Garbage Collection and Run-time Typing as a C++ Library

David Detlefs
Digital Equipment Corporation
Systems Research Center
130 Lytton Ave, Palo Alto CA 94301
detlefs@src.dec.com

June 18, 1992

1 Introduction

Automatic storage management, or *garbage collection,* is a feature that can ease program development and enhance program reliability. Many high-level languages other than C++ provide garbage collection. This paper proposes the use of "smart pointer" template classes as an interface for the use of garbage collection in C++. Template classes and operator overloading are techniques allowing language extension at the level of user code; I claim that using these techniques to create smart pointer classes provides a syntax for manipulating garbage-collected storage safely and conveniently. Further, the use of a smart-pointer template class offers the possibility of implementing the collector at the user-level, without requiring support from the compiler. If such a compiler-independent implementation is possible with adequate performance, then programmers can start to write code using garbage collection without waiting for language and compiler modifications. If the use of such a garbage-collection interface becomes widespread, then C++ compilation systems can be built to specially support the garbage collection interface, thereby allowing the use of collection algorithms with enhanced performance.

This paper presents such a garbage collection interface and implementation. In particular, the collection scheme has the following properties:

1. it requires no compiler support;
2. it requires no information from the programmer about the format of garbage-collected objects;
3. it has strategies to cope with bugs caused by aggressive optimizing compilers;
4. it allows the use of both automatically and explicitly managed storage in the same program; and
5. it invokes destructors on collected heap objects.[1]

[1]It is by no means clear that invoking destructors on collected objects is the proper approach for handling object finalization in garbage-collected C++ systems, but doing so at least allows the convenient handling of many common cases.

The "smart-pointer" class developed in this paper also supports *run-time type queries.* Such queries give programs the ability to ascertain the allocated type of an object at run-time, and is another feature offered in several high-level languages (e.g., CLU [22], Modula-3 [21].)

The discussion is organized as follows: Section 2 discusses related work; Section 3 presents the garbage-collection interface; Section 4 presents the template class framework and collection algorithm; Section 5 describes support for run-time type queries; Section 6 presents some further implementation considerations; Section 7 briefly discusses performance; Section 8 discusses areas for future work, giving special attention to problems that the current system does not solve; and Section 9 presents conclusions.

2 Related Work

There have been a number of garbage collection systems proposed and implemented for C++. To the best of my knowledge, all fail to provide at least one of the properties listed in Section 1.

Boehm and Weiser [5] developed a conservative mark-and-sweep collector aimed at use with C, but also suitable for use with C++. This collector has most of the properties listed above; the exception is that the collector does not invoke destructors on collected objects (property 5), and has insufficient information to do so. Another disadvantage is that every pointer-aligned field is considered to be a pointer, which may hurt collector performance and retain too much storage.

Bartlett [2] presents a mostly-copying collector for C++. In this scheme, stacks and registers are considered conservatively: in these areas, objects "pointed to" by bit patterns that might be pointers are retained but not copied. However, type information giving the offsets of pointers in heap objects allows copying of objects that are referenced only by pointers in the heap. Later work [3] casts this collection system in a generational framework. Bartlett's collector was originally designed as part of a Scheme runtime written in C. When used in general-purpose C++ programming, Bartlett's collector requires the user to invoke special macros indicating the presence of pointer fields in classes, violating property 2. Also, no provision is made for invoking destructors of collected objects, violating property 5. In previous work [6], I extended Bartlett's work by modifying a compiler to produce the pointer-location information, and allowing concurrent mutator activity in the style of Appel, Ellis, and Li [19]; however, a collection strategy requiring a modified compiler violates property 1.

Ferreira [11] presents a C++ collector framework that allows explicitly managed storage and calls destructors. Collectibility is determined on a per-class basis; programmers must invoke a macro in the definition of each collected class that registers the class with the collector, violating property 2. The default collection algorithm is a conservative mark-and-sweep scheme similar to the Boehm-Weiser collector. There are a set of "supplementary rules" that the programmer can follow to increase performance; following these rules allows the use of other, often more efficient, collection algorithms. For example, if the programmer guarantees that each class provides a s̃canref method that invokes a collection procedure for each pointer in the class, then a potentially more efficient generational algorithm may be used. If any of the supplemental rules are used, they must be used consistently and correctly everywhere, or else the collector may malfunction. Ferreira modified a C++ compiler to follow the supplemental rules automatically; this version violates property 1.

Seliger [23] describes “Extended-C++”, a language extension that provides garbage collection as well as some other useful features. This extension defines a new language (implemented by a translator that produces C++ as output), and therefore violates property 1. Kuse and Kamimura [16] take a similar approach, designing their own object-oriented C extension that includes garbage collection.

Edelson and Pohl [9] present a copying collector based on smart pointers. The constructor for a smart pointer class performs *root registration;* it inserts the address of the pointer into a data structure, and the smart pointer destructor removes it. The collector traverses this data structure to obtain the accurate pointer identification necessary to enable the pointer modification required in copying collection. A major drawback of this collector is that the implementor of a garbage-collected class must implement a member function to copy and scan instances of that class, violating property 2. This collector does not maintain sufficient information to invoke destructors.

Later work by Edelson [10] points out some shortcomings in the copying collector described above, and describes an alternative mark-and-sweep collector based on pointer-indirection tables. This collector still violates property 2 by requiring programmers to implement a `mark` function in each garbage-collected class to mark recursively the targets of pointers contained in objects of that class. This collector is designed to allow asynchronous object finalization, as in Cedar [17] and Modula-2+ [18], in lieu of destructor invocation.

Wang [26] describes a collector based on smart pointers implemented with template classes. This system offers both reference-counting collection and “fake copy” collection; the latter, which is similar to generational mark-and-sweep, reclaims cyclic garbage. In this system, programmers use a distinguished constructor to declare that a particular reference should be registered as a root. Every garbage-collected class must include certain macros in its definition, violating property 2.

