# A COMPILER-BASED APPROACH TO IMPLEMENTING SMART POINTERS

by

Stephen Hoskins
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Science

Committee:

| | |
|---|---|
| [illegible] | Dr. Elizabeth White, Dissertation Director |
| [illegible] | Dr. Hakan Aydin, Committee Member |
| [illegible] | Dr. Pearl Wang, Committee Member |
| Sanjeev Setia | Dr. Sanjeev Setia, Department Chair |
| [illegible] | Dr. Daniel Menascé, Associate Dean for Research and Graduate Studies |
| [illegible] | Dr. Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering |
| Date: Dec. 3, 2007 | Fall Semester 2007<br>George Mason University<br>Fairfax, VA |

# A Compiler-Based Approach to Implementing Smart Pointers

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Stephen Hoskins
Bachelor of Science
Virginia Polytechnic Institute and State University, 2003

Director: Elizabeth White, Professor
Department of Computer Science

Fall Semester 2007
George Mason University
Fairfax, VA

## DEDICATION

This is dedicated to my girlfriend, Tam Nguyen, who has had to wait patiently for me to finish this paper.

## ACKNOWLEDGEMENTS

I would like to express sincere appreciation to Professor White for her assistance in the preparation of this manuscript.  Furthermore, I would like to express sincere appreciation to all the members of the committee for their invaluable help.

TABLE OF CONTENTS

## LIST OF FIGURES

ABSTRACT

A COMPILER-BASED APPROACH TO IMPLEMENTING SMART POINTERS

Stephen Hoskins, M.S.

George Mason University, 2007

Thesis Director: Dr. Elizabeth White

Because of the growing popularity of programming languages with garbage collectors, such as C# and Java, there is a clearly a desire for languages that support automated memory management. However, as a result of the inefficiencies of the garbage collectors of C# and Java, there is a requirement that programmers have a better understanding of the underlying implementations of the garbage collectors in order to make applications more robust or so that they can run on real-time systems. Using an implementation of smart pointers written from scratch, this paper attempts to address this problem by exploring techniques that are used by garbage collectors and ultimately concluding which features of object-oriented languages make the task of automating efficient garbage collection more difficult. As a result of the conclusions produced in this paper, it may be possible to create a brand new language with the simplicity and elegance of Java and the robustness and efficiency of C without the developer ever needing to perform memory management.

# 1. Introduction

Within recent years, Java has emerged as one of the most popular programming languages to use, with forty-five percent of developers saying that "they use Java for at least some of their coding activities," which is based on a poll of four hundred and thirty developers from the United States and Canada (Morgan, 2006). Furthermore, a Gartner survey shows that seventy-eight percent of all universities have made Java a mandatory course within their computer science and MIS departments (Weinstein, 2001). However, using languages like Java comes with a price.

With the emergence of Java, programmers of real time systems had to become acquainted with the concept of determining optimal times to use garbage collection in order to achieve optimal memory usage. If garbage collection happens to be performed at an inappropriate time, then this could result in the system bogging down during time critical situations, which is completely unacceptable for any real-time system or any system that deals with life or death situations. In order to deal with this situation, many books have been written that explain how to program in Java for real time systems, such as *Real-Time Specification for Java* by Benjamin Brosgol et al. (2000), *Concurrent and Real-Time Programming in Java* by Andrew Wellings (2004), and *Real-Time Java Platform Programming* by Peter Dibble (2002). However, if the memory has been managed properly within the programmer's code using a language like C++, then the programmer has no need to determine the optimal time to perform garbage collection because at any given time that the application is running it will only be using the memory that it needs in order to function properly.

However, the major drawback to using programming languages like C and C++ has always been that they become increasingly burdensome for tracking memory related issues as the systems become increasingly larger and more complex. With the government and corporations spending huge amounts of money for the development time that is needed to work with complex languages like C++ and for memory leak detection software such as Purify, languages like C++ are becoming less desirable. Languages like Java and C# have taken the center stage because they do not burden the programmer with the difficult task of memory management and they require significantly less debugging/development time. Unfortunately this has also resulted with developers settling with producing less robust and memory efficient software systems.

However, by using a compiler-based approach to smart pointers we hope to show that you can have a programming language with the elegance and simplicity of a fast prototyping language like Java and have the efficient memory management that can be achieved with a language like C++, and the programmer does not even have to use pointers! The primary difference with smart pointers being used behind the scenes by the compiler is that the same type of memory management that C++ programmers had to deal with will now be handled for them automatically in the same way that they used to handle memory management themselves, which is much more efficient than the garbage collectors being used by languages like Java and C# today. Therefore, with smart pointers being incorporated into the compiler behind the scenes, we hope to show that programmers will be able to return back to the days of developing memory efficient applications that are not bogged down by “stop-the-world” garbage collectors and they will not even have to worry about performing memory management.

# 2. Background

## GARBAGE COLLECTION

The concept of garbage collection has been around for a long time and it is most notably used by programming languages, such as Java and C#. Its purpose is to identify blocks of memory that have become unreachable by the user program and it serves as a great tool for automating the task of memory management. This alleviates programmers from having to manually perform the task of memory management themselves. However, automating memory management can have an impact on the application being developed.

Some of the things that can be considered when determining the impact of incorporating a garbage collector into an application are:

- **Pause Time**: This is the amount of time it takes to perform garbage collection. The length of the pause time may depend on how much memory needs to be de-allocated. However, it may not be necessary to de-allocate all garbage at once (Goetz, 2003).
- **Throughput**: This is the percentage of the execution time not being used for garbage collection over a long period of time (Davidson, 2005).

- **Pause Predictability**: Will pauses occur consistently or in-consistently? Will pauses occur when most convenient for a user? For instance, in a video game it might be most convenient for pauses associated with garbage collection to occur when the user is waiting for the next level to load (Davidson, 2005).
- **CPU Usage**: What percentage of the total available CPU time is being spent collecting garbage (Davison, 2005)?
- **Memory Footprint**: This refers to the total amount of main memory that is used by an application while it is running. The result of using a garbage collector is that the heap may end up being several times larger than the maximum heap residency of the user program (Goetz, 2003).
- **Memory Reuse**: This raises the question, "How quickly after an object is de-referenced is its memory available for a new object?" (Davison, 2005).
- **Virtual Memory Interaction**: If a computer system has low physical memory, then performing a full garbage cleanup may result with pages being faulted into memory. Because the costs of page faults are high, it is desired that a garbage collector maintain its locality of reference (Goetz, 2003).
- **Cache Interaction**: Garbage collectors may have the effect that they flush out all the data being used by the user program from the cache. This results with a performance cost being imposed on the user program. (Goetz, 2003).
- **Compiler and Runtime Impact**: What is the performance impact of a garbage collector's bookkeeping requirements on the runtime? (Goetz, 2003).

All of the aforementioned heuristics are very important things to consider when assessing how good a particular garbage collector is. However, within this paper, we will be most interested in observing the impact that a garbage collector has during runtime. In particular, we will be measuring the execution time in order to compare how well a garbage collector performs.

## Reference Counting

Reference counting is one of the most straight forward mechanisms for performing garbage collection. Essentially a count of incoming references is stored for each object and whenever the count drops to zero, it is then safe to de-allocate the object. (Boehm, 2004) There are both pros and cons to using this technique.

The pros to using reference counting are (Boehm, 2004):

- It allows memory that is no longer needed to be almost immediately reused.
- It can be a very space-efficient technique.
- Implementations tend to be much simpler than other methods for garbage collection.

However, the biggest con to using reference counting is that it fails to reclaim “cyclic garbage” or referential loops that reside within memory that are no longer accessible (Boehm, 2004). If some kind of a workaround can be implemented to solve this problem that retains all the pros described earlier, then this may prove to be a good way to implement a garbage collector. However, many things will need to be considered in order to come up with a viable solution.

## Smart Pointers

Smart pointers are essentially instances of classes that are wrapped around built-in pointers and are substituted in for basic, language-defined pointers (or raw pointers) in C++ (Eyre-Todd, 1993). This can enable a variety of applications, such as garbage collection, persistence, and distributed objects (Hamilton, 1997). The smart pointer library that is provided by Boost Software provides six smart pointer class templates to choose from for automating garbage collection (Colvin et al., 1999):

- **scoped_ptr**: This smart pointer has sole ownership of single objects and is not copyable.
- **scoped_array**: This has sole ownership of arrays of objects and is also not copyable.
- **shared_ptr**: This allows object ownership to be shared among multiple pointers.
- **shared_array**: This allows an array of objects to be shared among multiple pointers.
- **weak_ptr**: This is for simply observing objects owned by a "shared_ptr" and does not have any ownership privileges.
- **intrusive_ptr**: This is for sharing ownership of objects and has an embedded reference count.

As can be seen from all the smart pointer templates that are provided, there is a lot that programmers must be aware of when deciding which template to use. This places a burden on the programmers because they will have to learn how to use each of these constructs. Furthermore, changes to a software system may require developers to change the templates being used throughout the entire system. Therefore, as a result of using smart pointers, a developer may effectively be doing just as much work as if using regular raw pointers.

Note that the smart pointers being used in this paper will be different than the ones developed by Boost Software. To begin with, the smart pointers being used here will be implemented from scratch. Furthermore, there will not be different templates to choose from. However, the smart pointers in this paper will be similar because they are implemented as classes and they will handle garbage collection, but they will do so with the help of some function calls, which will be described later.

## Mark-sweep Collectors

Mark and sweep collectors are the most basic types of tracing garbage collectors and were first proposed by Lisp inventor, John McCarthy in 1960 (Goetz, 2003). These collectors operate by stopping all program threads, marking any object that is accessible by the user program, and then recursively marking any object that is being pointed to by a marked object. Any objects that are left over that are not marked are then *swept* –i.e. they are reclaimed so that they can be used again by the application for different purposes.

After a mark and sweep has been performed, all the program threads are resumed (Azatchi et al., 2003).

The pros to mark-sweep collectors are that they are simple to implement, they can reclaim cyclic garbage, and they do not place any burden on the compiler like reference counting does.  However, the problem with mark-sweep collectors is that they have really long pause times and they sweep through the entire heap, which may result with a lot of page faults occurring.  Therefore, there is a cost that you must pay for the simplicity that comes with mark-sweep collectors.  (Goetz, 2003).

**Copying Collectors**

Copying collectors are another form of tracing collectors where two memory spaces of equal size are required.  These spaces may be denoted as *From-Space* and *To-Space*. Allocation of objects always makes use of memory from the *From-Space*.  If it is determined that no more memory can be allocated from the *From-Space*, then garbage collection occurs.  (Huelsbergen et al., 1993).

When garbage collection occurs, a breadth-first search for live objects is performed on the *From-Space* and they are then copied to the *To-Space*.  After this occurs, the roles of the spaces are reversed and memory allocation of objects is performed on the *To-Space*. This has the benefit that only live objects will need to be examined during a trace and garbage can simply be ignored.  (Huelsbergen et al., 1993).

Another great benefit of copying collectors is that all live objects are copied into the bottom of the memory spaces. This is good because this improves the locality of reference of the user program and it helps eliminate heap fragmentation. Furthermore, this is good because whenever new memory needs to be allocated, this can be taken from the next available block of memory on the active memory space, which is just simple pointer arithmetic –i.e., memory does not actually need to be allocated because it is already available for use. Therefore, performing memory allocation in a language that uses a copying collector may be significantly cheaper than performing manual memory allocation in a language in C or C++. (Goetz, 2003).

However, the obvious problem with copying collectors is that they require twice the heap space. Furthermore, copying collectors have the added cost of copying objects from one memory space to another. This also means that references will have to be updated so that they are pointing to the new copies of the objects. Therefore, there is some significant overhead associated with using copying collectors. (Goetz, 2003).

**Mark-compact Collectors**

Mark-compact collectors combine both the techniques of mark-sweep collectors and copying collectors. Like the mark-sweep collectors, mark-compact collectors mark all objects that are accessible by the user program. However, instead of performing a sweep afterwards, it copies the live objects in such a way that they get compacted at the bottom of the memory space.  This just requires a little bit more complexity than copying the objects onto another memory space.  (Goetz, 2003).

Like copying collectors, mark-compact collectors only need to examine live objects and can just ignore garbage.  Also like copying collectors, mark-compact collectors can improve the locality of reference of a user program, help eliminate heap fragmentation, and make allocation of memory a simple procedure.  However, in addition to having all the same advantages as copying collectors, mark-compact collectors do not need twice the memory space.  Also, since long-lived objects tend to accumulate at the bottom of the memory space, there is no need to repeatedly copy these objects.  (Goetz, 2003).

The disadvantage to using mark-compact collectors is that they may or may not take longer to copy live objects than copying collectors based on how many long-lived objects there are.  If there are not many long-lived objects, then the mark-compact collector may take longer if it has to ensure that it does not overwrite other live objects on the memory space when doing compaction.  Furthermore, for the objects that need to be copied, references pointing to these objects will have to be updated to point to the new copies.  Therefore, there is still some overhead associated with using mark-compact collectors.

## PROGRAMMING LANGUAGES THAT FACILITATE AUTOMATED GARBAGE COLLECTION

Many programming languages have incorporated garbage collectors into their software so that programmers would not have to be burdened with performing memory management manually. Each of these languages has its own strengths and weaknesses associated with it. This section takes a brief look at some of the languages that provide garbage collection.

### Java

Java is quite possibly one of the best languages ever created in the 20th century (aside from its highly influential predecessors, such as Smalltalk and C++). Java's application programming interface (API) provides programmers with a huge selection of packages that contain everything from hash tables to GUI components so that programmers do not have to constantly reinvent the wheel in order to use some of the most basic code constructs within their software. Also, its consistency with naming conventions and API designs help to ensure that programmers will know what to expect when they need to incorporate another component of Java's API into their software system. Java's exceptional error handling system also makes debugging in Java much easier than in programming languages such as C. However, by far the most incredible feature that Java offers is its garbage collector, which eliminates the gruesome task of performing memory management.

However, as was mentioned in the Introduction to this paper, in order for developers to use Java for the development of time-critical applications or applications that can be used on real-time systems, there is a lot that they have to be aware of. In general, it is arguably the case that there is a lot that a developer must be aware of no matter which programming language is being used when developing time-critical applications or when developing on real-time systems. However, it would be more preferable if developers were not burdened with needing to have an understanding of the underlying implementation of the Java garbage collector when developing these kinds of applications and instead the language provided a solution to this problem to begin with.

**D**

D is a systems programming language that attempts to combine the power of a high performance language like C with the elegance of a programming language like Java. Like Java, D has a garbage collector. Furthermore, D is syntactically very similar to Java. However, unlike Java, D is compiled down directly to native code. So this language attempts to achieve the power and performance of a language like C by eliminating any virtual machines or interpreters that are used by languages like Java. (Digital Mars, 2007).

However, the problem with D is that there may still be performance issues as a result of its garbage collector. Because it has a garbage collector, any of the issues discussed in the previous sections could arise depending on which techniques that it is using.

However, if it could use a garbage collector that does not suffer from any of these problems, then this language could perhaps be used as a very powerful alternative to Java.

**C#**

Microsoft Visual C# is an attempt at trying to unify the elegance of Java with the robustness of C++. Like Java, C# uses a tracing collector. However, unlike Java, the implementation for its garbage collector is left proprietary. Pointers can be used, however they must be used within blocks of code that are marked as “unsafe” and the memory that is allocated to these pointers cannot be freed explicitly. Furthermore, Microsoft Visual C# is compiled down to a Common Intermediate Language (CIL) that subsequently runs on top of what is called the Common Language Runtime (CLR) (Archer et al., 2002). This is similar to the concept behind the Java Virtual machine and may further hinder the performance of the overall application. Therefore, it seems that C# has just more or less tried to mimic Java, rather than attempting to create a language that truly has the robustness of a language like C or C++.

## STATEMENT OF PROBLEM

For each technique that is described in the section, Garbage Collection, there exists a particular set of problems that must be dealt with. With reference counting, there is the problem of detecting cyclic garbage. With smart pointers, there is the large learning curve involved with using the smart pointer templates. With tracing collectors, such as mark-sweep, copying, and mark-compact, there are various performance problems that can arise when embedding them into applications. Lastly, for each language discussed in the previous section (Programming Languages that Facilitate Automated Garbage Collection), there are considerations that must be taken into account, such as, "What are the consequences of having an embedded garbage collector within the application that I develop?"

## PURPOSE OF STUDY

The purpose of this study is to get hands-on experience with implementing a garbage collector from scratch in order to get a better understanding of the implementation issues that must be dealt with. It is hoped that by doing this research, a new approach for handling garbage collection can be invented. Furthermore, it is desired that the techniques found will enable the memory management to be as efficient as handling the memory management manually in a language like C or C++.

