const char * const sigtype = "SIGBUS";
#else
#define SIG_PROTECT SIGSEGV
const char * const sigtype = "SIGSEGV";
#endif


/*
 * Catch the SIG_PROTECT.
 * Determine which protected_obj got the fault by traversing
 * the linked list comparing base_addresses to the fault address.
 * That's why the linked list of protected addresses is needed.
 * Then call the associated handler with its arg.
```

```
// it from the list. It's linked list entry is the one that

// points at "this."

protected_item::~protected_item()
{
        protected_item ** linkpp = &list_head;
        while (*linkpp ≠ this)
                linkpp = &((*linkpp)→next);
        *linkpp = (*linkpp)→next;
}

/*
 * Class protected_obj Member Function Implementations
 */

char * protected_obj::getlimit()
{
        return firstpage + (page_count ≪ PGLOG);
}

/*
 * Add my R/O page address to the linked list
 */
protected_obj::protected_obj(client * cp, int pages) : link(cp)
{
        page_count = pages-1;

        // Allocate protected space

        firstpage = palloc(pages);
        link.status = protected_item::RO;

        // Assign the protected page address into my linked list entry

        char * r_o_page_addr = firstpage + (page_count ≪ PGLOG);
        link.setbase(r_o_page_addr);

        // This should really be done once, not on every constructor call.

        // trap sigsegv

        set_signal();
```

```
extern "C" { int getpagesize(); }
extern int log2(int);

static const int PGSIZE = getpagesize();
static const int PGLOG = log2(PGSIZE);

/*
 * The Static Member of Class protected_item
 */

/*
 * The static data member that points at the linked-list of
 * protected addresses. It is iniallly an empty list.
 */
protected_item * protected_item::list_head = NULL;

/*
 * Prototypes of non-member functions defined herein
 */


typedef int (*sig_handler_type)(); // Any signal handler


#ifdef ultrix
int sighandler(int, char *, struct sigcontext *);
#else
int sighandler(int, int, struct sigcontext *, char *);
#endif

static char * palloc(int page_count);
static void set_signal();
static int samepage(char *, char *);
static inline int samepage(char * adr1, char * adr2)
{
        return (int(adr1) >> PGLOG) == (int(adr2) >> PGLOG);
}

/*
 * Class protected_item Member Function Implementations
 */

// This destructor finds its linked list entry and unlinks
```

```
        heapmode setmode(heapmode m) { heapmode t = mode; mode = m; return
t; }
        chunk * newspace();
        void set_gc(vfp f) { gc = f, mode = collect; }
        size_t set_incr(size_t i) { size_t t = incr; incr = i; return t;}

        byte * get(size_t n) {
                free_ptr[n] = 0;
                register byte * p = free_ptr;
                free_ptr += n;
                return p;
        }

        mallor(vfp = NULL, size_t = CHUNK_SIZE, heapmode = grow);
        ~mallor();
};
```

A.4.5 Protected Memory `class`

```
#ifndef ultrix
#include <vm/faultcode.h>
#endif

#include <iostream.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/mman.h>
#include "signal.h"
#include "mprotect.h"
#include "low_align.h"

/*
 * Externals Used
 */

extern "C" { int mprotect(void *, int, int); }

extern int errno;
extern char *sys_errlist[];

/*
 * Constants
 */
```

```
 * its link to the next item in the list. The chunks must be retained
 * so they can be deallocated when the Heap is destroyed.
 *
 * The last page of c chunk is write-protected so that no bounds check is
 * needed at allocation time, keeping allocation to a minimal number of
 * instructions.
 *
 * To allocate an object of n bytes, the current free_ptr is incremented
 * by n. The old value will be returned as the objects's address. First
 * however the allocator does a bounds check. It writes a byte to the
 * free_ptr's new address. If it has overrun the last write-allowed page
 * of the chunk this will cause a segmentation violation. The handler will
 * catch the trap, unprotect the page, and assign NULL to the return
 * value. It is up to the caller what to do next, whether to allocate
 * another chunk or to garbage collect.
 *
 */

typedef char byte;

typedef byte * Pbyte;

typedef protected_obj chunk;

typedef void (*vfp)();

extern const size_t CHUNK_SIZE;

class mallor : public protection_client {
public:
        enum heapmode {grow, collect};
private:
        chunk * curr_chunk;
        byte * free_ptr;
        size_t incr;

        void fault_handler();
        void (*gc)();

        heapmode mode;

public:

        static void release(chunk * chunk_linked_list);
```

```
                delete tmp;
                tmp = cp;
        }
}
/*
 * A simple memory allocator
 */

/*
 *
 * The mallor type implements a simple, fast memory allocator similar to
 * the one described by Appel in "Simple Generational Garbage Collection
 * and Fast Allocation." A mallor consists initially of a chunk of
 * bytes. It has a pointer into the data that indicates the current
 * allocation point. The last page of the array is write-protected through
 * the virtual memory hardware.
 *
 * +------+
 * | Page 0 |
 * +------+
 * | Page 1 |
 * +------+
 * | Page 2 |
 * +------+
 * | Page 3 |
 * |write-proctd|
 * +------+
 *
 * The first four bytes (sizeof void*) are reserved. The reason is
 * described in the next paragraph.
 *
 * The allocator allocates out of one chunk until the chunk is out of
 * room. When that happens the allocator has a choice. It may:
 *
 * 1) Instigate a copying collection
 * 2) Continue allocating from another chunk
 *
 * Case one is handled by a function pointer member of the Heap. It
 * simply calls through the function pointer to begin a collection.
 * If the function pointer is NULL then the only option is #2.
 * The allocator maintains a linked list of chunks that have been used
 * to satisfy requests. When a chunk is exhausted and the allocator
 * wants to continue with another one, it inserts the exhausted chunk
 * into a linked list. The first four bytes of the chunk are used for
```

```
mallor::~mallor()
{
        release(curr_chunk);
}

void
mallor::fault_handler()
{
        if (mode == collect) {
                if (gc ≠ NULL) {
                        gc();
                        return;
                }
                cerr ≪ "mallor::fault_handler:  can't garbage collect
because no\n";
                cerr ≪ "collection routine is set.  Growing instead.\n";
        }

        // Grow the chunk linked list by one more chunk

        register chunk * tmp = newspace();
        *(chunk**)curr_chunk→getbase() = tmp;
}

chunk *
mallor::newspace()
{
        register chunk * tmp = curr_chunk;
        curr_chunk = new protected_obj(this,incr/PGSIZE);
        free_ptr = curr_chunk→getbase();
        *(chunk**)free_ptr = NULL;
        free_ptr += sizeof(chunk*);

        return tmp;
}

void
mallor::release(chunk * cp)
{
        protected_obj *tmp = cp;

        while(cp) {
                cp = *(protected_obj**) cp→getbase();
```

```
 * Appel in "Simple Generational Garbage Collection and Fast
 * Allocation" except that it supports discontinuous chucks and
 * is encapsulated, thus making it multiply instantiatable.
 *
 * A chunk is a sequence of pages. The last page is write-protected.
 * The chunk is obtained from the protected_obj class. Every object
 * is touched when it is allocated. When an object crosses the
 * protection boundary there will be a write-fault. The protected-
 * obj fault handler will trap the signal and call our fault
 * handler. We can either initiate a garbage collection or extend
 * the space by obtaining a new chunk. A space is a linked list of
 * chunks. The first word (pointer-sized) quantity of every chunk
 * is reserved for forming the list.
 *
 * Another version of the allocator does not use virtual memory
 * protection, instead it performs an explicit test.
 */


#include <iostream.h>
#include <stdlib.h>

#include "mprotect.h"
#include "faulting_mallor.h"

extern "C" { int getpagesize(); }

static const int PGSIZE = getpagesize();
static const size_t BLOCK = (32 * 1024);
const size_t CHUNK_SIZE = BLOCK < PGSIZE ? PGSIZE : BLOCK;

mallor::mallor(void (*f)(void), size_t size, heapmode m)
        : mode(m), gc(f)
{
        if (!size)
                size = BLOCK < PGSIZE ? PGSIZE : BLOCK; // default value bug


        incr = size;
        newspace();
}

// Delete all the chunks
```

```
 *
 * This should use the binary buddy system.
 */


const int MIN_EXPONENT = 10;
const int MAX_EXPONENT = 24;
const int MIN_BLOCK = 1 << MIN_EXPONENT; /* 1024 bytes */
const int MAX_BLOCK = 1 << MAX_EXPONENT; /* 16 Megabytes */
const int TOTAL_SIZES = MAX_EXPONENT - MIN_EXPONENT + 1;

void * low_alloc(int encoded_size_of_request);
void low_free(void * addr, int encoded_block_size);

int log2(int);

inline int map_size(int sz)
{
    return log2(sz) - MIN_EXPONENT;
}

inline int unmap_size(int code)
{
    return 1 << (code + MIN_EXPONENT);
}
```

A.4.4 Faulting Block Buffer Allocator

```
/*
 *
 * faulting_mallor.c
 *
 * Memory allocator.
 *
 * Implementation File.
 *
 *
 *
 * The class described in this pair of files implements
 * a memory allocator. It allocates storage out of discontinuous
 * chunks that are obtained from a low-level allocator. It write-
 * protects the end of each chunk and detects an out-of-bounds
 * with the resulting write-fault. It uses the protected_obj
 * to accomplish this. The scheme is based on that advocated by
```

```
                /* Free block of the desired size? */
                if (tmp ≠ 0) {
                        free_lists[list_num] = *tmp;
                        return tmp;
                }
        }

        /*
         * Obtain a block from the operating system.
         * If the current break is not on a page boundary waste some memory.
         */
        register int curr, diff;
        curr = int(sbrk(0));
        diff = curr + (PGSIZE-1);
        diff &= ~(PGSIZE-1);
        diff = diff - curr;
        curr = int(sbrk(unmap_size(list_num) + diff));
        if (curr == -1)
                /* No block available from the operating system */
                error("Sbrk provided no more memory.");
        return (int**) (curr + diff);
}
/*
 *
 * low_align.h
 *
 * Low level memory allocator.
 *
 * Header File.
 *
 *
 *
 * The class described in this pair of files implements
 * a low-level memory allocator. It allocates blocks in sizes
 * that are powers of two. The smallest block it will allocate
 * is 1024 bytes. The largest is 16Meg.
 *
 * This version of the low-level allocator page-aligns the blocks
 * it returns to make use of the mprotect system call more
 * convenient.
 *
 * It maintains linked lists of blocks of each size. When a size
 * that it requires is unavailable it obtains one from sbrk.
```

```
static const int PGSIZE = getpagesize();
extern char * form(const char *, ...);

static void * free_lists[TOTAL_SIZES] = {
    0, 0, 0,
    0, 0, 0,
    0, 0, 0,
    0, 0, 0,
    0, 0, 0
};

static void error(char * s)
{
        fprintf(stderr, "%s\n",s );
    exit(2);
}

int log2(int x)
{
    register int log = 0;

    while ((x&1) == 0) {
        log++;
        x = x >> 1;
    }

    if (x >> 1)
        error("Size is not an even power of two.");

    return log;
}

void low_free(void * addr, register int list_num)
{
    register int * * tmp = (int * *) addr;

    *tmp = (int*) free_lists[list_num];
    free_lists[list_num] = tmp;
}

void * low_alloc(int list_num)
{
        {
                register int * * tmp = (int * *) free_lists[list_num];
```

```
        }

    mallor(vfp = NULL, size_t = CHUNK_SIZE, heapmode = grow);
    ~mallor();
};
```

A.4.3 Low Level Page-Aligned Allocator

```
/*
 *
 * low_align.c
 *
 * Low level memory allocator.
 *
 * Implementation File.
 *
 *
 *
 * The class described in this pair of files implements
 * a low-level memory allocator. It allocates blocks in sizes
 * that are powers of two. The smallest block it will allocate
 * is 1024 bytes. The largest is 16Meg.
 *
 * This version of the low-level allocator page-aligns the blocks
 * it returns to make use of the mprotect system call more
 * convenient.
 *
 * It maintains linked lists of blocks of each size. When a size
 * that it requires is unavailable it obtains one from sbrk.
 *
 * This should use the binary buddy system.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "low_align.h"


extern "C" { void exit(int); }
extern "C" { char * sbrk(int); }
extern "C" { int getpagesize(); }
```

```
extern int fault_count;

typedef char byte;
typedef byte * Pbyte;

struct chunk {
    int size_code;
    chunk * next;
};

typedef void (*vfp)();

extern const size_t CHUNK_SIZE;

class mallor {
public:
    enum heapmode {grow, collect};
private:
    chunk * curr_chunk;
    byte * free_ptr;
    byte * lower_limit;
    int size_code;
    heapmode mode;
    byte * fault();
    void (*gc)();

public:

    static void release(chunk * chunk_linked_list);
    heapmode setmode(heapmode m)
                { register heapmode t = mode; mode = m; return t; }
    chunk * flip();
    void set_gc(vfp f) { gc = f, mode = collect; }

    int set_incr(int i) {
                int t = unmap_size(size_code);
                size_code = map_size(i);
                return t;
        }

    byte * get(int n) {
                 if ((free_ptr -= n) < lower_limit)
                         free_ptr = fault() - n;
                 return free_ptr;
```

```
    return tmp;
}

void
mallor::release(register chunk * cp)
{
    register chunk * tmp;

    while(cp) {
        tmp = cp→next;
        low_free(cp, cp→size_code);
        cp = tmp;
    }
}
/*
 *
 * testing_mallor.h
 *
 * Memory allocator.
 *
 * Header file.
 *
 *
 *
 * The class described in this pair of files implements
 * a memory allocator. It allocates storage out of discontinuous
 * chunks that are obtained from a low-level allocator.
 * This version performs an explicit bounds check to know when
 * it is out of space.
 *
 * A chunk is a sequence of bytes obtained from the low level alloc.
 * On a fault we can either initiate a garbage collection or extend
 * the space by obtaining a new chunk. A space is a linked list of
 * chunks. The first word (pointer-sized) quantity of every chunk
 * is reserved for forming the list.
 *
 */
// A simple memory allocator

#include <stdlib.h>
#include <stddef.h>

extern char * mem_start;
```

```
// Delete all the chunks

mallor::~mallor()
{
    release(curr_chunk);
}

byte *
mallor::fault()
{
    if (mode == collect && gc ≠ NULL) {
        gc();
        return free_ptr;
    }

    // Grow the chunk linked list by one more chunk


    // allocate a new chunk

    register chunk * tmp = curr_chunk;
    curr_chunk = (chunk*) low_alloc(size_code);
    curr_chunk→size_code = size_code;
    curr_chunk→next = tmp;

    // reinitialize the boundary

    lower_limit = (byte *) (curr_chunk + 1);

    // return the current allocation point, end of the chunk

    return unmap_size(size_code) + (byte*) curr_chunk;
}

chunk *
mallor::flip()
{
    chunk * tmp = curr_chunk;

    curr_chunk = (chunk *) low_alloc(size_code);
    curr_chunk→size_code = size_code;
    curr_chunk→next = NULL;
    lower_limit = (byte *) (curr_chunk+1);
    free_ptr = unmap_size(size_code) + (char*) curr_chunk;
```

```
 *
 * Implementation File.
 *
 *
 *
 * The class described in this pair of files implements
 * a memory allocator. It allocates storage out of discontinuous
 * chunks that are obtained from a low-level allocator.
 * This version performs an explicit bounds check to know when
 * it is out of space.
 *
 * A chunk is a sequence of bytes obtained from the low level alloc.
 * On a fault we can either initiate a garbage collection or extend
 * the space by obtaining a new chunk. A space is a linked list of
 * chunks. The first word (pointer-sized) quantity of every chunk
 * is reserved for forming the list.
 *
 */
#include <stdlib.h>
#include <stddef.h>
#include "low_mallor.h"
#include "testing_mallor.h"

char * mem_start = 0;

static const size_t BLOCK = (32 * 1024);
extern "C" { int getpagesize(); }
extern "C" { char * sbrk(int); }
const int PGSIZE = getpagesize();
const size_t CHUNK_SIZE = BLOCK < PGSIZE ? PGSIZE : BLOCK;

mallor::mallor(void (*f)(void), size_t size, heapmode m) : mode(m), gc(f)
{
    if (!size)
        size = BLOCK < PGSIZE ? PGSIZE : BLOCK; // default value bug


    if (mem_start == 0)
        mem_start = sbrk(0);

    size_code = map_size(size);
    (void) flip();
}
```

```
            /* No block available from the operating system */
            error("Sbrk provided no more memory.");
    }
    return tmp;
}

/*
 * The class described in this pair of files implements
 * a low-level memory allocator. It allocates blocks in sizes
 * that are powers of two. The smallest block it will allocate
 * is 1024 bytes. The largest is 16Meg.
 *
 * It maintains linked lists of blocks of each size. When a size
 * that it requires is unavailable it obtains one from sbrk.
 *
 */

const int MIN_EXPONENT = 10;
const int MAX_EXPONENT = 24;
const int MIN_BLOCK = 1 << MIN_EXPONENT; /* 1024 bytes */
const int MAX_BLOCK = 1 << MAX_EXPONENT; /* 16 Megabytes */
const int TOTAL_SIZES = MAX_EXPONENT - MIN_EXPONENT + 1;

void * low_alloc(int encoded_size_of_request);
void low_free(void * addr, int encoded_block_size);

int log2(int);

inline int map_size(int sz)
{
    return log2(sz) - MIN_EXPONENT;
}

inline int unmap_size(int code)
{
    return 1 << (code + MIN_EXPONENT);
}
```

A.4.2 Testing Block Buffer Allocator

```
/*
 *
 * testing_mallor.c
 *
 * Memory allocator.
```

```
{
    cerr << s << endl;
    exit(2);
}

int log2(int x)
{
    int log = 0;

    if (!x)
        error("Size is zero.");

    while ((x&1) == 0) {
        log++;
        x = x >> 1;
    }

    if (x >> 1 != 0) {
        error("Size is not an even power of two.");
        exit(2);
    }

    return log;
}

void low_free(void * addr, register int list_num)
{
    register int * * tmp = (int * *) addr;

    *tmp = (int*) free_lists[list_num];
    free_lists[list_num] = tmp;
}

void * low_alloc(int list_num)
{
    register int * * tmp = (int * *) free_lists[list_num];

    /* Free block of the desired size? */
    if (tmp != 0)
        free_lists[list_num] = *tmp;
    else {
        /* Obtain a block from the operating system. */
        tmp = (int**) sbrk(unmap_size(list_num));
        if (tmp == (int**) -1)
```

```
#ifdef NONE
    t = NULL; // copy no data

#endif
#ifdef HALF
    t→trunc1(); // copy half of the data

#endif
#ifndef NODELETE
    Node::gc();
#endif
}
```

A.4 Code

A.4.1 Low Level Allocator

```
/*
 * The class described in this pair of files implements
 * a low-level memory allocator. It allocates blocks in sizes
 * that are powers of two. The smallest block it will allocate
 * is 1024 bytes. The largest is 16Meg.
 *
 * It maintains linked lists of blocks of each size. When a size
 * that it requires is unavailable it obtains one from sbrk.
 *
 */

#include <iostream.h>
#include <stdlib.h>
#include "low_mallor.h"

extern "C" { void exit(int); }
extern "C" { char * sbrk(int); }
extern char * form(const char *, ...);

static void * free_lists[TOTAL_SIZES] = {
     0, 0, 0,
     0, 0, 0,
     0, 0, 0,
     0, 0, 0,
     0, 0, 0
};

static void error(char * s)
```

```
Tree::Tree(char* op, Node * n)
{
    p = new UnaryNode(op, n);
}

Tree::Tree(char* op, Node * left, Node * right)
{
    p = new BinaryNode(op, left, right);
}

Tree::Tree(char* op, Tree left, Tree right)
{
    p = new BinaryNode(op, left.p, right.p);
}

ostream & operator<<(ostream & o, Node * p)
{
        p->print(o);
        return o;
}

ostream & operator<<(ostream & o, R_Node & t)
{
        t->print(o);
        return o;
}

Tree MakeTree(int depth)
{
        if (depth == 1)
                return Tree("+", Tree(1), Tree(1));
        else
                return Tree("+", MakeTree(depth-1), MakeTree(depth-1));
}

main(int argc,char ** argv)
{
    if (argc==1) {
          cerr << "Need depth on command line.\n";
          exit(2);
        }
    int depth = atoi(argv[1]);
    R_Node t;
    t = MakeTree(depth);
```

```
    int value();
    virtual void copy(Node * & ptr);
    virtual void trunc1() { left = NULL; }
};

void BinaryNode::copy(Node * & ptr)
{
  register BinaryNode * newptr;

  if (!(ptr = (Node*) get_forward())) {
          ptr = newptr = new BinaryNode(*this); // copy object

          set_forward(newptr); // set forward

          if (left) left->copy(newptr->left); // copy child

          if (right) right->copy(newptr->right); // copy child

  }
}

int BinaryNode::value()
{
          switch (*op) {
          case '-': return left->value()-right->value();
          case '+': return left->value()+right->value();
          case '*': return left->value()*right->value();
          case '/': return left->value()/right->value();
          case '^': return left->value()^right->value();
          default: cerr << "Binary Node ``" << op << "'' unknown.\n";
          }
          return 1;
}

Tree::Tree(int n)
{
    p = new IntNode(n);
}

Tree::Tree(char* op, Tree t)
{
    p = new UnaryNode(op, t.p);
}
```

```
class UnaryNode: public Node {
    friend class Tree;
    char * op;
    Node * opnd;
    UnaryNode(char* a, Node * b): op(a), opnd(b) {}
    void print (ostream& o) {o << "(" << op << opnd << ")";}
    int value();
    virtual void copy(Node * & ptr);
};

void UnaryNode::copy(Node * & ptr)
{
  register UnaryNode * newptr;

#ifdef DEBUG
  copied++;
#endif
  if (!(ptr = (Node*) get_forward())) {
          ptr = newptr = new UnaryNode(*this); // copy object

          set_forward(newptr); // set forward

          if (opnd) opnd->copy(newptr->opnd); // copy child

  }
}

int UnaryNode::value()
{
          switch (*op) {
          case '-': return -opnd->value();
          case '+': return opnd->value();
          default: cerr << "Unary Node ``" << op << "'' unknown.\n";
          }
          return 1;
}

class BinaryNode: public Node {
    friend class Tree;
    char* op;
    Node * left;
    Node * right;
    BinaryNode(char* a, Node * b, Node * c): op(a), left(b), right(c){}
    void print (ostream& o) {o << "(" << left << op << right << ")";}
```

```
class Tree {
    friend class Node;
    friend ostream& operator≪(ostream&, const Tree&);
    Node * p;
public:
    Tree(int);
    Tree(Node * n) : p(n) { }
    Tree(char*, Tree);
    Tree(char*, Tree, Tree);
    Tree(char*, Node *);
    Tree(char*, Node *, Node *);
    Tree(const Tree& t) { p = t.p; }
    void operator=(const Tree& t);

    int value() { return p→value(); }
    operator R_Node() { return p; }
    operator Node *() { return p; }
};

void Tree:: operator=(const Tree& t)
{
    p = t.p;
}

ostream& operator≪(ostream& o, const Tree& t)
{
    t.p → print(o);
    return (o);
}

class IntNode: public Node {
    friend class Tree;
    int n;
    IntNode(int k): n (k) {}
    void print(ostream& o) { o ≪ n; }
    int value() { return n; }
    virtual void copy(Node * & ptr)
    {
        if ((ptr = (Node*) get_forward()) == NULL)
            set_forward(ptr = new IntNode(*this));
    }
};
```

A.3.3 Garbage Collected

```
// C++ example JOOP august 1988 Andrew Koenig

// Heavily modified


// Necessary modifications:

// In the copy routine need to copy not a member but

// rather a member of a member.