Kennedy [15] describes the support for garbage collection in the OATH class library. Kennedy points out some dangers associated with using smart pointers to get garbage collection, and advocates the use of *accessors,* which are essentially “smart references,” instead. Accessor classes must provide the same interfaces as the classes that they access, which imposes some burden on the programmer. The OATH collector is a simple reference-counting collector, but also offers the option of periodically invoking a more expensive collector that collects cyclic garbage. This backup collector is novel in that it uses a three-pass algorithm that involves reference count manipulations and does not require the identification of root pointers. This algorithm does require the identification of pointers within collected objects, and the paper does not specify how the collector gets that information; one must assume that the programmer supplies it, violating property 2 [12].

Ginter [12] surveys C++ garbage collectors, especially those involving smart pointers. He describes some difficulties with these approaches, and proposes language modifications to eliminate these difficulties. This paper addresses the concerns raised by Ginter without proposing language modifications.

3 Basic Garbage Collection Interface

This section presents the most basic form of the garbage collection interface, which takes the form of a template class `Ptr`.

Figure 1 shows the basic definition of the `Ptr` class; note that private members are elided. Every class `Ptr<T>` inherits from `PtrAny`; as we shall see in Section 5, `PtrAny` is

```
class PtrAny {
   public:
      PtrAny();
      int operator==(const PtrAny& pa);
};

template<class T> class Ptr:  public PtrAny {
   public:
      Ptr(); // Default constructor; sets value to NIL.
      void New(); // Allocates a new T on the heap and makes this Ptr point to it.

      // Copy-constructor and assignment operator.
      Ptr(const Ptr<T>& pt);
      Ptr<T>& operator=(const Ptr<T>& pt);

      // Overloaded pointer operations.
      T& operator*();
      T* operator->();
};

const PtrAny PtrNil;
```

Figure 1: `Ptr` template class

somewhat analogous to `void*` for non-collected pointer types. The default constructor for `Ptr` sets the value of the `Ptr` to NIL (= 0). The member function `New` allocates a new `T` and causes the `Ptr` to refer to it. This is the only way to allocate a garbage collected object. `Ptr` defines copy constructors and overloaded assignment operators; these must at least ensure that the `Ptr` that is constructed or assigned to gets the same value as the argument; as we will see later, in some implementations these operations may also have other side-effects, such as the maintenance of reference counts. The overloaded `operator*` and `operator->` functions allow `Ptr<T>` variables to be used syntactically much as if they were of type `T*`. A deliberate exception to this rule is the omission of operators for pointer arithmetic, which interacts poorly with many collection algorithms. Finally, note that the `PtrAny` class, rather than `Ptr<T>`, provides an equality operator. This placement of the operator is essential to allow comparisons with a single NIL value; if the equality operator were in `Ptr<T>`, and required a `Ptr<T>&` argument, then we would need a different NIL value for every `T`.

`Ptr` is intended to be a complete interface for garbage collection. Given a program using explicit storage management, the main changes required to convert that program to use garbage collection are modifications of the types of `T*` variables to `Ptr<T>`.[2] In particular, the only modification that a programmer need make to a class containing pointers to collected objects is to use `Ptr` members to represent those pointers.

[2]Other changes may be required to work around the use of pointer arithmetic and the address-of operator in the original program, but these changes are generally straightforward.

4 Basic `Ptr` Implementation

My implementation of the `Ptr` interface has two aspects. The first is a framework that uses somewhat subtle properties of the C++ template system to allow the efficient and convenient generation of information useful to garbage collection algorithms, particularly the offsets of garbage-collected pointers within objects. This framework requires neither compiler support nor any information from the user beyond the use of `Ptr` types. The second aspect is the implementation of a particular collection algorithm that I feel is well suited for providing safe, portable C++ collection. The two aspects are in large measure separable; one could imagine using the template framework with several of the algorithms mentioned in Section 2 that required user-specification of the locations of pointers in objects, thus removing their violation of property 2. Sections 4.1 and 4.2 will present the framework in two parts, describing, respectively, a class hierarchy that allows the collector to treat all heap objects uniformly, and a system for generating type description information automatically; Section 4.3 will then present the particular collection algorithm.

4.1 Basic Class Hierarchy

This section presents the first part of the framework: those aspects of the implementation of `Ptr` that would be shared by any collection algorithm.

```
class WrapperBase {
   protected:
      WrapperBase();

   public:
      virtual  WrapperBase() {}
      // ...Whatever other members and functions are needed by
      // the collection algorithm...
};

template<class T> class Wrapper:  public WrapperBase {
      T elem;
   public:
      // Allows us to specify a special allocator for collected objects.
      void* operator new(size_t st);

      Wrapper() {}
      sl // Only used by Ptr<T>.
      T* GetElem()  return &elem;

      virtual W̃rapper() {};
};
```

Figure 2: `Wrapper` template class

The first implementation detail we reveal is the existence of the class `WrapperBase` and the template class `Wrapper<T>`, shown in figure 2. The `WrapperBase` class presents a "garbage collected object" interface to the collector; it may include data members such as reference counts or mark bits, or virtual functions such as "`markMyChildren`". Different collection algorithms may require different `WrapperBase` classes, but the important point is that the collector treats all objects in the garbage-collected heap uniformly as instances of class `WrapperBase`.

Each `Wrapper<T>` class inherits the functionality of `WrapperBase`, and adds a `T` member; the `GetElem` operation returns the address of this component. `Wrapper<T>` overloads `operator new` to use an allocator appropriate for the collection algorithm. `Wrapper<T>` may also implement some virtual functions of `WrapperBase` in a way that depends on `T`; we will see an example of this in Section 4.3. Finally, note that both `WrapperBase` and `Wrapper<T>` have virtual destructors with empty bodies. This allows the collector to `delete` a `WrapperBase*` representing a reclaimed object, causing the invocation of the destructor for the associated `Wrapper<T>`; if class `T` has a destructor, it will be called by the `Wrapper<T>` destructor.