# 3. Attempting to Build a Better Garbage Collector

In the previous chapter we discussed a brief history of garbage collection and many of the popular programming languages that relinquish the burden of managing memory for programmers. Despite the fact that a lot of progress has been made with "stop-the-world" garbage collectors, they still create problems when developing applications where timing is crucial. If we can possibly find a way to use the reference counting approach within our own implementation of smart pointers so that it would not suffer from the problems associated with cyclic garbage, then we can possibly produce a compiler that is as easy to use as Java, but that runs more efficiently and does not burden the programmer with learning how to manage the garbage collector when programming time-critical applications.

In this chapter, we shall explore using the reference counting approach in constructs very similar to smart pointers in order to perform garbage collection. However, instead of passing the burden of learning how to use smart pointers onto the programmer, we will make the smart pointers transparent to the programmer by making a compiler that determines how to best use smart pointers within the code that was written by the programmer. Note that making changes to the programmer's code so that smart pointers are used both efficiently and effectively is a compile-time issue whereas de-allocating memory that is no longer useful to the application at the appropriate time while the program is running is a runtime issue. By exploring the difficulties encountered with this approach it is hoped that solutions can be developed to overcome the obstacles encountered with efficient and effective automated memory management.

## OVERVIEW OF THE BUILD PROCESS

The language that we will be using will be referred to as the F language and it is assumed to be Java-esque –i.e., everything will be defined in classes using C++-styled syntax. Furthermore, this language will facilitate all the common object-oriented features, such as inheritance, polymorphism, and information hiding. Also, much like Java, it will be assumed that every class type will inherit methods and attributes from a base Object type. This will be important to know later on because the base Object type will contain all the methods for adding and removing references and determining if its allocated memory has become inaccessible and must therefore be garbage collected. In order to get a complete description of what the language will be like, refer to Appendix A: F Language Reference Guide.

Code written in F will be compiled using an F language compiler that was implemented in lex & yacc, which will generate C++ code containing additional code for tracking references and performing garbage collection. Furthermore, every variable that existed in the F language code will be converted into a smart pointer. However, in order to compile the generated C++, additional header files will be needed because the generated C++ will contain "#include" statements that import header files containing implementations for performing garbage collection. After the C++ has been generated, the generated code along with additional header files will be compiled into an executable image using Microsoft Visual Studio 2005. The resulting executable image can be tested via a command prompt. An overview of the build process for producing an executable image from F language code is presented in Figure 1.

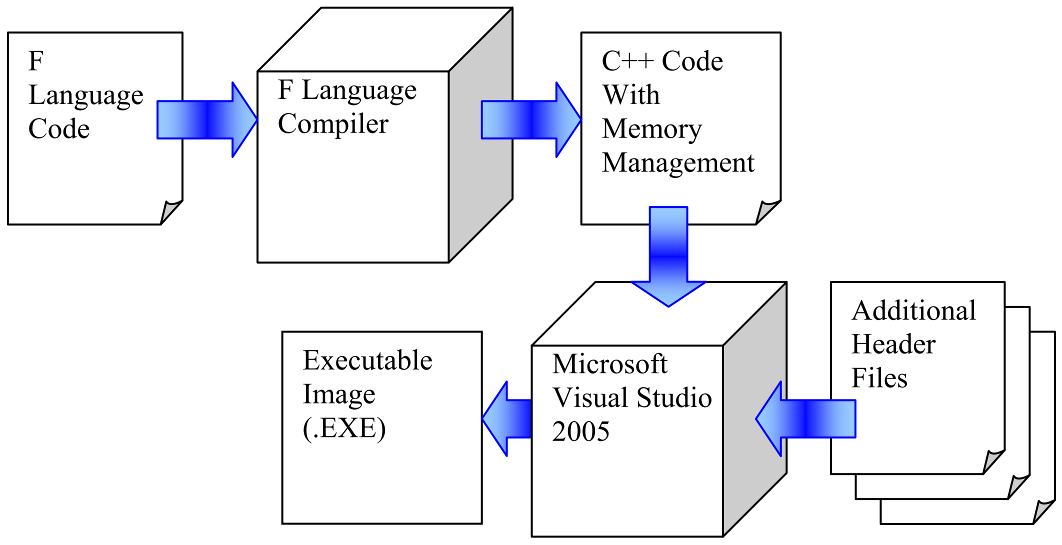
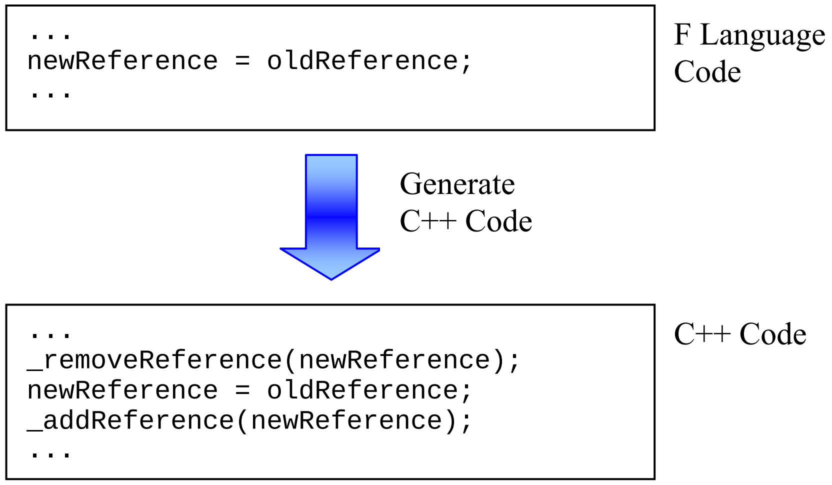


Figure 1 – Overview of F Language Compilation Process

## REFERENCE COUNTING

As shown in Figure 1 from the previous section, the F language compiler will be responsible for producing C++ from the F language code and the generated C++ will need to contain code for performing memory management. However, many things will need to be considered for automating the task of memory management. To begin with, we will assume that each object, which is incidentally also a smart pointer, is holding on to a reference count, which is stored in the base class type, Object, and is inherited by every object in the F language. However, as we will see, this method will prove to be inadequate and we may have to store more information in order to appropriately de-allocate memory that is inaccessible by the user program.

The reference count for each object will need to be updated during runtime in order for it to be determined whether or not garbage collection should be performed. The number of references that an object has pointing to it can be determined by examining assignment statements within the code. So whenever an assignment statement occurs within the programmer's code, the reference count for the previous object being referenced by the variable on the left hand side of the assignment statement will have its reference count decremented (or left alone if there was no previous object) and the new object that the variable is referencing will have its reference count incremented. The reference counting will be handled in this way because the object that was previously being referenced by the variable on the left hand side of the assignment will no longer be pointed to by this variable. Therefore, it follows that the previously referenced object should have its reference count decremented. Similarly, the new object that the variable is referencing should have its reference count incremented. Figure 2 illustrates how the programmer's code can be compiled down into C++ to handle incrementing and decrementing the reference count.



```
...
newReference = oldReference;
...
```

```
...
_removeReference(newReference);
newReference = oldReference;
_addReference(newReference);
...
```

Figure 2 – Generating C++ From F Language Code Using Reference Counting.

The functions, _removeReference and _addReference, will be used to decrement and increment the reference count for the variable, newReference, respectively. Furthermore, the reference count will be stored as a data member, _numReferences, in each object (although this cannot be seen in Figure 2). Also, even though it cannot been seen in Figure 2, all the variables in the programmer's code will be represented as pointers in the generated C++.

The code for incrementing and decrementing the reference count is more complicated than using the simple ++ and -- operators available in C++ on _numReferences because if the reference count drops to zero, garbage collection will have to be performed. Also, it will be shown later that other checks will have to be performed on newReference before updates to its data member, _referenceCount, can be made. Therefore, it is preferred that functions, _removeReference and _addReference, be used for updating the reference counts for objects. Figure 3 gives the code for _removeReference and _addReference.

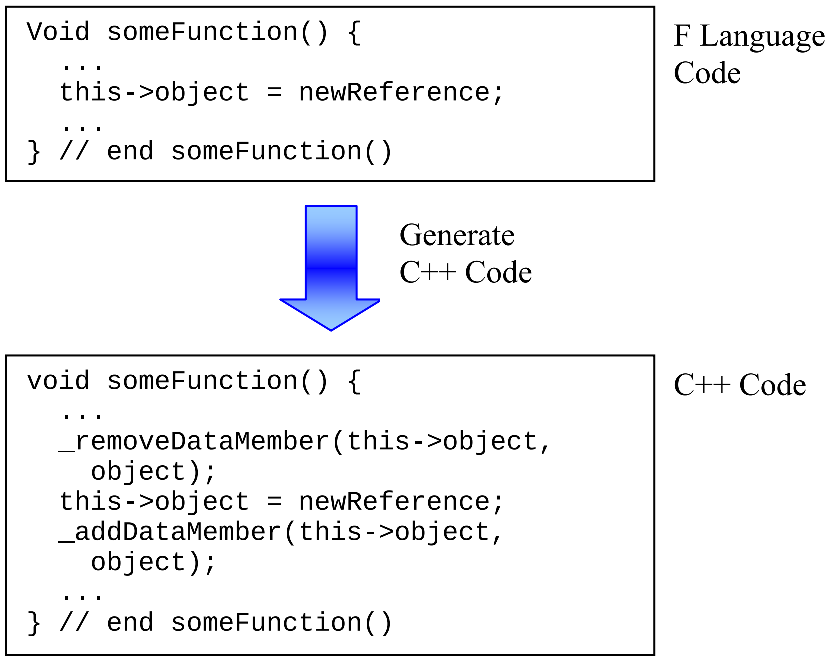
```
/**
 * Updates the number of references that exist for the given object.
 * NOTE: Do not use this function.  Call the other _removeReference()
 * function instead.
 * @param object the given object
 * @param visitedReferences stores all the objects that are visited from
 * recursive calls made to _removeReference()
 */
void _removeReference(Object *object, hash_map<Object*, bool> *visitedReferences) {

  if (object == NULL) return;
  (*visitedReferences)[object] = true;
  object->_numReferences--;
  if (object->_numReferences == 0) {
      hash_map<Object*, int>::iterator iter = object->_dataMembers.begin();
      while (iter != object->_dataMembers.end()) {
        if (visitedReferences->find(iter->first) ==
            visitedReferences->end()) {
          _removeReference(iter->first, visitedReferences);
        }
        object->_dataMembers.erase(iter->first);
        iter = object->_dataMembers.begin();
      }
      delete object;
  }
} // end _removeReference()

/**
 * Updates the number of references that exist for the given object.
 * @param object the given object
 */
void _removeReference(Object *object) {
  _removeReference(object, new hash_map<Object*, bool>());
} // end _removeReference()

/**
 * Updates the number of references that exist for the given object.
 * @param object the given object
 */
void _addReference(Object *object) {
  if (object == null) return;
  object->_numReferences++;
} // end _addReference()
```

Figure 3 – C++ Code For the Functions, _removeReference() and _addReference()

The first function contains the actual implementation for decrementing the reference count and performing garbage collection if needed. However, this function is not directly called from the C++ that is generated from the F language code. The parameter variable, visitedReferences, contains references to all the objects visited and is used to ensure that this function will not get caught in an infinite loop when it makes recursive calls to itself. First, this function stores a reference to the given object being examined in visitedReferences. Afterwards, it decrements the given object's reference count. If the reference count is equal to zero afterwards, it iterates through all the given object's data members' references stored in _dataMembers and makes a call to _removeReference for each reference if the reference has not already been stored in visitedReferences. Next, it removes the data member reference from _dataMembers and the function makes a call to delete the given object.

The second function defines the _removeReference that is called from the generated code. This function merely creates new hash maps to store visited nodes that are subsequently passed to the first definition of _removeReferences. The main program logic for decrementing the reference count is not stored in this function.

In order for an object to appropriately notify all its data members that it no longer has a reference pointing to them, a traversal through the object's data member references stored in _dataMembers will have to be performed.  However, in order to ensure that _dataMembers will contain the references to its data members, _dataMembers will have to be appropriately populated.  Whenever the variable in the left hand side of an assignment statement happens to be a member variable of an object, we know that whatever the member variable is referencing after the assignment can be stored in _dataMembers.  Furthermore, whatever the member variable was referencing before the assignment will have to be removed from _dataMembers.  Figure 4 gives an example of how this can be performed within the generated C++.



```
Void someFunction() {
  ...
  this->object = newReference;
  ...
} // end someFunction()
```

```
void someFunction() {
  ...
  _removeDataMember(this->object,
    object);
  this->object = newReference;
  _addDataMember(this->object,
    object);
  ...
} // end someFunction()
```

Figure 4 – Example Where References Are
Added and Removed From _dataMembers

Once again, because of the complexity of the code, function calls are being used to perform the desired operations. In Figure 4, the function, _removeDataMember, removes the previous reference being stored by the variable, this->object, from the hash map, _dataMembers, that is stored inside of the variable, object. The function, _addDataMember, adds the new reference being stored by the variable, this->object, into _dataMembers. The member variable, _dataMembers can be implemented as a hash map, where the key set is data member references and the corresponding value set is Booleans. The code for _removeDataMember and _addDataMember is shown in Figure 5.

```
/**
 * This removes the given object to the given dependent's _dataMembers.
 */
void _removeDataMember(Object *object, Object *dependent) {
  if (dependent != NULL) {
    dependent->_dataMembers.erase(object);
  }
} // end _removeDataMember()

/**
 * This adds the given object to the given dependent's _dataMembers.
 */
void _addDataMember(Object *object, Object *dependent) {
  if (dependent != NULL) {
    dependent->_dataMembers[object] = true;
  }
} // end _addDataMember()
```

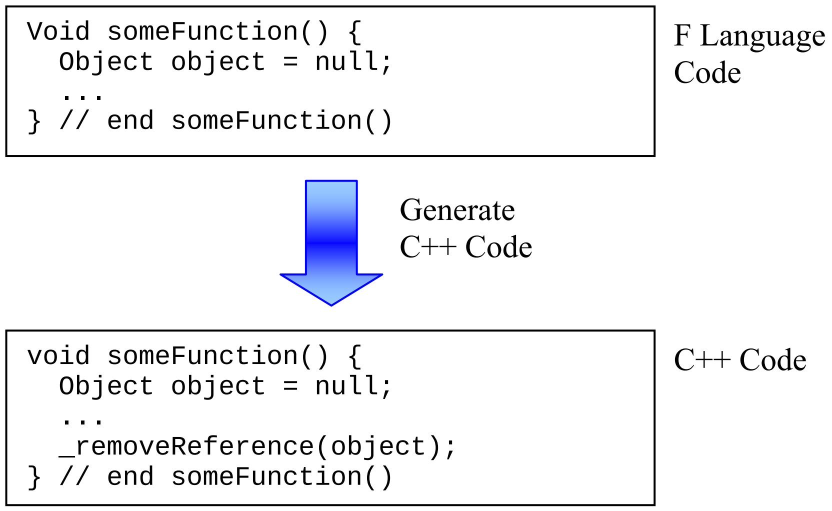
Figure 5 – C++ Source For the Functions, _removeDataMember() and _addDataMember()

It should be noted that if _dataMembers is a linked list, then this would require a search to be performed every time a particular reference needed to be removed from the list, which is what would have to be performed every time there is an assignment statement that deals with data members. This search would have a time complexity of $T(N) \equiv O(N)$, where N is the number of data members. However, it is expected that assignment statements be performed in constant time because virtually every programming language performs assignment statements in constant time, and it is desired that the F language be no exception. Therefore, _dataMembers is being implemented as a hash map to help ensure that searches for references be executed in constant time, which would also help ensure that assignment statements dealing with data members would also be performed in constant time.

It should be noted that there are inefficiencies with implementing _dataMembers as a hash map as well. In particular, what happens when the probability of collisions occurring increases when inserting new elements into the hash map? Then it may take longer to insert into the hash map (Shaffer, 1997). Furthermore, what would happen if the hash map needed to be resized as a result of running out of unoccupied positions? Time would need to be spent for this as well. There is a much better solution for storing data members than using hash maps, which is discussed in Appendix H: A Better Solution to De-allocating Member Variables. However, within the implementation of the F language compiler, a hash map implementation for storing references to data members is currently being used.

Notice that in the definitions for _removeReference, _addReference, _removeDataMember, and _addDataMember that there are if statements to ensure that the variable, object, is not equal to NULL before attempting to access its data members. Whenever a variable is not pointing to an instance of its class type, it will be assigned the NULL value. NULL is simply a constant with the value of zero. If it turns out that a reference is equal to NULL, then it means that no memory has been allocated to it and referencing the data member, _referenceCount, would result in an access violation occurring. Therefore, this check must be performed in order to ensure that an access violation will not occur when attempting to update the reference count of an object.

