Precompiling C++ for Garbage Collection[⋆]

Daniel R. Edelson

INRIA Project SOR, Rocquencourt BP 105, 78153 Le Chesnay CEDEX, FRANCE
Daniel.Edelson@inria.fr

Abstract. Our research is concerned with compiler-independent, efficient and convenient garbage collection for C++. Most collectors proposed for C++ have either been implemented in a library or in a compiler. As an intermediate step between those two, this paper proposes using precompilation techniques to augment a C++ source program with code to allow mostly type-accurate garbage collection. There are two key precompiler transformations. The first is automatic generation of *smart pointer* classes. The precompiler defines the smart pointer classes and the user utilizes them instead of raw pointers. These smart pointers supply functionality that allows the collector to locate the root set for collection. The second transformation augments the C++ program with code that allows the garbage collector to locate internal pointers within objects. This paper describes the precompiler and the garbage collector. The paper includes a brief (1500 word) survey of related techniques.

1 Introduction

The lack of garbage collection (GC) in C++ decreases productivity and increases memory management errors. This situation persists principally because the common ways of implementing GC are deemed inappropriate for C++. In particular, tagged pointers are unacceptable because of the impact they have on the efficiency of integer arithmetic, and because the cost is not localized.

In spite of the difficulty, an enormous amount of work has been and continues to be done in attempting to provide garbage collection in C++. The proposals span the entire spectrum of techniques including:

- compiler-based concurrent atomic mostly-copying garbage collection [12],
- library-based reference counting and mark-and-sweep GC [27],
- library-based mostly copying generational garbage collection [5],
- library-based reference counting through *smart pointers* [28, 29] (Smart pointers are discussed momentarily),
- library-based mark-and-sweep GC using smart pointers [16],
- compiler-based GC using smart pointers [22],
- library-based mark-and-sweep and generational copying collection using macros [21], and,
- library-based conservative generational mark-and-sweep GC [8, 11].

[⋆] This paper appears in *Proc. 1992 International Workshop on Memory Management*, Springer-Verlag, 1992

The vast number of proposals, without the widespread acceptance of any one, reflects how hard the problem is.

In the past, we have proposed implementing GC strictly in application-code: GC implemented in a library. The problem with this approach is that it requires too much effort on the part of the end-user. The user must first customize/instantiate the library, and then follow its rules. This is a tedious and error-prone process.

To solve our goal of compiler-independence, while keeping the associated complexity to the user to a minimum, we are now proposing *precompiling* C++ programs to augment them for garbage collection. The user still needs to cooperate with the collector, but the likelihood of errors is reduced. In addition, the precompiler can perform transformations that are independent of the actual garbage collection algorithm in use, making it very useful for experimentation in GC techniques.

A Word About Smart Pointers

A number of the systems that are considered in the related work section use smart pointers, as does this collector. Therefore, this paper begins with an introduction to the term.

C++ provides the ability to use class objects like pointers; these objects are often called smart pointers [38]. Smart pointers allow the programmer to benefit from additional pointer semantics, while keeping the syntax of the program largely unchanged. Smart pointers use *operator overloading* to be usable in expressions with the same syntax as normal pointers. For example, the overloaded assignment operator = permits raw pointers to be assigned to smart pointers and the overloading indirect member selection operator -> permits smart pointers to be used to access data members and operations of the referenced object.

Smart pointers can be used for a variety for purposes. For example:

- reference counting [10, 27, 28, 29, 39],
- convenient access to both transient and persistent objects [36, 39],
- uniform access to local or distributed objects [24, 35, 37],
- synchronizing operations on objects [39, p. 464],
- tracing garbage collection [16, 18, 27],
- instrumenting the code,
- or others.

Section 2 discusses how existing systems use smart pointers for memory management. A discussion of various issues concerning smart pointers can be found in [17]. Our implementation of smart pointers is presented in §3.1.

2 A Brief Survey of Related Work

There is a significant body of related work, in the general field of GC, in C++ software tools, and specifically in collectors for C++.

2.1 Conservative GC

Conservative garbage collection is a technique in which the collector does not have access to type information so it assumes that anything that might be a pointer actually is a pointer [7, 8]. For example, upon examining a quantity that the program interprets as an integer, but whose value is such that it also could be a pointer, the collector assumes the value to be a pointer. This is a useful technique for accomplishing compiler-independent garbage collection in programming languages that do not use tagged pointers.