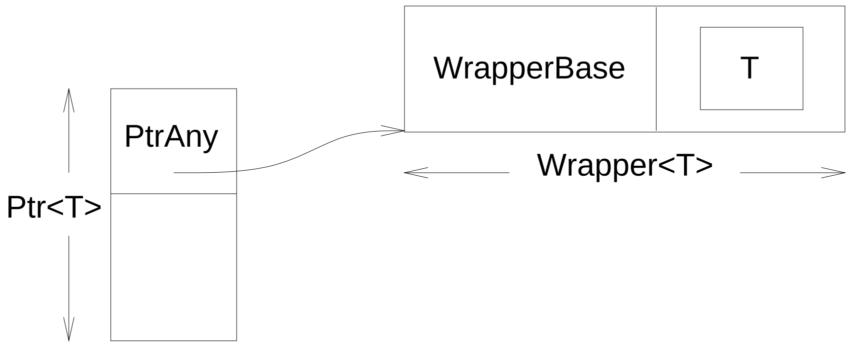

Figure 3: Relation of Ptr and Wrapper Classes

Conceptually, `Ptr<T>` contains a single member `ptr` of type `Wrapper<T>*`, and the `New` function allocates a new `Wrapper<T>` and assigns its address to `ptr`. So that `PtrAny` can be used in a similar way to `void*` (see Section 5), `ptr` is actually defined to be a protected member of `PtrAny` of type `void*`, and is cast to and from `Wrapper<T>*` by the operations of `Ptr<T>`. The "`*`" and "`->`" operators of `Ptr<T>` work by invoking the `GetElem` operation of `Wrapper<T>`. Figure 3 shows the relationships between the `PtrAny, Ptr, WrapperBase,` and `Wrapper` classes.

4.2 Automatic Generation of RC Maps

Most collection algorithms require a method for identifying pointers in heap objects. Many Lisp systems used *tagged data*, where one or more bits in each data word is devoted to a flag indicating the type of the value in the word. This approach requires compiler support, and has performance overhead that would not be acceptable to the C community. The method we implement uses per-type *RC maps* (reference-containing maps) to indicate which elements of aggregate types contain reference-counted pointers. This section presents the part of the framework that allows RC maps to be generated automatically, without requiring compiler support.

A number of the C++ collectors discussed in Section 2 require the user to provide information equivalent to RC maps: Bartlett's system requires the user to invoke a macro identifying each pointer in a `struct`; Ferreira's collector has a similar scheme in its non-conservative mode; Edelson and Pohl's collector requires the user to implement a `copy` member function for each garbage-collected class. It would obviously be more convenient and less error prone if RC maps could be generated without programmer intervention. The rest of this section presents a scheme that accomplishes this goal.

We assume the existence of an `RCMap` class providing functions for noting the offsets of `Ptr`'s within objects and for yielding these offsets in a completed `RCMap`.

```
class PtrAny {
      static RCMap* curMap;
      static void* containerStart;

      static void CtorInit(void* ptrStart, void*&);
      static void CtorNorm(void*, void*& ptr);

      static void SetCtorInit();
      static void SetCtorNorm();

      static void (*Ctor)(void*, void*&);
      friend class RCMapperBase;

   public:
      PtrAny() { Ctor(this, ptr); }
      int operator==(const PtrAny& ra);
};
```

Figure 4: Changes to `PtrAny` for RC Maps

We add several private members to `PtrAny`, as shown in Figure 4. The basic idea is that the `PtrAny` class is always in one of two states, INIT or NORMAL. In the INIT state we are building an `RCMap` for some class C that may contain `Ptr`'s; a pointer to the `RCMap` being built is stored in `PtrAny::curMap`. We cause the construction of an object O of class C, and cause each `Ptr` constructed during the construction of O to determine its offset within O and add that offset to the RC map.

The `SetCtorInit` function puts `PtrAny` in the INIT state. The default constructor for `PtrAny` always calls the function indicated by the function pointer `Ctor`; the `SetCtorInit` function, as the name implies, sets `Ctor` to `CtorInit`. `CtorInit` simply computes the difference between the address of the `PtrAny` being constructed (passed as the first argument) and the start of the container class, which it assumes to be stored in `PtrAny::ContainerStart`.[3] This difference is appended to the current RC map.

`SetCtorNorm` sets the value of the function pointer `Ctor` back to `CtorNorm`, which simply does what the default constructor of `PtrAny` did in the previous version; namely, initialize the pointer to NIL. Having this constructor perform an indirect function call where it was inlined before obviously imposes some cost in the NORMAL state.

When we have built an `RCMap` for a class `T`, we store it as a static member of `Wrapper<T>`. We also create a virtual member function of `WrapperBase` that returns the RC map associated with the `Wrapper<T>` object of which the `WrapperBase` is a part. This virtual member

[3]To be more precise, only those `Ptr`'s whose addresses fall within the bounds of the original container object contribute to the RC map, since the constructor for the containing object might create other `Ptr`'s or `Ref`-containing objects at arbitrary locations.

function allows the collector to obtain RC maps for objects while still treating all objects uniformly as `WrapperBase`'s.

Next we introduce a template class `RCMapper<T>`, shown in Figure 5, to do the work of constructing an RC map for a class `T`. As we will see, constructing a single object of class `RCMapper<T>` creates an `RCMap` for type `T` and stores it as a static member of `Wrapper<T>`.

```
class RCMapperBase {
   protected:
      // start is the start address of the current object.
      RCMapperBase(void* start);

      // This is called by the Wrapper and PtrArrayTarget special constructors
      // when they are finished.
      RCMap* SetCtorNorm();
};

// This exists only to get the RCMap for T built.
template<class T> class RCMapper:  public RCMapperBase {
      T elem;
   public:
      RCMapper() :  RCMapperBase(&elem) {
         RefTarget<T>::rcMap = SetCtorNorm();
      }
};
```

Figure 5: `RCMapper` template class

The `RCMapper<T>` constructor executes after the `RCMapperBase` constructor. The body of this constructor is not shown, but its function is to set up RC map generation. Since `RCMapperBase` is a friend of `PtrAny`, its constructor can access the private members of `PtrAny` shown in Figure 4; it sets `containerStart` to the address of the `T` object in the `RCMapper<T>`, sets `curMap` to a new, empty `RCMap`, and calls `SetCtorInit` to put `PtrAny` in the INIT state. We now resume the `RCMapper<T>` constructor, which next invokes the constructor for its single member `elem`. If the class `T` contains any `Ptr`'s, their constructors execute; since `PtrAny` is in INIT mode, the offsets of these `Ptr`'s are added to `PtrAny::curMap`. When the constructor for `elem` is complete, we finally execute the body of the `RCMapper<T>` constructor, which sets `PtrAny` back to the NORMAL state and stores the completed RC map for class `T` in `Wrapper<T>::rcMap`.