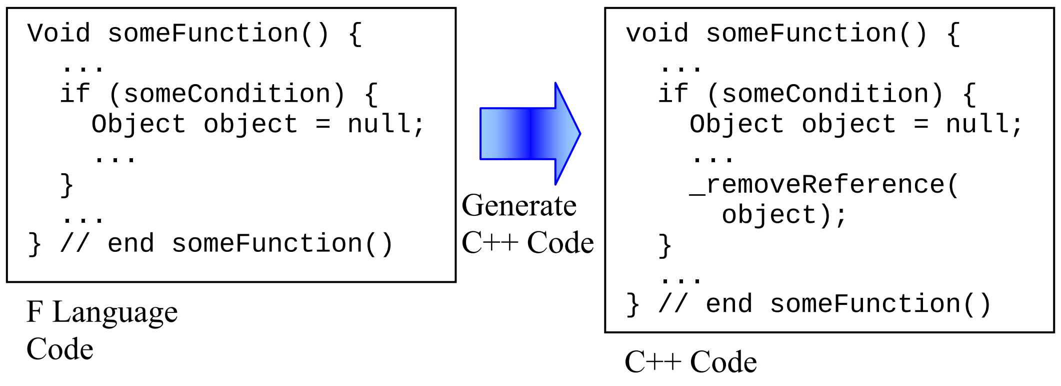
The other thing that has to be looked out for with each variable is if it falls out of scope within a function or if a function terminates, unless the variable is being returned by the function. If the variable is returned by the function, then the reference count will not be decremented until after the line of code where the function was called. This gives the calling function a chance to add its own references and increment the reference count for the variable before its memory is de-allocated (assuming the variable has no other references pointing to it). Figure 6 - Figure 8 give examples of this.



```
Void someFunction() {
  Object object = null;
  ...
} // end someFunction()
```

```
void someFunction() {
  Object object = null;
  ...
  _removeReference(object);
} // end someFunction()
```

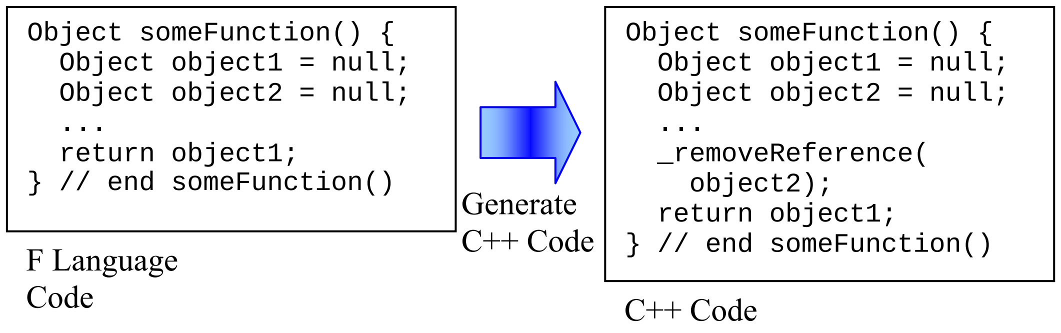
Figure 6 – Example Where Reference Count is Decremented When a Function Terminates



```
Void someFunction() {
  ...
  if (someCondition) {
    Object object = null;
    ...
  }
  ...
} // end someFunction()
```

```
void someFunction() {
  ...
  if (someCondition) {
    Object object = null;
    ...
    _removeReference(
      object);
  }
  ...
} // end someFunction()
```

Figure 7 – Example Where Reference Count is Decremented When a Variable Falls Out of Scope



```
Object someFunction() {
  Object object1 = null;
  Object object2 = null;
  ...
  return object1;
} // end someFunction()
```

```
Object someFunction() {
  Object object1 = null;
  Object object2 = null;
  ...
  _removeReference(
    object2);
  return object1;
} // end someFunction()
```

Figure 8 – Example Where Reference Count is Decremented When a Function Terminates Except If a Variable Is Being Returned

The benefit to using this approach to keep track of references is that the implementation is very simple. However, the problem with using this approach is that it does not properly determine when to garbage collect in all cases. In particular, this method is unable to detect objects that are inaccessible by the user program when they are being referenced within referential loops.

## REFERENTIAL LOOP PROBLEM

Consider for a moment that references were being kept track of by simply maintaining a reference count within each object that stores the number of objects that happen to be referencing them (as described in the previous section). Then there would be a problem with determining when to de-allocate memory using this method. For instance, what if the programmer decided to create a singly-circularly-linked list? Generally this would entail that the linked list structure would store a reference to the first node in the list. However, the last node in the list would also contain a reference that points back to the first node in the list. This can result in the formation of objects that is shown in Figure 9.

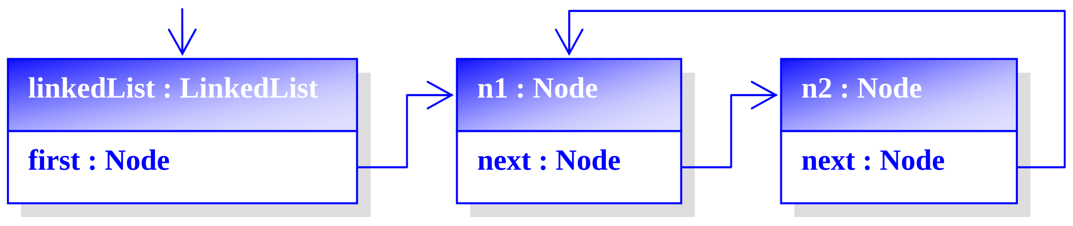


Figure 9 – A Singly-Circularly-Linked List with Two Elements

Now, consider what would happen if the reference count for the linked list went to zero. Since it would no longer contain any references, it would then begin the process of destructing itself and it would then make calls to each of its member data notifying the data that they each have one less reference. This would result with the first node (or n1 in Figure 9) being notified since it is member data. However, the first node in the list would have a reference count of two since both the linked list and the last node in the list are referencing it. This means that after being notified, its reference count would be reduced to one. Therefore, since its reference count would not be zero, the first node would not begin the process of destructing itself. Therefore the first and last nodes would continue to have memory allocated to them. However, since the linked list structure destructed itself, these nodes would be inaccessible since there would be no references pointing to them other than the references that they have pointing to themselves. This would result in a memory leak (see Figure 10).

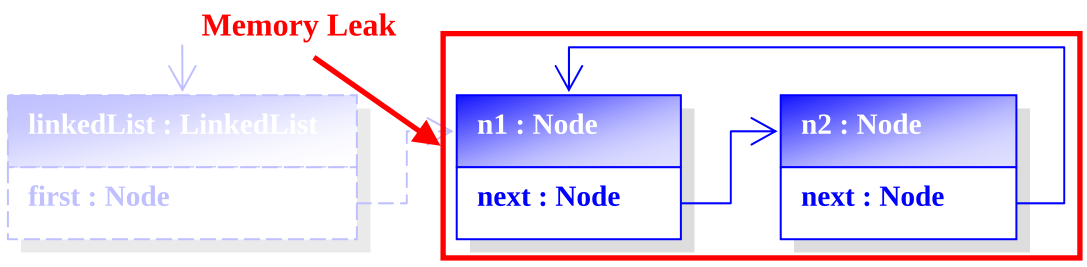


Figure 10 – Memory Leak Formed by Circularly-
Linked List

One way to prevent this from happening is to restrict the programmer from being able to create cyclic references. This can be achieved by disallowing classes from having attributes with types that are the same as the class that they are defined in. So if you have a class, A, then it may not have any attributes defined with the type, A.  This would prevent self-referential loops from occurring with objects and this would prevent the sort of design that was used to create the linked list shown in Figure 9.

However, this would not prevent a more convoluted design where the nodes in the list would alternate between different types.  So the first node would contain a reference to a node of a different type and the second node would contain a reference to a node that is the same type as the first node.  This would require that the last node be a different type than the first node so that the last node could reference the first node.  Figure 11 gives an example of how this would be done.

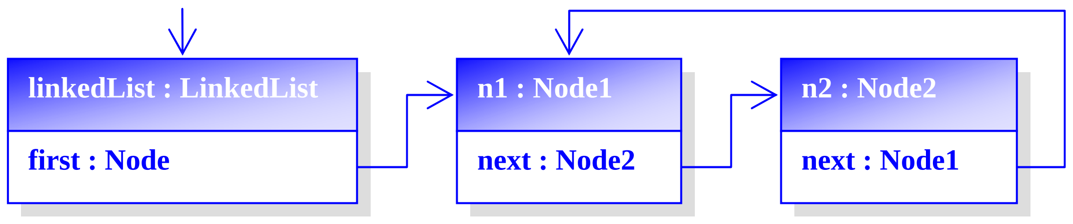


Figure 11 – A Singly-Cicularly-Linked List with Alternating Node Types

More restrictions could be placed on the programmer to prevent this sort of thing from happening.  The programming language can require that the definition of a particular class be placed before it gets used and disallow forward declarations.  Since the class, Node2, has an attribute of type Node1, Node1 will have to be defined before Node2. However, since Node1 has an attribute of type Node2, Node2 will have to be defined before Node1. However, since forward declarations are not allowed, this would be impossible and the structure shown in Figure 11 could not exist.

However, assuming that polymorphism is a component of this language, these restrictions still would not prevent yet another convoluted design where the nodes in the list would reference each other through attributes of type Object. So each node in the list would contain a reference to another node through a reference of type Object. This would require that these references be typecast to a class of type Node in order to access the methods specific to Node. However, regardless of the annoyance that this could cause from constantly having to typecast, this would still be allowed and would provide a way for cyclic references to occur. Figure 12 gives an example of this.

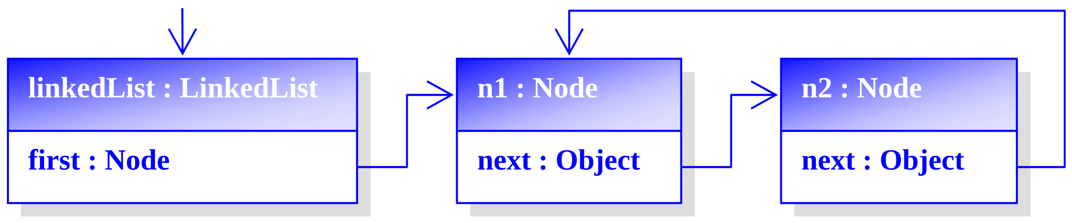


Figure 12 – A Singly-Circularly-Linked List with
Polymorphic References to Neighboring Nodes

More restrictions could yet again be placed on the programmer to prevent this sort of a thing from occurring. A check can be performed on assignment operations dealing with attributes where polymorphism is being performed. If the type of a variable being assigned to an attribute is the same type as the class that contains the attribute, then a compiler error could be generated. This would prohibit the programmer from creating the cyclic structure shown in Figure 12.

However, it should now be apparent that this language is becoming increasingly more restrictive for the programmer and as more loopholes are discovered the restrictions may only get worse. Even if no other loopholes are discovered, the restrictions that were just discussed might already be considered unacceptable for meeting the needs of many programmers. Also, even if this language did turn out to be extremely efficient, the lack of flexibility might cause programmers to still continue using languages like Java. In fact, if developers were only concerned about efficiency, then all of the software systems that are being developed today would still be implemented in straight assembly. However, this is clearly not the case. Therefore, it is desired that this language not be too restrictive with its capabilities.

## STRONG AND WEAK REFERENCES

One way to handle the referential loop problem is to use what are called strong and weak references. Since loops can exist that do not have any references pointing to them from the user program, the references that they do possess can be considered weak references. Any references to objects that come from the user program will be considered strong references. Whenever it is detected that an object cannot be traced to the user program, it is de-allocated.

Note that this technique can be integrated with the reference counting techniques that were discussed in the section, Reference Counting. In fact, we will still be using the same techniques for performing tracing and garbage collection that were used for reference counting, however we will be adding some additional checks to the code and we will classify certain references as being "strong" or "weak," which will require code that is more specialized for handling each reference type. So, it should be known that the techniques learned from reference counting will not be totally abandoned.

There will be a slight difference with the way that references are depicted in figures to accommodate representing strong and weak references. Strong references will be represented in figures with edges that are labeled with an 'S'. All other edges are assumed to be weak references. An example of how strong references and weak references are notated is shown in Figure 13.

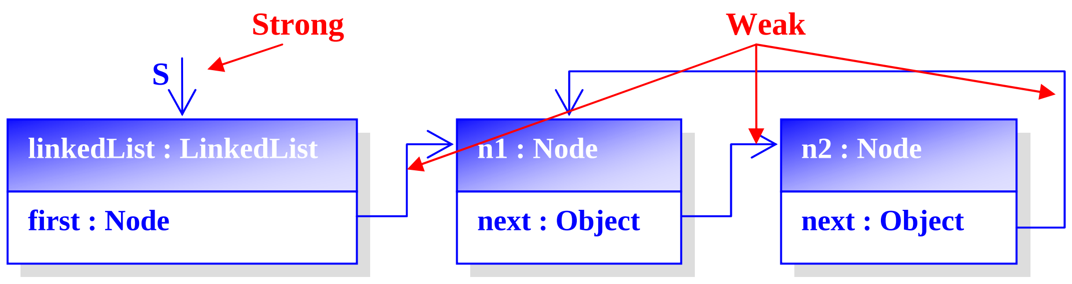


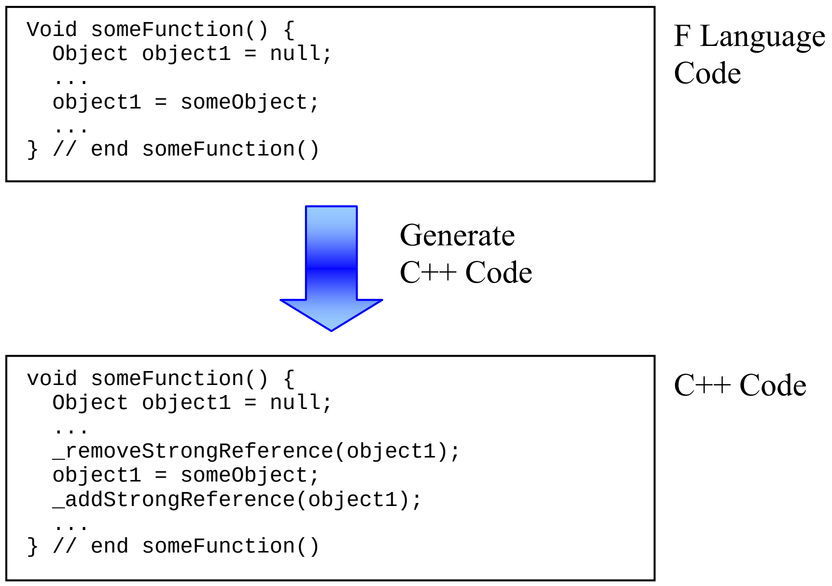
Figure 13 – An Example of Both Strong and Weak References

Since weak references are not the deciding factor as to whether or not an object should be de-allocated, it is important to keep track of where these references are coming from. The reason is because if it turns out that an object only has weak references pointing to it, then it must inspect the objects that are referencing it and determine if they have strong references. If they do not contain any strong references, then their weak references will also have to be inspected for strong references. This traversal will have to continue until at least one strong reference can be found because otherwise the memory will have to be de-allocated. Therefore, it is important that each object maintains what all the objects are that have weak references pointing to it and not just contain a reference count of weak references pointing to it.

Whenever it is determined that an object no longer has any strong references pointing to it, it must send a message to each of its attributes informing them that it no longer contains weak references to them. Each attribute that receives this message must in turn remove the object from its list of objects that are referencing it. After every attribute has been properly notified that the object is no longer referencing them, the object may de-allocate itself. If it turns out that the attributes can no longer be traced to a strong reference, then they too must send messages out to their attributes and de-allocate themselves after every attribute has been notified.

Strong references are added and removed from objects in the same way as described in the previous section. That is, strong references are added and removed as a result of assignment statements, variables falling out of scope, and functions terminating. However, weak references are only added and removed as a result of assignment statements. Furthermore, weak references are added only when the variable on the left hand side of the assignment is a member variable of an object. For all other assignments, strong references are added and removed.