//


#include <iostream.h>
#include <stdlib.h>
#include "low_mallor.h"
#include "testing_mallor.h"
#include "root.h"
#include "gc.h"

class R_Node;

class Node {
    friend class Tree;
    friend ostream& operator≪(ostream&, const Tree&);
    friend ostream & operator≪(ostream &, Node *);
    friend ostream & operator≪(ostream &, R_Node &);
protected:
    virtual void print(ostream&) = 0;
public:
    virtual void copy(Node * & ptr) = 0;
    virtual void trunc1() { }
    COLLECTION_MEMBERS(Node)
    virtual int value() = 0;
};

mallor Node::heap(Node::gc,32 * 1024);

ROOT(Node)
DROOT(Node)
DECLARE_GC(Node)
```

```
    Node * left;
    Node * right;
public:
    BinaryNode(char* a, Node * b, Node * c): op(a), left(b), right(c){}
    void print (ostream& o) {o << "(" << left << op << right << ")";}
    int value();
    ~BinaryNode() { }
};

int BinaryNode::value()
{
        switch (*op) {
        case '-': return left->value()-right->value();
        case '+': return left->value()+right->value();
        case '*': return left->value()*right->value();
        case '/': return left->value()/right->value();
        case '^': return left->value()^right->value();
        default: cerr << "Binary node `" << op << "' unknown.\n";
        }
        return 1;
}

Node * MakeTree(int depth)
{
        if (depth == 1)
                return new BinaryNode("+", new IntNode(1), new IntNode(1));
        else
                return new BinaryNode("+", MakeTree(depth-1),
MakeTree(depth-1));
}

main(int argc,char ** argv)
{
    if (argc==1) {
          cerr << "Need depth on command line.\n";
          exit(2);
        }
    int depth = atoi(argv[1]);
    Node * t = MakeTree(depth);
#ifndef NODELETE
    delete t;
#endif
}
```

```
#include <iostream.h>
#include <stdlib.h>

class Node {
public:
    virtual int value() = 0;
    virtual void print(ostream&) = 0;
    Node(){}
    virtual ~Node() {} //note virtual destructor

};

class IntNode: public Node {
    int n;
public:
    IntNode(int k): n (k) {}
    ~IntNode() {}
    void print(ostream& o) { o << n; }
    int value() { return n; }
};

class UnaryNode: public Node {
    char* op;
    Node * opnd;
public:
    UnaryNode(char* a, Node * b): op(a), opnd(b) {}
    void print (ostream& o) {o << "(" << op << opnd << ")";}
    int value();
    ~UnaryNode() { }
};

int UnaryNode::value()
{
        switch (*op) {
        case '-': return -opnd->value();
        case '+': return opnd->value();
        default: cerr << "Unary node `" << op << "' unknown.\n";
        }
        return 1;
}

class BinaryNode: public Node {
    char* op;
```

```
Tree::Tree(int n)
{
    p = new IntNode(n);
}

Tree::Tree(char* op, Tree t)
{
    p = new UnaryNode(op, t);
}

Tree::Tree(char* op, Tree left, Tree right)
{
    p = new BinaryNode(op, left, right);
}

Tree MakeTree(int depth)
{
        if (depth == 1)
                return Tree("+", Tree(1), Tree(1));
        else
                return Tree("+", MakeTree(depth-1), MakeTree(depth-1));
}

main(int argc,char ** argv)
{
    if (argc==1) {
            cerr << "Need depth on command line.\n";
            exit(2);
        }
    int depth = atoi(argv[1]);
    Tree t = MakeTree(depth);
    //cout << "Value is " << t.value() << endl;

#ifdef NODELETE
        ++t.p->use;
#endif
}
```

A.3.2 Manually Reclaimed

```
//C++ example JOOP august 1988 Andrew Koenig

// Totally rewritten
```

```
};

class UnaryNode: public Node {
    friend class Tree;
    char* op;
    Tree opnd;
    UnaryNode(char* a, Tree b): op(a), opnd(b) {}
    void print (ostream& o) {o << "(" << op << opnd << ")";}
    int value();
};

int UnaryNode::value()
{
        switch (*op) {
        case '-': return -opnd.value();
        case '+': return opnd.value();
        default: cerr << "Unary node `" << op << "' unknown.\n";
        }
        return 1;
}

class BinaryNode: public Node {
    friend class Tree;
    char* op;
    Tree left;
    Tree right;
    BinaryNode(char* a, Tree b, Tree c): op(a), left(b), right(c){}
    void print (ostream& o) {o << "(" << left << op << right << ")";}
    int value();
};

int BinaryNode::value()
{
        switch (*op) {
        case '-': return left.value()-right.value();
        case '+': return left.value()+right.value();
        case '*': return left.value()*right.value();
        case '/': return left.value()/right.value();
        case '^': return left.value()^right.value();
        default: cerr << "Binary node `" << op << "' unknown.\n";
        }
        return 1;
}
```

```
    Node() { use = 1; }
    virtual int value() = 0;
    virtual void print(ostream&) = 0;
    virtual ~Node() {} //note virtual destructor

};

class Tree {
friend int main(int, char**);
    friend class Node;
    friend ostream& operator≪(ostream&, const Tree&);
    Node * p;
public:
    Tree(int);
    Tree(char*, Tree);
    Tree(char*, Tree, Tree);
    Tree(const Tree& t) { p = t.p; ++p → use;}
    ~Tree() { if (--p→use == 0) delete p; } //ref count

    void operator=(const Tree& t);

    int value() { return p→value(); }
};

void Tree:: operator=(const Tree& t)
{
    ++t.p → use;
    if ( --p → use == 0)
        delete p;
    p = t.p;
}

ostream& operator≪(ostream& o, const Tree& t)
{
    t.p → print(o);
    return (o);
}

class IntNode: public Node {
    friend class Tree;
    int n;
    IntNode(int k): n (k) {}
    void print(ostream& o) { o ≪ n; }
    int value() { return n; }
```

```
void
main(int argc, char ** argv)
{
    int total;
    int chunks;
    sscanf(argv[1], "%d", &total);
    sscanf(argv[2], "%d", &chunks);
#if FAULT || TEST
    node::heap = new mallor(NULL, chunks * 1024);
#endif

    register node * p;
    register int limit = 1024 * total / sizeof(node);
    while (limit--) {
        p = new node;
        if (!p) {
            fprintf(stderr, "Allocation failed.\n");
            exit(1);
        }
        p->data[0] = 0;
    }
}
```

A.3 Expression Tree Example

The following code was used to create the expression tree measurements of §4.5.

A.3.1 Reference Counted

//C++ example JOOP august 1988 Andrew Koenig

//Extensively modified by Daniel Edelson

```
#include <iostream.h>
#include <stdlib.h>

class Node {
   friend int main(int, char**);
   friend class Tree;
   friend ostream& operator<<(ostream&, const Tree&);
private:
   int use;
protected:
```

```
main(int argc, char * * argv)
{
    int allocated = 0;
    char *curr_brk = 0, *old_brk;
    int INCR = 0;
    int counter = 0, counter_max = 10;
    double expansion;

    sscanf(argv[1], "%d", &INCR));  // get request size
    sbrk(0);                        // make brk stabilize
    char *mem_start = old_brk = sbrk(0);

    while (1) {
        new char[INCR];
        curr_brk = sbrk(0);
        if (curr_brk != old_brk)
            ++counter;
        if (counter == counter_max)
            break;
        old_brk = curr_brk;
        allocated += INCR;
    }
    expansion = double(curr_brk - mem_start) / allocated;
    cout << double(INCR) << tab << expansion << endl;
}
```

The following program was used to obtain allocator timing information. It was `exec`'ed by another program that used the `wait3` system call to obtain timing information. It was compiled with one of four `#define` macros turned on to yield one of for programs. The four test respectively, `::new`, `malloc`, the custom allocator with an explicit test, and the custom allocator with write protection.

```
struct node {
    char data[20];
#if FAULT || TEST
    static mallor * heap;
    void * operator new(size_t size) { return heap->get(size); }
    void operator delete(void * p) { }
#endif
};

#if FAULT || TEST
mallor * node::heap = NULL;
#endif
```

```
    for (int i=0; i<2000000; i++)
        f();
}

void f()
{
    node  * r = NULL;
}

int
main()
{
    for (int i=0; i<2000000; i++)
        f();
}

void f()
{
}

int
main()
{
    for (int i=0; i<2000000; i++)
        f();
}

void f()
{
    DR_node r = NULL;
}

int
main()
{
    for (int i=0; i<2000000; i++)
        f();
}
```

A.2 Allocator Measurements

The following program was used to determine the space efficiency of the various allocators. The version shown below tested the standard allocator. It was run for every block size from 1 to 2499 bytes.

6 Future Work

There is justification for the assumption that a generation-based collector would not be appropriate for an imperative language like C++. However, this needs to be explored. Burgess has implemented a generation-based mark-and-sweep collector for Sil [9], an imperative object-oriented programming language based on Pascal. He found that generations improved the performance of the system by a factor of approximately 10. This is in spite of the fact that the system checks on every assignment to determine if a back pointer is being created. This is in the context of a free-store of M bytes where the system has $M/2$ bytes of real memory. The generation-based collector examines fewer objects so it causes fewer page faults. If that works for a non-copying generation-based collector in an imperative object-oriented language, it should certainly work for a copying collector. This would best serve applications that maintain a very large amount of living, dynamic data.

In a system with several allocators, our low-level allocator should use the buddy-system to reduce external fragmentation. Determination of the optimal chunk size for the main allocator is an open problem.

Our intention is to develop an incremental, concurrent version of the collector.

The most important future step, now that we've validated the approach, is reimplementing the collector in a C++ compiler. This requires a new syntax because it does not seem appropriate to treat "collectible" as a system-defined base class. This raises new problems, such as how to treat incomplete declarations of collected types, that are beyond the scope of this thesis.

A Appendices

A.1 Root Measurements

A.1.1 Creating and Destroying Roots

The following programs were executed with the UNIX C-shell `time` command to measure the cost of creating and destroying roots and droots. The first program creates and destroys two million roots. The second program creates and destroys two million pointers. The third program does nothing two million times. By subtracting the third program's time from those of the first two, the expense of creating roots versus pointers can be measured. The fourth program creates and destroys two million droots.