To reach the goal of generating RC maps automatically, we must cause a static object of class `RCMapper<T>` to be initialized for each type `T` for which `Ptr<T>` occurs in the program. This is simple; we include a static member of class `RCMapper<T>` in class `Ptr<T>`. Figure 6 shows changes to the `Ptr` class to support the generation of RC maps.

With this structure, the use of a `Ptr` type such as `Ptr<Foo>` causes an `RCMap` for class `Foo` to be created and stored as a static member of `Wrapper<Foo>`. This assumes, of course, implementations that support all the template capabilities defined in the draft ANSI standard [24]; in the partial implementations that are now becoming available, it may be necessary, for example, to instantiate templates explicitly in some source file.

```
template<class T> class Ptr:  public PtrAny {
   private:
      static RCMapper<T> rcmt;
      // ...
};

// In the implementation file for Ptr:
template<class T> RCMapper<T> Ptr<T>::rcmt;
```

Figure 6: Changes to Ptr template class for RC maps

4.3 A Particular Collection Algorithm: Deferred Reference Counting

This section presents the implementation of a particular collection algorithm within the framework described in Sections 4.1 and 4.2. The algorithm chosen is a *conservative reference-counting* collector. This algorithm has a number of desirable features for C++ collection:

- it does not move objects, avoiding a variety of problems associated with object movement in systems without compiler support;
- collection interruptions are short, allowing good interactive response;
- it is particularly simple to implement using smart pointer classes; and
- it treats pointers on the stack conservatively, avoiding a number of problems that can occur when collectors are used with aggressive optimizing compilers.

This algorithm also has some disadvantages:

- maintaining reference counts can impose considerable run-time overhead, and
- reference-counting alone cannot reclaim cyclic garbage, so a full-scale mark-and-sweep collection must be invoked occasionally.

Conservative reference counting is based on one of the two essential ideas of *deferred reference-counting* [7]. Both ideas are aimed at reducing the run-time cost of maintaining reference counts. I borrow the idea of treating stack pointers conservatively: the reference counts stored in objects count only references from heap or global objects; an object's count does not include references from the stack. When an object's reference count becomes zero, it is placed on a *Zero Count List* (ZCL) rather than deleted. The collector periodically interrupts the mutator to perform a collection. It scans the mutator stack(s) and registers to construct a *Found-On-Stack* (FOS) table which maps addresses to boolean values; if an address is found on the stack, it maps to **true** in this table. The collector considers each object on the ZCL: only objects not found in the FOS table are deleted.[4]

[4]The aspect of deferred-reference counting that I do *not* use is the idea of logging operations that change reference counts to a *transaction queue,* which is processed off-line to bring reference counts up to date. This makes sense, for example, in a multiprocessor environment where a separate processor can process the transaction queues, but probably does not gain any advantage on a current-technology uniprocessor.

The hope of conservative reference counting is that a large fraction of pointer assignments occur on the stack, thus avoiding the overhead of modifying reference counts. This strategy is usually implemented with compiler cooperation; for example, the SRC Modula-2+ system [18] used a deferred reference counting collector, and its compiler generated code to adjust reference counts only for assignments to non-stack `REF` (pointer-to-collected-object) variables.

Copying collectors move objects; reference-counting and (non-compacting) mark-and-sweep collectors do not. Disallowing object movement has a number of advantages for C++ collection, especially for systems that are intended to work without compiler support. To move an object, the complete and accurate set of pointers to the object must be identified; this set may include *derived pointers,* created by the compiler optimizations, that correspond to no program variable, as well as intra-object pointers used in implementing virtual base classes in multiple inheritance. Many have proposed modifying compilers to produce tables providing sufficient information to locate all pointers to objects (including, recently Diwan [8]), but this approach obviously requires a compiler closely integrated with the collector.

4.4 Implementing Basic RC Collection

```
class WrapperBase {
      int ref_count:  30;
      int on_zcl_bit:  1;
      int mark_bit:  1;

   protected:
      WrapperBase();

   public:

      void Inc();
      int Dec();
      int NonZeroRC();

      void SetZCLBit();
      void ResetZCLBit();

      void SetMarkBit();
      void ResetMarkBit();

      virtual ~WrapperBase() {}
};
```

Figure 7: Changes to `WrapperBase` class for Conservative RC Collection

It is quite easy to implement conservative reference-counting collection in the template framework I've described. The `WrapperBase` class is revised as shown in Figure 7. We add three private data members to `WrapperBase,` encoded in one 32-bit word using bitfields: the object's reference count, a bit indicating whether it is on the Zero Count List, and a mark bit reserved for use by the "backup" mark-and-sweep collector. The `Inc, Dec,` and `NonZeroRC` member functions manipulate the reference counts. `Inc` increments the count, `Dec` decrements it and returns a non-zero result if the count becomes zero, and `NonZeroRC` similarly returns a non-zero result if the count is non-zero. The remaining new member functions set and reset the ZCL and mark bits.

With these changes to `WrapperBase`, changing the `Ptr` template class to perform reference counting is straightforward. The `New` member function, the copy constructor and the overloaded assignment operator are each modified to perform as follows:

1. Check whether the current value of the `Ptr` is non-NIL.
2. If so, decrement the reference count of the original referent; if the count becomes zero, put the object on the Zero Count List and set the object's `on_zcl` bit.
3. In any case, if the new value of the `Ptr` is non-NIL, increment the count of the new referent.

To complete the collector, we modify the storage allocator to perform a collection every n^{th} allocation. As described above, a collection scans thread stacks to construct a Found-On-Stack table, then considers each object on the Zero Count List, deleting those objects whose counts remained zero and whose addresses are not found on the stack.