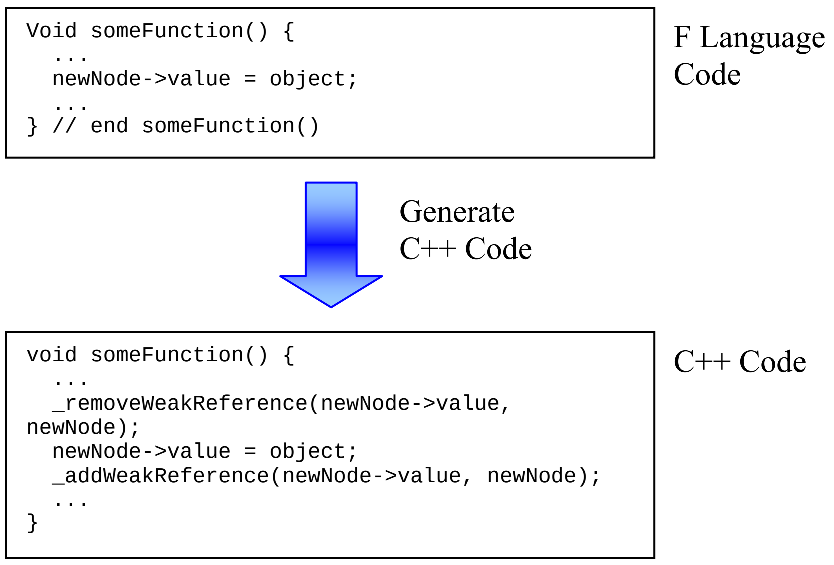
In order to update strong and weak references we will be using the functions, _addStrongReference, _removeStrongReference, _addWeakReference, and _removeWeakReference. Since weak references will be created whenever assignments are performed with data members, _addWeakReference and _removeWeakReference will also have to deal with populating and depopulating _dataMembers. However, since strong references will only be references from the user program, _addStrongReference and _removeStrongReference will not need to deal with populating and depopulating _dataMembers. Examples of what the generated code will look like are presented in Figure 14 - Figure 15.



```
Void someFunction() {
  Object object1 = null;
  ...
  object1 = someObject;
  ...
} // end someFunction()
```

```
void someFunction() {
  Object object1 = null;
  ...
  _removeStrongReference(object1);
  object1 = someObject;
  _addStrongReference(object1);
  ...
} // end someFunction()
```

Figure 14 – Example Where Strong References Are Used



```
Void someFunction() {
  ...
  newNode->value = object;
  ...
} // end someFunction()
```

```
void someFunction() {
  ...
  _removeWeakReference(newNode->value,
newNode);
  newNode->value = object;
  _addWeakReference(newNode->value, newNode);
  ...
}
```

Figure 15 – Example Where Weak References Are Used

In Figure 15 _removeWeakReference and _addWeakReference are added to the assignment statement that was written in the F language code. This is because the variable, newNode->value, on the left hand side of the assignment statement is a member variable. The object that will contain a weak reference to this variable will be newNode since newNode->value is its member data. This weak reference will be stored in the hash map, _dataMembers, for newNode and in the hash map, _weakReferences, for newNode->value. The code for _addStrongReference, _removeStrongReference, _addWeakReference, and _removeWeakReference are presented in Figure 16

```
/**
 * Stores that the given object has a weak reference to it by the given
 * dependent.
 * @param object the given object
 * @param dependent the given dependent
 */
void _addWeakReference(Object *object, Object *dependent) {

  if (object == null || dependent == null) return;

  if (object->_weakReferences.find(dependent) ==
      object->_weakReferences.end())
  {
    object->_weakReferences[dependent] = 0;
  }
  object->_weakReferences[dependent] = object->_weakReferences[dependent] + 1;

  if (dependent->_dataMembers.find(object) ==
      dependent->_dataMembers.end())
  {
    dependent->_dataMembers[object] = 0;
  }
  dependent->_dataMembers[object] = dependent->_dataMembers[object] + 1;
} // end _addWeakReference()

/**
 * Stores that the given object no longer has a weak reference to it by the
 * given dependent.  NOTE: Do not use this function.  Call the other
 * _removeWeakReference() function instead.
 * @param object the given object
 * @param dependent the given dependent
 * @param visitedReferences stores all the objects that are visited from
 * recursive calls made to _removeWeakReference() and _hasStrongReference()
 * @param deletedReferences stores all the objects that are deleted from
 * recursive calls made to _removeWeakReference()
 */
void _removeWeakReference(Object *object, Object *dependent,
    hash_map<Object*, bool, ObjectHasher> *visitedReferences,
    hash_map<Object*, bool, ObjectHasher> *deletedReferences) {

  if (object == null || dependent == null) return;

  int numReferences =
    dependent->_dataMembers[object] = dependent->_dataMembers[object] - 1;
  if (numReferences <= 0) {
    dependent->_dataMembers.erase(object);
  }

  numReferences =
    object->_weakReferences[dependent] = object->_weakReferences[dependent] - 1;
  if (numReferences <= 0) {
    object->_weakReferences.erase(dependent);
    if (!object->_hasStrongReference(visitedReferences)) {
      (*deletedReferences)[object] = true;
      hash_map<Object*, int, ObjectHasher>::iterator iter =
        object->_dataMembers.begin();
      while (iter != object->_dataMembers.end()) {
        if (deletedReferences->find(iter->first) ==
            deletedReferences->end()) {
          _removeWeakReference(iter->first, object, visitedReferences,
              deletedReferences);
        }
        else {
          object->_dataMembers.erase(iter->first);
        }
        iter = object->_dataMembers.begin();
      }
      delete object;
    }
  }
} // end _removeWeakReference()
```

```
/**
 * Stores that the given object no longer has a weak reference to it by the
 * given dependent.
 * @param object the given object
 * @param dependent the given dependent
 */
void _removeWeakReference(Object *object, Object *dependent) {
  _removeWeakReference(object, dependent, new hash_map<Object*, bool,
      ObjectHasher>(), new hash_map<Object*, bool, ObjectHasher>());
} // end _removeWeakReference()

/**
 * Updates the number of strong references that exist for the given object.
 * @param object the given object
 */
void _addStrongReference(Object *object) {

  if (object == null) return;
  object->_numStrongReferences++;
} // end _addStrongReference()

/**
 * Updates the number of strong references that exist for the given object.
 * NOTE: Do not use this function.  Call the other _removeStrongReference()
 * function instead.
 * @param object the given object
 * @param visitedReferences stores all the objects that are visited from
 * recursive calls made to _removeWeakReference() and _hasStrongReference()
 * @param deletedReferences stores all the objects that are deleted from
 * recursive calls made to _removeWeakReference() and _hasStrongReference()
 */
void _removeStrongReference(Object *object,
    hash_map<Object*, bool, ObjectHasher> *visitedReferences,
    hash_map<Object*, bool, ObjectHasher> *deletedReferences) {

  if (object == null) return;
  object->_numStrongReferences--;
  if (object->_numStrongReferences == 0) {
    if (!object->_hasStrongReference(visitedReferences)) {
      (*deletedReferences)[object] = true;
      hash_map<Object*, int, ObjectHasher>::iterator iter =
        object->_dataMembers.begin();
      while (iter != object->_dataMembers.end()) {
        if (deletedReferences->find(iter->first) ==
            deletedReferences->end()) {
          _removeWeakReference(iter->first, object, visitedReferences,
              deletedReferences);
        }
        else {
          object->_dataMembers.erase(iter->first);
        }
        iter = object->_dataMembers.begin();
      }
      delete object;
    }
  }
} // end _removeStrongReference()

/**
 * Updates the number of strong references that exist for the given object.
 * @param object the given object
 */
```

```
void _removeStrongReference(Object *object) {
  _removeStrongReference(object, new hash_map<Object*, bool, ObjectHasher>(),
      new hash_map<Object*, bool, ObjectHasher>());
} // end _removeStrongReference()
```

Figure 16 – Source Code For _addStrongReference, _removeStrongReference, _addWeakReference, and _removeWeakReference

The first function defines _addWeakReference. The variable, object, is the given object and the variable, dependent, is the given dependent. This method increments the number of weak references that the given dependent has pointing to the given object. It does this by updating the _weakReferences hash map for the given object. Within the key set for _weakReferences it stores objects that have weak references pointing to the given object and within its value set it stores the number of weak references that the objects have pointing to the given object. It also must update the _dataMembers hash map for the given dependent. Within the key set for _dataMembers it stores objects that the given dependent has weak references pointing to and within its value set it stores the number of weak references that the given dependent has pointing to the objects.

The second function defines _removeWeakReference, although this is not the function that will be called in the generated code. It gathers the number of weak references that are dependent on the given object by the given dependent. If it turns out the number of weak references going from the given dependent to the given object is zero, then it checks if it still has any strong references by making a call to _hasStrongReference. If there are no strong references, then it makes a call to _removeWeakReference to all of its attributes. The variable, _deletedReferences, is used to store all the references that had been deleted. This information is necessary to ensure that the program does not call _removeWeakReferences on an object that has already been deleted. The variable, _visitedReferences, is used to ensure that an infinite loop does not occur.

The third function defines the _removeWeakReference that is called from the generated code. This function merely creates new hash maps to store visited nodes and deleted nodes that are subsequently passed to the first definition of _removeWeakReferences. The main program logic for removing weak references is not stored in this function.

The fourth function defines _addStrongReference. This function is extremely simple because all it has to do is increment a reference count. This simplicity is derived from the fact that strong references are references from the user program. This means that there is no need to store references into a hash map, such as _dataMembers.

The fifth function defines _removeStrongReference. This function decrements the number of strong references that are pointing to the given object. If it turns out that the number of strong references has been reduced to zero, then it checks if it has any strong references pointing to the given object through its weak references by making a call to _hasStrongReference. If there are no strong references, then it makes a call to _removeWeakReference to all of its attributes and de-allocates itself afterwards.

Lastly, the sixth function defines the _removeStrongReference that is called from the generated code. This function merely creates new hash maps to store visited nodes and deleted nodes that are subsequently passed to the first definition of _removeStrongReference. The main program logic for removing strong references is not stored in this function.

Whenever the strong reference count falls to zero for the given object it should be apparent that tracing will have to be performed. If there are no immediate strong references pointing to the given object, then it must be determined if there are any strong references pointing to any of the given object's dependents, which requires a trace being performed unless that information was already pre-determined. However, in order to facilitate that this information be pre-determined, this would require that traces be performed every time weak references and strong references are added to objects as opposed to performing traces whenever they are removed from objects, which would not further improve the number of computations that would need to be performed in order to determine if garbage collection is required.

It should be equally apparent that tracing must be performed every time a given object loses all weak references from a given dependent. This is because whenever a given dependent no longer has any weak references pointing to a given object, there may be a possibility that the only strong reference that the given object traced to was pointing to the dependent (either directly or indirectly). In this case, once the weak reference going from the dependent to the object is removed, the given object will no longer be accessible by the user program.

However, there is also the possibility that if a given object loses a weak reference from a given dependent that no garbage collection will have to be performed at all. Unfortunately, even if garbage collection does not need to be performed, a search for strong references is still performed (according to the algorithms in Figure 16) to ensure that garbage collection is not needed.  Figure 17 gives an example of a situation where a weak reference may be removed without needing any garbage collection to be performed afterwards.

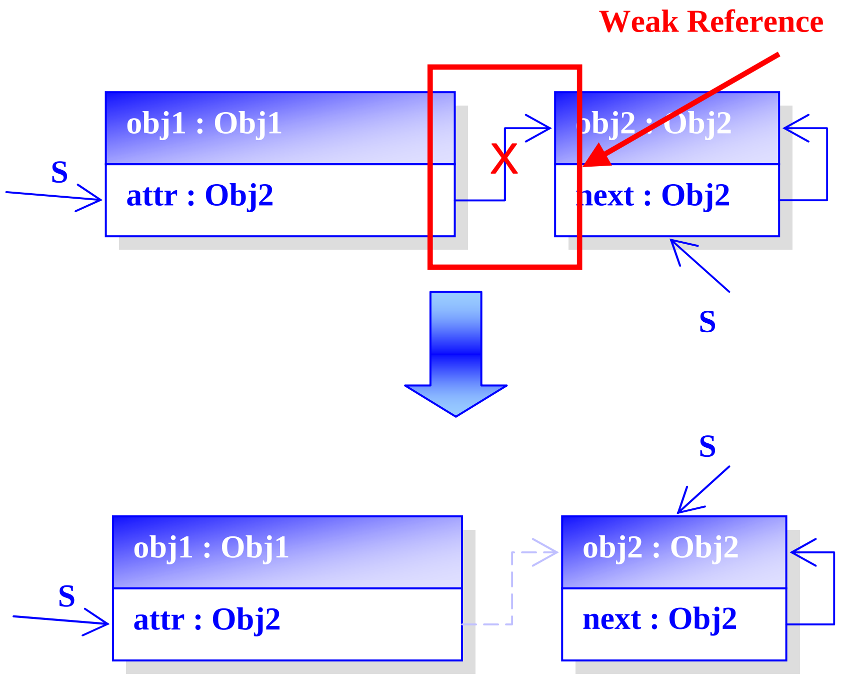


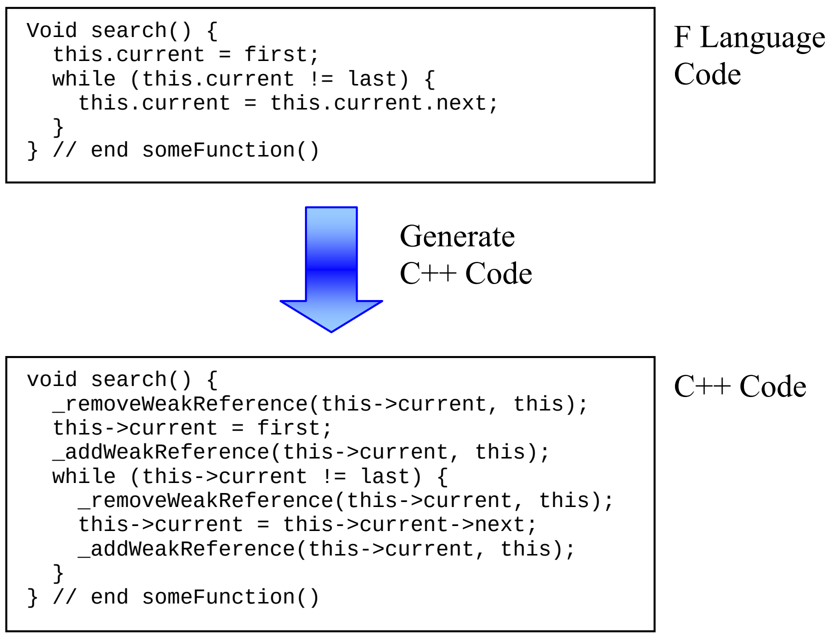
Figure 17 – Example Where Garbage Collection Does Not Need to Be Performed After an Object Loses a Weak Reference from a Dependent

All the garbage is collected in the same way that it is described in the section, Reference Counting, for non-cyclic structures. Furthermore, for every situation where garbage collection would be performed when using reference counting, garbage collection would also be performed if using strong and weak references. Therefore, every non-cyclic structure that would be de-allocated using reference counting would also be de-allocated when using strong and weak references.

Since strong references will only be made for objects accessible by the user program, referential loops cannot be created by strong references. The only way that referential loops can be created in memory is by weak references. Therefore it follows that if no strong references are pointing to a referential loop, the referential loop can be garbage collected since it will only consist of weak references. Therefore, by using strong and weak references we can ensure that garbage collection will de-allocate any cyclic structures that reside in memory that are inaccessible by the user program.

## PROBLEMS WITH STRONG AND WEAK REFERENCES

The benefit to using strong and weak references was made apparent in the previous section by showing that if you perform checks on objects to determine if they are traceable to the user program, then you can ensure that no memory leaks will occur with cyclic structures. However, the problem with using this method is that it can alter the expected time complexities of algorithms written by the programmer. In order to get a better grasp of this, consider the example given in Figure 18.



```
Void search() {
  this.current = first;
  while (this.current != last) {
    this.current = this.current.next;
  }
} // end someFunction()
```

```
void search() {
  _removeWeakReference(this->current, this);
  this->current = first;
  _addWeakReference(this->current, this);
  while (this->current != last) {
    _removeWeakReference(this->current, this);
    this->current = this->current->next;
    _addWeakReference(this->current, this);
  }
} // end someFunction()
```

Figure 18 – Example Where Generated C++ Code Affects the Expected Time Complexity of an Algorithm

In Figure 18, we have a function, search, which is a member function of a class containing the member variables, current, first, and last. Assume that current, first, and last are all references to linked list nodes and that "first" points to the head of a linked list and "last" points to the tail. The function, search, iterates through all the nodes in the linked list and assigns the member variable, current, each of the nodes that are encountered.