Boehm, Demers, et al. describe conservative, generational, parallel mark-and-sweep garbage collection [7, 8, 11] for languages such as C. Russo has adapted these techniques for use in an object-oriented operating system written in C++ [32, 34]. Since they are fully conservative, during a collection these collectors must examine every word of the stack, of global data, and of every marked object. Boehm discusses compiler changes to preclude optimizations that would cause a conservative garbage collector to reclaim data that is actually accessible [6]. Zorn has measured the cost of conservative garbage collection and found that it compares favorably not just with manual allocation, but even with optimized manual allocation [44].

Conservative collectors sometimes retain more garbage than type-accurate collectors because conservative collectors interpret non-pointer data as pointers. Often, the amount of retained garbage is small, and conservative collection succeeds quite well. Other times, conservative techniques are not satisfactory. For example, Wentworth has found that conservative garbage collection performs poorly in densely populated address spaces [41, 42]. Russo has found that the programming style must take into account the conservative garbage collector: naive programming leads to inconveniently large amounts of garbage escaping collection [33, 34]. For example, he has found it necessary to disguise pointers and manually break garbage cycles [33]. To aid the programming task, he is investigating augmenting the conservative garbage collector with *weak pointers* [30], i.e. references that do not cause objects to be retained. Finally, we have tested conservative garbage collection with a CAD software tool called ITEM [16, 26]. This application creates large data structures that are strongly connected when they become garbage. A single false pointer into the data structure keeps the entire mass of data from being reclaimed. Thus, our brief efforts with conservative collection in this application proved unsuccessful.

As these examples illustrate, conservative collection is a very useful technique, but it is not a panacea. Since it has its bad cases, it is worthwhile to investigate type-accurate techniques for C++.

2.2 Partially Conservative

Bartlett's *Mostly Copying Collector* is a generational garbage collector for Scheme [14] and C++ [39] that uses both conservative and copying techniques [4, 5]. This collector divides the heap into logical pages, each of which has a *space-identifier*. During a collection an object can be promoted in one of two ways: it can be physically copied to a to-space page or the space-identifier of its present page can be advanced.

Bartlett's collector conservatively scans the stack and global data seeking pointers. Any word the collector interprets as a pointer may in fact be either a pointer or some other quantity. Objects referenced by such roots must not be moved because, as the roots are not definitely known to be pointers, the roots cannot be modified. Such objects are promoted by having the space identifiers of their pages advanced. Then, the root-referenced objects are scanned with the help of information provided by the application programmer; the objects they reference are compactly copied to the new space. This collector works with non-polymorphic C++ data structures, and requires that the programmer make a few declarations to enable the collector to locate the internal pointers within collected objects.

Detlefs implements Bartlett's algorithm in a compiler and uses type information available to the compiler to generalize the collector. Bartlett's first version contains two restrictions, the first of which is later eliminated:

1. internal pointers must be located at the beginnings of objects, and
2. heap-allocated objects may not contain *unsure* pointers.

An unsure pointer is a quantity that is statically typed to be either a pointer or a non-pointer. For example, in "union { int i; node * p; } x;" x is an unsure pointer.

Detlefs relaxes these by maintaining type-specific map information in a header in front of every object. During a collection the collector interprets the map information to locate internal pointers. The header can represent information about both sure pointers and unsure pointers. The collector treats sure pointers accurately and unsure pointers conservatively. Detlefs' collector is concurrent and is implemented in the *cfront* C++ compiler.

2.3 Type-Accurate Techniques

Kennedy describes a C++ type hierarchy called OATH that uses both reference counting and mark-and-sweep garbage collection [27]. In OATH, objects are accessed exclusively through application-level references called *accessors*, that are very similar to stubs because they duplicate the interfaces of their target objects. Accessors implement reference counting on the objects that they reference. The reference counts are used to implement a three-phase mark-and-sweep garbage collection algorithm [9] that proceeds as follows. First, OATH scans the objects to eliminate from the reference counts all references between objects. After that, all objects with non-zero reference counts are root-referenced. The root-referenced objects serve as the roots for a standard mark-and-sweep collection, during which the reference counts are restored. Like normal reference counting, this algorithm incrementally reclaims some memory. In addition, however, this algorithm reclaims garbage cycles.

In OATH, a method is invoked on an object by invoking an identically-named method on an accessor to the object. The accessor's method forwards the call through a private pointer to the object. This requires that an accessor implement all the same methods as the object that it references. Kennedy implements this using preprocessor macros so that the methods only need to be defined once. The macros cause both the OATH objects and their accessors to be defined with the given list of methods. While not overly verbose, the programming style that this utilizes is quite different from the standard C++ style and such long macros can make debugging difficult.