As described so far, the collector does not gain any performance advantage from being conservative; if `Ptr<T>` variables are used for all pointers into the heap, then all pointer assignments and initializations are reference-counted. To allow pointers on the stack to avoid reference counting, we create a variant of `Ptr<T>` called `SPtr<T>`, for "stack pointer." An `SPtr` supports the same operations as `Ptr`, but performs no reference counting; the use of inline functions for assignment and dereference operations make the performance of an `SPtr` similar to that of a "raw pointer." Conversion and assignment operators are defined in `SPtr` and `Ptr` in a way that make them interchangeable from the point of view of client programmers.

Since the collection algorithm includes a conservative stack scan, programmers may choose to substitute `SPtr`'s for `Ptr`'s in certain places to enhance performance. Any `Ptr` variable that the compiler will store on the stack may safely become an `SPtr`. `Ptr`'s appearing as members of classes must in general remain `Ptr`'s, since instances of the class may be allocated on the heap. There is no way (without compiler support) to check the correctness of these substitutions; programmers wanting absolute safety could use `Ptr` everywhere, but `SPtr` offers enhanced performance and maintains correctness if one follows the simple rule of using only `SPtr`'s whose storage class is `automatic`.

Using a conservative scan of the stack not only offers the possibility of enhancing performance, but also addresses many of the safety concerns that have been raised about collectors based on smart pointers. Kennedy [15] and Ginter [12] describe several scenarios which are dangerous for "smart-pointer" collectors. The dangers in most of these scenarios, particularly problems involving compiler temporaries, are avoided by using a non-copying collector that retains objects referenced by pointers found in a conservative root scan.

For example, Kennedy considers the statement

```
O2 = O1->makeCopy()->transform();
```

where `O1` and `O2` are smart pointer variables of the same type. The `makeCopy` member function makes a copy of the current object, and returns a smart pointer to the new object. The `transform` member function modifies the object to which it is applied, and returns a smart pointer to the modified object. The problem that Kennedy points out is that if the compiler generates a temporary for the result of `makeCopy`, this temporary may be the only smart pointer to the copy of `O1`. Once `operator->` is invoked on this smart pointer (returning a "dumb" pointer), the smart-pointer temporary is dead, and language rules allow it to be destructed at any time before the end of the current scope. If it is destructed immediately, and a collection follows immediately after that, the collection may reclaim the object to which `transform` is about to be applied.

We can avoid this danger by using a conservative scan that recognizes *interior* pointers (pointers into the interiors of objects). In the `Ptr` implementation, the `T*` returned by `operator->` points into the interior of the `Wrapper<T>` that the collector considers to be the allocated object. Thus, we require that the existence of such an interior pointer on the stack suffices to prevent collection of the object into which it points.[5] A number of schemes have been proposed that allow interior pointers to be mapped into the addresses of containing objects; see, e.g., [5, 6, 2].

Independent of any concerns about the destruction of temporaries, it may be that some optimizing compilers will modify the pointers contained in smart pointer records in ways that violate the assumptions of the conservative scan [4]. In this case, we could provide (at some cost) a completely safe implementation along the following lines. Modify the `Ptr` and `SPtr` classes so that each contains two copies of the `ptr` member. Use one as a "write-only" copy of the pointer; it is never dereferenced, and therefore never exposed to compiler optimization. (It is declared `volatile` to prevent the writes from being optimized away.) With this approach, the collector can rely on the existence of a pointer to the head of all objects referenced from the stack. The collector would check the FOS table only for the starting address of the `WrapperBase` object. The cost of this scheme may not be as large as it seems on first consideration; memory costs are decreasing at a constant geometric rate, and the extra write costs only a single store operation per assignment, and never incurs a memory read. Section 7 presents some measurements of the cost of this scheme.

Thus, we have a range of collection options that can collect safely in the face of a variety of compiler behaviors. The safest combination, retaining objects referenced by any interior pointers found in the roots, and maintaining a "write-only copy" of the pointer in a `Ptr`, should be proof against almost all compiler behaviors. Obviously, it would be helpful if compilers were designed with garbage collection in mind; with true compiler cooperation, collectors can make stronger assumptions that improve performance.

4.5 Mark-and-Sweep Collection

As mentioned in Section 4.3, a disadvantage of reference-counting collection is its inability to collect cyclic garbage. The usual strategy is to periodically invoke a different collector that can reclaim cyclic garbage, such as a mark-and-sweep collector; I see no reason to depart from this strategy.

Given RC maps, it is a fairly simple matter to code a mark-and-sweep collector that treats the stacks and global data area as roots. Since we are allowing explicitly managed storage to coexist with the garbage-collected heap, we must also treat this "malloc heap" conservatively as a source of collection roots. Supporting mark-and-sweep collection places two requirements on our heap data structures:

1. We must be able to identify the allocated portion of the explicitly managed heap, so

[5]One might be tempted to optimize by observing that the pointer returned by `operator->` always points to a fixed offset from the containing `Wrapper<T>`, but statements such as

```
O2 = O1->makeCopy()->xxx.transform();
```

show that any interior pointers must be sufficient to prevent collection: here we dereference the `xxx` member of the referent of the smart pointer, which is at an unknown offset from the start of the containing `Wrapper<T>`.

that it can be scanned for possible collection roots.

2. We must be able to identify the start and extent of every object in the garbage-collected heap, both to determine which object an interior pointer points to, and to locate unreferenced objects during the sweep phase.

There are many possible organizations of the heap that allow these requirements to be satisfied; I have not yet implemented one, and will not present one here. I will note that for programs that make extensive use of explicitly managed storage, treating the explicitly managed heap as a source of conservative roots may to be expensive, so it will probably be important to invoke the mark-and-sweep collector as infrequently as possible.