```
void f()
{
    R_node r = NULL;
}

int
main()
{
```

with the UNIX `mprotect` system call. Then it allocates objects and touches each one that it allocates. A SIGSEGV (write-fault) signal from the operating system indicates that the collector is out of space. Appel suggests this scheme in [2]. In C++ an object's initialization is under the control of the programmer; thus its fields may be initialized in any order, or not initialized at all. The allocator cannot rely on initialization to cause the fault; it must touch the object itself. This means that two instructions, a compare and a conditional branch, are eliminated, but one is added. This scheme is highly inappropriate when objects may be larger than one page. The performance of the faulting allocator did not justify its added complexity.

Dynamic allocation in C++ incurs additional overhead because the constructor must do nothing when the allocator returns zero. This test, plus the passing and saving of values on the stack, tends to reduce the importance of saving one or two instructions in the allocator. If allocation and construction could share code then initialization of a *vptr* could be used by the allocator to trigger a fault. This would save an instruction and improve the efficiency of the faulting-allocator. It probably still would not justify the complexity.

Since this collector can allocate low-level chunks incrementally it may do work proportional to the total size of the free-store. Using large chunks reduces the overhead due to freeing them. If only one large chunk is used this allocator emulates a standard one.

We have proposed tracking the roots of the data structure using an auxiliary last-in first-out (LIFO) list. The advantage of this is that it does not require tags or hardware support making it appropriate for C++. It also has the advantage that locating the roots of the data structure takes time proportional to the number of such roots, rather than the current stack depth and the total amount of global data.

This implementation of the collector requires no modifications to the compiler, but it does require assistance from the programmer. Another version in the compiler would require changes to the language. Such a collector could be more efficient than this one.

The system is composed of encapsulated data structures making it appropriate for an object-oriented imperative programming language. This collector shows a way that garbage collection can be made an efficient, non-invasive part of the C++ programming language.

In this system we have accomplished the following:

1. added efficient and reasonably convenient automatic storage reclamation to C++,

2. found an appropriate organization for dynamic memory reclamation in C++ that remains within the philosophy of the language,

3. developed a platform for research into new collection techniques and algorithms, particularly in copying collection and virtual memory issues, and

4. identified one way in which C++ could better support this activity, namely, by diverging from the stated intention of the language and allowing allocation and construction to interact.

The collector has a scheme for working with foreign roots. However, since destructors are not called for collected objects, a foreign (doubly-linked) root must not be inside an object that is reclaimed with copying garbage collection. If a collected object contained a `droot` to another object, the root would continue to exist even after the containing object was deallocated. Objects containing foreign roots must be reclaimed another way, such as manually.

Collecting a homomorphic data structure is quite easy from the perspective of the client programmer. It becomes more complicated when the data structure is highly polymorphic and derived classes have different numbers of internal pointers than the base class. New `copy` routines must be supplied for each derived class. This is not hard and is only done once when the classes are defined.

5 Conclusion

Efficient management of dynamic data is difficult. Only in simple cases can programmer-controlled deallocation be safe, efficient and simple. Reclamation of generalized dynamic graph data structures requires edge traversal to identify unused blocks. Unless blocks are reused objects will be scattered in memory causing excessive paging and using an unnecessarily large amount of backing store.

There are two predominant classes of reclamation algorithms: mark-and-sweep and copying. Mark-and-sweep collectors must be able to recognize pointers to collected objects. They take one pass to mark all living objects and one pass to deallocate inaccessible ones. Another pass may be used to compact objects. A conservative collector—conservative is a subclass of mark-and-sweep—can be implemented with minimal compiler support. These collectors need not differentiate between pointers and integers. They reclaim most, though not all, of the inaccessible memory. They can never compact objects in memory because they cannot adjust pointers. Neither mark-and-sweep nor conservative collection permits the use of very fast allocators. These collectors do work proportional to the total size of the free-store rather than the amount of living data. Copying collection collects and compacts in one pass. Its work is proportional to the amount of living data rather than the size of the free-store when it collects.

This thesis has presented a copying collector and its C++ implementation. The collector will work with lists, trees, DAGs, or cyclic graphs. It incorporates very fast allocation and a novel way of tracking roots that does not require tagged pointers or integers. The collector's work is proportional to the amount of living data, therefore when most objects die it is highly efficient. This is likely if the free-store is allowed to grow very large. When the amount of data that must be copied can be kept small, the collector is more efficient than reference counting or manual reclamation. This is because the allocator is faster and there is only $58\mu s$ (approximately 600 integer arithmetic instructions) overhead in a null collection, thus the deallocation overhead is very small if most objects die.

Two allocators accompany this collector. They obtain memory from a low-level allocator that is also provided. One of them performs an explicit bounds check on every allocation to ensure that space is available. The other write-protects the last usable page

followed, inaccessible memory will be recycled and live objects will be compacted. This increases programmer productivity by removing from the application programmer responsibility for deallocating data. It reduces paging by compacting objects.

Unlike the conservative collector described in [8] this collector works in the presence of foreign roots, i.e., roots contained in dynamic objects allocated from different free-stores. Unlike Appel's collector [3] this allocator does not dictate where in the processes address space the free-store must reside. This allows multiple, independent collected data structures. It removes the need for an extra copy to move the surviving objects back to the lowest addresses of the free-store. With this allocator the free-store can grow dynamically. Many copying collector allocators fix the size of the free-store. This allocator will more closely track the actual amount of memory required since, after a collection, it is off by no more than one chunk. The behavior of other allocators is easily emulated by using a very large (e.g., multi-megabyte) chunk size and disallowing expansion.

This collector demonstrates a new way of tracking roots of the data structure without compiler assistance. The efficiency of this scheme is reported in §4.

C++ provides the `new` and `delete` operators that are overloaded to use the efficient allocator. The allocator supplied with this package is fast. It is not as fast as the hypothetical one described by Appel, but that one had very limited scope. Appel's allocator integrated allocation with initialization. It would not be appropriate in C++ where base-classes and constructors define the order of sub-object and member initialization, respectively. The measurements of §4.1.1 show that the allocator supplied with this system is much faster than the allocator provided in the standard libraries.

4.7 Limitations

One problem, not with this implementation but with copying collection for C++, results from the fact that objects are never individually deallocated. Without deallocation of individual objects destructors cannot be called for collected objects. A programmer who provides a collected `class` with a destructor is likely to be surprised when the destructor is never invoked. Never examining dead objects increases the efficiency of copying collectors. This explains why copying collectors do work proportional to the number of living objects rather than number of dead plus living objects. If copying collection is preferred, this cannot be seen as a disadvantage. It is only a disadvantage because it is a divergence from the C++ style.

Certain pointer operations become more expensive under this root scheme. The primary inefficiency involves the current implementations of the root-stack. The linked-list implementation uses two words for roots making it inconvenient to put them in registers. The array implementation has inefficiencies in growing the array. The proposed implementation described in §3.5.4 solves both problems.

The system does not support arrays of collected objects, though it does support arrays of pointers to collected objects. All objects of a collected type must be dynamically allocated. A global or `static` object of such a type can be simulated using a global `root` that always references the same object.

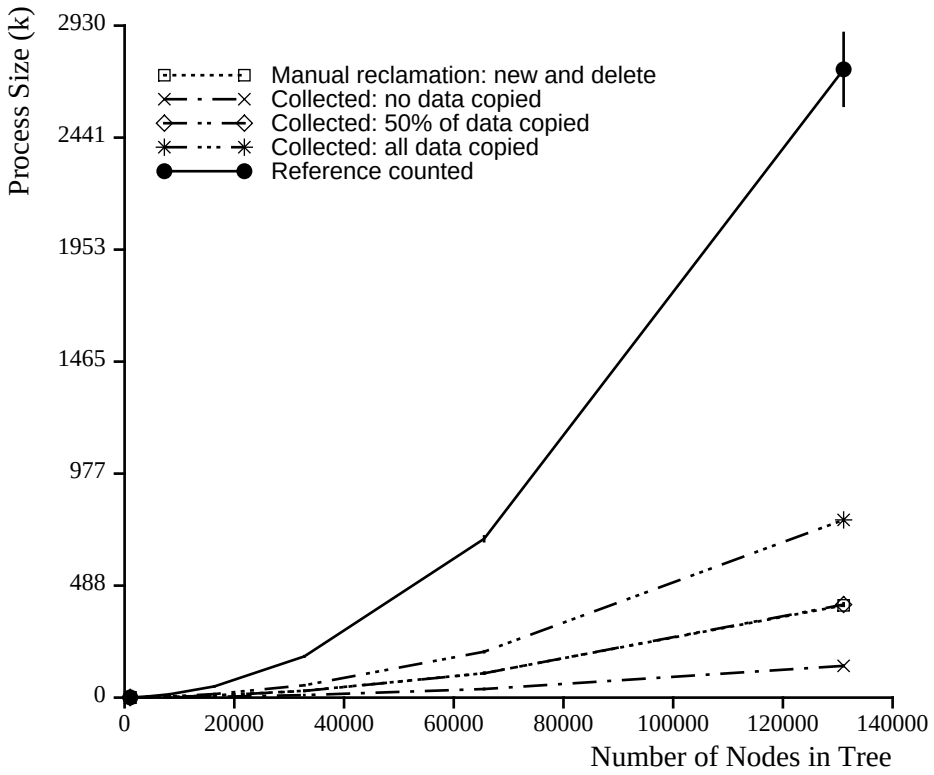

Figure 26: Process Size to Construct, Compute and Destroy the Expression Tree.

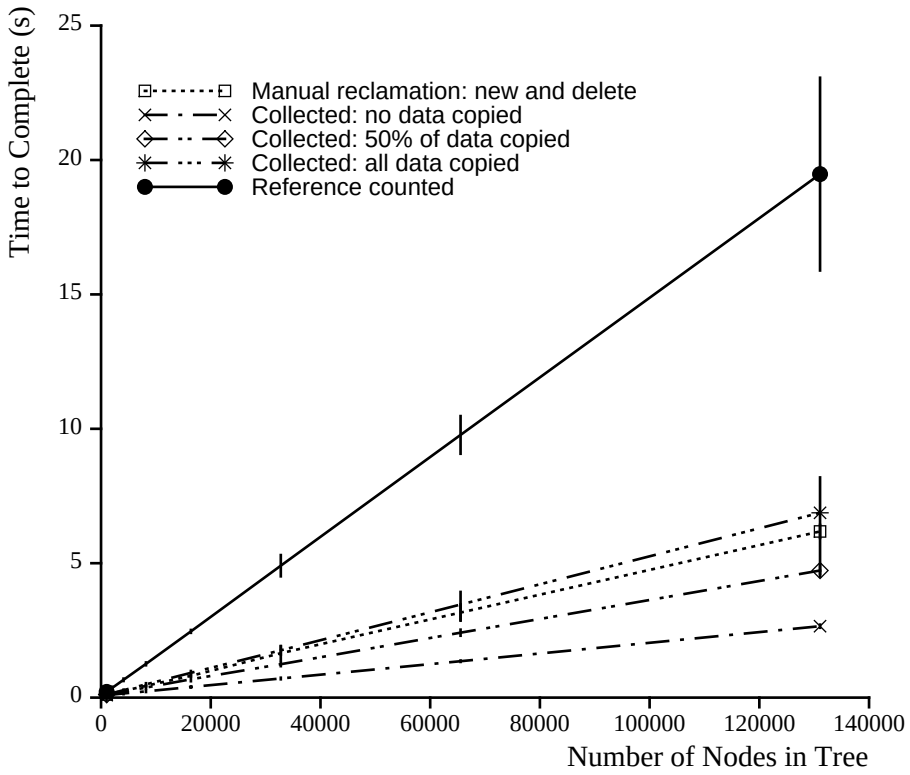

Figure 25: Time to Construct, Compute and Destroy the Expression Tree.

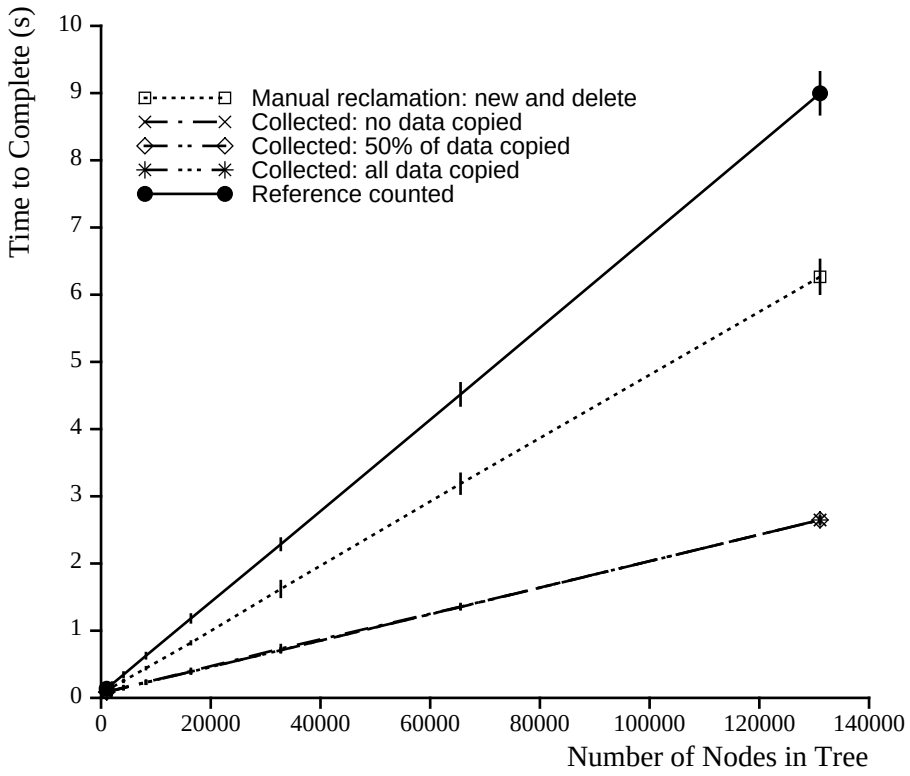

Figure 24: Time to Construct the Expression Tree: SPARCstation 1.

Each test was timed many times until reasonably tight 95% confidence intervals were obtained. Some tests required as few as 300 runs; others needed over 1500 executions. The 95% confidence intervals are shown in the graphs as vertical bars at every point. In many cases the width of the interval does not exceed the thickness of the line so no interval bar is visible. The time each program required to construct and evaluate the data structure is shown in the graph of figure 24, the total time each took to execute is shown in the graph of figure 25, and the space required is shown in figure 26. The three garbage collected versions took exactly the same amount of time to construct the data structure. This is consistent since they differ only in the amount of data they copy during the collection.

The results show that the reference counted version takes much longer and requires much more space than any other version. The garbage collected version can construct and copy the entire data structure in almost the same amount of time it takes manual reclamation to construct and free it.

4.6 Advantages

Reclaiming dynamically allocated objects can be difficult. As currently implemented, this collector requires only a little help from the application programmer and is compiler independent. Thereafter, provided the collector is correct and its restrictions are

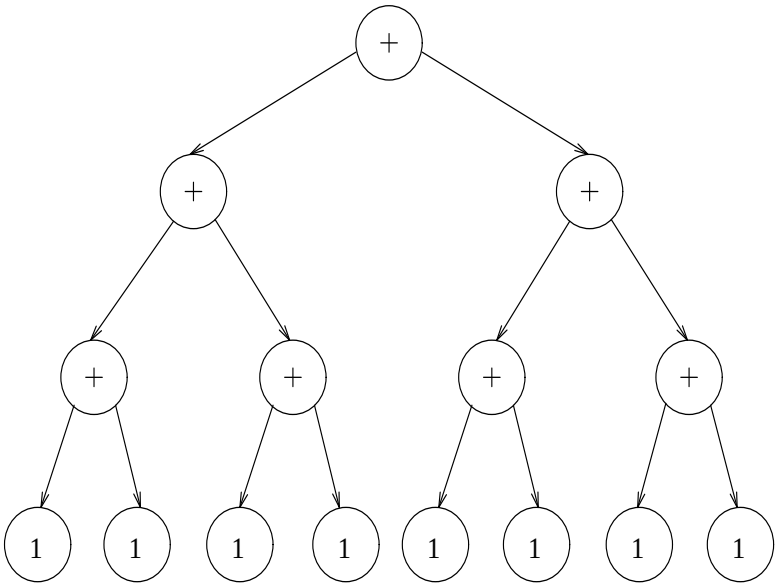

Figure 23: A sample data structure from the *expression* example.

4.5 Expression Tree Example

In [23] Koenig presents an example of using a polymorphic data structure in C++. The example uses inheritance and virtual functions to construct and print arithmetic expressions. Koenig's example uses reference counting to allow shared subgraphs. We have reimplemented the example to use both manual reclamation and garbage collection and compared the implementations for space and time efficiency. The three class definitions are shown in the appendix.

The data structure that we use is a balanced binary tree in which all internal nodes are "+" nodes and all leaf nodes are "1" nodes. A sample data structure is shown in figure 23. A tree data structure was used rather than a more general directed graph so that manual reclamation could be compared against the other forms. Trees with depths ranging from 10 to 17 were used.

The tests recursively construct and evaluate the data structure, and then free it. The reference counted and manually reclaimed versions destroy the data structure with destructors and calls to `delete`. Three cases of the collected version are shown. One deletes the entire data structure by overwriting the root with NULL. Another makes half the tree unreachable, and the third leaves the whole tree reachable. All three invoke a garbage collection. By overwriting pointers the versions cause 0%, 50% and 100%, respectively, of the data structure to be copied. The tests do not take into account the fact that a copying collection compacts objects.

The versions are compared for time to complete and process size. The measurements were obtained by a driver program written by the author that uses the UNIX `wait3` system call to obtain the data. Times are reported by the operating system with 17ms granularity; space is reported in 1024 byte units.

Table 3: Collector Efficiency Measurements

Nodes have 12 bytes of data, 4 bytes of forwarding pointer, and 4 bytes of *vptr*.
The data structure of $2^{17} - 1$ nodes requires 2559k bytes.

Operation	Time to Complete	
	VAXstation 3520	SPARCstation 1
Collect "null", 1 root, 1 chunk, no data	$58.5\mu s$	$11.5\mu s$
Build a binary tree: $2^{17} - 1$ nodes	$4.2s$	$2.4s$
Build and collect the tree	$12.3s$	$4.2s$
Time to copy $2^{17} - 1$ binary nodes	$6.0s$	$1.8s$
Time per node (minus overhead)	$45.8\mu s$	$13.7\mu s$
Nodes copied per second	21,845	72,992
Kbytes copied per second	426	1425

the base class type. In that case there is only one root stack adding 9 instructions or about $3\mu s$. Further, let us assume there are no `droot`s. The remaining cost of the collection is the time spent copying nodes. Copying a node entails: a virtual function call to the `copy` routine, a call to `new` to allocate the new memory, a call to the copy constructor, and virtual function calls through the internal pointers of the node. It is difficult to estimate the amount of time this will take because the application defines the copy constructor; it may not be simply linear in the size of the object.

Copying the data requires a call to the `copy` routine for every edge in the graph, plus calls to `new` and the copy constructor for every node. The main overhead associated with the copy routine is invoking it through the *vptr*.

Calls to the allocator take 6 instructions, except when a fault occurs. With 256k chunks faults will be very rare, there would be four if a megabyte were copied. Twenty byte objects, such as those benchmarked in this thesis, would require 5 instructions each to copy. If many objects are copied this expense will be significant. The key in copying collection is to wait to collect until many objects have died. That way copying will be fast.

A "null" collection when there is one root in one root stack and only one chunk has been allocated requires $60\mu s$ on the VAXstation. The collection time is independent of the amount of global data or the depth of the current stack frame because the collector need not search blindly for the roots; it can iterate over them directly.

For comparison purposes the collector was benchmarked collecting a graph of 20 byte nodes on both the VAXstation and a SPARCstation. The results are shown in table 3.

Table 2: Efficiency of creating `root`s compared to simple pointers.

Operation	Time
Create/initialize/destroy 2,000,000 pointers	$11.8s$
Create/initialize/destroy 2,000,000 roots	$14.2s$
Create/initialize/destroy 2,000,000 droots	$23.1s$
Startup, termination and function call overhead	$11.0s$
Time per pointer	$0.4\mu s$
Time per root	$1.6\mu s$
Time per droot	$6.1\mu s$

4.3.2 Creating and Destroying a Droot

Constructing and destroying `droot`s requires a doubly-linked list insertion and deletion, respectively. However, droots are only found within dynamic data structures. The allocator cannot be the ultra-fast block allocator of a copying collector because `droot` destructors must be called. The allocator must be a general purpose one such as Sequential Fit, Buddy System or Quick Fit. Taking a little more time to create and destroy droots should substantially impact program performance only in pathological cases. The time to create and destroy two million droots was 23.4 seconds. Subtracting the overhead and dividing by two million gives a time of $6.2\mu s$ per `droot`. Thus, doubly-linked roots are about four times as expensive as stackable roots. A doubly linked-list insertion requires four operations whereas a linked-stack insertion requires two. The construction takes four times as long rather than twice as long because two of the operations are doubly-indirected rather than singly.

4.4 Collecting

The next important measurement is the speed of a collection. A garbage collection pass visits every root, and then copies every node reachable from the root. The code that accomplishes this for a garbage-collected type `node` is shown in figure 16.

A garbage collection includes processing for the *flip* plus the deep copy of the data structure. The flip is constant time and fast. On the VAXstation, when a low-level block is available, it requires approximately 22 instructions. When the low level allocator must go to the operating system for more memory that adds 4 instructions plus a call to the `sbrk` system call. Deleting from-space adds approximately 12 instructions per chunk of from-space. With large chunks this time is very small. Assume four megabytes have been allocated in 256k chunks and that the low-level allocator has a free 256k chunk. Then on the two processor VAXstation 3520 the overhead for managing spaces during a collection would be 102 instructions. Estimating the machine's throughput at 3 MIPS gives a total of $34\mu s$ managing spaces.

A polymorphic data structure is often manipulated with only a single kind of root,

check. Assuming 20 byte objects that means 36k chunks, rounding up to a power of two as required by the low-level allocator gives 64k chunks.

4.3 Roots

Increasing the efficiency of root creation and object allocation is extremely important because these are very common operations. A program that uses a very large free-store may never need to garbage collect, yet it can still be slow if these operations are inefficient.

4.3.1 Creating and Destroying a Root

Creating a root means allocating and constructing a pointer variable. Global and `static` variables are created once at the beginning of the program. The efficiency of constructing them contributes little to the efficiency of a program. The critical roots are those allocated on the stack including: local variables, by-value function parameters, function return values, and temporaries.

Creating a pointer (root) on the stack normally requires two instructions: decrementing the stack pointer and initializing the root. The stack pointer allocation is performed with one subtraction for all the local variables to the function. Destroying such a variable normally requires no instructions because the stack pointer is restored during the normal function return sequence.

Replacing the pointer with a `root` adds three instructions for maintaining the root stack. The root's link gets the former stack head, and the stack head gets the new `root`'s address. This takes two instructions. Unlike a normal pointer variable, a `root`'s pointer component must not be uninitialized; it defaults to a NULL pointer. Initializing the pointer component of the root takes one instruction. The root destructor requires one instruction. Finally, the compilers move an unnecessary value into `r0` for the expression value. Creating, initializing and destroying a root, which should take four instructions, takes 5 under these compilers.

The figures in table 2 show the performance of root allocation and destruction. The code that yielded the measurement is provided in appendix A.

As predicted, creating roots is on the order of four times more expensive than creating pointers. It requires three machine instructions per root that are not required by a simple pointer.

Multiple roots will be created and destroyed when there are multiple local variables of type root. An unsophisticated compiler might not optimize their construction and destruction. However, the compiler could safely eliminate all but one `mov` to set the list head, and all but one `mov` to restore the old list head. Thus, the obvious code to construct and destroy n roots will use $3n$ instructions, however, optimization reduces that to $n + 2$ instructions.

Table 1: Efficiency of Trapping a Fault

Operation	Time to Complete	
	VAXstation 3520	SPARCstation 1
`getpid` (trivial) system call	55μs	29μs
`mprotect` system call	117μs	39μs
fault/trap/return	$\sim 394\mu$s	$\sim 795\mu s$
Decrement and conditional branch	0.7μs	$0.12\mu s$

4.2 Write-Protection and Faulting

How effective is circumventing the explicit bound check with write-protection and faulting? This is useful because every instruction is critical in common operations, and because conditional branches tend to disrupt pipelines. The benefit of this depends on its efficiency and on the number of instructions it replaces. The faulting implementation of the allocator described in this thesis performs two `mprotect` system calls plus additional processing on every write-fault.[7] These calls unprotect the faulting page and protect a new boundary. This memory-management requires $584\mu s$. That time equals approximately 834 decrement-compare sequences. The system permits any number of coexisting allocators, therefore the trap had to be directed to the fault handler of the correct object. This required a linked-list search and a virtual function call. In this system the time to handle and return from a trap was $630\mu s$ on the VAXstation. The fault/trap/return numbers shown in table 1 do not include the time for the two `mprotect` system calls. The SPARCstation, while much faster than the VAX for most operations, handles and returns from the faults more slowly. The discrepancy may result from the VAXstation's multiple processors. On the SPARCstation a fault/trap/return takes approximately $717\mu s$. Warning: these numbers are very rough approximations and should be used as such.

This data indicates that replacing a compare and branch with a fault is an improvement (on the VAXstation) when:

1. The two instructions really disappear, i.e., an extra instruction to force the trap is not required, and,

2. More than 900 operations occur between faults, hopefully much more than 900. If it's only an even trade the additional complexity is unjustified.

This allocator needed an extra instruction to force the fault. Therefore the number of operations to parity is closer to 1800 than 900. For blocks of size n the chunks would have to be $1800n$ bytes long for this scheme to approximately equal an explicit bounds

[7] The `mprotect` system call changes the protection status of a region of memory, e.g., it can be used to write-protect a page.

per chunk in the faulting allocator. Two factors lead to external fragmentation. The use of large chunks increases external fragmentation unless the entire chunk is used. The use of small chunks leads to fragmentation when there is insufficient space remaining in the chunk to satisfy a request. The former is ameliorated through the use of small chunks; this aggravates the latter. Static type information can be used to select the most appropriate chunk size. The write-protected page is not important since even though it consumes virtual address space it doesn't use real memory because it is never touched. If protected pages are reused by the faulting allocator then they will not take up secondary storage since there will not be any data on them to page out.

The initial version of the faulting allocator used the `valloc` library routine for its low-level allocation. This routine returns a block of memory subject to alignment constraints, for example, page-aligned memory. The use of this library function doubled the memory consumption of the allocator. Using customized low-level allocators brought the memory consumption of both allocators to a reasonable amount that, for 20 byte blocks, is significantly less than that used by the standard allocator.

4.1.3 Allocator Summary

With this copying collector is provided a customized memory allocator. One version of the allocator uses Appel's suggestion of avoiding a compare and conditional branch through virtual memory protection. This is less appropriate for C++ because of the complexity of object initialization. The protection only replaces one instruction, not two as in Appel's. When objects may be larger than one page it is much less efficient then a simple comparison.

The protected-page implementation of the allocator is roughly the same speed as the explicit-test version. Both are more than twice as fast as the standard allocator. 16k chunks are too small for the faulting allocator. The testing version was never observed to be significantly slower than the faulting one.

The standard allocator suffers from severe fragmentation when requests are in powers of two bytes. The application has no way to avoid this if it is allocating near a breakpoint. This also applies to the specialized allocators such as `valloc`. The custom allocators suffer no internal fragmentation. The use of large chunks increases external fragmentation unless the entire chunk is used. The use of small chunks leads to fragmentation when there is insufficient space remaining in the chunk to satisfy a request. The former is ameliorated through the use of small chunks; this aggravates the latter. Determination of the optimal chunk size is beyond the scope of this thesis.

The custom allocator comes with a low-level allocator that obtains memory directly from the operating system. Either version of the custom allocator is a significant improvement over the standard one, in terms of both time and space. This is unsurprising given the simple allocation strategy that copying collection permits. The faulting allocator is not fast enough to justify its added complexity. With 16k chunks it is significantly slower than the testing allocator. The following section suggests that the smallest appropriate chunk size for the faulting allocator is 64k bytes.

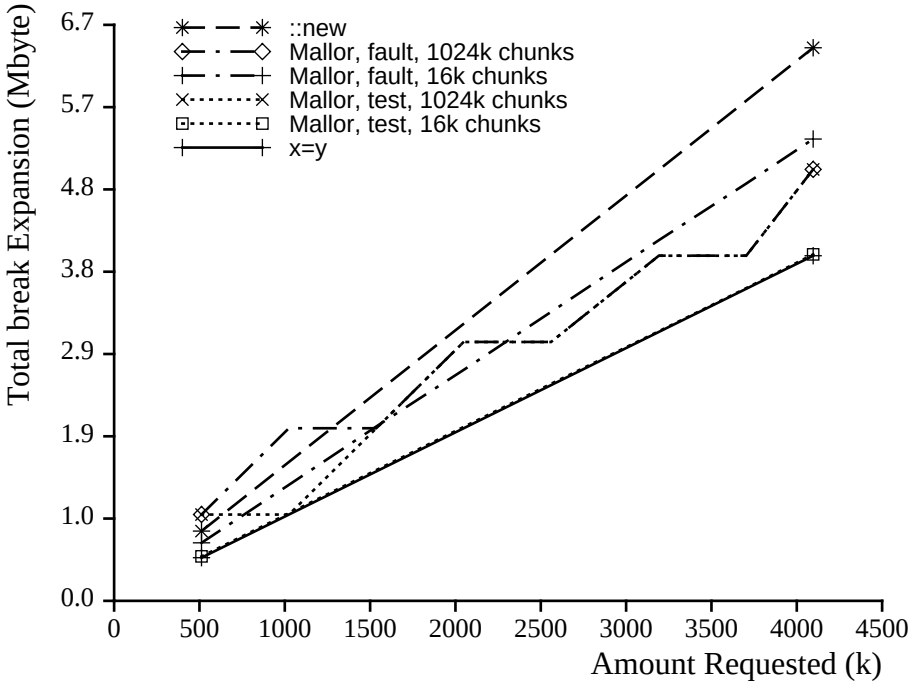

Figure 22: Total memory allocated, SPARCSTATION 1.

versions of the custom allocator. Both are roughly 2-3 times faster than the standard allocator. The faulting allocator with 16k chunks is significantly slower than the other configurations for reasons discussed in §4.2.

4.1.2 Space Efficiency

Another question is how much memory does each allocator waste? Waste, in this context, is memory obtained from the operating system but not available to the user. This can be determined by examining the program *break* before and after all the allocations. Under UNIX the *break* is the program's dynamic storage space limit. It grows toward higher addresses as the program obtains memory from the operating system. The `sbrk` system call with an argument of zero returns the current *break* without changing it. The total memory obtained from the operating system by each allocator is shown in figure 22.

In this test the standard allocator always obtained about 1.5 times as much memory as it provided. This fraction is dependent on the request size. When requests are equal to or slightly larger than a power of two the allocator is very space inefficient. When allocating objects in sizes that are powers of 2, for example, 32 byte objects, it obtains $2n$ bytes from the operating system to provide n bytes to the application. Very efficient sizes are one word smaller than a power of two such as 124 bytes. In those cases its overhead is very small.

There is no per-object space overhead in the custom allocator. The losses are due to external fragmentation, one word per chunk for making lists, and a write-protected page

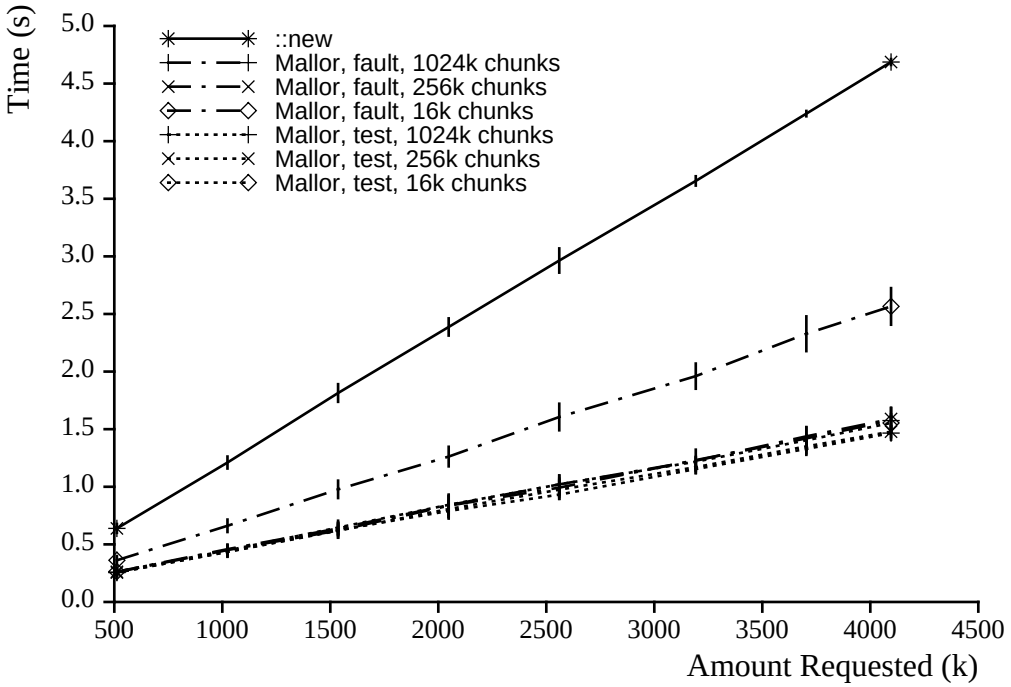

Figure 21: Time to Allocate and Touch: 20 Byte Requests, SPARCstation 1

Mallor: means the custom memory allocator
fault: the version that uses write-faulting to avoid the test
test: the version with an explicit bounds check