Goldberg describes tag-free garbage collection for polymorphic statically-typed languages using compile-time information [23], building on work by Appel [2], who in turn builds on techniques that were invented for Algol-68 and Pascal. Goldberg's compiler emits functions that know how to locate the pointers in all necessary activation records of the program. For example, if some function $\mathcal{F}$ contains two pointers as local variables, then another function would be emitted to mark from those pointers during a collection. The emitted function would be called once for every active invocation of $\mathcal{F}$ to trace or copy the part of the datastructure that is reachable from each pointer. The collector follows the chain of return addresses up the runtime stack. As each stack frame is visited, the correct garbage collection function is invoked. A function may have more than one garbage collection routine because different variables are live at different points in the function. Clearly, this collector is very tightly coupled to the compiler.

Yasugi and Yonezawa discuss user-level garbage collection for the concurrent object-oriented programming language ABCL/1 [43]. Their programming language is based on active objects, thus, the garbage collection requirements for this language are basically the same as for garbage collection of Actors [13, 25].

Ferreira discusses a C++ library that provides garbage collection for C++ programs [21]. The library supplies both incremental mark-and-sweep and generational copy collection, and supports pointers to the interiors of objects. The programmer renders the program suitable for garbage collection by placing macro definitions at various places in the program. For example, every constructor must invoke a macro to register the object and every destructor must invoke a complementary macro to un-register the object. Another macro must be invoked in the class definition to add GC members to the class, based on the number of base classes it has. To implement the remembered set for generations, the collector requires a macro invocation on every assignment to an internal pointer. Ferreira's collector requires that the programmer supply functions to locate internal pointers. It can also scan objects conservatively to work without these functions.

Maeder describes a C++ library for symbolic computation systems whose implementation uses smart pointers and reference counting [29]. The library contains class hierarchies for *expressions*, *strings*, *symbols*, and other objects that are called *normal*, and reference-counting smart pointers are used exclusively to access the objects. To improve the efficiency of assignment of reference counted pointer assignment, the address of a discrete object serves as a replacement for the NULL pointer. This means that pointers do not need to be compared with NULL before being dereferenced to modify the reference count. The smart pointers support debugging by allowing the programmer to detect dangling references: rather than being deleted, an object is marked *deleted* and subsequent accesses to the object cause an error to be reported. Other functionality allows the programmer to detect memory leaks by reporting objects that are still alive when the program terminates.

Madany et al. discuss the use of reference counting in the *Choices* object-oriented operating system [28]. The hierarchy of operating system classes is shadowed by parallel smart pointer classes, called *ObjectStars*. By programmer convention, the system classes are accessed exclusively through ObjectStars, which implement reference counting on their referents. As identified by Kennedy in [27], returning reference counting smart pointers from functions can sometimes result in dangling references. This was observed to be true of the ObjectStars, and therefore the following convention was adopted: Whenever an ObjectStar is returned from a function, it must first be assigned to a variable; it cannot be immediately dereferenced [15]. This prevents that particular error.

3 Garbage Collecting C++ Code

The program's dynamically allocated garbage collected objects are collectively referred to as the *data structure*. The collector's job is to determine which objects in the data structure are no longer in use and to reclaim their memory. The application has pointers into the data structure; these pointers are called *roots* and are collectively referred to as the *root set*. Any object in the data structure that can be reached by following a chain of references from any root is *alive*. The other objects are *garbage* and should be reclaimed. The two hard problems are: 1) finding the roots, and 2), locating pointers inside objects, called *internal pointers*.

3.1 Roots and Smart Pointers

This system uses smart pointers that implement indirection through a root table. All of the direct pointers are concentrated in the root table and can therefore be located by the collector. The term *root* is used to refer to the smart pointer objects. In contrast, the built-in pointers, i.e. the pointers that are directly supported by the compiler and the hardware, are called *raw pointers*.

A problem with smart pointers is that they can be nontrivial to code [17]. The problem arises from emulating the implicit type conversions of raw pointers. For example, a raw pointer of type T∗ can be implicitly converted to type const T∗, based on the safety of converting an unrestricted pointer to a pointer that permits only read accesses. Also, derived class pointers can be converted to base class pointers, reflecting the *isa* relationship between a derived class and its base classes. C++ allows smart pointers to emulate these type conversions using *user-defined* type conversions. The need to add these user-defined type conversions makes generation of the smart pointer classes inconvenient. They cannot be automatically produced from a parameterized type, a *template*, because that does not supply the necessary type conversions. While macros or inheritance can abbreviate the process, some coding is required. Emitting smart pointer class definitions, rather than necessitating hand coding, is one of the tasks of the precompiler.

The Root Table. The data structure that allows the collector to find the root set is the *root table*. It it implemented as a linked list of *cell arrays*. Each cell array contains its list link and many direct pointer *cells*. A cell may be *active*, in which case it contains a direct pointer value, or it may be *free*, in which case it is in the free list. A diagram of this data structure is presented in Fig. 1.