5 Run-Time Type Information

The `Ptr` class also include another feature called *typecodes,* inspired by Modula-2+ `REF` type. Typecodes are a minimal form of run-time type information: a distinct integer is associated with each type `T` for which a `Ptr<T>` type is instantiated in the program. This integer can be found given either the type `Ptr<T>` or an instance of that type:

```
Ptr<T> pt;
pt.New();
if (pt.TypeCode() == Ptr<T>::StatTypeCode()) { ...  }
```

What makes this capability useful is that any `Ptr<T>` can be converted into a `PtrAny`, and `PtrAny` is defined so that one can still query the type code. Further, `Ptr<T>` assignment operator for `PtrAny` arguments allows the assignment not only of `PtrNil` to a `Ptr<T>`, but also of non-NIL `PtrAny` values whose typecodes indicate that they are really `Ptr<T>`'s. If the typecode of the `PtrAny` does not match the typecode of `Ptr<T>`, the copy constructor and assignment operator raise an exception. The copy constructor for `PtrAny` arguments is defined in the same way.

While in general it is a good idea to search for ways of substituting statically-type-checked mechanisms such as virtual functions or template classes for run-time type discrimination, the technique remains useful in some circumstances. For example, run-time type queries enable the use of heterogeneous collections; in my system, such collections would have element type `PtrAny`. Another example occurs when a virtual function's implementation would differ only slightly in derived classes; it may be more convenient to implement the function once in the base class, using run-time type queries to produce the appropriate subclass-dependent behavior.

A number of systems have been proposed for adding run-time type information to C++. Gorlen's NIHLib [13] provides the functionality of typecodes and more; programmers may query whether an object is derived from any type, and acquire the name of the an object's actual type as a string. To offer this functionality, all classes must be derived from a distinguished base class `Object`, and class implementors must manually provide the information used by the run-time mechanism. The ET++ user-interface framework [1] provides similar capabilities. Interrante and Linton [14] propose a `Dossier` class as a standard interface for class information; the functionality provided by this interface subsumes the functionality of typecodes. Interrante and Linton propose extending the language to generate a virtual function automatically for each class that returns an appropriate `Dossier` object; in the absence of such a language extension, however, programmers must follow the convention of implementing such a virtual function manually in each class. Lenkov, Mehta and Unni [20] also propose a set of language extensions that would enable run-time type queries.

Again, I emphasize that the mechanism proposed here is quite minimal; features such as returning the class name as a string, or querying whether the actual type of a `PtrAny` is some subtype of a given static type, are not supported. The main strength of this proposal is that it does not require

- any extension to the language definition or compilers;
- a common base class shared by all classes; or
- additional code in every class supplied explicitly by programmers.

In contrast, each of the proposals above requires one or more of these properties.

The template framework we have defined for collection makes implementing typecodes quite simple. `TypeCode` is a simple class with a single public integer member; the constructor of `TypeCode` guarantees that each `TypeCode` object has a distinct value for this member. `Typecode` ensures this property by using a static member as a monotonically increasing counter. Every `Wrapper<T>` class has a static `TypeCode` member whose initializer invokes this constructor.

6 Implementation Considerations

This section presents implementation details that would have complicated the previous discussion.

6.1 Dynamically-Sized Arrays

The program fragment

```
int* ip = new int[j];
```

allocates an array of integers whose size is determined at run-time by the value of the variable `j`. It is not possible to use the `Ptr` class to allocate a dynamically-sized, garbage-collected array. To allow the use of such arrays, I provide a class similar to `Ptr` called `PtrArray`.

For the most part, `PtrArray`'s are identical to `Ptr`'s. The main differences are:

- The `New` operation of `PtrArray` takes an integer parameter telling how many elements will be in the newly-allocated array.
- `PtrArray` does not provide overloaded pointer operations. Instead, the only mechanism it provides for accessing its contents is an overloaded `operator[]`, which provides bounds-checked access to the array elements.[6]
- `RCMap` is modified slightly to work more efficiently with `RefArray`: an `RCMap` also includes the number of bytes between the last `Ptr` in an object and the beginning of the next object when the object is included in an array. This information is generated with the same automatic mechanism used before.

[6]Bounds-checking can be turned off by setting a preprocessor variable.

6.2 Circular Definitions

When using pointers, it is desirable to be able to write definitions such as:

```
struct IntList {
      int i;
      IntList* next;
};
```

If Ptr's are to be as powerful as pointers, one should be able to write

```
struct IntList {
      int i;
      Ptr<IntList> next;
};
```

This generates the constraint that it must be possible to instantiate at least the declaration of Ptr<T> when T is an incomplete type. (Recall that the *declaration* of a class gives its signature; the *definition* its implementation.) Thus, the result of instantiating the declaration of Ptr<T> may not include any classes that have elements of type T. Note that this constraint does not apply to the the definition of Ptr<T>, which may use T freely.

Two "tricks" are used to to avoid using T in the declaration of Ptr<T>. The first trick has already been mentioned: the ptr member of PtrAny is declared as a void*, but treated as a Wrapper<T>* in the bodies of the implementations of Ptr<T> via casting. The second trick involves the RCMapper<T> type used to generate RC maps automatically. Section 4.2 stated that every class Ptr<T> includes a static member of type RCMapper<T>. This is not actually true, because RCMapper<T> defines a member of type T, which would cause instantiations with incomplete types to give a compile-time error. Instead, Ptr<T> has a static member whose type is RCMapperIndirect<T>*. Nothing is revealed about the class RCMapperIndirect<T> in the declaration of Ptr. The implementation reveals that RCMapperIndirect<T> has a static member of type RCMapper<T>; initializing this member creates the RC map for type T.

6.3 Complete Class Hierarchy

We started with class Ptr, and added a number of related classes, as follows:

- SPtr allows programmers to get pointer-like efficiency for stack-allocated Ptr's; objects referenced by these stack pointers will be retained because of the conservative stack scan used by the collector.
- PtrAny enabled run-time type queries and heterogeneous collections.
- PtrArray supported garbage-collected references to arrays of objects.

Figure 8 shows the complete class hierarchy associated with Ptr's. The new features introduced by this figure are described below.

- The pair of classes PtrAny and SPtrAny are related in the same way as Ptr and Sptr, which we have already discussed. PtrArray and SPtrArray also have this relationship.
- BPtr<T> abstracts the functionality common to Ptr<T> and SPtr<T>, such as the dereference operators. (Read the "B" as "Base.") Similarly, BPtrAny factors out functionality common to both PtrAny and SPtrAny, such as the typeCode virtual function, and BPtrArray defines the functionality common to PtrArray and SPtrArray, such as the overloaded operator[].