```
struct node {  /* sizeof node == 20 */
    static mallor * heap;
    void * operator new(size_t size) { return heap->get(size); }
    void operator delete(void * p)   { }
    char data[20];
};
```

Figure 20: The object used in allocator tests.

This code compiles to the following. (Liberties have been taken with identifier names.)

```
// This routine is mallor::get().
// r11 is used for the ``pointer'' value being generated.
// The first two instructions compute the pointer value.
// The cmpl and the bgeq check its bounds.
// Then it's stored back into the allocator state and used.
// The instructions marked with `*' are the common case.
get:     subl3   $20,free_ptr,r11   //* compute new free_ptr
         cmpl    r11,lower_limit    //* test against bound
         bgeq    success            //* if ok, save it
         pushal  this               //  allocator fault
         calls   $1,fault           //  get more space
         subl3   $20,r0,r11         //  compute new ptr
success: movl    r11,free_ptr       //* save new free_ptr
```

A faulting allocation request compiles to 4 inline VAX instructions all of which are executed. A larger number of instructions are being executed when objects are allocated than these low counts would suggest. The constructor is comparing `this` to zero before it invokes the allocator. The allocator is saving its return value in a temporary on the stack. The constructor is then comparing the return value to zero. The extra work reduces the significance of a one or two instruction difference.

The time measurements shown in the graph of figure 21 were obtained as follows. The allocators were used to allocate 20 byte objects such as shown in figure 20. After allocation one byte was initialized in each object. This seems fair because uninitialized dynamically allocated objects should be rare. The test program repeatedly allocates new objects until a fixed amount of memory has been obtained. The test was run to obtain 512k through 4M bytes. The custom allocators are parameterized by their internal chunk size. Chunks of 16k, 256k and 1M bytes were tested. Timing information was obtained with the `wait3` system call because it provides better granularity than either the `time` program or the C-shell `time` function. The times reported are user time plus system time. The vertical bars at each point show 95% confidence intervals.

The data presented in the graph of figure 21 show little difference between the two

```
/* Coded by the application programmer in node.h */
class node {
    ... // whatever required by the application
    ... // normal constructors
    ... // no destructor
    COLLECTION_MEMBERS
    virtual void copy(node * &);
};

void node::copy(node * &)
{
    ...
}
```

Figure 19: Declaring a collected `node` type.

We wish to measure the speed of the allocators by issuing many allocation requests and seeing how long they take to complete. This was done once and the times showed that the faulting allocator was much slower than the testing allocator, though still much faster than the standard one. The complication was that the faulting allocator was writing to every object while the other two were not. Since it modified the objects, their pages had to be written out to backing store before the memory could be reattached. This made the faulting test program take many times longer to complete than the testing version. After this two observations influenced the testing:

- Object are typically initialized when they are allocated.
- We expect to allocate nodes in polymorphic data structures, therefore objects will have *vptr*s that require immediate initialization.

Given these two guidelines, it appears appropriate to touch every object when benchmarking the speeds of the allocators. To accomplish this we write a byte into every object to dirty the page that contains the object. In the faulting allocator this is in addition to the write that causes a fault.

A testing allocation request compiles to 7 inline VAX instructions of which 4 are executed in the common case. This instruction count does not include object construction. The Sparc code is considerably less dense. A testing allocation request calls:

```
inline byte * mallor::get(size_t n)
{
    if ((free_ptr -= n) < lower_limit)
        free_ptr = fault() - n;
    return free_ptr;
}
```

```
/* Supplied in gc.h */
#define COLLECTION_MEMBERS(T)                                   \
private:                                                        \
    static mallor freestore;                                    \
    struct T/**/_forward {                                      \
        T *     forward;                                        \
        T/**/_forward() : forward(NULL) { }                     \
    } fa;                                                       \
    void    set_forward(void * p) { fa.forward = p; }           \
    void *  get_forward()  { return fa.forward; }               \
public:                                                         \
    void *  operator new(size_t n)                              \
                { return freestore.get(n); }                    \
    void    operator delete(void * p)   { }                     \
    static  void gc_collect()                                   \
                { freestore.setmode(mallor::collect); }         \
    static  void gc_grow()                                      \
                { freestore.setmode(mallor::grow); }            \
    static  void gc();
```

Figure 18: The necessary members of a collected type.

The tests reported here were compiled with optimization enabled on a two processor VAXstation 3520, cfront 2.0 and the Ultrix 3.0 C compiler. For comparison purposes many tests were also run on a Sun SPARCstation 1 running SunOS 4.0.3 and cfront 2.0 and are so indicated.

In analyzing the performance of individual components we examine the following operations:

- allocating an object
- creating and destroying roots, both stacked and doubly-linked
- collecting a data structure

4.1 The Allocator

We compare three different allocators for time and space: the faulting version, the testing version, and the global `::new` allocator on top of the standard libraries.

4.1.1 Allocation Speed

When comparing the time-efficiencies (rather than space-efficiencies) of the allocators, a complication arose that made it difficult to isolate the allocation overhead.

```
// Copy from every root in a stack.
// Linked list implementation of the stack.
void root_list_head_node::copy()
{
    R_node * rootp = R_node::head.root_list;

    while (rootp) {
        if (rootp->ptr)
            rootp->ptr->copy(rootp->ptr);
        rootp = rootp->link;
    }
}
```

Figure 17: Copying `node` objects from a root-stack.

1. the overloaded `new` operator to allocate memory from the current to-space
2. the static member `gc()` that is used to garbage collect all objects of the type, and
3. a virtual `copy` function to recursively copy objects and update pointers.

The overloaded `delete` operator is nonfunctional. It is implemented so that, should the application programmer accidently delete a pointer, the global `::delete` operator will not try to free the memory. When the argument to `delete` is an l-value, it can overwrite the pointer with NULL; this will reduce the amount of garbage that is copied.

Garbage collection is triggered either by the application, when it knows it is done allocating, or by the allocator when it runs out of space. The garbage collection algorithm first switches the allocator to a new space, a flip, and saves a pointer to the old space. Then from every root it traverses and copies the reachable data structure. The virtual `copy` function accomplishes the copy and updates the pointers. The function must be reimplemented for every collected `class`. It is simple to implement, generally requiring less than ten lines. Every object is copied the first time it is visited. If an already copied object is revisited a forwarding pointer indicates that fact. After the reachable data structure has been copied the from-space pointer is used to recycle that space.

This implementation of the collector is not generation-based. If Appel is correct, however, generations might be of little benefit in an assignment-based Algol-like language such as C++ [3].

4 Analysis

We have shown why this dynamic memory management organization does not impact code that does not use it, thereby satisfying the first important efficiency criterion. The other is that code that does use the collector is also efficient.

```
void node::gc()
{
    chunk * fromspace = freestore->flip();

    // for each root stack
    root_stack * stackp = root_stack_node::head;
    while (stackp) {
        stackp->copy();           // fix each root in the stack
        stackp = stackp->next;
    }

    // for each doubly-linked root list
    droot_base_node * curr = dummy_node::dummy.next;
    droot_base_node * limit = &dummy_node::dummy;
    while (curr != limit) {
        curr->copy();              // copy from the droot
        curr = curr->next;
    }

    mallor::release(fromspace);    // recycle from-space
}
```

Figure 16: Garbage collecting objects of base type `node`.

This is for the linked-list implementation of the root-stack.

3.9 Review

We have now described all the principle components of the copying collector: the allocator, tracking roots, and the collector algorithm. We now give a detailed review of the system. In this discussion the type `node` is a garbage collected type with two internal pointers as shown in figure 13.

Roots are of two types: (1) `R_node` for fast, `auto` and global roots, and (2) `DR_node` for the rare roots that must be linked into the doubly linked list. These two types of roots can be located for a collection by searching the list of node_root stacks and the list of node_root doubly-linked lists, respectively. The head of the list of stacks is a `static` member of the `R_node` class; the head of the list of doubly-linked lists is a `static` member of the `DR_node` type. The doubly-linked list of `droot`s should be empty for most applications.

The data structure's memory allocator is a static member of the `node` type. The overloaded `node::new()` routine uses this object to allocate memory.

The are three critical member functions of the collected type:

```
class dnode : public node {
    node * center;
    ...
    virtual void copy(node * &);
    virtual void copy(dnode * &);
};

void dnode::copy(node * & n)
{
    ...
    n = new dnode(*this);  // safe, implicit conversion
    ...
}

void dnode::copy(dnode * & n)
{
    ...
    n = new dnode(*this);  // No conversion needed
    ...
}
```

Figure 15: Overloaded copy functions for a derived `node` type.

Polymorphism in C++ requires a base class. The programmer defines this base class and adds to it the collection members. The collection members include the forwarding address object, the overloaded `new` operator, and the `static` free-store object. The parameterized macro `COLLECTION_MEMBERS(T)` provides all the requisite members except the copy function. This macro is shown in figure 18. The copy function must be coded by the programmer. An example of the way a collected `class` is declared is shown in figure 19.

Only one thing must be done after the type is declared: the copy routine must be defined. When the collector is in the compiler, the compiler will implement these routines but this version requires the programmer to supply it.

The copy routine has four tasks:

1. reallocate (copy) the object
2. set the forwarding pointer
3. recursively copy descendents of the object
4. update the pointer that led to the object

A sample `copy` routine was shown in figure 13.

```
// assume node is a collected type
class derived_node : public other_type, public node {
    ...
} obj;
```

Figure 14: An object such that `(node*) &obj != &obj`.

forwarding pointer is not known until runtime. The field that contains the forwarding address is of type *base* `*`, where *base* is the base `class` of the polymorphic type hierarchy. Type safety would be preserved by redefining the pointer in every derived class but this would be inefficient. Only the `copy` routine reads or writes the forwarding pointer and the `copy` routine, because it is virtual, knows the runtime type of the pointer. Therefore it may convert the forwarding pointer to and from *base* `*` safely.

3.7 A Collection

The collector is nonincremental and copying. It is invoked by the allocator when an allocation request cannot be satisfied. The collector takes the following steps:

1. make the allocator "flip" to a new space, saving the from-space pointer,
2. traverse the data structure, reallocating every object in the new space and updating pointers,
3. release the old space, and
4. resume the interrupted allocation request.

The collector "flips" the allocator with a call to the `mallor::flip()` routine. This returns a pointer to from-space. Future allocation requests are satisfied out of the new space.

After the flip the collector calls the `traverse` virtual member of each root stack. That routine performs a depth-first traversal from each root in the stack. Every object encountered is copied to the new space by its `copy` virtual function. After the entire data structure has been copied the old space is released with a call to the allocator `release()` static member function.

The routine of figure 16 is the collection routine for a `node` type. The routine shown in figure 17 copies the subtree rooted at each root in a root-stack.

3.8 Using the Collector

This section describes how the collector is realized and how it is imported into an application. We assume that the objects to be collected are polymorphic because the homomorphic case is less interesting and easier.

This collector requires that every collected type have a `copy()` virtual function. This function reallocates the object, sets a forwarding pointer, copies the descendents, and updates the pointer that led to the object. A sample copy function is shown in figure 13.

A traditional system could accomplish this by using the structure tag to index into a table of function pointers. The problem with using that scheme in C++ is the fact that a base class pointer may not actually point at a base class object. The copy must be accomplished—or the structure tag must be obtained—with a `virtual` function call.

```
class node {
    node * left, * right;
    ...
    virtual void copy(node * &);
};

void node::copy(node * & n)
{
    node * newptr;