The expected time complexity for the algorithm written in the F language code in Figure 18 is $T(N) \equiv O(N)$, where N is the number of nodes in the linked list. However, as a result of the call being made to _removeWeakReference() within the while loop for the generated C++ code, the function, _hasStrongReference() will be called for each iteration (see Figure 16). This call will effectively be making traversals through the linked list for every iteration through the while loop just so it can search for strong references. As a result, this will make the actual time complexity for the algorithm written by the developer $T(N) \equiv O(N^2)$ (ignoring all overhead associated with hash maps).

In fact, it should be pointed out that every time a call is made to _removeWeakReference, there will be a chance that _hasStrongReference will need to be called and a search for strong references will then be conducted. If the number of weak references being traversed through is small, then this may not make a noticeable change during the runtime. However for container classes, such as linked lists, which may hold thousands of objects during runtime, this could cause huge delays in the runtime.

It should be noted here that the techniques described in the section, Reference Counting, do not suffer from this problem. This is because when you use mere reference counting, garbage collection is only performed when the reference count for an object drops to zero. Therefore there are not any unnecessary searches for strong references being performed as a result of assignment statements where member variables happen to appear on the left hand side of the assignment. Therefore, despite the fact that reference counting is inadequate at garbage collecting cyclic structures, it performs better than strong and weak references in the sense that it does not alter the expected time complexities of a programmer's algorithms.

It should also be noted here that by using strong and weak references, we are effectively creating another "stop-the-world" garbage collector because traces are being performed at times when the programmer would not have otherwise been managing memory if she had coded it up herself in C++.  In fact, it may be better if the program waited for a more appropriate time to perform traces and garbage cleanup similarly to how Java does it. This would at least significantly reduce the problem of affecting the expected time complexities of the programmers' code.

# 4. Examples

## QUEUE

In order to ensure that strong and weak references are being used appropriately within the generated C++ and are functioning properly, it is necessary to have an example where assignments are being performed on member variables of a class. To do this, an F language implementation of a Queue was chosen. The code is shown in Figure 19.

```
/*****************************************************************************
 *                                                                           *
 *                            Stephen Hoskins                                *
 *                              G-00337381                                   *
 *                        George Mason University                            *
 *                                 CS707                                     *
 *                              Summer 2007                                  *
 *                             Thesis Project                                *
 *                                                                           *
 *****************************************************************************/

/**
 * A queue for storing the objects given to it.
 * @author Stephen Hoskins, George Mason University
 * @version 1.0, 07/29/07
 */
public class Queue {

  /**
   * Internal class for representing a queue node.
   * @author Stephen Hoskins, George Mason University
   * @version 1.0, 07/29/07
   */
  private class QueueNode {

    /**
     * Stores the next node in the queue.
     */
    public QueueNode next;

    /**
     * Stores the value for the node.
     */
    public Object value;
    /**
     * Creates a new QueueNode object.
     */
    public QueueNode() {
      next = null;
      value = null;
    } // end default constructor

  } // end class QueueNode

  /**
   * Stores the first node in the queue.
   */
  private QueueNode first;

  /**
   * Stores the last node in the queue.
   */
  private QueueNode last;

  /**
   * Creates a new Queue object.
   */
  public Queue() {
    first = null;
    last = null;
  } // end default constructor

  /**
   * Adds the given object to the end of the queue of objects.
   * @param object the given object
   */
  public void enqueue(Object object) {
    if (last != null) {
      QueueNode newNode = new QueueNode();
      newNode.value = object;
      last.next = newNode;
      last = newNode;
    }
    else {
      last = new QueueNode();
      last.value = object;
      first = last;
    }
  } // end enqueue()

  /**
   * Removes and returns an object from the beginning of the queue of objects.
   * @return an object from the beginning of the queue of objects
   */
  public Object dequeue() {

    QueueNode node = first;
    if (first == null) {
      return null;
    }
    else if (first.next == null) {
      first = null;
      last = null;
      return node.value;
    }
    else {
      first = first.next;
      return node.value;
    }
  } // end dequeue()
```

```
  /**
   * The main entry-point into the application.  Performs example enqueueing
   * and dequeueing on a queue.
   */
  public static void main() {
    Queue queue = new Queue();
    Integer object1 = 1;
    Integer object2 = 2;
    queue.enqueue(object1);
    queue.enqueue(object2);
    object1 = (Integer) queue.dequeue();
    object2 = (Integer) queue.dequeue();
    println(object1);
    println(object2);
  } // end main()
} // end class Queue
```

Figure 19 – Queue F Language Implementation

The code in Figure 19 implements a class, Queue, that contains an inner-class, QueueNode, and the member functions, enqueue, dequeue, and main(). The inner class, QueueNode, contains the member variables, next and value, which reference the next node in the queue and the value being stored in the node respectively. The member function, enqueue, adds a new node to the back of the queue and the member function, dequeue, removes and returns a node from the front of the queue. The main function is the main entry point into the application, which creates an instance of a queue, enqueues and dequeues two integers, and prints out the values of the integers that were dequeued.

The code in Figure 19 was successfully parsed and converted into C++ code. Indentation and extra spacing (formatting) was added to the output to make it more legible. In addition, “couts” were added to display how long it took the program to run in seconds. The results of the parse with modifications are shown in Figure 20.

```
#include <iostream>
```

```
#include <iomanip>
#include <time.h>

using namespace std;

#include "FLang.h"

class Queue : public Object {

  Queue *_Queue_this;

  private:
  class QueueNode : public Object {
    QueueNode *_QueueNode_this;
    public:
    QueueNode *next;
    public:
    Object *value;
    public:
    QueueNode() {
      _QueueNode_this = this;
      next = null;
      value = null;
      _removeWeakReference(next, _QueueNode_this);
      next = null;
      _addWeakReference(next, _QueueNode_this);
      _removeWeakReference(value, _QueueNode_this);
      value = null;
      _addWeakReference(value, _QueueNode_this);
    }
  };

  private:
  QueueNode *first;
  private:
  QueueNode *last;

  public:
  Queue() {
    _Queue_this = this;
    first = null;
    last = null;
    _removeWeakReference(first, _Queue_this);
    first = null;
    _addWeakReference(first, _Queue_this);
    _removeWeakReference(last, _Queue_this);
    last = null;
    _addWeakReference(last, _Queue_this);
  }

  public:
  void enqueue(Object *object) {
    QueueNode *newNode = null;
    if (last != null) {
      newNode = new QueueNode();
      _addStrongReference(newNode);

      _removeWeakReference(newNode->value, newNode);
      newNode->value = object;
      _addWeakReference(newNode->value, newNode);
      _removeWeakReference(last->next, last);
      last->next = newNode;
      _addWeakReference(last->next, last);
      _removeWeakReference(last, _Queue_this);
      last = newNode;
      _addWeakReference(last, _Queue_this);
    }
    else {
      _removeWeakReference(last, _Queue_this);
      last = new QueueNode();
      _addWeakReference(last, _Queue_this);
      _removeWeakReference(last->value, last);
      last->value = object;
      _addWeakReference(last->value, last);
      _removeWeakReference(first, _Queue_this);
      first = last;
      _addWeakReference(first, _Queue_this);
    }
    _removeStrongReference(newNode);
  }

  public:
  Object *dequeue() {
    QueueNode *node = null;
    node = first;
    _addStrongReference(node);

    if (first == null) {
      _tmp = null;
      _addStrongReference(_tmp);
      _removeStrongReference(node);
      return _tmp;
    }
    else {
      _removeWeakReference(first, _Queue_this);
      first = first->next;
      _addWeakReference(first, _Queue_this);
      _tmp = node->value;
      _addStrongReference(_tmp);
      _removeStrongReference(node);
      return _tmp;
    }
    _removeStrongReference(node);
  }
};

void main() {

  clock_t time = clock();

  Queue *queue = null;
  Integer *object1 = null;
  Integer *object2 = null;
  queue = new Queue();
  _addStrongReference(queue);

  object1 = new Integer(1);
  _addStrongReference(object1);

  object2 = new Integer(2);
  _addStrongReference(object2);

  queue->enqueue(object1);
  queue->enqueue(object2);
  _removeStrongReference(object1);
  object1 = (Integer*) queue->dequeue();
  _addStrongReference(object1);
  _removeStrongReference(object1);
  _removeStrongReference(object2);
  object2 = (Integer*) queue->dequeue();
  _addStrongReference(object2);
  _removeStrongReference(object2);
  println(object1);
  println(object2);
```

```
    _removeStrongReference(queue);
    _removeStrongReference(object1);
    _removeStrongReference(object2);

    cout << "****TIME TO RUN APPLICATION****" << endl;
    double deltaTime = ((double) clock() - time) / ((double) CLOCKS_PER_SEC);
    cout << fixed << setprecision(12) << deltaTime << endl;
  }
```

Figure 20 – C++ Generated From Queue F Language Implementation (With Formatting Added)

It appears that the strong and weak references were all updated in the appropriated places within the code in Figure 20. Furthermore, the code in Figure 20 could be successfully compiled and executed in Microsoft Visual Studio 2005. The program ran without crashing and produced the results that were expected from the code that was written in the F language. The results are shown in Figure 21.

```
1
2
****TIME TO RUN APPLICATION****
0.000000000000
```

Figure 21 – Output Generated From the Executable Produced From the Code in Figure 20

The output in Figure 21 shows the results that were expected from the example queue implementation written in the F language. In particular, the integer, 1, was dequeued first, followed by the integer, 2, which was the order that the integers were enqueued onto the queue. Furthermore, the output shows that this program completed very fast, taking virtually no time to complete.

Next, it would be appropriate to show that garbage collection occurred when it was expected to.  To do so, more "cout" statements will need to be added to the main function.  The modified main function is shown in Figure 22.

```
void main() {

  clock_t time = clock();

  Queue *queue = null;
  Integer *object1 = null;
  Integer *object2 = null;
  queue = new Queue();
  _addStrongReference(queue);

  object1 = new Integer(1);
  _addStrongReference(object1);

  object2 = new Integer(2);
  _addStrongReference(object2);

  queue->enqueue(object1);
  queue->enqueue(object2);
  _removeStrongReference(object1);
  object1 = (Integer*) queue->dequeue();
  _addStrongReference(object1);
  _removeStrongReference(object1);
  _removeStrongReference(object2);
  object2 = (Integer*) queue->dequeue();
  _addStrongReference(object2);
  _removeStrongReference(object2);
  println(object1);
  println(object2);

  cout << endl;
  cout << "Number of Strong References Pointing to Queue:" << endl;
  cout << queue->_numStrongReferences << endl;
  cout << "Number of Strong References Pointing to object1:" << endl;
  cout << object1->_numStrongReferences << endl;
  cout << "Number of Strong References Pointing to object2:" << endl;
  cout << object2->_numStrongReferences << endl;
  cout << endl;
  cout << "****REMOVING REFERENCES FROM QUEUE****" << endl;
  cout << endl;

  _removeStrongReference(queue);

  cout << endl;
  cout << "Number of Strong References Pointing to Queue:" << endl;
  cout << queue->_numStrongReferences << endl;
  cout << "Number of Strong References Pointing to object1:" << endl;
  cout << object1->_numStrongReferences << endl;
  cout << "Number of Strong References Pointing to object2:" << endl;
  cout << object2->_numStrongReferences << endl;
  cout << endl;
  cout << "****REMOVING REFERENCES FROM OBJECT1****" << endl;
  cout << endl;

  _removeStrongReference(object1);
```

```
    cout << endl;
    cout << "Number of Strong References Pointing to Queue:" << endl;
    cout << queue->_numStrongReferences << endl;
    cout << "Number of Strong References Pointing to object1:" << endl;
    cout << object1->_numStrongReferences << endl;
    cout << "Number of Strong References Pointing to object2:" << endl;
    cout << object2->_numStrongReferences << endl;
    cout << endl;
    cout << "****REMOVING REFERENCES FROM OBJECT2****" << endl;
    cout << endl;

    _removeStrongReference(object2);

    cout << "Number of Strong References Pointing to Queue:" << endl;
    cout << queue->_numStrongReferences << endl;
    cout << "Number of Strong References Pointing to object1:" << endl;
    cout << object1->_numStrongReferences << endl;
    cout << "Number of Strong References Pointing to object2:" << endl;
    cout << object2->_numStrongReferences << endl;

    cout << endl;
    cout << "****TIME TO RUN APPLICATION****" << endl;
      double deltaTime = ((double) clock() - time) / ((double) CLOCKS_PER_SEC);
      cout << fixed << setprecision(12) << deltaTime << endl;
  }
```

Figure 22 – Modified Main Function for Generated C++ Produced From Queue F Implementation

As can be seen in Figure 22, "couts" were added to show the number of strong references pointing to the variables, queue, object1, and object2, as strong references are removed from them right before the function terminates. It is expected that the variables, queue, object1, and object2, will each have one strong reference right before the line, "****REMOVING REFERENCES FROM QUEUE****," is displayed. As the references are removed from the variables, queue, object1, and object2, it is expected that garbage values will be displayed for the number of strong references pointing to these variables since they will have been garbage collected. The results of the new "cout" statements are shown in Figure 23.

1

```
2

Number of Strong References Pointing to Queue:
1
Number of Strong References Pointing to object1:
1
Number of Strong References Pointing to object2:
1

****REMOVING REFERENCES FROM QUEUE****


Number of Strong References Pointing to Queue:
-572662307
Number of Strong References Pointing to object1:
1
Number of Strong References Pointing to object2:
1

****REMOVING REFERENCES FROM OBJECT1****


Number of Strong References Pointing to Queue:
24
Number of Strong References Pointing to object1:
-572662307
Number of Strong References Pointing to object2:
1

****REMOVING REFERENCES FROM OBJECT2****

Number of Strong References Pointing to Queue:
24
Number of Strong References Pointing to object1:
24
Number of Strong References Pointing to object2:
-572662307

****TIME TO RUN APPLICATION****
0.015000000000
```

Figure 23 – Results From Modified Main Function For Generated C++ Produced From Queue F Implementation

Right before the line, "****REMOVING REFERENCES FROM QUEUE****," it can be seen that the variables, queue, object1, and object2, each have one strong reference, which is how many they should have. After the line, "****REMOVING REFERENCES FROM QUEUE****," a garbage value of -572662307 can be seen for the number of strong references pointing to the variable, queue, which is expected since it should have been garbage collected as a result of nothing pointing to it. Furthermore, it can be seen that after the line, "****REMOVING REFERENCES FROM OBJECT1****," that a garbage value of -572662307 exists for the number of strong references pointing to the variable, object1, which is also expected. And finally, after the line, "****REMOVING REFERENCES FROM OBJECT2****," it can be seen that all the variables, queue, object1, and object2, have been garbage collected.

Notice that the time for the application to complete has increased. However, this change in time is attributed to the fact that so much more output is being generated by the application. Therefore, this increase in time has nothing to do with the garbage collection.