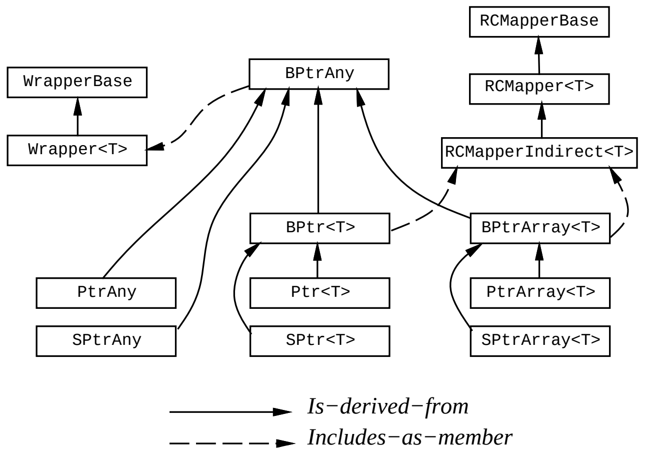

Figure 8: Garbage Collector Class Hierarchy

7 Performance

I have modified two non-trivial programs to use the garbage collection system described in this paper. One, GROBNER, is an implementation of the Grobner basis algorithm [25], a theorem-proving application. The other, HYPER, is a simulation of a hypercube architecture, done by Donald Lindsey at Carnegie-Mellon University.

		GROBNER		HYPER	
1	ET-no-GC	4.4 sec		10.4 sec	
2	ET-GC *(/ ET-no-GC)*	8.8 sec	*(2.00)*	16.5	*(1.59)*
3	# Allocations	163301		11334	
4	# RC ops	515735		5496586	
5	On stack *(%)*	188783	*(37%)*	2873609	*(52%)*
6	Non-stack *(%)*	326952	*(63%)*	2622977	*(48%)*
7	# Collections	18		3	
8	Collect Time	2.11 sec		0.09 sec	
9	Avg. collect time	0.11 sec		0.03 sec	
10	Max. collect time	0.17 sec		0.08 sec	

Figure 9: Performance of Reference-Counting Garbage Collection

Figure 9 shows measurements of these programs running on representative input files. All measurements were taken on a MIPS R3000-based DECStation 5000 with 96 MByte of memory, running Ultrix. Memory size was sufficient to prevent paging in all tests. "ET-no-GC" shows the elapsed time of the program using explicit storage management. The numbers shown in this line and all others are the averages over 10 runs. "ET-GC" shows the elapsed time of the same program using garbage collection, and includes the ratio with the first line. "# Allocations" indicates the number of allocations performed during the run. "# RC ops" is the number of *reference-counting operations* that occurred in the run; these operations are the assignment operators and copy constructors of `Ptr` and `SPtr` types. The "On stack" line indicates the number of these operations that occurred on `SPtr`'s, thus avoiding reference counting; the remainder are shown in the "Non-stack" line. The "# Collections" line shows the number of collections that occurred in the run. The last three lines show the total time spent in collections, and the average and maximum durations of collection interruptions.

The overall performance of the garbage collection algorithm in these tests is somewhat disappointing, as shown by the increase in overall elapsed time. Section 4.3 stated that a hope of conservative reference counting is that the majority of pointer assignments occur on the stack, and are therefore not reference counted; lines 3-6 show that this hope is not fulfilled in the programs tested. Two programs is still a small sample size; further measurements are obviously desirable.

One positive aspect of these measurements is that the interruptions due to conservative reference-counting collections are quite short, as shown in lines 9 and 10. Thus, the cost of collection can be smoothly distributed, which would be an advantage in interactive applications. The number of allocations between collections was set at 10,000 entries in these tests, but can be modified by the user. Increasing this number to 100,000 for GROBNER decreased the number of collections to 3, and decreased total collection time by 19%. However, the length of the average collection interruption increased from 0.11 sec to 0.57 sec.

A final observation is that using the two-pointer scheme described in Section 4 for enhanced safety does not significantly increase the garbage collection overhead for these programs. Using this scheme for GROBNER gave no detectable difference in the overall elapsed time; for HYPER, elapsed time increased by 5%.

8 Future Work

This section examines opportunities for future work in this area. Some opportunities are made available by strengths of this collection interface; others, unfortunately, come out of weaknesses.

8.1 Strengths: Other Collection Algorithms

A strength of this collection interface is that it should admit several implementations. In fact, other collection algorithms should be able to re-use much of the template framework presented in Section 4. For example, one can easily imagine implementing the `Ptr<T>` interface with Bartlett's mostly-copying algorithm instead of reference counting. Using the `Ptr<T>` syntax for pointers to garbage-collected objects would enable the coexistence of garbage-collected and explicitly managed storage in Bartlett's algorithm.[7] The main advantage gained would be the automatic generation of RC maps, avoiding the requirement that programmers provide equivalent information explicitly.

[7] Note that the explicitly managed heap would have to be treated as part of the conservative root set, as noted in Section 4.5.

8.2 Weaknesses

The current interface has several weaknesses. Solutions to these problems would be interesting future work.

One weakness is that a class `T` used to instantiate `Ptr<T>` must have a constructor of no arguments if it has any constructors. If `T` has any constructors, the constructor of no arguments is the only one that can initialize a `T` object allocated using `Ptr<T>::New`. This requirement is at best inconvenient; for some types, it is worse, since it may not be possible to sensibly define a constructor of no arguments for the type. (Consider, for example, something like `NonEmptySet`.)

Another weakness of the interface concerns "casting up," or *widening*. Casting up is a common, natural, and safe C++ idiom:

```
class A { ... };

class B : public A { ... };

B* bp = new B;
A* ap = bp;          // Implicitly casts bp to A*.
```

If `Ptr`'s are to be semantically equivalent to pointers, there should be a similar idiom involving `Ptr`'s; we should be able to convert a `Ptr<B>` to a `Ptr<A>`. Ideally, this would be done with the same syntax used by pointers.