    if (n = (node*) get_forward()) return;  // already done

    n = newptr = new node(*this);           // copy object
    set_forward(newptr);                    // set forward
    if (left) left->copy(newptr->left);     // copy children
    if (right) right->copy(newptr->right);  // copy children
}
```

Figure 13: A copy function for a `node` type.

When an object is copied the pointer that led to the object must be updated. If the object is of a derived class type then the pointer may be either a base class pointer or a derived class pointer. If it is a derived class pointer then the new address may be used directly, otherwise the new address must be converted to be a base class pointer type. This conversion, in the presense of multiple inheritance, may change the value of the pointer. Therefore it is mandatory that the copy routine of a derived class have available the actual type of the pointer that led to the object. In the `derived_node` class of figure 14 converting a pointer of type `derived_node *` to type `node *` changes the value of the pointer. This problem is solved in this collector with function overloading.

The `copy` virtual function is overloaded based on the type of the pointer that leads to the object. A derived class must have a copy function for every type of pointer that can lead to such an object. A simple derived class based on the `node` class of figure 13 is shown with its `copy` functions in figure 15.

The copy routine still requires an explicit type coercion because the type of the

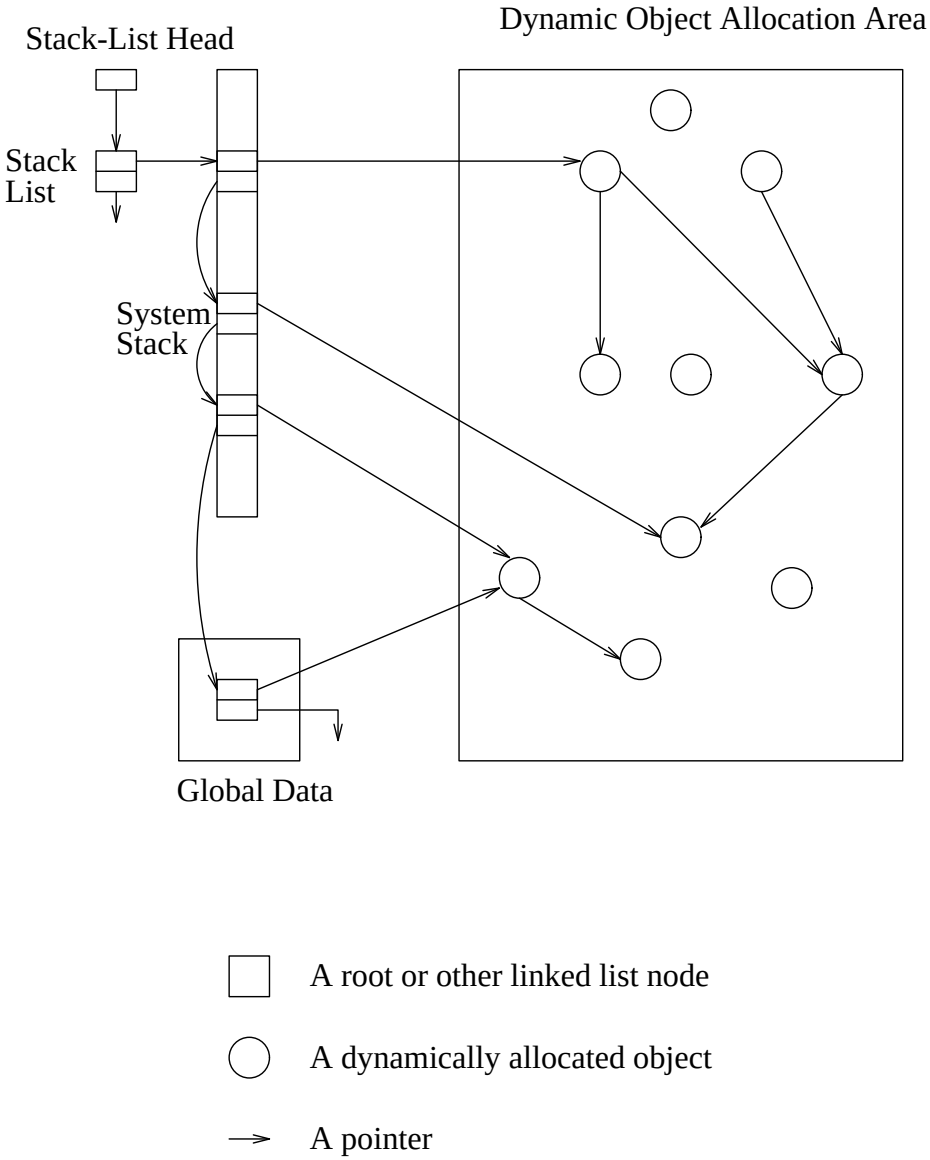

Figure 12: Runtime organization of the copying collector.

The linked-list implementation of the root-stack is shown.
No doubly-linked roots are shown.

a linked list when they're constructed. Since the collector can find the head of this list it can find all the root stacks.

This implementation, similarly to the array implementation, requires two words per root. One is the pointer the other is the list link. Since roots are two words long it is inconvenient to pass or return them in registers.

The runtime organization of this data structure is shown in figure 12.

3.5.4 Proposed Third Implementation

The array implementation has the advantage that roots are only one word so they are easily passed in registers. The problem involves growing the array. If it could be treated by the operating system the same way the system stack is treated this problem would be eliminated.

The system stack grows down from a fixed address. Virtual addresses below the stack are backed by real memory as they are required. If this can be done for the root-stack then no copy will be required. As before, the first word beyond the stack is write-protected. Upon a fault the protected page is unprotected and one or more new pages are backed with real memory. The new end of the stack is then write-protected. The stack remains contiguous and no copies are required as the array grows. This implementation is better than the linked stack implementation because these roots are only one word long so they can be passed in registers conveniently.

3.5.5 Doubly-Linked Roots

Dynamically allocated roots that are not in the data structure need not have LIFO lifetimes, therefore they cannot be tracked with a stack. Boehm's conservative collector referred to these as *foreign* roots and disallowed them [8]. This collector tracks them with a doubly linked list rather than a stack. The doubly linked list permits deletion of any list element to support non-last-in first-out (LIFO) insertion and removal.

These roots must have a distinct type. Dynamic roots to objects of type T are of type `DR_`T. They are called *droots* because they insert themselves into a doubly linked list when they are created (doubly-linked roots.) As with stacked-roots there is one list for every kind of root and the heads of the lists link themselves into a list. The collector can find the meta-list head and thus can find all the lists, therefore it can find all the pointers. The head of all the lists is a `static` member of the `DR_`T class.

Doubly-linked roots take up three words in memory. Their construction and destruction entail doubly-linked list insertion and removal.

3.6 Copying

Tracking the roots is the first difficult operation. Copying the data structure is the second. Other collectors have accomplished this by giving every structure a tag field and using that to determine how to copy the object. Our approach is essentially equivalent, except that since it is currently implemented in C++ it remains within the language.

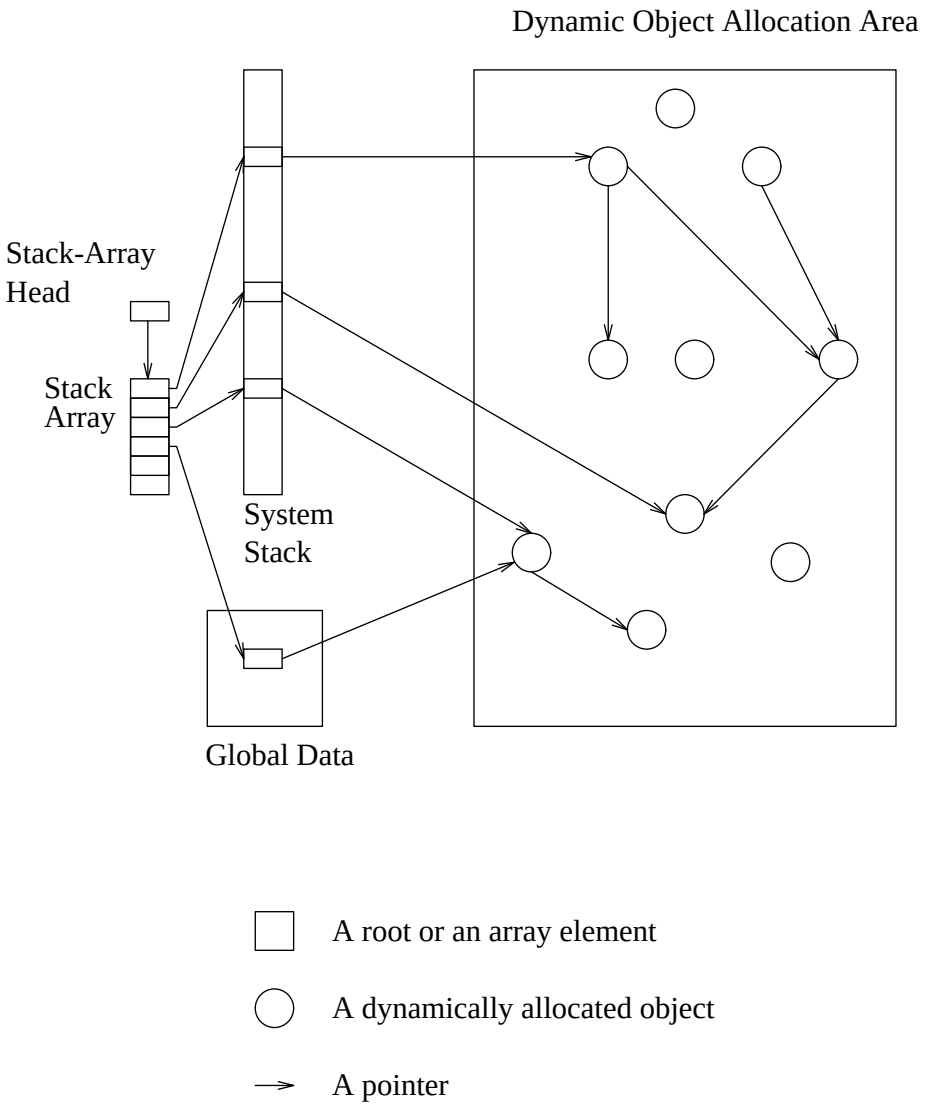

Figure 11: Runtime organization of the copying collector.

The array implementation of the root-stack is shown.

popped. The instance data of a root is the actual pointer. The array entry points at the pointer.

A stack push, which copies the pointer's address into the array cell and increments the limit, requires two instructions. A stack pop decrements the limit using one instruction. During a collection all the roots can be located because the base address and size of the stack are available to the collector.

The first page beyond the array is write-protected so that no bounds check is required on the critical *push* operation. The problem is what to do when a write-fault indicates that the stack is full.

When the array runs out of data it has to be extended. Two options are available. We can reallocate and copy the stack, or add a segment. Adding a segment makes the common push operation expensive because the array is no longer contiguous. Copying the array is also inefficient and becomes worse as the stack grows.

The disadvantage to this scheme is that either a *push* is slow because the array is discontiguous or else copies are required decreasing scalability of the system. We intend the system to be efficient with very large applications so this is a problem. A proposal in §3.5.4 shows how to implement the root stack as an array with neither of these difficulties. The advantage to the array implementation is that a root's instance data is only one pointer which may be conveniently passed and returned in a register.

Under this scheme each root requires two words, its pointer value and its array entry. The runtime organization of this data structure is shown in figure 11.

3.5.3 List Implementation of Stackable Roots

The second implementation of the root stack uses a linked list for the stack. Every root consists of its object pointer and its list link.

The stack head is a `static` member of class `R_`T. When a root is created its constructor pushes (links) its address into the stack; its destructor removes it.

A program that manipulates a polymorphic data structure may require several kinds of roots. For example functions that allocate new nodes require pointers to the correct kind of object. When the node is inserted into the data structure the pointer is converted to a base class pointer. To support this we require polymorphic roots.

There is a choice about whether the different kinds of roots share the same stack. If they were tracked with the same stack then going from root to root, i.e. traversing the stack, would require a virtual function call per list link. This would also give roots a *vptr* field slowing down root construction, a critical operation.

Instead there is a separate stack for every kind of root. The stacks are themselves linked into a list so that they can all be found from a single pointer. Advancing from stack to stack requires a virtual function call, but traversing a single stack entails only normal function calls or inline code. This way collecting is more efficient and allocating roots is more efficient because they do not contain *vptr*s .

In a polymorphic data structure with n types of roots there would be n stacks. If only one pointer type, say the base pointer type, was ever used as a root, then only one stack would be needed. All of the root stacks have stack heads that link themselves into

the fault. This field is more appropriate than any application-defined data because it's initialization is controlled by the compiler.

If initialization of a *vptr* is to be used to trigger a fault then allocation must be performed from high addresses to low ones, i.e., the write-protected page must be at a lower address then all the available pages.

3.5 Roots

A copying collector must be able to identify the roots of the data structure. Traditional collectors scan the program's stack, global data and registers looking for pointers to collected objects. This is impossible for our system because the collector cannot identify the pointers. (Tagged pointers have already been rejected.) Furthermore, this is inefficient in that the time it takes to find the roots depends not on the number of roots, but on the stack depth and the amount of global data. This leads to the case of a very small collected data structure requiring a comparatively long time to collect because other program components have a lot of data and a very deep call stack. For example, an operating system might have more than 250k bytes of global data. It could not garbage collect even a small data structure efficiently if a collection required scanning all that data. This collector uses auxiliary data structures to track roots.

Roots of a data structure are a parameterized type; for nodes of some type T, roots are of type `R_`T. They are currently generated with `#define` macros because no C++ compiler that implements Stroustrup's *template* mechanism [17] is available. Their parameter is the type of object they reference. This allows them to act as "smart pointers" [32]. A root is a class object that is convertible to a normal pointer type. It can be dereferenced just as a normal pointer. It has a constructor and a destructor that are used to track its lifetime.

The roots of the collection are located using two data structures. Most roots are either global or `static` variables, or `auto` variables. These roots can be tracked with a stack. Other roots, such as those contained in other dynamic data structures, cannot be tracked using a stack. Their addresses are kept in a doubly linked list.

3.5.1 Stackable Roots

Every time a global root is created, or a root is allocated on the runtime stack, its address is pushed onto a stack. When the root is destroyed its address is popped from the root stack. Global objects are pushed when the program starts to run; `auto` roots' addresses are pushed as the roots are constructed. At any moment all the global and local roots in a program can be found by examining the stack. This data structure was implemented two ways and a third is proposed.

3.5.2 Array Implementation of Stackable Roots

The first implementation uses an array for the stack. The cells of an array point at the roots. An index associated with the array indicates the current stack top. When a root is created its address is pushed onto the stack. When a root is destroyed the stack is

allocator while smaller objects used the faulting version.

The class definition of the testing allocator is shown in figure 10. The faulting implementation lacks the `lower_limit` bound since it does not perform an explicit test. It is a derived `class` of `protection_client`.

Section 4.1.1 compares the two implementations and explains one of the difficulties in benchmarking the faulting allocator against the testing and standard allocators.

```
class mallor {              // A memory allocator class
public:
    enum heapmode {grow, collect};
private:
    int      chunk_size;    // for low-level allocation requests
    chunk *  curr_chunk;    // ``to-space'' chunk list
    byte *   free_ptr;      // current allocation point
    byte *   lower_limit;   // lower-limit of current space
    heapmode mode;          // to collect or to grow on fault
    byte *   fault();       // handle ``insufficient space''
    void     (*gc)();       // pointer to collection routine

public:
    byte *   get(int n);        // for an allocation request
    chunk *  flip();            // switch spaces, return old one
    void     release(chunk *);  // deallocate a space
    heapmode setmode(heapmode); // ``collect'' versus ``grow''

    mallor(void (*)(), size_t, heapmode);
    ~mallor();
};
```

Figure 10: The memory allocator class, *testing* version.

3.4.2 Proposed Third Implementation

The version of the allocator that avoids bound checking with write-protection adds an instruction to force the fault. This might be circumventable if the implementation were in the compiler.

The overhead of the extra assignment can be eliminated if a required initialization can be used. This can be realized if the object is laid-out with a known, compiler-initialized field at an end of the object. The high-offsets of an object contain user-defined data. The low offsets also contain user data, but in some cases a satisfactory field can be placed in the first word of the object. The *vptr*s are under the control of the compiler. When it can put a *vptr* in the first word of the object it can use initialization of that field to cause

3.4.1 Two Implementations

The allocator has two implementations that differ in how they know when there is insufficient space in the current chunk to satisfy the current allocation request. One performs an explicit comparison during each allocation request. The other uses virtual memory protection to avoid the explicit test as suggested by Appel [2]. The last page of the current chunk is write-protected. Every object is touched as it is allocated. When an object crosses the write-protection boundary this will cause a protection violation. The operating system sends a signal to the program that is trapped by the `protected_obj` class and routed to the faulting allocator. The allocator either collects or allocates a new chunk and resumes the interrupted request. We refer to the two versions of the allocator as the *faulting* version and the *testing* version.

Both versions maintain a linked list of chunks, i.e., a *space*, that constitutes *to-space*. They also keep a pointer into the lead chunk that demarks the current allocation point. The chunk at the head of the list is the only one used to satisfy allocation requests. The other chunks are never again examined by the allocator until they are returned to the low-level allocator.

The low-level allocator quickly allocates large blocks of memory. It allocates blocks in sizes of powers of two between 2^{10} and 2^{24} bytes. The version for the faulting allocator always returns page-aligned blocks. The low-level allocator obtains its memory from the `sbrk` system call. When chunks are deallocated they are linked into lists and used for future allocation requests. This allocator could also sort chunks by address and shrink the address space when possible. This would reduce secondary storage use.

Using write-protection to circumvent the test is more difficult in C++ programs than it is in the simple allocator described by Appel. In that case allocation and initialization occur together. The allocator can attempt to cause a write-fault during normal initialization. This hides the allocation overhead. In C++ allocation and initialization are distinct operations. Initialization is almost entirely under the control of the application. Initialization of members and base-class sub-objects is generally performed by constructors. Constructors may have side-effects. If the fault occurs after one or more constructors have executed it may be impossible to undo the effects of those constructors. For this reason, a write to the object must cause the fault before any sub-object constructors are invoked. Unfortunately, when an object crosses the boundary into the write-protected memory, the part of the object in the protected region may be initialized at the application's discretion with a constructor, or not initialized at all. Therefore, the normal initialization sequence cannot be used to trigger the fault; an extra instruction must be inserted. Thus we are replacing two instructions with one. The instruction being eliminated is a conditional branch. These instructions disrupt pipelines on some architectures so the trade-off may still be worthwhile in some cases. A way to eliminate the extra instruction is described in section 3.4.2. Objects that are larger than one page cannot benefit from bypassing the explicit test with write-protection because every page must be touched. This requires two or more instructions making it no less expensive than an explicit test. The lack of dynamic arrays in C++ makes it easily determinable at compile time what objects are larger than a page. These objects can use the testing

Through that pointer the client's fault handler is invoked upon a write-fault.

This is a general way of allowing multiple protected-objects to be conveniently used within a program. This is useful because they are a powerful tool.

3.4 The Memory Allocator

The strategy used in this allocator is based on the block allocation scheme described in §1.1.5, however, it uses a variation that supports discontiguous spaces. A *chunk* is a large block with which the allocator satisfies allocation requests. A *space* is a linked-list of chunks. The allocator reserves the first word (pointer-sized region) of each chunk for creating the linked list. The chunk from which the allocator is currently satisfying allocation requests heads the list. When a chunk is exhausted the allocator may either begin a collection or it may obtain another chunk from the low-level allocator, link it into the list, and satisfy the current allocation request out of the new chunk.

This permits the free-store to shrink and grow dynamically to accommodate the changing needs of the data structure. After a collection the free-store is exactly the necessary size, subject to the granularity of the chunk size. The allocator may permit the application to decide whether the allocator should expand or collect, the allocator may base its decision on the anticipated needs of the application, or it may decide based on the current paging or swapping load of the virtual memory system.

The allocator is encapsulated in a C++ `class` called a `mallor`. The name `mallor` was selected because it suggests *memory allocator*. The common term `malloc` was not used to preclude confusion with the ANSI C [1] library routine of that name. In this discussion the noun *an allocator* denotes an instance of `class mallor`.

Individual objects can not be individually freed therefore headers and footers are not required. An entire space is deallocated by returning its chunks to the low-level allocator.

The allocator is initialized with a function pointer to the collection routine. When it runs out of space it can either garbage collect by calling through this pointer or continue allocating from a new chunk. To continue allocating it allocates a new chunk and links it to the beginning of to-space. It resets the pointers to indicate the current allocation point and, in the testing version, the lower bound. The first word of every chunk is reserved for constructing this linked list.

By calling through its `gc` function pointer the allocator invokes the garbage collector. The collector causes the allocator to begin using a new allocation space with a call to the allocator `flip()` member. This function returns to the collector a pointer to the old space, that is, a pointer to *from-space*. The collector garbage collects the objects into the new space by reallocating and copying them. Then it releases the old memory with the `static` allocator routine `release()`. Finally, the collector returns and the allocator resumes the interrupted allocation request.

An allocator is a `static` member of the base class of the collected data structure. The overloaded `new` operator of the class obtains memory from the `mallor`.

3.3 Virtual Memory Protection

Several recent dynamic memory management systems have used standard virtual memory hardware to increase efficiency. Appel, Ellis and Li disallow accesses to pages of memory as a form of medium-grained synchronization between the concurrent mutator and collector [4]. Appel uses write-protection to avoid an explicit bound check in the allocator. His allocator write-protects a page beyond the current allocation area and allocates until the initialization of an object produces a write-fault [3]. This avoids a compare and conditional branch in the allocation-request code.

A `protected_obj` is a C++ `class` that encapsulates this functionality. Such an object contains a sequence of pages, the last (or first) of which is write-protected. A client of the class provides a fault-handler that is invoked when a write-fault occurs. This is used in one version of the memory allocator to avoid an explicit bound check.

A write-protected page of memory may be read normally. An attempt to write to the page results in a write-fault. Under UNIX this interrupts the program with a signal. This signal causes the program to terminate unless the signal is trapped. The `class` that encapsulates this data structure traps the signal. The `protected_obj` module may have many clients because there may be any number of protected objects in existence at a given moment. The address of the protection violation is compared with the addresses of all the write-protected pages. The fault-handler associated with the object on which the fault occurred is invoked. Figure 9 shows the public interface of the `protected_obj` class.

```
class protection_client {
protected:
    virtual void fault_handler() = 0;
};

class protected_obj {
private:
    ...
public:
    protected_obj(protection_client * cp, int npages);
    ~protected_obj();