The application's smart pointers point to pointer cells rather than directly to objects; the cells, in turn, contain the direct pointers. C++ objects implement this in the following way. The initialization code for a root, i.e. the *constructor*, gets a cell from the free list, optionally initializes the cell, and makes the root point to the cell. The de-initialization code for a root, the *destructor*, adds the root's cell to the free list. The overloaded indirection operators first dereference the indirect pointer

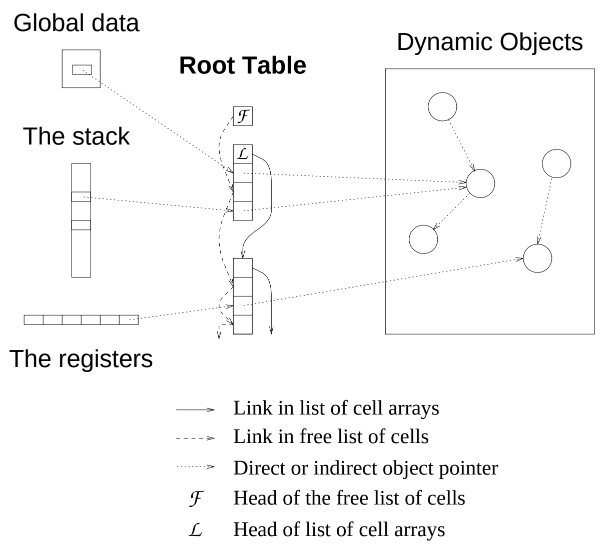
Global data
Root Table
Dynamic Objects
The stack
The registers
$\mathcal{F}$
$\mathcal{L}$
Link in list of cell arrays
Link in free list of cells
Direct or indirect object pointer
$\mathcal{F}$ Head of the free list of cells
$\mathcal{L}$ Head of list of cell arrays

Fig. 1. The root table

to fetch the direct pointer and then dereference the direct pointer. The overloaded assignment operator causes assignment to a root to assign to the direct pointer rather than to the indirect pointer.

Linked list removal usually requires a test and conditional branch to check for the end of the list. In this implementation, however, when a cell is removed from the free list, its value is immediately fetched. That fetch is used to avoid the test and branch. The last page of the last cell array is *read-protected* [3]. Attempting to load the link stored in the first cell on the read-protected page causes the program to receive a signal. The signal handler unprotects the page, links in and initializes a new cell array, and read-protects the last page of the new array. A new diagram of a cell array is presented in Fig. 2; the shaded area illustrates the read-protected region.

Smart Pointer Class Definitions. For every application class two smart pointer classes are generated. One of them emulates pointers to mutable objects and the other emulates pointers to `const` objects. When the application classes are related through inheritance, the precompiler gives the derived class smart pointers user-defined type conversions to the base class smart pointer types. A detailed description of this organization can be found in [17].

The precompiler parses the program to determine what smart pointer classes are needed and writes the classes to a file. Then, the preprocessed and otherwise transformed application code is appended.

A typical smart pointer class is shown in Fig. 3. This shows the smart pointer class for `const` objects. The associated smart pointer class for mutable objects derives from this class.