The best mechanism I have been able to devise for casting up `Ptr`'s uses a template class `Widen<From, To>`. The constructor for `Widen` takes a `Ptr<From>&` argument; `Widen` provides a single operation `ToPtr` that returns a `Ptr<To>`. Using `Ptr`'s and `Widen`, the example above would be rewritten as

```
Ptr<B> bp;
bp.New();
Ptr<A> ap = Widen<B, A>(bp).ToPtr();        // Convert bp to Ptr<A>.
```

`Widen` is guaranteed to give a compile-time error message if `To` is not a base class of `From`. The current version of `Widen` works only with single inheritance hierarchies implemented in the usual way, where casting up a pointer preserves the pointer value. I have a design for a version of `Widen` that would work with multiple inheritance, but would require the `Ptr` and `SPtr` classes to contain two pointers instead of one. This scheme could be combined with the two-pointer scheme described for added safety in Section 4.4.

Another capability that would be convenient but is not supported by the current system is the capability to "cast down" (or *narrow*) to a legal type other than the allocation type of an object. That is, continuing the example above, if one had a `PtrAny` whose allocated type is `Ref<B>`, one would like to be able to query whether the `PtrAny` is also of type `Ptr<A>`, and convert the `PtrAny` to that type if the answer is **true**. The system presented in this paper provides no support for this operation. It may be that many future compilers will generate run-time data structures containing information sufficient to support these queries in order to implement the C++ exception model; if so, it may be useful to extend the C++ language to make some of this information available to programmers [20]. In any case, C++ presently offers no guarantees for the results of casting down; the situation is no worse for in the system I propose.

9 Conclusions

This paper has presented a smart pointer template class interface to garbage collected storage, and has argued that such an interface may be sufficiently convenient to be used even if language changes are made to support collection. It also presented a compiler-independent implementation of this interface. This implementation was divided into a “framework” that could support a number of different collection algorithms, and a particular collector based on conservative reference counting. While the performance of this implementation is not especially exciting, its existence allows the interface to be used without any special compiler support. If the interface (or one like it) is found to be useful, gains popularity, and is later standardized, one could imagine compilation systems giving special treatment to the `Ptr` class that would enable more efficient collection algorithms. This is a potential scenario for the wide-spread adoption of garbage-collection by C++ programmers.

References

[1] Gamma; Andre Weinand; Erich and Rudolf Marty. ET++ – an object-oriented application framework in C++. In *Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 46–57, New York, NY, 1988. ACM.

[2] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, February 1988.

[3] Joel F. Bartlett. Mostly-copying collection picks up generations and C++. Technical Report TN-12, Digital Equipment Corporation Western Research Laboratory, October 1989.

[4] Hans-Juergen Boehm. Simple GC-safe compilation. Position paper in 1991 OOPSLA Garbage-Collection Workshop, 1991.

[5] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[6] David L. Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie-Mellon University, May 1990.

[7] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[8] Amer Diwan. Stack tracing in a statically typed language. Position paper in 1991 OOPSLA Garbage-Collection Workshop, 1991.

[9] Daniel Edelson and Ira Pohl. A copying collector for C++. In *Usenix C++ Conference Proceedings*, pages 85–102, Berkeley, CA, 1991. Usenix Association.

[10] Daniel R. Edelson. A mark-and-sweep collector for C++. In *Proceedings of the ACM Conference on Principles of Programming Languages*, New York, NY, 1992. ACM. Edelson has made a longer version of this paper available electronically.

[11] Paulo Ferreira. Garbage collection in C++. Position paper in 1991 OOPSLA Garbage-Collection Workshop, 1991.

[12] Andrew Ginter. Cooperative garbage collectors using smart pointers in the C++ programming language. Master’s thesis, University of Calgary, December 1991.

[13] K. E. Gorlen. An object-oriented class library for C++ programs. *Software Practice and Experience*, 17(12):899–922, December 1987.

[14] John A. Interrante and Mark A. Linton. Runtime access to type information in C++. In *Usenix C++ Conference Proceedings*, pages 233–240, Berkeley, CA, 1990. Usenix Association.

[15] Brian Kennedy. The features of the Object-oriented Abstract Type Hierarchy (OATH). In *Usenix C++ Conference Proceedings*, pages 41–50, Berkeley, CA, 1991. Usenix Association.

[16] Kazushi Kuse and Tsutomu Kamimura. Generational garbage collection for C-based object-oriented languages. Position paper in 1991 OOPSLA Garbage-Collection Workshop, 1991.

[17] Butler W. Lampson. A description of the Cedar language; a Cedar language reference manual. Technical Report CSL-83-15, Xerox PARC, 1983.

[18] Paul Rovner; Roy Levin; and John Wick. On Extending Modula-2 For Building Large, Integrated Systems. Research Report 3, Digital Equipment Corporation Systems Research Center, 1985.

[19] John R. Ellis; Kai Li; and Andrew W. Appel. Real-time Concurrent Collection on Stock Multiprocessors. Research Report 25, Digital Equipment Corporation Systems Research Center, February 1988.

[20] Dmitry Lenkov; Michey Mehta; and Shankar Unni. Type identification in C++. In *Usenix C++ Conference Proceedings*, pages 103–118, Berkeley, CA, 1991. Usenix Association.

[21] Greg Nelson, editor. *Systems Programming in Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[22] B. Liskov; R. Atkinson; T. Bloom; E. Moss; C. Schaffert; R. Scheifler; and A. Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

[23] Robert Seliger. Extending C++ to support remote procedure call, concurrency, exception handling, and garbage collection. In *Usenix C++ Conference Proceedings*, pages 241–264, Berkeley, CA, 1990. Usenix Association.

[24] Bjarne Stroustrup. *The C++ Programming Language: Second Edition*. Addison-Wesley, Reading, Massachusetts, 1991.

[25] J. P. Vidal. The computation of Grobner bases on a shared memory multiprocessor. Technical Report CMU-CS-90-163, Carnegie Mellon University, August 1990.

[26] Thomas Wang. The MM garbage collector for C++. Master's thesis, California Polytechnic State University, San Luis Obispo, October 1989.