    char * getbase();     // lowest address of the segment
    char * getlimit();    // lowest write-protected address
};
```

Figure 9: The public interface of the `protected_obj` class.

An object that wants to use a `protected_obj` makes itself a publicly derived `class` from `protection_client`. That gives it a `fault_handler` virtual function that it must implement. The client passes its address (`this`) to the `protected_obj` constructor.

research: the creation of a useful system for comparing collection techniques. In this sense the programmer who codes a collection algorithm is as much a client of the system as the application programmer who uses one of the provided collectors. An indication of the success of this project, and the usefulness of C++ toward these goals, will be the simplicity of the interface. This goal is essentially vacuously satisfied. Conservative collection could be implemented with practically no impact on the programmer's style. These two collection paradigms, copying and conservative, would be a basis for interesting comparisons in efficiency, convenience, and virtual memory performance.

3.2 Overview

This thesis describes the implementation of an efficient allocator with a copying collector for C++. It has the following features:

- The system is modular, encapsulated and very flexible. The application may communicate with the memory manager to configure it appropriately. Any number of collectors and allocators can exist concurrently in an application on disjoint data structures.
- The allocator is fast; it compiles inline to 7 VAX instructions of which 4 are executed in the common case. It supports discontiguous chunks.
- The collector is a copying collector that collects and compacts in one pass.
- This implementation is entirely compiler external and independent.
- The components are small, simple and efficient. In many cases it is more efficient than a standard manual memory allocator such as the global `new` and `delete` operators or *malloc* and *free*.

This system is implemented as a memory allocator class called `mallor` and a set of parameterized macros for defining garbage collected types. The macros generate the requisite members of the collected type and `class` types for roots of the data structure. Two versions of the allocator are presented and compared. One avoids bounds-checking by write-protecting the last page of its available memory, touching every object it allocates, and intercepting the write-fault signal. The other version simply tests the bounds of the segment on every allocation.

If a C++ compiler supporting *templates* were available it would not be necessary to use the preprocessor to define parameterized types [33,17].

The collector uses a non-incremental stop-and-copy algorithm. The system solves the critical copying collector operations: identifying roots and deeply copying the data structure. It takes full advantage of C++ features such as virtual functions to be general, simple and safe.

This version of the collector requires some assistance from the client programmer: the programmer must give every collected type a member function that copies the object, and recursively, its descendents. Were the collector implemented in the compiler this requirement would disappear.

unused. Large chunks reduce the overhead from expanding and freeing the space. They also reduce external fragmentation which depends more on the number of chunks rather than their size.

Conservative collection may be rendered less efficient by discontiguous chunks. When a conservative collector identifies a quantity that may be a pointer it determines if the referent would be a dynamically allocated object. Pointers to non-dynamic objects need not be considered because the objects themselves will be scanned. When spaces may consist of discontiguous chunks identifying pointers to dynamic objects is more difficult. It may be accomplished in log time by searching a tree. It may be done by hashing. An argument could be made that in a process with contiguous global data it could be determined by elimination, but this argument ceases to be valid when foreign dynamic memory managers are permitted.

3.1.4 Other Requirements

A perceived advantage of the C programming language is transparency [29]. Tool libraries that are expensive in a non-obvious way make it hard to analyze the performance of a segment of a program. The important criterion is this: are program fragments that do not use the feature penalized? In the context of garbage collection adding functionality to the global `new` and `delete` operators would penalize all code, not just operations on collected types. This would not be a philosophically consistent implementation. It is mandatory that code that does not need the automatic reclamation not pay any penalty.

The notation for declaring and manipulating collected types should be consistent with the normal syntax. It will not be entirely possible to achieve this while the collector is outside the compiler. Collected types should be declared and used like uncollected types; they will be allocated with `new`, and other than not requiring deletion, should look like other types of objects.

Pointers to objects of the collected type should behave like normal pointers: pointer arithmetic, dereferencing operators, etc., however, the collector must have information about these pointers; it must be able to traverse all of the pointers from a set of unambiguously identifiable roots. Operator overloading makes it possible to use the standard `*` and `->` operators on user-defined types. Unfortunately, it will not be possible to declare a pointer to a tracked type `T` as an object of type `T *`. These pointers must be of a `class` type so that they can have overloaded operators and construction and destruction code. Our requirement of remaining within the language dictates that the programming style used will need to make minor accommodations. Such a pointer will require a type along the lines of `T_ptr`.

An overriding goal of this project is that the garbage collection interface should be simple to use. There should be few complexities exposed to the client programmer. If there are many exceptions that accompany the system it will not be a useful programming tool.

Along with ease of use there is the desire to compare algorithms. Therefore it should also be easy to code garbage collection algorithms to conform to the interface specified herein. Accomplishing this will directly contribute to one of the prime goals of this

3. It reduces compiler complexity.

4. It allows communication between the application and the collector to occur through the existing language. This will permit the flexibility we seek without requiring redefinition of the language.

5. Writing the collector as an application makes it easier for researchers and developers to write or modify a collector. It no longer requires a source license for the compiler.

6. It makes dissemination of the collector easier.

On the other hand there are compelling reasons to keep it in the compiler. The compiler has access to type information. The compiler can generate code that the client programmer would otherwise have to write. It can perform optimizations that the programmer cannot express in the language (this may well be a shortcoming of the language.) It can affect code that the application does not directly generate. In C++ *vptr*s are an example of such code. This project is an exploration of copying collection in the C++ implementation language. This version is outside of the compiler. If there is benefit to be derived from moving it into the compiler, and there is, then this research will identify the necessary functionality and propose an organization. We do not suggest a syntax for incorporating it into the language. That is beyond the scope of this thesis.

This research is predicated upon the assumption that no single collector is always the right one. However, one must be implemented initially. There is a choice of conservative, non-generational copying, or generational collection. The collector implemented for this thesis is a non-generational copying collector. While conservative collection would be easy to implement copying collectors are easier to encapsulate because they do not need private information to coexist with other dynamic memory allocators, and copying collection is appropriate for large virtual and real memories. The collector is not generational because C++ is assignment-based and tracking roots of the young generations could be expensive.

3.1.3 Discontiguous Chunks

The ability to handle discontiguous chunks gives the memory manager the flexibility to adjust to changing requirements or resources more quickly than an allocator that operates with immutable spaces. When sufficient real memory and paging throughput are available the collector can defer collection and continue allocating. This is advantageous for a copying collector because this gives objects more time in which to die. When many objects die the system is more efficient. On the other hand, when a system is paging heavily the allocator can force a collection in order to compact objects, deallocate virtual address space, and free frames of memory.

When an allocator grows its allocation space incrementally, immediately after a collection the free-store is exactly the right size subject to the granularity of chunks. Small chunks decrease internal fragmentation in chunks. For example, with megabyte chunks the average unallocated memory immediately after a collection is half of a megabyte regardless of the size of the data structure. With 16k chunks only 8k is expected to be

the philosophy of C++ because one module should not affect another unless they communicate. This resembles generation-based collection except that nodes are partitioned according to their type rather than their age.

Modules should be able to select appropriate parameters for their allocator and optional collector. Selecting the allocator's chunk size based on the application's allocation pattern can reduce external fragmentation.[6] For example, when requests to an allocator are homogeneous, chunks can be allocated in even multiples of the size of a request.

LISP systems formed the foundation for research into dynamic memory management. A LISP interpreter would work with a fixed-size free-store. It would allocate until it ran out of space, and then collect. That model is too simple for an object-oriented program. In the presence of copying collection, providing too much memory to a data structure can lead to bad locality, excessive paging and poor performance. Providing too little memory leads to excessive copying and worse performance. Of course, a more sophisticated model such as the one presented in this thesis can trivially emulate the simple one.

Who should decide when to collect? In many cases the application can best decide. A compiler performs much of its work in discrete steps. For example, it has actions to perform for every function of a file, for every block, for every statement, and for every expression. One compiler known to the author managed its dynamic storage to take advantage of this property. While compiling each statement it would allocate storage. After it was done with the statement it would free much of the associated storage in a single constant-time operation; it recycled the entire space exactly the way a copying collector does. This functionality is useful and easy to provide if the model is general and the interface is safe.

3.1.2 The Compiler

Dynamic memory management and garbage collection has traditionally been implemented in the compiler. This is sensible because the compiler often has better access to type information, for example, tags. LISP [25] and Smalltalk-80 [20] systems always have a garbage collector. Modula-3 [11] provides optional garbage collection. Most compilers have provided only a single reclamation algorithm, not a choice. This has forced all applications and all modules of one application to use the same collector and allocator. This is too restrictive since the choice of the proper memory manager or parameters depend on the characteristics of application. Indeed many programs may be best served if different collection algorithms can be applied to different data structures.

It is possible to provide the flexibility that we seek in the compiler. Multiple collectors can be supplied, either in the compiler or in libraries. There are, however, other advantages—especially for a researcher—in moving the collector out of the compiler.

1. Keeping the collector outside of the compiler facilitates benchmark comparisons of collectors in the absence of compiler modifications.

2. It keeps changes to the collector from requiring recompilation of the compiler.

[6] Allocators for copying collectors do not suffer from internal fragmentation so external fragmentation is the most important source of wasted memory.

3 A Copying Collector for C++

3.1 Motivation

Managing dynamic memory is difficult; Meyer has called garbage collection a critical feature of an object-oriented programming environment [26] because of the usefulness of the feature. C++ is already a complicated language even without the added complexity of efficiently deallocating dynamic objects. Providing convenient, efficient garbage collection in C++ makes the language easier and more convenient to use. Copying collection has the added advantage of compacting objects in memory after only one pass. The research reported in this thesis had several goals, the most important of which is to add convenient, efficient, automatic dynamic storage reclamation to C++. Other goals were:

- to implement an appropriate model for dynamic memory management in object-oriented imperative programming languages,
- to develop a platform for research in collection techniques and algorithms, in particular virtual memory issues, and
- to identify ways C++ could better support these activities.

3.1.1 The Model

Top-down design and stepwise refinement leads to monolithic programs in which an identifiable top level decomposes into more specific modules that are themselves further refined. Under this model all of the modules are contributing toward the end result, for example, a working LISP interpreter. The modules are specifically designed for the application. Programs developed under this paradigm have typically used a global allocator (and global collector, when present) for dynamic object allocation throughout the code.

By contrast, in the object-oriented universe, modules are not designed toward a common goal. They are designed to be reusable: to be useful and efficient in a wide variety of applications. In this model it is not reasonable for modules with disjoint dynamic data structures to share a dynamic memory manager. Some components of a program may benefit greatly from garbage collection, other parts of the program will not. The parts that will not must not be penalized for coexisting peacefully with a module that does. For example, reducing the range of all integers by using a tag bit would be unacceptable. Encapsulating the dynamic memory management functionality makes it easier to support the concurrent use of copying and manual reclamation within a program on different data structures. Since the dynamic memory systems are not global and do not communicate, they do not interfere with each other.

In a object-oriented system, there is more opportunity to tune the dynamic memory manager to improve locality and efficiency. For example, a program may generate two data structures at different rates. Segregating their storage allows the intensive one to be collected without copying nodes of the other data structure. This is consistent with

2.8 Discussion

Modern collectors are of two main varieties: conservative or copying. An important subclass of copying is generational collectors. Conservative collectors are easy to implement in the absence of compiler support and require only minimal operating system support. While they may not reclaim as much garbage as a copying collector, the discrepancy is probably quite small in most cases. Conservative collectors cannot update pointers, therefore they cannot compact memory. Since they are based on mark-and-sweep their work is proportional to the total memory allocated.

When a conservative collector identifies a quantity that may be a pointer to a dynamic object it scans the object to find other potential pointers. This requires that the collector know the size of the object. Therefore a system with such a collector cannot have a separate, uncooperative allocator. This makes conservative collection less appropriate for object-oriented systems in which a module's implementation details are unavailable to other modules.

Copying collectors are efficient but not without limitations. Their advantages include the fact that their work is proportional to the amount of living objects rather than the total number of allocated objects. They interact well with very simple, fast memory allocators. They compact memory thus improving locality and virtual memory performance. Their disadvantage is that copying collectors require more information than conservative collectors. They must be able to unambiguously identify roots of the data structure and internal pointers of objects. Their peak memory requirements are twice the size of the usable free-store.

Copying collectors are appropriate for systems with large virtual and real memories. Conservative collection may be more appropriate for systems with very limited amounts of real memory or backing store, since all of the free-store is usable at any given time.

An important technique in modern collectors is the use of virtual memory protection to avoid bounds checks. Any frequent operation that requires a pointer bounds check is a good candidate for using protection to avoid the need for an explicit check. It is used in Appel's collector to determine when the allocator has insufficient memory to satisfy a request. It is used in the Appel/Ellis/Li collector to detect references to unscanned objects. It is used in two implementations of two data structures in the collector that is presented in this thesis.

Generational collectors can further improve the efficiency of copying collectors for appropriate applications. These collectors must identify pointers from old objects to young ones. These pointers are roots of a collection of the young generation. In languages that rarely use assignment there will be few such pointers because they can only be created by assignments. LISP is the typical example of such a language. Generational collectors differ in how they identify these pointers. Most of the existing generational collectors rely on there being a minimal number of such assignments to remain efficient. In an assignment-based Algol-like language, Appel suggests the overhead of tracking such pointers would offset the benefits of generational collection [3].

ML [41]. A linked-list of roots similar to this one is used in the collector described later in the paper.

As one last noteworthy item, the collector requires that old, living objects be at a fixed end of the space. However, after a major collection they have been copied to the middle of the space. The collector uses a block copy to move them down to the end where it requires them. This is an additional copy of all the living objects. After the copy all of the pointers to the objects must be updated.

This collector is simple and appears efficient. It uses virtual memory protection to avoid bound checks. It does not support allocation from discontiguous chunks. It is appealing for its simplicity, however, it is specialized for its target language. The object-initialization code requires that the entire object be initialized when the object is allocated. The collector's scheme for tracking pointers from old-objects to young ones is efficient only provided assignments are rare.

2.7 Boehm/Weiser

Boehm and Weiser describe a conservative garbage collector (cf. §1.2.6) for use with statically type-checked languages like Pascal and C without compiler assistance [8]. They rejected tagged or limited integers or incompatibility with the standard libraries. They wanted to design a collector that would not penalize programs that didn't use it. Their goals are quite similar to those of the author's project. The primary difference is their use of conservative collection instead of copying collection. This is consistent because copying collection without compiler assistance would be impossible or prohibitively inconvenient for the application programmer in languages like C and Pascal. For reasons discussed in §2.8 this is less appropriate than other types of garbage collection for object-oriented imperative programming languages.

The free lists for small objects are organized as linked lists of blocks, so allocation takes four or five machine instructions including the test for an empty list. When the list is empty a 4k block is obtained from the low-level allocator. This allocation strategy allows the compiler or mutator to explicitly deallocate objects that are known to be inaccessible.

The low-level allocator uses 4k chunks that may be discontiguous. The sweep routine coalesces adjacent free blocks and identifies and returns to the low-level allocator free 4k chunks. Their allocation strategy allows them to identify a pointer-sized quantity that points at the beginning of an object, even though the free-store memory may be discontinuous. Handling pointers into the middle of objects is more difficult requiring an expensive hash.

The collector cannot tolerate roots contained in dynamically allocated objects from other free-stores.[5]

[5] See [8] page 812

very few VAX assembly instructions. Virtual memory protection is used to avoid explicit bound checks during allocation. When a new object is initialized, if a write-fault occurs, then the allocator is out of space and must garbage collect. This relys on the fact that initialization of objects in functional languages is comparatively straightforward; there are no uninitialized fields. In simple cases this scheme leads to very little (or no) allocation overhead because the only required instructions, the initialization of the object, would be required anyway.

Like all copying collectors his must differentiate between pointers and integers. The three schemes Appel identifies for accomplishing this in statically typed languages are:

1. allocating integers dynamically and from a distinct region of memory
2. tagging records with a format
3. obtaining a map from the compiler

Segregating types by address would require replacing all integers with pointers to integers allocated in a specific region. This would slow down integer arithmetic. In this collector he tags records with a value that identifies the type, and therefore the structure, of the object. By comparison, traditional mark-and-sweep LISP collectors frequently have used tagged pointers, and conservative collectors use no type information whatsoever. The collector described later in this paper uses a combination of dynamic and static type information.

Appel suggests that generational collectors may be most efficient when they use exactly two generations. The goal of a generational collector is to let objects die before it becomes necessary to copy them. A collector with only two generations maximizes the amount of memory that can be allocated to the youngest generation, therefore a long time can pass before that generation fills up and must be collected. This increases the probability that most objects in the generation will have had time to die.

This paper calls a collection of only the youngest generation a *minor* collection and a collection of both generations *major*. The roots of minor collections, just as in other generational collectors (e.g., Hewitt, Moon, Ungar) are the registers, the stack, global data, and pointers from old objects to young objects. Roots in old objects are the hardest ones to identify and maintain. Appel's collector requires that the mutator, through the compiler, maintain a linked list of old-objects that may be roots. An assignment into an old object causes the object's address to be inserted into a linked list of objects. At collection time the collector traverses the linked list looking for pointers into the young generation. Pointers to young objects can be identified by their value because the allocator uses contiguous chunks. Space for the linked list of roots is allocated out of the same free-store as other dynamic objects. Objects may be inserted into this list multiple times. The collector identifies duplications the same way other duplicate roots are identified: by the forwarding pointer left behind when an object is scavenged. One inefficiency with this scheme is that objects are inserted even if they do not reference a young object, e.g., if the assignment was of an old-space pointer. The ameliorating factor is that these assignments are comparatively rare in functional languages such as

2.5 Appel, Ellis, Li