Next, the results of this queue implementation will be compared with the results of writing a queue implementation directly in C++. In this particular case, the memory will not be managed with the automated procedure being used by the F compiler. The C++ equivalent of the F implementation of a queue is presented in Figure 24.

```
/******************************************************************************
 *                                                                            *
 *                             Stephen Hoskins                                *
 *                               G-00337381                                   *
 *                          George Mason University                           *
 *                                 CS707                                      *
 *                               Summer 2007                                  *
 *                              Thesis Project                                *
 *                                                                            *
 ******************************************************************************/

#include <iostream>
#include <iomanip>
#include <time.h>

using namespace std;

#define null 0
```

```
/**
 * A queue implemented in C++ for storing the values given to it.
 * @author Stephen Hoskins, George Mason University
 * @version 1.0, 11/19/07
 */
class Queue {

private:

  /**
   * Internal class for representing a queue node.
   * @author Stephen Hoskins, George Mason University
   * @version 1.0, 11/19/07
   */
  class QueueNode {

  public:

    /**
     * Stores the next node in the queue.
     */
    QueueNode *next;

    /**
     * Stores the value for the node.
     */
    void *value;

    /**
     * Creates a new QueueNode object.
     */
    QueueNode() {
      next = null;
      value = null;
    } // end default constructor

  }; // end class QueueNode

  /**
   * Stores the first node in the queue.
   */
  QueueNode *first;

  /**
   * Stores the last node in the queue.
   */
  QueueNode *last;

public:

  /**
   * Creates a new Queue object.
   */
  Queue() {
    first = null;
    last = null;
  } // end default constructor

  /**
   * Adds the given value to the end of the queue.
   * @param value the given value
   */
  void enqueue(void *value) {
    if (last != null) {
      QueueNode *newNode = new QueueNode();
      newNode->value = value;
      last->next = newNode;
      last = newNode;
    }
    else {
      last = new QueueNode();
      last->value = value;
      first = last;
    }
  } // end enqueue()

  /**
   * Removes and returns a value from the beginning of the queue.
   * @return a value from the beginning of the queue
   */
  void *dequeue() {

    void *value = null;
    QueueNode *node = null;
    if (first == null) {
      return null;
    }
    else if (first->next == null) {
      value = first->value;
      delete first;
      first = null;
      last = null;
      return value;
    }
    else {
      value = first->value;
      node = first;
      first = first->next;
      delete node;
      return value;
    }
  } // end dequeue()

  /**
   * Destructs the queue.
   */
  ~Queue() {
    while (first != null) dequeue();
  } // end destructor

}; // end class Queue

/**
 * The main entry-point into the application.  Performs example enqueueing
 * and dequeueing on a queue.
 */
int main() {

  clock_t time = clock();
  Queue *queue = new Queue();
  int *value1 = new int(1);
  int *value2 = new int(2);
  queue->enqueue(value1);
  queue->enqueue(value2);
  value1 = (int*) queue->dequeue();
  value2 = (int*) queue->dequeue();
  cout << *value1 << endl;
  cout << *value2 << endl;
  delete queue;
  delete value1;
  delete value2;

  cout << "****TIME TO RUN APPLICATION****" << endl;
  double deltaTime = ((double) clock() - time) / ((double) CLOCKS_PER_SEC);
  cout << fixed << setprecision(12) << deltaTime << endl;
  return 0;
} // end main()
```

Figure 24 – C++ Equivalent of the F
Implementation of a Queue

As can be seen from Figure 24, the code looks a little more complicated than the original F implementation of a queue as a result of having to code the memory management. However, this code does appear to be much simpler than the C++ that was produced from the F language compiler. Therefore, it is expected that this program will run faster than the F-implemented program. However since the F implementation finished in virtually zero time, it probably will not be possible to see the differences in execution time. The output produced from the C++ equivalent of the F implementation of a queue is shown in Figure 25.

```
1
2
****TIME TO RUN APPLICATION****
0.000000000000
```

Figure 25 – Output Generated from C++
Equivalent of the F Implementation of a Queue

As expected, no differences in the execution time could be detected between the F implementation of a queue and a C++ implementation of a queue. This is a good because it appears that adding automated memory management has not severely impacted the execution time. However, a more complicated example will be necessary to show the consequences of using strong and weak references, which is discussed in chapter III in the section, Problems with Strong and Weak References.

Next, we will compare these results with a Java implementation of a queue. In this particular case, the programmer will not need to manage memory since it will already be handled by the Java garbage collector. However, we will need to make a call to the garbage collector to perform cleanup in order to get a more accurate reading of the execution time. The Java equivalent of the F implementation of a queue is presented in Figure 26.

```
/*****************************************************************************
 *                                                                           *
 *                              Stephen Hoskins                              *
 *                                G-00337381                                 *
 *                          George Mason University                          *
 *                                  CS707                                    *
 *                                Summer 2007                                *
 *                              Thesis Project                               *
 *                                                                           *
 *****************************************************************************/

/**
 * A queue for storing the objects given to it.
 * @author Stephen Hoskins, George Mason University
 * @version 1.0, 07/29/07
 */
public class Queue {

  /**
   * Internal class for representing a queue node.
   * @author Stephen Hoskins, George Mason University
   * @version 1.0, 07/29/07
   */
  private class QueueNode {

    /**
     * Stores the next node in the queue.
     */
    public QueueNode next;

    /**
     * Stores the value for the node.
     */
    public Object value;

    /**
     * Creates a new QueueNode object.
     */
    public QueueNode() {
      next = null;
      value = null;
    } // end default constructor

  } // end class QueueNode

  /**
   * Stores the first node in the queue.
   */
  private QueueNode first;

  /**
   * Stores the last node in the queue.
   */
  private QueueNode last;

  /**
   * Creates a new Queue object.
   */
  public Queue() {
    first = null;
    last = null;
  } // end default constructor

  /**
   * Adds the given object to the end of the queue of objects.
   * @param object the given object
   */
  public void enqueue(Object object) {
    if (last != null) {
      QueueNode newNode = new QueueNode();
      newNode.value = object;
      last.next = newNode;
      last = newNode;
    }
    else {
      last = new QueueNode();
      last.value = object;
      first = last;
    }
  } // end enqueue()

  /**
   * Removes and returns an object from the beginning of the queue of objects.
   * @return an object from the beginning of the queue of objects
   */
  public Object dequeue() {

    QueueNode node = first;
    if (first == null) {
      return null;
    }
    else if (first.next == null) {
      first = null;
      last = null;
      return node.value;
    }
    else {
      first = first.next;
      return node.value;
    }
  } // end dequeue()

  /**
   * The main entry-point into the application.  Performs example enqueueing
```

```
     * and dequeueing on a queue.
     */
    public static void main(String args[]) {

      long time = System.currentTimeMillis();

      Queue queue = new Queue();
      Integer object1 = new Integer(1);
      Integer object2 = new Integer(2);
      queue.enqueue(object1);
      queue.enqueue(object2);
      object1 = (Integer) queue.dequeue();
      object2 = (Integer) queue.dequeue();
      System.out.println(object1);
      System.out.println(object2);

      // perform garbage collection
      queue = null;
      object1 = null;
      object2 = null;
      System.gc();

      double deltaTime = ((double) System.currentTimeMillis() - time) / 1000;
      System.out.println("****TIME TO RUN APPLICATION****");
      System.out.println(deltaTime);
    } // end main()
  } // end class Queue
```

Figure 26 – Java Equivalent of F Implementation of a Queue

As can be seen in Figure 26, the code is pretty similar to the F language implementation. However, notice that the code is a little bit more complicated when the programmer wishes for the garbage collector to perform cleanup at a specific time.  The results of the output generated from this program is in Figure 27.

```
1
2
****TIME TO RUN APPLICATION****
0.016
```

Figure 27 – Output Produced from the Java
Equivalent of F Implementation of a Queue

According to the output in Figure 27, the Java implementation actually took longer than the F implementation and the C++ implementation of the queue. However, this might be attributed to the fact that Java is also running on top of an interpreter –i.e., the Java Virtual Machine. So it is possible that this slow down was not necessarily the result of Java's garbage collector. However, it is clear that for whatever reason the Java implementation of the queue is slower. Note that the precision of the output is the same as the precisions in the previous examples, although Java does not print trailing zeros by default.

## QUEUE-EXHAUSTED

In the last section, it was determined that a more complicated example would have to be used in order to test the inefficiencies of strong and weak references. In order to achieve this, the main function in the queue example will be modified so that twenty-five hundred objects will be enqueued onto the queue and then subsequently dequeued to see if major differences in execution time occur. The main function in the C++ generated from the F implementation of the queue will be modified instead of the main function in the original F implementation since the purpose of this experiment is just to test out the garbage collector. The modified main function in the C++ generated from the F implementation of a queue is show in Figure 28.

```
void main() {

  clock_t time = clock();

  Queue *queue = null;
  Integer *object1 = null;

  queue = new Queue();
  _addStrongReference(queue);

  for (int i = 0; i < 2500; i++) {
    _removeStrongReference(object1);
    object1 = new Integer(i);
    _addStrongReference(object1);

    queue->enqueue(object1);
  }

  for (int i = 0; i < 2500; i++) {
    _removeStrongReference(object1);
    object1 = (Integer*) queue->dequeue();
    _addStrongReference(object1);
    _removeStrongReference(object1);
  }

  _removeStrongReference(queue);
  _removeStrongReference(object1);

  cout << "****TIME TO RUN APPLICATION****" << endl;
  double deltaTime = ((double) clock() - time) / ((double) CLOCKS_PER_SEC);
  cout << fixed << setprecision(12) << deltaTime << endl;
}
```

Figure 28 – Modified Main Function in the C++
Generated from the F Implementation of a Queue

It is expected that this program will run much longer than before as a result of searches being conducted through weak references in order to find strong references. In particular, for every iteration of each *for* loop, up to twenty-five hundred weak references may have to be traversed through before a strong reference can be found, which should make this program considerably slower. The output from this modified queue example is shown in Figure 29.

```
****TIME TO RUN APPLICATION****
2.937000000000
```

Figure 29 – Output From Modified Main Function in the C++ Generated from the F Implementation of a Queue

As expected, the program did run considerably slower than before. However, in order to ensure that this is related to the F garbage collector, we will need to subject the C++ and Java implementations of a queue to the same changes. The modified main function in the C++ implementation of a queue is shown in Figure 30.

```
/**
 * The main entry-point into the application.  Performs example enqueueing
 * and dequeueing on a queue.
 */
int main() {

  clock_t time = clock();
  Queue *queue = new Queue();
  int *value1 = null;

  for (int i = 0; i < 2500; i++) {
    value1 = new int(i);
    queue->enqueue(value1);
  }

  for (int i = 0; i < 2500; i++) {
    value1 = (int*) queue->dequeue();
    delete value1;
  }

  delete queue;

  cout << "****TIME TO RUN APPLICATION****" << endl;
  double deltaTime = ((double) clock() - time) / ((double) CLOCKS_PER_SEC);
  cout << fixed << setprecision(12) << deltaTime << endl;
  return 0;
} // end main()
```

Figure 30 – Modified Main Function in the C++ Implementation of a Queue

It is expected that this will run much faster since traversals through weak references will not need to be performed in order to perform garbage collection. However, if weak references had nothing to do with the execution time, then it is still expected that this program will run faster because it will not have all the overhead associated with retrieving data members from hash maps. The output produced from the modified C++ implementation of a queue is shown in Figure 31.

```
****TIME TO RUN APPLICATION****
0.000000000000
```

Figure 31 – Output Produced from the Modified C++ Implementation of a Queue

Interestingly enough, the modified C++ implementation of a queue still performed in virtually zero time. Clearly, if there is any hope of making the F language compiler a sensible alternative solution to C++, then the garbage collector will have to be modified to ensure that programs written in F will run at speeds more comparable to C++. It is unfortunate that the F language implementation did not perform as well, although this is not a surprise either given the discussion in the section, Problems with Strong and Weak References.

Finally, we will compare the results of the modified F queue implementation with a modified Java queue implementation. Once again, we will only need to modify the main function within the Java code. The modified main function within the Java implementation of a queue is shown in Figure 32.

```
/**
 * The main entry-point into the application.  Performs example enqueueing
 * and dequeueing on a queue.
 */
public static void main(String args[]) {

  long time = System.currentTimeMillis();

  Queue queue = new Queue();
  Integer object1 = null;

  for (int i = 0; i < 2500; i++) {
    object1 = new Integer(i);
    queue.enqueue(object1);
  }

  for (int i = 0; i < 2500; i++) {
    object1 = (Integer) queue.dequeue();
  }

  // perform garbage collection
  queue = null;
  object1 = null;
  System.gc();

  double deltaTime = ((double) System.currentTimeMillis() - time) / 1000;
  System.out.println("****TIME TO RUN APPLICATION****");
  System.out.println(deltaTime);
} // end main()
```

Figure 32 – Modified Main Function Within the Java Implementation of a Queue

Once again, it is expected that this implementation will run faster than the F implementation. However, it is not entirely certain how much better this will run because Java also has a built in garbage collector. However, unlike the F garbage collector, the Java garbage collector only performs traces through references when absolutely necessary or when it is told to by the programmer. Therefore, it is believed that the Java implementation will perform exceptionally better than the F implementation, although it is still possible that the performance may take a hit as a result of the built-in garbage collector. The output from the modifications made to the Java implementation is shown in Figure 33.

```
****TIME TO RUN APPLICATION****
0.016
```

Figure 33 – Output Generated from Modified
Java Implementation of a Queue

Interestingly enough, the modified Java implementation performed just as fast as it did in the original queue implementation. It was expected that the Java garbage collector would perform much better than the F garbage collector, but it was not known that it would be this efficient. Clearly improvements will have to be made to the F garbage collector because the drastic differences in performance will most likely be considered unacceptable by any programmer who wishes to produce robust and efficient software.

## CIRCULARLY-LINKED QUEUE

In the previous examples, it was shown that the F compiler was able to properly convert a simple queue example into C++ code, which could then be compiled into an executable and ran without suffering memory leak problems. Furthermore, the ramifications of using strong and weak references as they are implemented in this paper are shown in the section, Queue-Exhausted, which shows a significant decrease in performance when the

F language implementation of a queue was compared with a C++ implementation and a Java implementation. However, what has not been shown yet is whether or not the F language compiler is able to perform a cleanup on cyclic garbage. In this section, we hope to show that.

In order to show whether or not the F language compiler is able to perform a cleanup on cyclic garbage, we will create yet another modified example of the queue. However, this time we will modify the queue so that it has a circularly-linked list implementation instead of the linearly-linked list implementation that was being used before. The circularly-linked list implementation of a queue is shown in Figure 34.

```
/******************************************************************************
 *                                                                            *
 *                              Stephen Hoskins                               *
 *                                G-00337381                                  *
 *                          George Mason University                           *
 *                                  CS707                                     *
 *                               Summer 2007                                  *
 *                              Thesis Project                                *
 *                                                                            *
 ******************************************************************************/

/**
 * A queue for storing the objects given to it.
 * @author Stephen Hoskins, George Mason University
 * @version 1.0, 12/02/07
 */
public class Queue {

  /**
   * Internal class for representing a queue node.
   * @author Stephen Hoskins, George Mason University
   * @version 1.0, 07/29/07
   */
  private class QueueNode {

    /**
     * Stores the next node in the queue.
     */
    public QueueNode next;

    /**
     * Stores the value for the node.
     */
    public Object value;
    /**
     * Creates a new QueueNode object.
     */
    public QueueNode() {
      next = null;
      value = null;
    } // end default constructor

  } // end class QueueNode

  /**
   * Stores the first node in the queue.
   */
  private QueueNode first;

  /**
   * Stores the last node in the queue.
   */
  private QueueNode last;

  /**
   * Creates a new Queue object.
   */
  public Queue() {
    first = null;
    last = null;
  } // end default constructor

  /**
   * Adds the given object to the end of the queue of objects.
   * @param object the given object
   */
  public void enqueue(Object object) {
    if (last != null) {
      QueueNode newNode = new QueueNode();
      newNode.value = object;
      last.next = newNode;
      last = newNode;
      last.next = first; // new line
    }
    else {
      last = new QueueNode();
      last.value = object;
      last.next = last; // new line
      first = last;
    }
  } // end enqueue()

  /**
   * Removes and returns an object from the beginning of the queue of objects.
   * @return an object from the beginning of the queue of objects
   */
  public Object dequeue() {

    QueueNode node = first;
    if (first == null) {
      return null;
    }
    else if (first == last) { // modified line
      first = null;
      last = null;
      return node.value;
    }
    else {
      first = first.next;
      last.next = first; // new line
      return node.value;
```

```
      }
    } // end dequeue()

    /**
     * The main entry-point into the application.  Performs example enqueueing
     * and dequeueing on a queue.
     */
    public static void main() {
      Queue queue = new Queue();
      Integer object1 = 1;
      Integer object2 = 2;
      queue.enqueue(object1);
      queue.enqueue(object2);

      // dequeue calls were removed to keep cyclic references
      object1 = null; // new line
      object2 = null; // new line
    } // end main()
  } // end class Queue
```

Figure 34 – Circularly-Linked List F
Implementation of a Queue

As can be seen from Figure 34, lines were added to the functions, enqueue and dequeue, all the dequeue calls were removed from the main function, and the variables, object1 and object2, were assigned the value, null. The calls to dequeue were removed from main because we want to keep the cyclic structure that was created in the queue to see if it can be garbage collected. Lastly, the variables, object1 and object2, were assigned the value, null, because otherwise the entire cyclic structure will not be garbage collected when the variable, queue, falls out of scope.