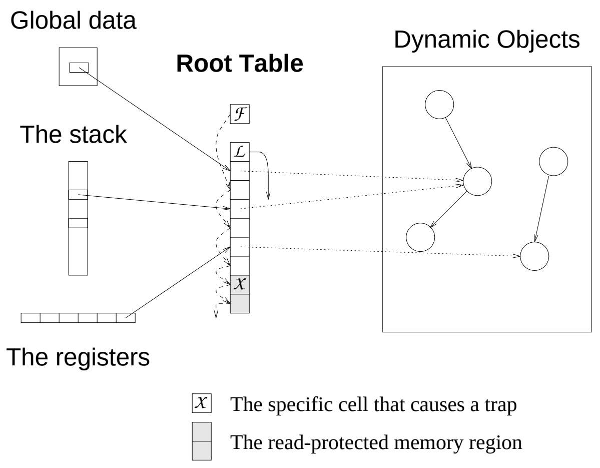

Fig. 2. The protected page of a cell array

The last cell array has its last page read protected. When the protection violation occurs, a new array is allocated and linked to the others.

```
class Root_C_T {
  protected:
    const T * * iptr;  // The indirect pointer

  public:
    const T & operator*()            const { return **iptr; }
    const T * operator->()           const { return *iptr; }
    void operator=(const T * p)            { *iptr = p; }
    void operator=(const Root_C_T r)       { *iptr = *r.iptr; }
    int operator==(const void * vp)  const { return *iptr == vp; }
    int operator==(const T * tp)     const { return *iptr == tp; }
    int operator!=(const void * vp)  const { return *iptr != vp; }
    int operator!=(const T * tp)     const { return *iptr != tp; }
    int operator==(const Root_C_T r) const { return *iptr == *r.iptr; }
    int operator!=(const Root_C_T r) const { return *iptr != *r.iptr; }

    const T * value()                const { return *iptr; }

    Root_C_T()                   { iptr = (T**) _gc_RootTable.pop(); }
    Root_C_T(const T * p)        { iptr = (T**) _gc_RootTable.pop(p); }
    Root_C_T(const Root_C_T & r) { iptr = (T**) _gc_RootTable.pop(*r.iptr); }
    ~Root_C_T()                  { _gc_RootTable.push(iptr); }
};
```

Fig. 3. A smart pointer class for const objects of type T

Smart Pointer Efficiency. Each smart pointer takes up two words in memory, one for the indirect pointer and one for the direct pointer. The actual space overhead is greater than that because the root table grows in increments of 8 kilobytes.

Measurements of the efficiency of these smart pointers show them to be more expensive than raw pointers but less expensive than reference counted pointers [16]. If a global register can be dedicated to the Root Table, then initializing a new smart pointer requires two memory references and destroying one requires one memory reference. Without a dedicated global register, the cost of each of construction and destruction is increased by one memory reference. Accesses through a smart pointer pay a one memory reference penalty due to the level of indirection.

3.2 Locating Internal Pointers

Locating pointers within managed objects is the second task of the precompiler: the precompiler parses type definitions and emits a *gc()* function per garbage-collected type. This function identifies the internal pointer members to the garbage collector.

Internal Pointers and Type Tags. For every managed type the precompiler emits a gc() function. The gc() function invokes an *internal pointer*, or *ip()*, function on every pointer member of an object. The ip() function is global to the program and defined inline for efficiency. As an example, in the existing mark-and-sweep collector, the ip() function pushes internal pointer values onto the mark stack.

The precompiler emits code to register each managed type with the collector. Registration consists of a call to _gc_register() that passes in the type's gc() function pointer. Each such registration causes the garbage collector to generate and return a new type tag. Subsequent memory allocation requests pass in the tag, which is stored in the object's allocator meta-information.

Three type tags are predefined: one for objects that contain no pointers, one for objects that are entirely pointers, and one for *foreign* objects. Foreign objects are only reclaimed manually, i.e. they are never garbage collected, and there is no type information available for them. They are called foreign because they are ignorant of the presence of the garbage collector. Support for foreign objects permits this memory allocator to be the only one in the program; it can satisfy the dynamic memory needs of the standard libraries by treating their allocation calls as requests for foreign objects. Foreign objects are not examined by the collector; they should only reference collected objects through smart pointers, not through raw pointers.

The C++ feature that makes this process convenient is overloadable dynamic storage allocation operators: new and delete. These operators permit every class to supply functions to handle memory allocation and deletion. In this case, operator new for a managed class passes in the type tag to the memory allocator. The default, global operator new passes in the type tag for foreign objects. A call to malloc(), which circumvents new, also allocates a foreign object.

Figure 4 shows some sample input to the precompiler; the transformations for locating internal pointers are shown in Fig. 5.

```
class CL {
private:
  CL    * ptr1;
  OTHER * ptr2;
  static void _gc_finalize(CL *);  /* optional */
  ...
public:
  ...
};
```

Fig. 4. A class with internal pointers

```
class CL {
private:
  CL    * ptr1;
  OTHER * ptr2;
  static void _gc_finalize(CL *);
  ...
public:
  ...
private:
  static _gc_tag_t _gc_tag;
  static void _gc_(CL *);
public:
  void * operator new(size_t sz)   { return malloc(sz,_gc_tag); }
  void   operator delete(void * p) { return free(p); }
};

// The inline ip() function...
inline void _gc_ip_(void * ptr) { _gc_MarkStack.push(ptr); }

// Emitted in exactly one .C file ...
void CL::_gc_(CL * ptr)                    // the type's gc() function
{
   _gc_ip_(ptr->ptr1);
   _gc_ip_(ptr->ptr2);
}

// register type CL with the collector
_gc_tag_t CL::_gc_tag = _gc_register(&CL::_gc_, &CL::_gc_finalize);
```

Fig. 5. The internal pointers transformation

3.3 Finalization

If the programmer specifies a static member function named T::_gc_finalize(T*), then that becomes the finalization function [31] for objects of type T. As in Cedar, finalization can be enabled or disabled for individual objects; the collector maintains a bit with every object indicating whether or not the object needs finalization. By default, finalization is enabled for an object whose class has a finalization function; a library call is available to disable or to re-enable finalization for any object.