Appel, Ellis and Li describe a realtime, concurrent garbage collector implemented on a DEC Firefly multiprocessor [4]. The algorithm allows the mutators to run concurrently with the collector. Synchronization is medium-grained and accomplished with virtual memory hardware.

Their collector is based on the Baker algorithm. It partitions memory into to-space and from-space; to-space is partitioned into two regions: the region from which new objects are allocated and the copy region. The copy region is partitioned into scanned objects and unscanned objects. The unscanned objects contain the only pointers into from-space that may exist. This structure is identical to that of the Baker algorithm (§2.1).

The collector sets the virtual memory protection of the pages of unscanned objects to no-access. The program traps when the mutator tries to read or write an object on an inaccessible page. The collector handles the trap and scans the faulting page. It scavenges the objects referenced by pointers on the page, leaves forwarding pointers, and updates the pointers on the page. Then it unprotects the page and resumes the mutator. When the mutator resumes the page contains only to-space pointers.

The collector also executes concurrently with the mutator. It scans unscanned pages and scavenges any referenced objects. It unprotects each page after scanning it. The more pages the collector can scan the fewer page traps the mutator will cause.

The collector performs a `flip` when it has insufficient free memory to satisfy an allocation request. For a flip, the collector stops all the mutator threads and scans all the unscanned to-space pages. Then it exchanges the roles of the spaces and reinitializes the to-space boundary pointers: `new`, `scanned` and `unscanned`. Then it scavenges the root objects and resumes the mutator threads.

This algorithm is incremental because pages are scanned when the mutator references them (or sooner). When to-space runs out of memory the amount of work that the collector must perform is unpredictable. Additionally an access of a from-space object may result in very little work or in a lot of work. The amortized cost of collection is still small, however, and according to the authors, the algorithm is efficient. The performance measurements of the Boyer benchmark from Gabriel [19] show a garbage collection overhead of 13%.

2.6 Appel

Appel describes a generational garbage collector suitable for use in functional language systems under UNIX [3]. The intent of his paper is to explain the major problems inherent in generational collectors and to offer efficient solutions. He identifies and incorporates UNIX features such as the memory layout of a process's address space and the virtual memory system calls.

In an earlier paper Appel argues that because copying collectors never deallocate individual objects garbage collection can be faster than stack allocation [2]. Therefore he stresses the importance of very fast allocation for improving efficiency of copying collectors. This collector uses a simple allocation strategy that can be hand coded in

2.4 Generation Scavenging

Generation scavenging is a memory reclamation algorithm designed by Ungar [36,38,37]. Like the two algorithms considered earlier in this section, Generation Scavenging separates young objects from old objects. Unlike Lieberman's and Moon's algorithms Generation Scavenging is not incremental.

Initially in Generation Scavenging all objects are young and they live in a single space. As that space becomes full it is scavenged. The number of scavenges that an object survives is recorded in the object. After an object survives some number of scavenges it is tenured to the next generation. In an early description of the algorithm [38] the number of scavenges that an object needed to survive was fixed. Later work by Ungar and Jackson discusses ways of using feedback to influence the tenure threshold [37].

Since the number of generations that an object survives must be counted the algorithm requires a counter per object. It also requires a single mark bit per object for marking during scavenges.

When a scavenge is performed all references to objects involved in the scavenge must be located, including back-pointers. In Lieberman's algorithm this was done with per-generation reference tables. In Moon's algorithm this was done with bitmaps of pages of objects. In Ungar's algorithm objects that have been tenured are distinguished based on whether or not they refer to younger objects. Objects that do refer to younger objects are put into a special set called the *remembered* set. This set is implemented as an array of object pointers. When a generation is scavenged, all the younger generations, the machine registers, the stack and all the remembered sets are the roots.

The Generation Scavenging algorithm partitions each generation into three spaces: *NewSpace*, *PastSurvivorSpace*, and *FutureSurvivorSpace*. These spaces serve as areas for new object allocation and for copying during scavenges.

After a scavenging pass if objects were tenured the remembered set of a generation may have grown. If it has, the new part needs to be scavenged. This may cause objects to be added to FutureSurvivorSpace. If any objects are added they need to be scavenged and that may cause the remembered set to grow again.

Generation Scavenging very rarely scavenges the oldest generation; doing so is very expensive. Therefore objects that live a long time and then die are not detected. When it happens, this results in wasted virtual address space, possibly wasted memory, and fragmentation. If that were all that happened, it might not be significant because the amount of memory involved is small. However, these dead objects may still contain references to other objects. Though invalid, these references will keep the other objects alive past when they should be reclaimed. Therefore the algorithm suffers when tenured objects die. To prevent this, Ungar inserts intermediate generations between the youngest (undergraduate) generation and the eldest (full professor) generation. By the time an object reaches full-professorhood it has lived a long time and is unlikely to die. That heuristic is not perfect and memory does get lost. Therefore, a full, compacting mark and sweep garbage collection is performed off-line about once a day.

may be in a condemned state and in the process of being scavenged. The scans move through the data spaces like waves from older regions to younger ones.

The collector can coalesce older regions when the number of objects between them shrinks to an amount that is appropriate for a single generation. The sizes of regions can be adjusted. Users can indicate the expected lifetimes of objects.

Every value fetched from an old object must be checked to see if it points at an assignment table; without hardware support this is expensive. The assignment tables are updated on every store into an old object, not just on stores of pointers to young objects. Thus the tables can become large giving an exaggerated root set for the collection [3].

2.3 Ephemeral Garbage Collector

The ephemeral garbage collector described by Moon in [28] is an incremental copying garbage collector based on the Baker algorithm. It segregates objects according to the expected longevity to concentrate effort where it is likely to be of most benefit. The collector works very closely with virtual memory and related dedicated hardware of the Symbolics 3600 LISP system to perform collection-related processing quickly and concurrently with the mutator.

This collector identifies three categories of objects. *Ephemeral* objects will probably become garbage shortly after they're created. The collector is designed to collect ephemeral objects efficiently. *Dynamic* objects will probably become garbage sometime in the future. Ephemeral objects that survive a certain number of collections are promoted to dynamic status. *Static* objects are not expected to become garbage. They are never collected except as the result of an explicit, slow command to "do a full garbage collection." Examples of static objects include compiled functions and internal LISP data structures such as hash tables.

This collector is incremental so that interactive response is not degraded by long pauses. The Baker algorithm identifies pointers to old-space objects in software and substitutes the forwarding pointer. Moon's collector uses hardware to implement a *barrier*. On the Symbolics 3600 pointers and integers are differentiated by tags. When the mutator loads a pointer from memory, the barrier hardware checks to see if it points at an old-space page. If so, the appropriate forwarding pointer is substituted.

The roots of a collection include all of the static objects. The normal LISP distribution includes approximately 4M words of static objects so unless the root set could be narrowed down the collector would be very inefficient. To keep track of roots the system maintains tables of locations that contain references to ephemeral objects. Whenever a pointer to an ephemeral object is created the address of the pointer is stored in the table. When ephemeral objects are collected these tables identify the roots. The table is maintained with dedicated hardware; when a word is stored into memory the barrier hardware determines if it points into an ephemeral page. If so, the address is added to a table of references. Pointers are not removed from these tables when they become invalid; the tables actually indicate a superset of the roots. The tables are implemented as bit-maps to indicate pages that contain roots. Separate tables indicated in-core pages and swapped pages since in-core pages need to be processed more efficiently.

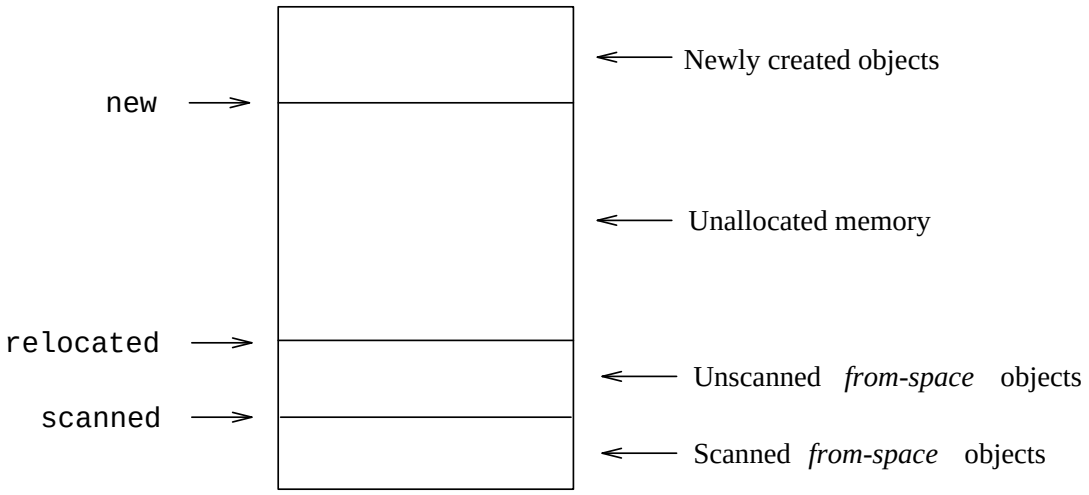

Figure 8: The structure of Baker's to-space.

2.2 Generation Garbage Collector

Lieberman and Hewitt designed an incremental copying garbage collection algorithm that segregates objects by their age [24]. The algorithm creates a number of regions of objects that are garbage collected at different rates. A region that is very young probably contains more garbage than an older region and therefore is garbage collected more frequently.

The *creation region* is used to allocate new objects. When the creation region is filled a new one is allocated. The system maintains a *current generation number*; when a region is created it is assigned the current generation number, which is periodically incremented.

The process of initiating garbage collection in a region is called *condemning* the region. Objects in a condemned region are called *obsolete*. When a region is condemned all the live objects in the region are scavenged into a new region with the same generation number but a higher version number. Then the memory allocated to the condemned region is recycled.

Objects are evacuated from a region to the next version of the region in the same way as in Baker's algorithm. In the common case, it is assumed that there will not be back-pointers. The algorithm is optimized for this case.

When a region is condemned all pointers to objects in the region must be updated. All the younger regions must be scanned for pointers into the condemned one. Given this fact it is much less expensive to condemn young regions than old ones. This is why the collector groups together young objects and scavenges them frequently.

Pointers from old regions to young ones are treated specially. Every region has associated with it a table that tracks back pointers (§1.2.8) into the region. When the region is condemned this table is used to update the back pointers without scanning their regions.

In this collector there may be multiple scavenges active concurrently. Several regions

Programmer controlled storage reclamation can be very efficient. It is effective on directed acyclic graph (DAG) data structures having no node with in-degree greater than one. It also works for vectors, however, this is less important in languages that have powerful array types. Since C++ lacks a real array type this is a primary use of the dynamic memory allocator in that language.

Even in simple cases programmer-controlled storage reclamation can be error-prone, especially in a language of the complexity of C++. The need to manage the memory takes effort away from the more important problem at hand. With generalized graph data structures the programmer must implement one of the two more complex schemes to ensure that appropriate memory is freed. If the functionality must be provided then it should not be the programmer's responsibility.

Reference counts are appropriate for some kinds of objects. They are suitable for generalized directed acyclic graphs but not for self-referential data structures. Reference counting can be inefficient because of the high cost of copying, initializing, and destroying pointers.

Garbage collection is the final alternative. Traditional garbage collection was very slow but modern copying collectors are comparatively efficient. Copying collectors collect and compact in a single pass. Their efficiency increases with the size of the virtual and real memories, making them a scalable memory management solution for the indefinite future. The presense of automatic, efficient garbage collection increases the value of a programming environment.

2 Related Work

2.1 Baker's Algorithm

Baker's algorithm partitions memory into two hemispaces: *from-space* and *to-space*. New objects are allocated from one end of to-space. In figure 8 the boundary `new` indicates the point at which new objects are allocated. During an allocation operation the collector copies some small number of objects from from-space to to-space. This process is called *scavenging*. Scavenging begins by copying the objects referenced by the roots to to-space. The region of scavenged objects grows up from the bottom of to-space and is delimited by the `relocated` pointer. After the roots have been scanned the scavenged objects themselves are scanned since they may contain pointers to from-space objects. The `scanned` marker partitions the to-space objects into those that have been scanned for from-space pointers and those that have not. When a pointer to a from-space object is encountered the object is scavenged to to-space. When all of the roots and all of the scavenged objects have been scanned there are no more live objects in from-space. This is the case when the `scanned` pointer catches up to the `relocated` pointer. At that point a *flip* occurs: the names of the hemi-spaces are swapped and the process resumes.

The algorithm performs its work while satisfying allocation requests. The amount of work that it performs during a request is proportional to the size of the request. While incremental and therefore by definition real-time, this algorithm is inefficient.

nism of polymorphism in C++. Private derivation allows the derived class to inherit functionality without being a subtype of the base class.

C++ provides a mechanism intended to help software developers use specialized dynamic storage allocators. The two free-store operators `new` and `delete` permit a `class` to define specialized allocation and deallocation strategies.

C++ supports object-oriented design (OOD). In OOD a system is decomposed into encapsulated modules that interact via message passing or procedure call. Modules, called *objects* in an object-oriented system (OOS), are not intended to affect each other except through their respective public interfaces. In C++ the `class` is the instantiation mechanism for objects.

The advantage to OOD is that it supports the development of maintainable, reusable software components. Encapsulation helps clients remain independent of the implementation of a `class` they utilize. This allows that implementation to change to reflect changing needs of the system. Such changes are localized. Inheritance promotes reusability because a `class` may be refined by a derived class to make it more appropriate for a particular application.

By contrast, in traditional monolithic programs the data structure is designed specifically for the application. This makes both reuse and maintenance more difficult because code that manipulates the data structure need not be localized, and because the data structure operates on fixed data types.[4] The client may manipulate the implementation of the data structure making it harder to change that implementation.

In spite of its advantages C++ is very complex and difficult to master. As in C, implementation details such as the use of dynamic or `static` data are visible to clients of a class [16]. Reducing the complexity of the task facing the programmer is a key to enhancing productivity and correctness. One way to achieve this is to remove the responsibility for reclaiming inaccessible dynamic objects from the programmer; automatic storage reclamation accomplishes this. Making the software developer's task easier by providing garbage collection is one of the goals of this thesis.

1.4 Summary

Dynamically allocated memory is a pervasive element of modern programming practices. It is vital in graph algorithms which constitute one of the most important abstractions in computer science. Managing dynamically allocated memory is nontrivial. Essentially there are three ways to determine when memory may be recycled:

1. The programmer can be responsible for freeing memory.

2. By maintaining reference counts some inaccessible storage can be detected and recycled.

3. Graph-traversal algorithms can be used to identify active memory and free inactive memory. This is known as garbage collection.

[4] Parameterized types are required to alleviate this limitation.

1.2.8 Generational Collection

In 1983–1984 three copying collectors were presented that improved efficiency by segregating objects according to their actual age or anticipated life expectancy [24,28,36]. These collectors by Lieberman and Hewitt, Moon, and Ungar respectively, were the first *generation-based* collectors. These collectors and more recent ones are described in the next chapter.

Generation-based collectors exploit the following empirically observed phenomenon: young objects are likely to become garbage quickly and old objects are likely to live for a long time. These collectors separate old objects from young objects. Young objects are likely to die quickly, therefore they are collected frequently. Old objects are unlikely to die so they are collected less frequently. The rationale is that garbage collecting old objects is unlikely to be profitable since few are expected to have died since the last collection.

Lieberman and Hewitt's was the first of this class. It defines several generations and conducts incremental collection at different rates in different generations. A generation is a set of objects that are of approximately the same age. Moon's collector uses virtual memory hardware and longevity-based object segregation to improve the efficiency of the Baker algorithm. It identifies *ephemeral* objects that are very short lived, and keeps them separate from static (permanent) and dynamic but longer-lived objects. Moon's and Lieberman's collectors are incremental while Ungar's is stop-and-copy.

When collecting a space, the roots for that space must be located. Generational collectors share a common problem of tracking pointers to young objects, particularly pointers inside old objects. These pointers must be located because, like pointers on the stack and in global data, they are roots for the collection. In discussions of generational collectors a *back pointer* is a pointer from an old object to a young one. The methods they use to handle back-pointers is one of the things that distinguishes these algorithms from each other.

1.3 C++

C++ [35,17] is a modern programming language that improves on the C programming language [21] in important ways [29,16].

C++ adds to C features that make it a safer and more convenient language. However, additions such as `const`, `inline`, function prototypes and references are minor compared to the addition of `class`es to support object-oriented programming [34].

C++ `class`es permit the compiler to enforce the separation of data objects into encapsulated state and public operations. The `public` members constitute the object's interface. The implementation of the object (that is, the `private` members) can change without affecting code that uses the `class` provided the interface remains unchanged.

`Class`es have a mechanism for inheritance. One `class` may inherit data and operations from another `class`, called the *base* `class`. C++ has multiple inheritance allowing one type to inherit from many base types. The resulting type hierarchy may form a directed acyclic graph (DAG). Inheritance may be `public` or `private`. In `public` derivation the derived type is a subtype of the base type; this is the primary mecha-

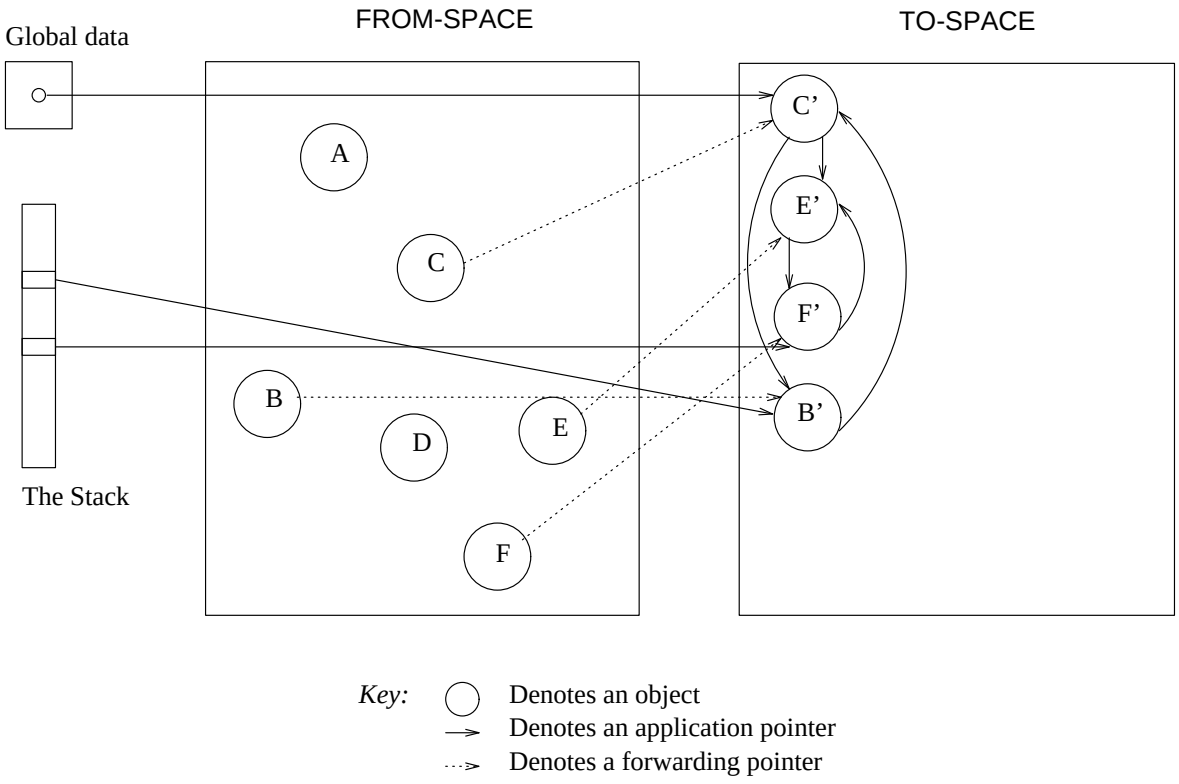

Figure 7: A snapshot after collection: all objects have been copied.

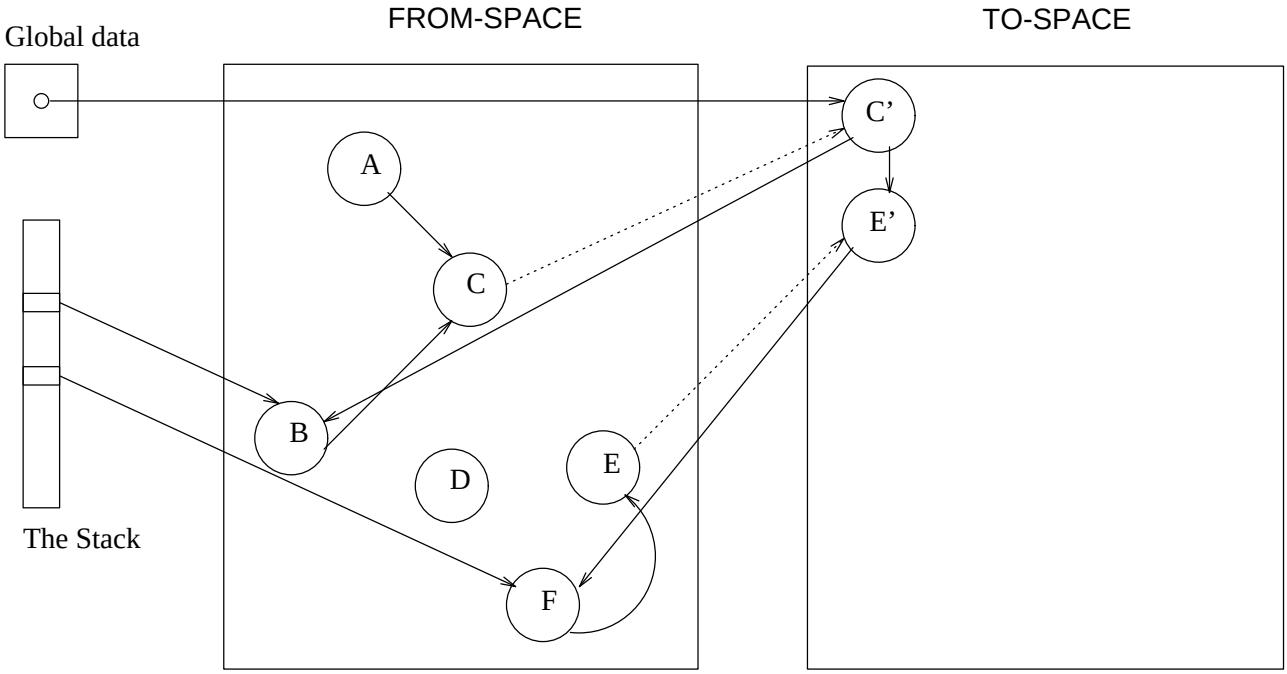

Figure 6: A snapshot during collection: two objects have been compactly copied.

When an object is visited it is scanned for values that resemble pointers. Any value that were it a pointer, would point at an allocated object, is assumed to be a pointer. The data structure that this technique marks is a supergraph of the actual live graph. After the graph traversal a sweep phase deallocates the unmarked allocated nodes.

Conservative collection cannot copy or compact objects. Copying objects implies updating pointers, but under conservative collection a value interpreted as a pointer may not actually be one.

Conservative collection is less efficient than copying collection, but it does not require tags or other type information. It is a useful technique for implementing garbage collection in languages like C [21]. Conservative collection is being used for the CEDAR project at Xerox PARC [14,13].

1.2.7 Incremental Collection

A variation on standard garbage collection is real-time collection. In real-time collection long periods of time in which the mutator is stopped are disallowed. Real-time collection is normally synonymous with *incremental* collection. Under this paradigm a small amount of garbage collection work is done frequently.

Reference counting is one incremental reclamation technique. Baker's 1978 algorithm was incremental, but inefficient. Baker's was the first incremental, copying collector [5].

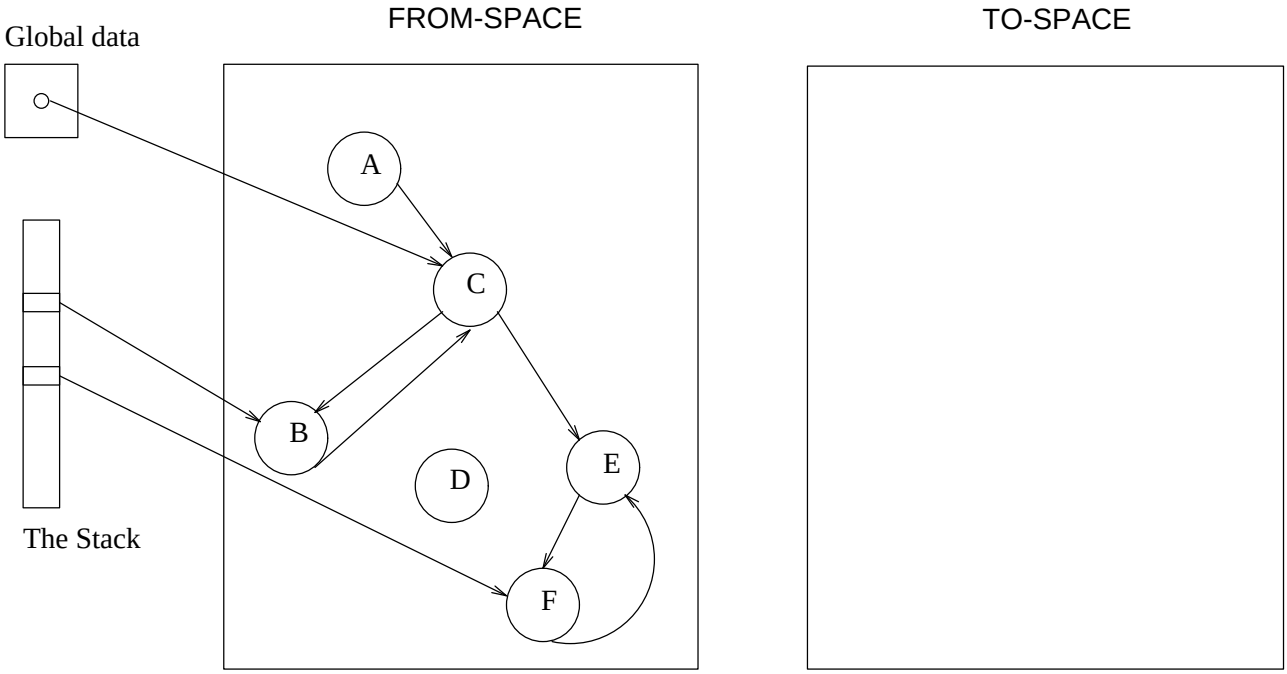

Figure 5: A snapshot of the data structure before a copying collection.

pointers it precludes conservative collection (§1.2.6).[3]

The key operations in a copying collector are twofold: 1) unambiguously identifying the roots, and 2) deeply copying the data structure. Figures 5, 6 and 7 show a data structure being copied by a copying collector. In those figures obsolescent pointers in from-space are omitted for clarity.

1.2.6 Conservative Collectors

Conservative garbage collection is described by Boehm and Weiser in [8] and by Bartlett in [6]. In other types of collectors it is necessary to differentiate between pointers and integers; conservative collection removes that requirement.

Conservative collection does not require that the roots of the data structure be explicitly identified. It begins by scanning all of the memory accessible to the program. On contemporary architectures this means the stack, the registers and the global data space. Any value found in that scan that *might* be a pointer is assumed actually to be a pointer. Thus an integer with a value that happens to resemble a pointer will be interpreted as a pointer. The quantities presumed to be pointers constitute the perceived roots of the data structure.

The perceived roots will be a superset of the real roots. Starting from the perceived roots the data structure is traversed and objects are marked (cf. mark-and-sweep §1.2.4).

[3]Papers by Demers et. al. [14] and Bartlett [6,7] discuss collectors that use both conservative and copying techniques.

structure has all the components of figure 3.

The collector identifies the root pointers of the data structure. It traverses the subgraph reachable from each root. Every node it visits is copied to the new memory space; the pointer that led to the object is redirected to the new location. In the old copy the traversal algorithm stores a *forwarding pointer* to the new copy. When the collector finds a pointer to an object that has already been copied, it updates the pointer with the forwarding address stored in the object.

The data structure can be traversed breadth-first with a queue or depth-first with a stack. The to-space region can supply memory for the queue or stack. Objects are copied into to-space from one end of the region; the process of collecting compacts all the living objects.

The collector algorithm is given in pseudo-code in figure 4. The algorithm is shown here in recursive form using the run-time stack rather than the memory in to-space. This presentation makes the algorithm easy to understand.