The F compiler managed to once again convert this code appropriately. In order to make the output from the F compiler more readable, formatting has been added. Furthermore, “cout” statements have been added to show the states of the variables while garbage collection is being performed. Two variables, test1 and test2, had to be inserted in the code as well in order to show the states of the variables. The output from the F compiler with formatting and debug code inserted is shown in Figure 35.

```
#include <iostream>
#include "FLang.h"

using namespace std;

class Queue : public Object {
  Queue *_Queue_this;
private:
  class QueueNode : public Object {
    QueueNode *_QueueNode_this;
  public:
    QueueNode *next;
  public:
    Object *value;
  public:
    QueueNode() {
      _QueueNode_this = this;
      next = null;
      value = null;
      _removeWeakReference(next, _QueueNode_this);
      next = null;
      _addWeakReference(next, _QueueNode_this);
      _removeWeakReference(value, _QueueNode_this);
      value = null;
      _addWeakReference(value, _QueueNode_this);
    }
  };
private:
  QueueNode *first;
private:
  QueueNode *last;
public:
  Queue() {
    _Queue_this = this;
    first = null;
    last = null;
    _removeWeakReference(first, _Queue_this);
    first = null;
    _addWeakReference(first, _Queue_this);
    _removeWeakReference(last, _Queue_this);
    last = null;
    _addWeakReference(last, _Queue_this);
  }
public:
  void enqueue(Object *object) {
    QueueNode *newNode = null;
    if (last != null) {
      newNode = new QueueNode();
      _addStrongReference(newNode);

      _removeWeakReference(newNode->value, newNode);
      newNode->value = object;
      _addWeakReference(newNode->value, newNode);
      _removeWeakReference(last->next, last);
      last->next = newNode;
      _addWeakReference(last->next, last);
      _removeWeakReference(last, _Queue_this);
      last = newNode;
      _addWeakReference(last, _Queue_this);
      _removeWeakReference(last->next, last);
      last->next = first;
      _addWeakReference(last->next, last);
    }
    else {
      _removeWeakReference(last, _Queue_this);
      last = new QueueNode();
      _addWeakReference(last, _Queue_this);
      _removeWeakReference(last->value, last);
      last->value = object;
      _addWeakReference(last->value, last);
      _removeWeakReference(last->next, last);
      last->next = last;
      _addWeakReference(last->next, last);
      _removeWeakReference(first, _Queue_this);
      first = last;
      _addWeakReference(first, _Queue_this);
    }
    _removeStrongReference(newNode);
  }
public:
  Object *dequeue() {
    QueueNode *node = null;
    node = first;
    _addStrongReference(node);

    if (first == null) {
      _tmp = null;
      _addStrongReference(_tmp);
      _removeStrongReference(node);
      return _tmp;
    }
    else {
      _removeWeakReference(first, _Queue_this);
      first = first->next;
      _addWeakReference(first, _Queue_this);
      _removeWeakReference(last->next, last);
      last->next = first;
      _addWeakReference(last->next, last);
      _tmp = node->value;
      _addStrongReference(_tmp);
      _removeStrongReference(node);
      return _tmp;
    }
    _removeStrongReference(node);
  }
};
void main() {
  Queue *queue = null;
  Integer *object1 = null;
  Integer *object2 = null;
  queue = new Queue();
  _addStrongReference(queue);

  object1 = new Integer(1);
  _addStrongReference(object1);

  object2 = new Integer(2);
  _addStrongReference(object2);

  queue->enqueue(object1);
  queue->enqueue(object2);

  Integer *test1 = object1;
  Integer *test2 = object2;
```

```
    cout << endl;
    cout << "Number of Strong References Pointing to Queue:" << endl;
    cout << queue->_numStrongReferences << endl;
    cout << "Number of Strong References Pointing to object1:" << endl;
    cout << test1->_numStrongReferences << endl;
    cout << "Number of Strong References Pointing to object2:" << endl;
    cout << test2->_numStrongReferences << endl;
    cout << endl;
    cout << "****REMOVING REFERENCES FROM OBJECTS****" << endl;
    cout << endl;
    _removeStrongReference(object1);
    object1 = null;
    _addStrongReference(object1);
    _removeStrongReference(object2);
    object2 = null;
    _addStrongReference(object2);

    cout << endl;
    cout << "Number of Strong References Pointing to Queue:" << endl;
    cout << queue->_numStrongReferences << endl;
    cout << "Number of Strong References Pointing to object1:" << endl;
    cout << test1->_numStrongReferences << endl;
    cout << "Number of Strong References Pointing to object2:" << endl;
    cout << test2->_numStrongReferences << endl;
    cout << endl;
    cout << "****REMOVING REFERENCES FROM QUEUE****" << endl;
    cout << endl;
    _removeStrongReference(queue);

    cout << endl;
    cout << "Number of Strong References Pointing to Queue:" << endl;
    cout << queue->_numStrongReferences << endl;
    cout << "Number of Strong References Pointing to object1:" << endl;
    cout << test1->_numStrongReferences << endl;
    cout << "Number of Strong References Pointing to object2:" << endl;
    cout << test2->_numStrongReferences << endl;
    _removeStrongReference(object1);
    _removeStrongReference(object2);
  }
```

Figure 35 – C++ Output Generated from the Circularly-Linked List F Implementation of a Queue

Once again, it appears that the F language compiler updated the strong and weak references appropriately. In order to see whether or not all the cyclic garbage was collected, the output from Figure 35 was compiled and executed. The results of running the executable are shown in Figure 36.

```
Number of Strong References Pointing to Queue:
1
```

```
Number of Strong References Pointing to object1:
1
Number of Strong References Pointing to object2:
1

****REMOVING REFERENCES FROM OBJECTS****


Number of Strong References Pointing to Queue:
1
Number of Strong References Pointing to object1:
0
Number of Strong References Pointing to object2:
0

****REMOVING REFERENCES FROM QUEUE****


Number of Strong References Pointing to Queue:
-572662307
Number of Strong References Pointing to object1:
-572662307
Number of Strong References Pointing to object2:
-572662307
```

Figure 36 – Output Generated from the F
Implementation of a Circularly-Linked Queue

As can be seen, the variables, queue, object1, and object2, each had a strong reference count of one before assigning the variables, object1 and object2, the value, null. Furthermore, after the variables, object1 and object2, were assigned the value, null, the strong reference count for the variables, queue, object1, and object2, became one, zero, and zero respectively. Lastly, after garbage collection was performed on the queue, the strong reference counts became garbage values for all three variables because they had been deleted. Therefore, it appears that cyclic garbage was successfully cleaned up in this particular example.

# 5. Findings and Discussion

## DESCRIPTION OF FINDINGS

In the previous chapter we were able to show that by using strong and weak references in constructs similar to smart pointers, we are able to compile Java-esque code into C++ code that can be compiled and executed properly without producing any memory leaks or any other memory-related problems, and the programmer does not have to perform any kind of memory management within the code. However, as was discussed in the section, Problems with Strong and Weak References, by using strong and weak references within our implementation of smart pointers, we may be producing executable images that are much more inefficient than what we would like them to be. This was certainly proven to be true when a comparison was done between F, C++, and Java in the section, Queue-Exhausted. However, the experience acquired from this experiment has provided some valuable insight into how a better garbage collector/compiler could possibly be made.

The cyclic structures that reference counting has difficulty garbage collecting are produced by member variables of objects. For the reasons discussed in the section, Referential Loop Problem, it was not desired to restrict the programmer from being able to create cyclic references with member data because of the fear that the programming language would be too restrictive. However, if the programming language can provide another way for programmers to create cyclic structures without giving them the capability to have objects reference each other cyclically, then it is possible that this programming language would be able to facilitate all the needs of all programmers.

If we impose a restriction on cyclic referencing, then we only need to use the techniques discussed in the section, Reference Counting, in order to perform garbage collection. As discussed in the section, Problems with Strong and Weak References, referencing counting would not change the expected time complexities of a programmer's code. In fact, if reference counting is used, traces will never be performed unnecessarily to determine if garbage exists; traces will only be performed when there is in fact garbage. Furthermore, the time complexity of performing the trace and performing garbage cleanup can be equivalent to the time complexity of the algorithms produced by the programmer if she were to perform efficient memory cleanup herself in a language like C or C++.

## FUTURE WORK

As a result of the findings produced by the experiments described in this paper, it is desired that strong and weak references are abandoned and instead create a language that uses a simpler reference counting implementation of smart pointers in its C++ code that is generated from the corresponding F language code. Furthermore, the F language will prohibit cyclic referencing from occurring within the programmers' code by producing compile-time errors whenever the programmer attempts to assign an object's member variable an object of the same class type or subclass type and disallow forward referencing of class types without their types already being defined. Despite the fact that this would make the language much more restrictive, as mentioned in the section, Referential Loop Problem, the F language will provide a construct to compensate for these restrictions so that the programmer will be able to produce any program that can be produced with a programming language like C, C#, or Java.

To begin with, the F language will provide all the container class types that are common to Java, such as hash maps, array lists, linked lists, tree maps, etc. Besides the more obvious reason that these classes should be provided to prevent programmers from re-inventing the wheel, these classes will be fundamental to the programming language because they will provide the only means by which a programmer can create lists of objects. In fact, all these class types will be considered as primitives by the F language and there will be no need to have "import" or "#include" statements to use any of them.

Next, the F language will provide a construct that is not standard in Java's API. The F language will provide a way to construct graphs using a graph container class type that will also be considered a primitive type by the F language. This class will allow programmers to store objects in it much like a hash map, where there will be an associated key/value pair for every object that is added. Furthermore, the graph container class will allow relationships or edges to be created between pairs of objects stored within it.

By providing a graph container class, programmers will be able to produce all the types of structures that can be produced in C, C++, and Java. They will be able to create B+ trees, PR-quad-trees, scene graphs, and any other structure that the F language will not provide to the programmer by default. Furthermore, they will be able to create these structures with greater ease because they will not have to re-invent the mechanism by which objects reference each other in such a structure; they simply need to add objects and create the relationships among them by using the methods already implemented in the graph container class.

By creating a language that provides all these features, it is possible that a programming language can be created that is much more efficient than languages like Java and C# and it will not burden the programmer with having to perform memory management or making any kind of calls to the garbage collector in order to make an application compatible with a real-time system. Also, it can be said that by imposing the aforementioned restrictions on the programmer that this language will also force programmers into adopting more proper software engineering principles. Therefore, this language might produce better engineered code in addition to it being easier to use and more memory efficient.

# SUMMARY

Because of the growing number of programming languages that do not burden programmers with having to manage memory, there is clearly a desire for languages that support automated memory management. Furthermore, there has been a lot of progress made with tracing collectors, such as the ones used by Java and C#. However, as a result of the inefficiencies with these garbage collectors, there is a requirement to have an understanding of how these garbage collectors work in order to make applications coded in these languages compatible with real-time systems. Furthermore, it is desired that garbage collectors be as efficient as if the programmers were to manage the memory efficiently themselves. In order to determine if it is possible to make a programming language that does not require memory management from the programmer and that does not require the programmer to have any kind of knowledge about the underlying implementation of the garbage collector, it was desired to get some hands on experience with creating a programming language that performs automated memory management, even if the techniques have already been used before, in order to get a better lower-level understanding of the work that is required.

Two techniques were used for automating memory management in a Java-esque language called, F, which was developed exclusively for doing this kind of research. The first technique was using standard reference counting. However, it was determined that it was inadequate at detecting referential loops in memory that were no longer accessible by the

user program. Therefore, a technique of using what were referred to as strong and weak references were integrated with reference counting so that referential loops could be dealt with. However, it was determined that integrating strong and weak references with reference counting created an unfortunate side-effect that the C++ generated from the F language code contained methods that affected the expected time complexities of the algorithms written by the developers.

Using strong and weak references along with reference counting, we were able to create a programming language that performed automated memory management appropriately for three examples. The first example was a simple queue example, which demonstrated that memory was being automatically managed appropriately with simple assignment statements. The second example was a more complicated queue example that involved adding two thousand five hundred objects to the queue, which showed the impact of using strong and weak references as they are implemented in this paper. The third example was a circularly-linked list implementation of a Queue, which demonstrated that cyclic garbage could be collected using strong and weak references as they are implemented in this paper.

As a result of the hands on experience with creating a programming language that automates the task of memory management, it was decided that it would be better if the strong and weak references were not used and that the language impose restrictions on the users, prohibiting them from creating cyclic references with their objects. With these restrictions, simple reference counting would be sufficient enough for determining when garbage collection should be performed. This would ensure that we would not have a "stop-the-world" garbage collector. Furthermore, the language would provide a whole suite of container classes, such as array lists, linked lists, hash maps, and tree maps for creating lists of objects. In addition, the language would provide an alternative means for creating cyclic structures by providing a graph container class as a primitive language type, which allows developers to create all kinds of structures, such as B+ trees, PR Quad trees, scene graphs, etc.

# APPENDIX A: F LANGUAGE REFERENCE GUIDE

The F Language is a Java-esque programming language where everything will be defined within classes using C++-styled syntax. Like Java and all object-oriented languages, F will facilitate programmers with inheritance, polymorphism, and information hiding in addition to the standard programming constructs typically encountered within imperative languages, such as subprograms (functions), variables, and complex expressions. This language does not facilitate some of the more advanced features offered by languages like C++ and Java, such as the ability to import files into code through the use of #include or import statements.

## LEXICAL DESCRIPTION

A description of the lexemes encountered in the F language is provided in the following bulleted list. This list is not meant to be comprehensive but rather a general overview of the lexical structures found in the F language. In order to get a complete description of the lexical structures that exist in the F language, refer to Appendix B: F Language Lex Source Code.

- F is case sensitive
- **IDs (Identifiers)**: An identifier consists of a single alphabetic character followed by a sequence of alphanumeric characters and underscores.

- **Reserved Keywords**: else Boolean Character class private protected public Real return false if Integer new null static String true Void while
- **Assignment Operator**: =
- **Arithmetic Operators**: * + / -
- **Increment**: ++
- **Relational and Equality Operators**: == >= > <= < !=
- **Logical Operators**: ! && ||
- **Index Range Operator**: ..
- **Miscellaneous Symbols**: { } , [ ] . ( ) ;
- **Real Literal**: One or more numeric values followed a dot and one or more numeric values.
- **Integer Literal**: One or more numeric values.
- **String Literal**: A quote character followed by zero or more non-quote and non-backslash characters or a backslash character followed by any character, terminated with an ending quote character.
- **Comments**: Comments may begin with // and end with a newline character or they may begin with /* and end with */. The comments cannot be nested.

The index range operator essentially specifies a range of values being indexed within an array. This makes it easier for the programmer to obtain substrings and to perform memory copies for specified ranges of arrays. An example is given in Figure 37.

```
...
// copying elements 0 through 2 of someArray2 into
// someArray1
someArray1[0..2] = someArray2[0..2];
...
```

Figure 37 – Example Where the Index Range Operator is Being Used to Copy Array Elements

## SYNTAX

In order to learn how to write in F, we should probably start off by using an example. As our first example, we shall use the notorious "Hello, World!" program with a little more complexity added to show some more of the features provided by the language. A "Hello, World!" example is shown in Figure 38.

```
/**
 * A simple HelloWorld class.
 * @author Stephen Hoskins, George Mason University
 * @version 1.0, 11/16/07
 */
public class HelloWorld {

  /**
   * Stores the "Hello, World!" message.
   */
  private String helloWorldMessage;

  /**
   * Creates a new instance of a HelloWorld class.
   */
  public HelloWorld() {

    helloWorldMessage = "Hello, World!";
  } // end default constructor

  /**
   * The main entry-point into the application.
   */
  public static void main() {
    // integer used for doing a countdown
    Integer i = new Integer(3);
    while (i > 0) {
      println(i);
      i++;
    }

    // print "Hello, World!"
    println(helloWorldMessage);
  } // end main()

} // end class HelloWorld
```

Figure 38 – A “Hello, World!” Example Written in F

As described earlier, everything in the F language is written within classes. As shown in Figure 38, you may have variables declared at the class level, such as the variable, helloWorldMessage, and you may have variables declared at the function level, such as the variable, i, in the function, main. Furthermore, in order for a program to be converted into an executable, it must have a function, main, which is the main entry-point into the application. The output generated by this program is:

```
3
2
1
Hello, World!
```

## Access Modifiers

Access modifiers facilitate information hiding within F. In particular, they specify whether a given class, member function, or member variable will be accessible from other classes through references or through inheritance. Anything specified as private, can only be accessed by other members of the same class. Anything specified as protected can only be accessed by other members of the same class and all subclasses.

Anything specified as public can be accessed anywhere, as long as it is within scope. The syntax for access modifiers is:

access_modifiers → “private” | “protected” | “public”

**Variables**

Variables are structures that store information relevant to a program. In addition, they may exist as various different types. In order to see how to declare a variable to be of a particular type, see the section, Declarations.

Variables may be used in many different ways in order to extract the information that they contain. If a variable is a class type for instance, then member variables of that class type can be referenced by the variable by using the dot operator (‘.’) and then specifying the name of the member variable. If the variable is an array, then particular elements of the array can be referenced by using brackets (“[ ]”) to specify the index of the array. The different ways that variables can be referenced are:

- **ID**: The simplest way to reference a variable is by using its identifier. However, if the variable is a class type and you wish to access its member variables of if the variable is an array and you wish to reference a specific index of the array, then more sophisticated methods will have to be used.

- **Indexing**: If it is desired that a specific index or a specific range of an array is to be referenced, then brackets ("[ ]") will have to be used –e.g., someArray[5] or someArray[1..3].
- **Member Variable**: If a variable is a class type, then member variables can referenced by using the dot operator ('.') if the member variable is made public by the class –e.g., object . memberVariable.
- **Function call**: A variable can be referenced through a function call that happens to return a variable –e.g., returnHello(), which happens to return a string, "Hello."
- **New Instance**: A variable can be referenced though the creation of a new instance of a variable using the operator, new –e.g., new Object().