There are no restrictions on what a finalization function can do. This means that a finalization function, which is only called when the object is unreachable, may make the object reachable. Therefore, in order not to create dangling references, an object is never reclaimed in a turn when it is finalized; it is only reclaimed after another collection confirms that it is unreachable and that finalization is disabled for it [19, 31].

A finalize function must be static, therefore, it may not be virtual (i.e. dynamically bound). However, since it is allowed to invoke virtual functions, the effect of a virtual finalize function is easily obtained.

3.4 Garbage Collection

The collector divides the heap into blocks that are used to allocate objects of uniform size. Using an integer division operation and knowledge of where blocks begin and end, the collector is able to make a pointer to the interior of an object (an *interior pointer*) point to the beginning of the object. This is potentially expensive because integer division can be expensive on RISC processors. Nonetheless, this ability is needed because a pointer to the beginning of the object is necessary to locate the object's type tag and mark bit. Forbidding interior pointers is impossible, firstly because the collector is sometimes conservative, also because multiple inheritance in C++ is generally implemented using interior pointers.

Garbage collection begins by examining every cell of the root table. For each cell, the collector determines if the cell points to a page that is part of the heap. If so, the value is pushed onto the mark stack. After all the roots have been pushed, the collector begins the marking traversal.

Every time a value is popped from the mark stack, the collector determines whether or not the value points into the heap, and if so, what object it references. The collector fetches the object's mark bit. If the mark bit was already set, the pointer is ignored. Otherwise, the bit is set and the type tag is fetched from the allocator's meta-information. The type tag indexes into an array of type descriptors that contain the gc() and finalization function pointers. The gc() function is called with a pointer to the object; the gc() function pushes the internal pointers onto the mark stack.

After the mark phase, the collector performs finalization and reclamation. For every object, one of three cases is true:

1. The object is unmarked and has finalization enabled: The object is finalized and its finalization bit is unset.

2. The object is allocated, unmarked, has finalization disabled, and is not a foreign object: The object is reclaimed. If the object's page is now empty of objects, then the page is added to the free page list. Otherwise, the object is added to the free list for its size.
3. Neither of the above is true: No action is taken.

After this, garbage collection is finished and the application resumes execution. In the next version this phase will be incremental.

3.5 `this` Pointers

In C++, whenever a method is invoked on an object, a pointer to the object is passed to the method on the stack. This pointer is called the this pointer. Through the this pointer the method can access the object's instance data. These pointers are part of the root set, so the garbage collector should consider them.

There are a number of different ways of finding the this pointers. For example, the precompiler could add a root local variable to every member function and assign the this pointer to the root. This would be invasive and inefficient for small member functions. Another way is to coarsely decode the stack and treat the first argument to every function conservatively in case the argument is a this pointer. This requires information about the stack frame layout that only the compiler has. In particular, the first argument to a function call is not necessarily always placed at a consistent offset in the stack frame. Thus, this either requires knowledge about the stack frame layout for each individual function, or it requires treating virtually the entire stack conservatively.

A pure copying collector must accurately find all pointers during every collection. However, a collector that does not move *all* objects does not necessarily need to find all the pointers. We do not to attempt to locate the this pointers. Instead, the programmer must ensure that the following property is always true:

> *There may not be an object whose* only *reference is through one or more* this *pointers.*

For example, the following code is illegal:

```
int main(void)
{
  (new T)->method_X();
  ...
}
```

This code is invalid because in method_X(), the object's only reference is the this pointer.

If the programmer suspects that this restriction is accidentally being violated, the collector can be configured to scan the stack conservatively in addition to using the root table. Then, the collector can report the presence of pointers on the stack to objects that would otherwise be reclaimed. The debugger can then be used to determine what code is responsible.

3.6 Controlling the Precompiler

By default, all of the class, struct, and union types that the precompiler sees are assumed to be garbage collected. Thus, for all such types, the precompiler performs its two transformations. In fact, a great many of these types are likely not to be managed. For example, while the vast majority of C++ files include the standard header file <iostream.h>, emitting smart pointer types for the iostream classes would unnecessarily slow down compilation because there is no need for them. As an optimization, therefore, there are precompiler-specific #pragmas to control generation of garbage collection information, either at the granularity of the individual type, or at much coarser granularity. (This functionality permits programmers to take control of storage management for certain types if they so choose.)

3.7 Translation Unit Management

The precompiler processes every file in a multi-file program. Therefore, it is likely to see the class definitions multiple times. Some of the transformations are performed every time a file is compiled; others must be performed more selectively. In particular, the modifications to the class definitions are always performed so that all of the code in the program sees the same definitions. However, the gc() functions must not be replicated every time the class definitions are seeing because that would define these functions multiple times.