```
void collect(root-set)
{
    rename to-space to from-space
    allocate a new to-space
    For each root, r, in root-set
        r.traverse()
    free from-space
}

void traverse(by-reference: pointer)
{
    if pointer refers to a copied object then
        pointer = pointer->forwarding-address
    else
        new_pointer = copy the object into to-space
        pointer->forwarding-address = new_pointer
        pointer = new_pointer
        for every internal pointer, ip, in the object
            ip.traverse()
}
```

Figure 4: The copying collector algorithm.

Copying collection requires time proportional to the size of the reachable data structure, not the total size of the space. Since memory is divided into hemispaces, only half of the required memory is usable. When virtual and real memory spaces are large this trade-off is well worth the performance benefits. Since copying collection updates

1.2.5 Copying Collectors

Modern copying collectors are based on the work of Baker [5], Fenichel and Yochelson [18], and Minsky [27]. Copying collectors allocate objects from one region and then copy all live objects into another region. These collectors do work proportional to the amount of live objects rather than the total amount of allocated objects; this is a vast improvement when many objects become garbage. These collectors *compact* the objects into the new region improving virtual memory performance. Since they never deallocate individual objects a very simple, fast storage allocator such as that described in §1.1.5 can be used with copying collectors.

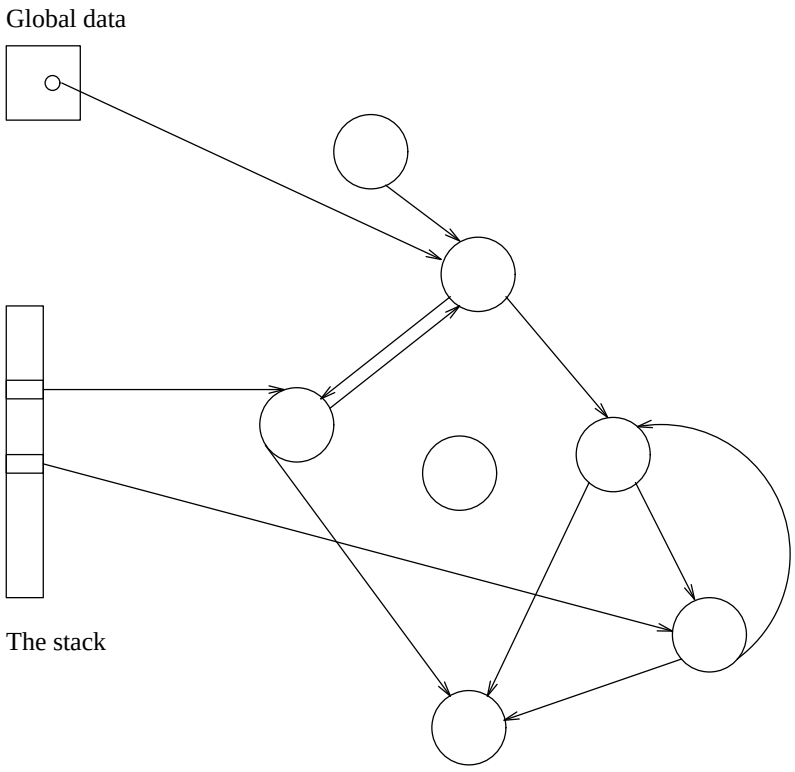

Figure 3: A representative data structure for a collector.

Large circles represent graph nodes.
The tall rectangle represents the runtime stack with two roots.
The square represents all global data with one root.

Garbage collection is triggered by the allocator when the allocation space does not have sufficient free memory to satisfy an allocation request. At this point the data

Mutator	This term refers to client or application processes. They are given this name because they continuously alter the data structure by allocating new objects and overwriting existing pointers.
Collector	The term "collector" can refer to either the garbage collection process or to the algorithm.
The data structure	The data structure denotes the set of allocated objects and their interconnections. An edge in this graph is a pointer. A node in the graph is a dynamically allocated object.
Roots	The roots are pointers into the data structure that the mutator can read. Roots are found on the stack, in global data, and in other dynamic data structures.
Internal pointer	An internal pointer is a pointer inside a node that references another node. To traverse the data structure the collector must be able to identify internal pointers.
From-space	This term and the next one apply to *copying garbage collectors* (§1.2.5). This space is the pool of memory that contains the dynamically allocated objects when a collection begins. The living objects are copied to a new space so that this space may be recycled.
To-space	During a collection by a copying collector, this is the pool of memory into which objects are copied and from which new objects are allocated. When a collection starts the old to-space is renamed from-space and a new to-space is initialized.

1.2.4 Mark-and-Sweep

Garbage collection has been a field of active research for thirty years. In that period efficiency has improved dramatically. Early—and current—collectors based on the mark-and-sweep algorithm [22,12] took time proportional to the size of the heap.[2] In its simplest form mark-and-sweep garbage collection traverses the data structure marking every object that it visits. That is the 'mark' phase. Then it traverses all of the allocated objects deallocating each one that is not marked. That is the sweep phase. The primary disadvantages to this technique are 1) it requires time proportional to total number of allocated objects, and 2) its requires two full passes. Variations that compact objects in memory require three passes. In some systems with mark-and-sweep collection large LISP programs can spend 30% of their time managing dynamic storage [38]. There are many variations on the mark-and-sweep algorithm that have become obsolete. An excellent survey of early garbage collection techniques was published by Cohen in 1981 and can be found in [12].

[2] In the remainder of this document we use *free-store* rather than *heap* to preclude ambiguity with the *heap* data structure.

4. Copy the pointer value.

It is important that the increment happen before the decrement otherwise copying a pointer to itself could deallocate the object.

The second problem with reference counting is demonstrated in figure 2. Suppose the pointer "P" is the only reference of the indicated data structure. The data structure will not be reclaimed when "P" is destroyed because the counter associated with "A" will not go to zero. Reference counting can not reclaim cyclic data structures.

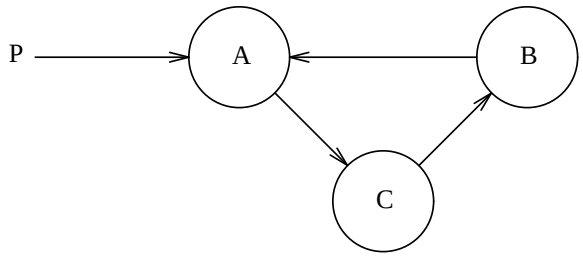

Figure 2: A reference to a cyclic data structure.

The solution to this problem for self-referential data structures is *automatic storage reclamation*, commonly called, *garbage collection*.

1.2.3 Garbage Collection: Problem and Terminology

Garbage collection is a process in which dynamically allocated objects that the program cannot reference are deallocated. All other objects are retained. The amount of research that has been done in garbage collection has resulted in a well-defined problem with standard terminology.

Certain classes of garbage collection schemes compact objects in memory and do not increase the cost of most pointer operations. They have no difficulty reclaiming cyclic data structures.

The dynamically allocated objects are interpreted as nodes in a graph. Edges in the graph are pointers that the application maintains, as well as internal pointers of objects. The graph may be disconnected and/or cyclic. The graph is often referred to as the *data structure*.

The client program is called the *mutator*. If it is a parallel program then there are *mutators*, referring to the parallel threads of the application. The name "mutator" is used because the client program operates by continuously changing, or *mutating*, the data structure.

The mutators have pointers into the graph. These pointers are called the *roots* of the data structure. The nodes of the subgraph reachable by following edges from the roots are the *live* nodes. All other nodes are *garbage*. Any node that is garbage cannot be accessed by the mutator and should be delivered to the memory allocator for deallocation. One or more *collector* processes identify and deallocate garbage objects. The terms *mutator* and *collector* were introduced in 1978 in [15] and have since become standard.

reachable data structure and deallocates the remainder. There are numerous ways of accomplishing this that are described in the remainder of this section.

1.2.1 Manual Reclamation

One way to manage deallocation is to require that the programmer issue deallocation requests explicitly. Giving responsibility to the programmer forces that individual to engage in reasoning that is secondary to the main task. It can be error-prone and tedious.

From the perspective of the programmer the logical time to deallocate a block of memory is when a pointer to the memory is destroyed. Consider figure 1. In this example two nodes each reference a third. Suppose the node labeled "A" is deleted. It contains a pointer to node "C." Node "C" should also be deleted if and only if "A" contains the only reference to "C." Nodes that may have in-degree greater than one cannot be deleted simply when any pointer to the node is destroyed.

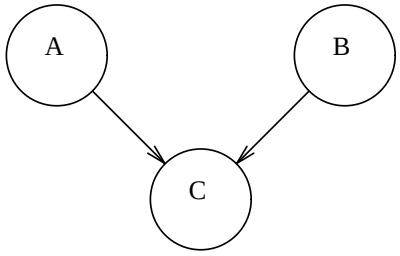

Figure 1: A node with in-degree greater than 1.

1.2.2 Reference Count Reclamation

One solution to the problem of identifying deallocatable data is to add a reference count to the data. The counter tracks the number of pointers that currently refer to the data. The pointer-copy operation copies the pointer and increments the reference count on the data. Destruction of a pointer decrements the counter of the referenced object. When a counter goes to zero the object is deallocated.

There are two problems with reference counting. The overhead that reference counting imposes on pointer operations is substantial. Creating, copying or destroying a pointer is normally inexpensive. When tracking a reference-counted object, however, these operations become expensive. Normally copying a pointer is one machine instruction. With reference counting it may require five or more instructions even when no counter goes to zero.

1. Increment the counter of the object referenced by the source of the copy.
2. Decrement the counter of the object referenced by the destination of the copy.
3. Compare and branch if that counter went to zero.

1.1.5 Block Allocation from a Buffer

A memory management method that will be described later in this paper is *copying garbage collection*. This technique is unique in that individually allocated blocks are *never* individually deallocated. Instead, the entire space is deallocated at once. The three allocation strategies mentioned previously, Sequential Fit, Buddy System and Quick Fit all expect to handle deallocation requests. The first two coalesce adjacent free blocks. All three entail unnecessary overhead for a system based on copying garbage collection.

In copying collection the allocator can satisfy allocation requests out of a large chunk of memory. To allocate a block, the current allocation-point pointer is incremented (or decremented) by the size of the request and the old (or new) value is returned. The bound must be checked to ensure that there is available storage. Allocated blocks do not need headers or footers containing the size and allocation status of the block. Allocated blocks cannot be individually deallocated. Instead the entire space is deallocated *en masse*. This allocation strategy, when applicable, is more memory efficient and faster than the others described herein.

1.1.6 Deallocation

The deallocation operation of the memory allocator causes a previously allocated block to be marked as free. If the newly freed block is adjacent to another free block the two may be coalesced to form a larger free block. This is done in Sequential Fit methods and in the Buddy System to reduce external fragmentation. Quick Fit saves time by not coalescing adjacent free blocks.

After the block has been deallocated its memory may be used to satisfy a future allocation request. The strategy and data structures used by the memory allocator dictate the cost of executing a deallocation request.

1.2 Reclamation

Reclamation is the process of returning allocated blocks to the allocator, in effect, recycling the memory. In the presense of huge virtual memories one might question the need for reclamation, however, it remains crucial.

Programs that allocate a large amount of data over time will use arbitrarily large amounts of virtual address space and secondary storage unless unused memory is reclaimed. These programs will have data distributed throughout the virtual address space unless there is compaction. Such distribution results in poor locality and excessive paging.

There are three paradigms for storage reclamation. Manual reclamation requires that the programmer of the application issue deallocation requests explicitly. This is the normal technique in many imperative programming languages such as Pascal, C and Ada. Reference counting keeps track of the number of pointers that reference a block. When that count drops to zero the block is deallocated. The Modula-2+ compiler uses reference counting [4,30]. The final alternative is garbage collection. This process traverses the

1.1.3 Buddy System Allocation

Buddy System allocation is a family of strategies that attempt to be fast while minimizing external fragmentation. The most common strategy is based on blocks whose sizes are powers of two, another is based on the fibonacci sequence.

The Binary Buddy System allocates blocks in sizes that are powers of two. Block sizes of 8, 16, 32 or 1024 units (bytes, words) might be allocated. The allocator keeps linked lists of free blocks of each size and satisfies a request with the smallest size block that is sufficiently large. For example, a request for 10 bytes would be satisfied with a 16 byte block. How this is accomplished if no 16 byte block is available is described in the following paragraphs.

Every block has its *buddy* [22]. A block of size 2^k is always aligned so that the last $k-1$ bits of its address are zero. The k^{th} bit may be either one or zero. The buddy of a block is the unique block of the same size whose address is different only in the k^{th} bit.

For example, suppose a 64 byte block is at binary address $abc100000$. The digits abc represent "don't care" values. The buddy of this block is the 64 byte block at address $abc000000$.

To satisfy a request of size n the allocator examines the linked list of free blocks of size 2^k where k is the smallest integer such that $2^k \geq n$. In the best case that list is nonempty and a block is used to satisfy the request. If that list is empty the next larger list is examined. If a larger block is available the block is broken into its component buddies. It is removed from its former free list. One component is added to the smaller free list while the other is returned. If the next larger list is empty the algorithm continues up lists until it finds a nonempty one.

When a block is deallocated the allocator determines if its buddy is free. If the buddy is also free the allocator combines them into one larger free block and checks if its buddy is free, etc. Finally the free block is inserted into the list of its size.

1.1.4 Quick Fit

Many programs allocate blocks in a small number of discrete sizes. In these cases a very simple allocation strategy can be very efficient.

Linked lists of free blocks are maintained for each discrete size. When a block is requested a free block is taken off the correct list. When a block is deallocated the block is added to the list of blocks of its size.

When an allocation request is issued and the corresponding free-list is empty, the memory allocator invokes a lower-level allocator. The lower-level can be simple or complicated. Hopefully it is not invoked too often.

The second level is backed-up by the operating system memory allocator. This is the Quick Fit strategy described by Weinstock in his doctoral thesis [40] and by Standish [31]. Weinstock found that in some cases First Fit is faster, and in some cases Best Fit results in less fragmentation, but on the average Quick Fit is the overall best strategy. The first level Quick Fit allocator is very simple to implement. For many programs good efficiency can be obtained when the second level is simple also.

2. whether or not the proceeding block is free

3. whether or not the following block is free

Proceeding and following are defined in terms of absolute address, not linked-list order. Thus, let a ten byte block begin at address 1000. The proceeding block has its last byte at address 999; this byte is part of the block's footer. The trailing block begins at address 1010; this byte is part of that block's header. This administrative information is needed for *coalescing* adjacent free blocks.

A *roving pointer* indicates the last block in the ring that the allocator examined. To satisfy an allocation request the allocator examines some number of blocks starting at the block referenced by the roving pointer. It selects a block with which to satisfy the request. If the ring contains no sufficiently large block the request fails. If the block it finds is very close to the requested size then it uses the block to satisfy the request. The block is marked as allocated in its header and footer. The roving pointer is advanced past the block; the block is removed from the ring and returned to the application.

Often the block chosen is so much larger than the request that returning it would entail excessive internal fragmentation. In these cases the block is broken into two. One part is used to satisfy the request and the other is left in the free ring.

When a block is deallocated the allocator attempts to coalesce it with adjacent free blocks. Using the headers and footers the allocator can tell whether or not the adjacent blocks are free. If one or both are free the allocator combines them into one larger free block.

The variations on this technique are *First Fit*, *Best Fit*, *Worst Fit*, and *Optimal Fit*.

First Fit satisfies an allocation request with the first block it finds that is sufficiently large. It frequently chooses a block that is much larger. This can lead to external fragmentation. It is attractive because allocation can be very fast.

Best Fit always scans the entire ring. It selects the block that is most nearly the exact size required. It is the best in terms of fragmentation, external and internal. It is generally slower than First Fit.

Worst Fit always scans the entire ring and uses the largest block it finds. The justification is that this will not create many small blocks that cause external fragmentation.

Optimal Fit scans part of the ring to get a representative sample of its contents [10]. After scanning some fraction of the ring it then selects the next block it finds that is better than all the ones it has seen. Optimal Fit examines fewer blocks than Best Fit so it is faster. It examines more blocks than First Fit so it causes less fragmentation.

A naive Sequential Fit allocator might use a linear linked list rather than a circular one. Knuth found that this leads to many very small fragments at the beginning of the list [22]. This slows down the First Fit and Optimal Fit strategies. This was an important observation because First Fit was the most common strategy at the time it was made. Using a ring distributes the fragments better.

request is available the allocator returns an indication of failure. A deallocation request informs the memory allocator that a block of memory previously obtained through an allocation request is no longer needed by the program. The memory allocator may use that memory to satisfy a future allocation request.

Generally a first-level memory allocator is backed-up by a second-level memory allocator. On UNIX systems the first memory allocator is in the programming language standard library, for example, `malloc`. The second level allocator is often part of the operating system, such as the UNIX `sbrk` system call. This system call extends the process' *heap* region to make more memory available.

Dynamic memory allocation occurs continuously in languages such as LISP and Prolog. Some programming languages allocate function activation records dynamically. In imperative languages like Pascal, ADA and C, memory allocation occurs at the explicit request of the program. The program issues allocation requests when it needs room for a computation and issues a deallocation request when it identifies a block of memory that has become free. Dynamic memory is used in these languages for building graph data structures and for holding objects of variable size such as strings and vectors.

1.1.1 Allocation

Efficient memory allocation has been a field of active research since the late 1950's. Running time of LISP programs is significantly impacted by the efficiency of the memory allocator. Research into improving efficiency has resulted in a wide variety of strategies. There are four common classes of memory allocation strategies: Sequential Fit, Buddy System, Segregated Free-list, and Buffer Block allocation.[1] Knuth [22] and Standish [31] are good references for detailed descriptions of the issues that arise in implementing these strategies. Knuth's book predates the Quick Fit Segregated Free-list method.

Dynamic allocation strategies are difficult to analyze analytically but easy to measure empirically. Weinstock compared allocation strategies in [40]. Improving the efficiency of these techniques resulted in substantially faster LISP systems.

1.1.2 Sequential Fit Allocation

Sequential Fit allocation is a family of allocation strategies that use the same data structure but differ in how they choose the block to satisfy a request. They use a circular doubly-linked list of free blocks. Every free block contains its size and pointers to its neighbors in the ring. The initial free list consists of one block that points to itself in both directions.

Every block has a header and footer consisting of a small number of bytes that contain the block's size, its ring pointers, and its current status, namely, allocated or currently free. Given a pointer to a block the allocator can tell three things:

1. whether or not the block is currently free

[1] This is a nonstandard term that the author uses to describe the simple allocation scheme for a copying collector described by Appel in [3].

Introduction

This thesis studies the problem of efficient dynamic memory management for object-oriented imperative programming languages. The two aspects of the problem are allocation and reclamation. The memory allocator provides storage to the program upon demand in a way that is expected to avoid excessive fragmentation. Reclamation is the process of recycling inaccessible memory so that it may be used to satisfy future allocation requests. On virtual memory (VM) systems this keeps processes small and reduces paging. On non-VM systems this permits a program to allocate more data over the course of its lifetime than the machine has available memory.

Manual storage reclamation is error-prone and, on dynamic generalized, cyclic graph data structures, impossible without graph traversal. The lack of automatic reclamation makes many common imperative programming languages such as C and Pascal inconvenient for such manipulations. The lack of automatic reclamation also makes the programmer expend creativity in a way that is secondary to the primary goal.

Since the 1970's hierarchical design and stepwise refinement have led to monolithic software systems. In such systems, a global dynamic memory manager was efficient and appropriate. However, the software design paradigm is evolving into independent components linked into client-server relationships, the *object-oriented* approach. Under this model the global dynamic memory manager must be abandoned to be replaced by multiple localized memory managers that can be customized for the data structures they administrate.

C++ is a modern, general purpose programming language that extends the C programming language into the object-oriented design domain. C++ is a useful tool, but it is more difficult to use than it should be because it lacks automatic memory reclamation. This thesis presents a dynamic storage management organization for C++ based on *copying garbage collection.*

This document is organized as follows. Section 1 describes the components of a dynamic memory manager including the allocator and the optional collector. Section 2 discusses related work in automatic reclamation. Section 3 presents our approach and its implementation. Section 4 analyzes the new dynamic memory manager for efficiency and flexibility. Section 5 concludes with a summary of what has been presented while section 6 looks to the future. Section A contains appendices and source code.

1 Dynamic Memory Management

1.1 The Memory Allocator

Nearly all modern, general purpose programming languages provide a memory allocation interface. A component of a program that accepts and satisfies allocation and deallocation requests is called a *memory allocator*. An allocation request is a message from the program's main computational engine to the memory allocator that asks for a block of memory of a particular size. The memory allocator identifies a block of the requested size and returns its address to the program. If no block large enough to satisfy the

Abstract of the Thesis

Dynamic Storage Reclamation in C++

Daniel Ross Edelson

Abstract

Dynamically allocated memory is a pervasive element of modern programming practices, but efficient dynamic memory reclamation is difficult. One class of algorithms that accomplish this is *copying garbage collection*. These algorithms require one pass to collect and compact objects in memory. It is the preferred form of garbage collection for systems with large virtual memories and large free-stores because the work of these algorithms is proportional to the amount of living data rather than the total data allocated.

A copying collector organization appropriate for object-oriented imperative programming languages is presented. The system is both encapsulated and efficient. The efficiency of reclaiming a data structure depends solely on the size of the data structure, not the size of the program. It remains within the philosophy of its implementation language, C++, because code fragments that do not use the collector are not affected in any way. Any number of collectors can coexist concurrently in an application on disjoint data structures, making it an appropriate component of an object-oriented system.

This system adds an important feature to C++ that increases productivity and makes the language more convenient. It makes dynamic memory management more efficient than manual reclamation for an important class of problems. It serves as a platform for research into reclamation techniques, particularly virtual memory issues, and, it provides an appropriate organization for automatic dynamic memory reclamation in object-oriented imperative programming languages.

List of Tables

List of Figures

Contents

Dynamic Storage Reclamation in C++

Daniel Ross Edelson
daniel@cis.ucsc.edu

UCSC-CRL-90-19
June 1990

Board of Studies in Computer and Information Sciences
University of California at Santa Cruz
Santa Cruz, CA 95064