The precompiler uses the following heuristic to make the decision in most cases: Produce the gc() functions in the same file that defines the first non-inline function of the class. This rule tells the precompiler when to emit the gc() function for every class that has at least one non-inline function. If a managed class does not have a non-inline function member, then the precompiler will issue a warning. The user must then add a #pragma to the class telling the precompiler what file should contain the definition of the class's gc() function. This technique is used in some C++ compilers to determine when to emit the *vtbl* for a class [20, §10.8.1c].

3.8 Status

The design and development of this system are both underway. The smart pointers and the garbage collector are operational. The precompiler has been prototyped using an existing C++ compiler as the starting point. The modified C++ compiler parses the user's C++ code and emits smart pointers and other declarations. The precompiler does not yet reintegrate the emitted code back into the original source program. A complete reimplementation of the precompiler is in progress.

The SOR group at INRIA Rocquencourt has designed and is developing a distributed garbage collection algorithm [35]. The distributed garbage collector requires local garbage collectors with support for finalization. This garbage collector serves as the foundation for the distributed garbage collector.

3.9 Future Work

This collector will be used as a platform for research on the interaction between the collector and the virtual memory system. The areas of future research include

VM synchronized incremental and generational collection, and influencing collection decisions based on the state of the virtual memory system.

4 Conclusions

C++ is a very well designed language considering its goals, however, the complexity of its semantics is daunting. Adding to that complexity by requiring manual storage reclamation makes programming in C++ difficult and error-prone.

Precompiling C++ programs for garbage collection is more convenient for the programmer than a pure library-based approach. Simultaneously, it is portable and not tied to any particular compiler technology. Also, it should reclaim more garbage then a purely conservative approach.

A number of other systems use smart pointers, generally for reference counting. Automatic generation of smart pointer classes can be of benefit to those projects. Similarly, the transformation that locates internal pointers is independent of the implementation of the GC algorithm, and could be used by other C++ garbage collectors.

There are three main benefits to our approach. First, the precompiler can be used as a garbage collector front-end and as a smart pointer generator. Second, this is a convenient platform for research in garbage collection techniques and issues, and will be used as such. Finally, the collector makes programming in C++ less complex and safer, and may make garbage collection available to a large part of the C++ programming community.

Acknowledgements

This work has been supported in part by Esprit project 5279 *Harness*.
I am grateful to **Marc Shapiro** for supporting this work.

References

1. ACM. *Proc. PLDI '91* (June 1991). SIGPLAN Not. 26(6).
2. Appel, A. W. Runtime tags aren't necessary. In *Lisp and Symbolic Computation* (1989), vol. 2, pp. 153–162.
3. Appel, A. W., and Li, K. Virtual memory primitives for user programs. In *ASPLOS Inter. Conf. Architectural Support for Programming Languages and Operating Systems* (Santa Clara, CA (USA), Apr. 1991), pp. 96–107. SIGPLAN Not. 26(4).
4. Bartlett, J. F. Compacting garbage collection with ambiguous roots. Tech. Rep. 88/2, Digital Equipment Corporation, Western Research Laboratory, Palo Alto, California, Feb. 1988.
5. Bartlett, J. F. Mostly copying garbage collection picks up generations and C++. Tech. Rep. TN–12, DEC WRL, Oct. 1989.
6. Boehm, H.-J. Simple gc-safe compilation. Workshop on GC in Object Oriented Systems at OOPSLA '91, 1991.
7. Boehm, H.-J., Demers, A. J., and Shenker, S. Mostly parallel garbage collection. In *Proc. PLDI '91* [1], pp. 157–164. SIGPLAN Not. 26(6).

8. Boehm, H.-J., and Weiser, M. Garbage collection in an uncooperative environment. *Softw. – Pract. Exp. 18*, 9 (Sept. 1988), 807–820.
9. Christopher, T. W. Reference count garbage collection. *Softw. – Pract. Exp. 14*, 6 (1984), 503–508.
10. Coplien, J. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.
11. Demers, A., Weiser, M., Hayes, B., Boehm, H., Bobrow, D., and Shenker, S. Combining generational and conservative garbage collection: Framework and implementations. In *Proc. POPL '90* (Jan. 1990), ACM, ACM, pp. 261–269.
12. Detlefs, D. Concurrent garbage collection for C++. Tech. Rep. CMU-CS-90-119, Carnegie Mellon, 1990.
13. Dickman, P. Trading space for time in the garbage collection of actors. In unpublished form, 1992.
14. Dybvig, K. R. *The SCHEME Programming Language*. Prentice Hall, Englewood Cliffs, N.J., 1987.
15. Dykstra, D. Conventions on the use of ObjectStars, 1992. Private communication.
16. Edelson, D. R. Comparing two garbage collectors for C++. In unpublished form, 1992.
17. Edelson, D. R. Smart pointers: They're smart but they're not pointers. In *Proc. Usenix C++ Technical Conference* (Aug. 1992), Usenix Association, pp. 1–19.
18. Edelson, D. R., and Pohl, I. A copying collector for C++. In *Proc. Usenix C++ Conference* [40], pp. 85–102.
19. Ellis, J. Confirmation of unreachability after finalization, 1992. Private communication.
20. Ellis, M. A., and Stroustrup, B. *The Annotated C++ Reference Manual*. Addison-Wesley, Feb. 1990.
21. Ferreira, P. Garbage collection in C++. Workshop on GC in Object Oriented Systems at OOPSLA '91, July 1991.
22. Ginter, A. Cooperative garbage collectors using smart pointers in the C++ programming language. Master's thesis, Dept. of Computer Science, University of Calgary, Dec. 1991. Tech. Rpt. 91/451/45.
23. Goldberg, B. Tag-free garbage collection for strongly typed programming languages. In *Proc. PLDI '91* [1], pp. 165–176. SIGPLAN Not. 26(6).
24. Grossman, E. Using smart pointers for transparent access to objects on disk or across a network, 1992. Private communication.
25. Kafura, D., Washabaugh, D., and Nelson, J. Garbage collection of actors. In *Proc. OOPSLA/ECOOP* (Oct. 1990), pp. 126–134. SIGPLAN Not. 25(10).
26. Karplus, K. Using if-then-else DAGs for multi-level logic minimization. In *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI* (Pasadena, CA, 20-22 March 1989), C. L. Seitz, Ed., MIT Press, pp. 101–118.
27. Kennedy, B. The features of the object-oriented abstract type hierarchy (OATH). In *Proc. Usenix C++ Conference* [40], pp. 41–50.
28. Madany, P. W., Islam, N., Kougiouris, P., and Campbell, R. H. Reification and reflection in C++: An operating systems perspective. Tech. Rep. UIUCDCS–R–92–1736, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Mar. 1992.
29. Maeder, R. E. A provably correct reference count scheme for a symbolic computation system. In unpublished form, 1992.
30. Miller, J. S. *Multischeme: A Parallel Processing System Based on MIT Scheme*. PhD thesis, MIT, 1987. MIT/LCS/Tech. Rep.-402.
31. Rovner, P. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Tech. Rep. CSL–84–7, Xerox PARC, 1984.

32. Russo, V. Garbage collecting an object-oriented operating system kernel. Workshop on GC in Object Oriented Systems at OOPSLA '91, 1991.
33. Russo, V. There's no free lunch in conservative garbage collection of an operating system, 1991. Private communication.
34. Russo, V. Using the parallel Boehm/Weiser/Demers collector in an operating system, 1991. Private communication.
35. Shapiro, M., Dickman, P., and Plainfossé, D. Robust, distributed references and acyclic garbage collection. In *Symp. on Principles of Distributed Computing* (Vancouver, Canada, Aug. 1992), ACM.
36. Shapiro, M., Gourhant, Y., Habert, S., Mosseri, L., Ruffin, M., and Valot, C. SOS: An object-oriented operating system—assessment and perspectives. *Comput. Syst. 2*, 4 (Dec. 1989), 287–338.
37. Shapiro, M., Maisonneuve, J., and Collet, P. Implementing references as chains of links. In *Workshop on Object Orientation in Operating Systems* (1992). *To appear.*
38. Stroustrup, B. The evolution of C++ 1985 to 1987. In *Proc. Usenix C++ Workshop* (Nov. 1987), Usenix Association, pp. 1–22.
39. Stroustrup, B. *The C++ Programming Language*, 2nd ed. Addison-Wesley, 1991.
40. Usenix Association. *Proc. Usenix C++ Conference* (Apr. 1991).
41. Wentworth, E. P. *An environment for investigating functional languages and implementations.* PhD thesis, University of Port Elizabeth, 1988.
42. Wentworth, E. P. Pitfalls of conservative garbage collection. *Softw. – Pract. Exp.* (July 1990), 719–727.
43. Yasugi, M., and Yonezawa, A. Towards user (application) language-level garbage collection in object-oriented concurrent languages. Workshop on GC in Object Oriented Systems at OOPSLA '91, 1991.
44. Zorn, B. The measured cost of conservative garbage collection. Tech. Rep. CU-CS-573-92, University of Colorado at Boulder, 1992.

This article was processed using the LaTeX macro package with LLNCS style