## Types

There are two kinds of types that a variable may be in the F language. It may either be one of the existing primitive types that exist within the F language or it may be a user defined type. The primitive types within the F language are Boolean, Character, Integer, Object, Real, String, and Void. User defined types are simply classes and once a programmer defines a class, a variable of that type may be created. The syntax for a type in F is:

type → "Boolean" | "Character" | **ID** | "Integer" | "Object" | "Real" | "String" | "Void"

## Expressions

Expressions are fundamental components in the description of the F language syntax. These components make it possible to perform arithmetic computations within the F language, however they may also be used for other things as well. The different types of expressions are:

- **Arithmetic**: Expressions are of the form *expression op expression*, where *op* is +, *, -, or /. Arithmetic expressions require the parameters to be either of type Integer or Real and produces a result either of type Integer or Real respectively. All operators are left associative and the operators, multiplication and division, have a higher precedence than the operators, addition and subtraction.

  expression → expression “+” expression | expression “–” expression |
  expression “*” expression | expression “/” expression

- **Negation**: Expressions either of type Integer or Real may have their values negated by prefixing a minus sign to the beginning of the expression.

  expression → “–” expression

- **Precedence Override**: The precedence of the operators in an expression may be overridden by the use of parenthetical expressions.

expression → “ ( ” expression “ ) ”

- **Single value**: Single value expressions merely consist of a single value or a variable. The syntax for single value expressions is:

expression → value

value → variable | variable “++” | **Integer_Literal** | **Real_Literal** | **String_Literal** | “null”

**Parameters**

Parameters are essentially used in two places with the F language. They are used within functions and they are used for indexing arrays. For arrays, it is required that parameters are of type, Integer. However, for functions they may be of any type. This requires that the types of the parameters be declared within function definitions. The syntax for parameters and parameter declarations respectively is:

parameters → parameters “ , ” expression | value

parameter_declarations → parameter_declarations “ , ” parameter_declaration | parameter_declaration

parameter_declaration → type **ID** | type **ID** “ [ ” “ ] ”

**Statements**

A statement is a single instruction that is to be performed in the F language. The different types of statements that can occur in F are declarations, conditionals, assigjnments, expressions and return statements. To learn more about declarations, conditionals, assignments, expressions, and return statements, see each of the corresponding sections that pertain to these topics. The syntax for statements is:

statements → statements statement

statements →

statement → declaration | conditional | assignment | expression " ; " | return_statement

**Declarations**

Declarations are statements that create variables of a specified type for use within the F language. The three main types of declarations that can be created in the F language are what are called global declarations, local declarations, and local array declarations. Each type of declaration is described in more detail below.

1. **Global Declarations**: Global declarations are declarations defined outside of functions as attributes to an encompassing class. These variables are global to all functions that reside within the class that these variables are attributes of. These kinds of declarations are restrictive in the sense that they cannot be initialized within the declaration statement itself; instead these variables will have to be

initialized within a constructor of the class. An example of a global declaration would be the declaration of the variable, helloWorldMessage, in Figure 38. The syntax for global declarations is:

global_declaration → access_modifiers type **ID** " ; "

2. **Local Declarations**: Local declarations are declarations defined within a function of a class. Unlike global declarations, local declarations can be initialized. If a variable is a class type, then it can be initialized by creating a new class using the "new" operator. An example of a local declaration being initialized with the new operator can be seen with the declaration of the variable, i, in Figure 38. The syntax for a local declaration is:

   declaration → type **ID** " ; " | type **ID** "=" expression " ; "

3. **Array Declarations (Local)**: Arrays may also be defined within a function of a class. The syntax for declaring an array is very similar to declaring a local variable except that the size of the array must be specified. The syntax for an array is:

   declaration → type **ID** " [ " parameters " ] " " ; "

## Functions

Functions contain portions of code that perform particular tasks within a program in F. Functions can be called from other functions and they have the ability to perform tasks based on the parameters that they are given. Furthermore, functions have the ability to return results to a calling statement. Functions play a very important role for modularizing functionality and for achieving better readability within code. To see an example of a function call, see the section on Variables. The syntax for defining functions is:

function_definition → access_modifiers type **ID** "(" opt_parameter_declarations ")" "{" statements "}" **|** access_modifiers "static" type **ID** "(" opt_parameter_declarations ")" "{" statements "}" | access_modifiers **ID** "(" opt_parameter_declarations ")" "{" statements "}"

opt_parameter_declarations → parameter_declarations

opt_parameter_declarations →

## Classes

Classes provide two very important features in F. One, classes provide a way for defining user-defined types in F. Two, classes also facilitate modularizing similar functionality in a program, which is good for readability.

Everything in F must reside within a class. This is to ensure that F is truly object-oriented. The syntax for defining classes is:

class_definition → access_modifiers “class” **ID** “{” definitions_and_declarations “}”

definitions_and_declarations → definitions_and_declarations
    definition_or_declaration

definitions_and_declarations →

definition_or_declaration → class_definition | function_definition |
    global_declaration

## Assignments

Assignments allow for the storage of information into variables in F. It can be used for storing the results from arithmetic expressions, but it can be used for storing strings, characters, and references as well. In addition, typecasting is supported within assignment statements. The syntax for assignments is:

assignment → variable “=” expression “ ; ” | variable “=” “(” type “)” expression “ ; ”

## Conditionals

Conditionals allow programs to make decisions within the F language. There are many ways that this can be done within F. The different ways that this can be done are:

1. **If/Else If/Else Statements**: If/else if/else statements evaluate the given conditions and if the given conditions turn out to be true, then it performs the given statements. If the given conditions are not true, then there are ways to specify

what should occur in these cases as well. The syntax for if/else if/else statements is:

> conditional → "if" "(" conditions ")" "{" statements "}" | "if" "(" conditions ")" "{" statements "}" "else" "{" statements "}" | "if" "(" conditions ")" "{" statements "}" elseifs "else" "{" statements "}" | "if" "(" conditions ")" "{" statements "}" elseifs | "if" "(" conditions ")" statement
>
> conditions → conditions "||" conditions | conditions "&&" conditions | " ! " conditions | "(" conditions ")" | condition
>
> condition → expression "<" expression | expression "<=" expression | expression ">" expression | expression ">=" expression | expression "==" expression | expression "!=" expression
>
> elseifs → elseifs elseif | elseif
>
> elseif → "else" "if" "(" conditions ")" "{" statements "}"

2. **While Loops**: While loops evaluate the given conditions and so long as the conditions are true, the given statements will continuously be performed until the given conditions should happen to be false. Note that it is possible to create infinite loops this way. The syntax for while loops is:

> conditional → "while" "(" conditions ")" "{" statements "}"

## Return Statements

Return statements are used for returning variables from a function call. Furthermore, return statements can be used for preemptively terminating a function early before the last line of code in the function is reached. In order for an executable image to be produced from a program written in F, it is important that the type of the variable being returned in a return statement match the return type that has been specified in the definition of the function. The syntax for return statements is:

return_statement → "return" " ; " | "return" expression " ; "

## Standard Library

A couple functions are predefined in F. These functions are:

- Object println(Integer integer) – Prints an integer to standard out with a newline character concatenated at the end.
- Void memcpy(Integer dest, Integer destLength, Integer src) – Copies the given source integer array into the given destination integer array of size, destLength.

## Complete F Syntax

program → class_definition

class_definition → access_modifiers "class" **ID** "{" definitions_and_declarations "}"

definitions_and_declarations → definitions_and_declarations
    definition_or_declaration

definitions_and_declarations →

definition_or_declaration → class_definition | function_definition |

global_declaration

function_definition → access_modifiers type **ID** "(" opt_parameter_declarations ")"

"{" statements "}" **|** access_modifiers "static" type **ID** "("

opt_parameter_declarations ")" "{" statements "}" | access_modifiers **ID** "("

opt_parameter_declarations ")" "{" statements "}"

opt_parameter_declarations → parameter_declarations

opt_parameter_declarations →

global_declaration → access_modifiers type **ID** " ; "

type → "Boolean" | "Character" | **ID** | "Integer" | "Object" | "Real" | "String" | "Void"

declaration → type **ID** " ; " | type **ID** "=" expression " ; " | type **ID** "[" parameters "]"

" ; "

expression → expression "+" expression | expression "–" expression | expression "*"

expression | expression "/" expression | "–" expression | "(" expression ")" | value

value → variable | variable "++" | **Integer_Literal** | **Real_Literal** | **String_Literal** |

"null"

conditional → "if" "(" conditions ")" "{" statements "}" | "if" "(" conditions ")" "{"

statements "}" "else" "{" statements "}" | "if" "(" conditions ")" "{" statements

"}" elseifs "else" "{" statements "}" | "if" "(" conditions ")" "{" statements "}"

elseifs | "if" "(" conditions ")" statement conditions → conditions “||” conditions | conditions “&&” conditions | “ ! ” conditions | “(” conditions “)” | condition

condition → expression “<” expression | expression “<=” expression | expression “>” expression | expression “>=” expression | expression “==” expression | expression “!=” expression

elseifs → elseifs elseif | elseif

elseif → “else” “if” “(” conditions “)” “{” statements “}”

parameters → parameters “ , ” expression | value

parameter_declarations → parameter_declarations “ , ” parameter_declaration | parameter_declaration

parameter_declaration → type **ID** | type **ID** “[” “]”

statements → statements statement

statements →

statement → declaration | conditional | assignment | expression “ ; ” | return_statement

return_statement → “return” “ ; ” | “return” expression “ ; ”

assignment → variable “=” expression “ ; ” | variable “=” “(” type “)” expression “ ; ”

access_modifiers → “private” | “protected” | “public”

variable → variable “ . ” variable | **ID** | **ID** “[” expression “]” | **ID** “[” expression “ . . ” expression “]” | **ID** “(” opt_parameters “)” | “new” **ID** “(” opt_parameters “)”

opt_parameters → parameters

opt_parameters →

# APPENDIX B: F LANGUAGE LEX SOURCE CODE

```
/*****************************************************************************
 *                                                                           *
 *                             Stephen Hoskins                               *
 *                               G-00337381                                  *
 *                         George Mason University                           *
 *                                  CS707                                    *
 *                               Summer 2007                                 *
 *                              Thesis Project                               *
 *                                                                           *
 *****************************************************************************/

%{

#include <string.h>
#include "y.tab.h"

/**
 * Stores the line currently being parsed.
 */
int lineNumber = 1;

%}

%x COMMENT1
%x COMMENT2

%%

"++" {
  return INCREMENT;
}

\+ {
  return ADD;
}

"==" {
  return EQUALS;
}

else {
  return ELSE;
}

\= {
  return ASSIGNMENT;
}

Boolean {
  return BOOLEAN;
}

Character {
  return CHARACTER;
}

\, {
  return COMMA;
}

class {
  return CLASS;
}

\/ {
  return DIVIDE;
}

".." {
  return DOT_DOT;
}

\. {
  return DOT;
}

private {
  return PRIVATE;
}

protected {
  return PROTECTED;
}

public {
  return PUBLIC;
}

Real {
  return REAL;
}

"return" {
  return RETURN;
}

false {
  return FALSE_T;
}

">=" {
  return GREATER_THAN_OR_EQUALS;
}

">" {
  return GREATER_THAN;
}

if {
  return IF;
}

Integer {
  return INTEGER;
}

"[" {
  return LEFT_BRACKET;
}

\{ {
  return LEFT_CURLY_BRACE;
}

\( {
  return LEFT_PARENTHESIS;
}
"<=" {
  return LESS_THAN_OR_EQUALS;
}

"<" {
  return LESS_THAN;
}

"&&" {
  return LOGICAL_AND;
}

"||" {
  return LOGICAL_OR;
}

\* {
  return MULTIPLY;
}

"new" {
  return NEW;
}

"null" {
  return NULL_VALUE;
}

"!=" {
  return NOT_EQUALS;
}

"!" {
  return NOT;
}

"]" {
  return RIGHT_BRACKET;
}

\} {
  return RIGHT_CURLY_BRACE;
}

\) {
  return RIGHT_PARENTHESIS;
}

\; {
  return SEMICOLON;
}

static {
  return STATIC;
}

String {
  return STRING;
}

\- {
  return SUBTRACT;
}

true {
  return TRUE_T;
}
```

```
Void {
  return VOID;
}

while {
  return WHILE;
}

[a-zA-Z][a-zA-Z0-9_]* {
  yylval.c_ptr = (char*) malloc(sizeof(char) * strlen(yytext));
  strcpy(yylval.c_ptr, yytext);
  return ID;
}

[0-9]+"."[0-9]+ {
  yylval.c_ptr = (char*) malloc(sizeof(char) * strlen(yytext));
  strcpy(yylval.c_ptr, yytext);
  return REAL_VALUE;
}

[0-9]+ {
  yylval.c_ptr = (char*) malloc(sizeof(char) * strlen(yytext));
  strcpy(yylval.c_ptr, yytext);
  return INTEGER_VALUE;
}

\"([^\"\\]|\\.)*\" {
  yylval.c_ptr = (char*) malloc(sizeof(char) * strlen(yytext));
  strcpy(yylval.c_ptr, yytext);
  return STRING_VALUE;
}

"//" {BEGIN COMMENT1;}

"/*" {BEGIN COMMENT2;}

[\n] {lineNumber++;}

[ \t\r] ;

<COMMENT1>\n {
  BEGIN 0;
  lineNumber++;
}

<COMMENT1>. ;

<COMMENT2>\n {lineNumber++;}

<COMMENT2>\*\/ {BEGIN 0;}

<COMMENT2>. ;

. {
  return -1;
}

%%
```

# APPENDIX C: F LANGUAGE YACC SOURCE CODE

```
/******************************************************************************
 *                                                                            *
 *                              Stephen Hoskins                               *
 *                                G-00337381                                  *
 *                           George Mason University                          *
 *                                   CS707                                    *
 *                                Summer 2007                                 *
 *                               Thesis Project                               *
 *                                                                            *
 ******************************************************************************/

%{

#include <deque>
#include <map>
#include <string>
#include <string.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <iostream>

using namespace std;

#ifndef null
#define null 0
#endif

/**
 * This specifies that the tokens captured in the non-terminal, value, are the
 * of token type, INTEGER_VALUE, REAL_VALUE, or STRING_VALUE.
 */
const int NON_VALUE = 0;

/**
 * Specifies that the non-terminal, statements, is empty.
 */
const int NO_STATEMENT = 0;

/**
 * Specifies that the non-terminal, statement, is a declaration.
 */
const int DECLARATION_STATEMENT = 1;

/**
 * Specifies that the non-terminal, statement, is a conditional.
 */
const int CONDITIONAL_STATEMENT = 2;

/**
 * Specifies that the non-terminal, statement, is an assignment.
 */
const int ASSIGNMENT_STATEMENT = 3;

/**
 * Specifies that the non-terminal, statement, is an expression.
 */
const int EXPRESSION_STATEMENT = 4;

/**
 * Specifies that the non-terminal, statement, is an return statement.
 */
const int RETURN_STATEMENT = 5;

/**
 * Prints out parse exceptions.
 */
int yyerror(char *s);

/**
 * Needed to compile.
 */
int yylex();

/**
 * Stores the name of the class currently being parsed.
 */
string className;

/**
 * Returns a string with the given format and variables.  Each time the
 * character, '$', is encountered it is replaced with a given variable
 * respectively.
 * @param format the given format
 * @param ... the given variables or 0 to specify that there are no more
 * variables; the memory for each variable will automatically be de-allocated
 * by this function
 * @return a string with the given format and variables
 */
char *createString(const char *format, ...);

/**
 * Returns the given type with all the asterisk characters removed.
 * @param type the given type
 */
char *removeAsterisks(char* type);

/**
 * Returns a copy of the given string.
 * @param string the given string
 * @return a copy of the given string
 */
char *copyString(const char* string);

/**
 * Returns everything to the left of the "->" operator in the given string.
 * @param string the given string
 * @return everything to the left of the "->" operator in the given string
 */
char *getDependent(const char* string);

/**
 * Pushes all the global variables, such as className and initializeAttributes,
 * onto their respective stacks and reinitializes them.
 */
void pushGlobals();

/**
 * Sets all the global variables to the last elements on their respective
 * stacks.
 */
void popGlobals();

/**
 * Set to true if a return variable from a function needs to have a reference
 removed; false otherwise.
 */
bool clearReturnVariable = false;
```

```
/**
 * Set to true if the class being parsed is the outer most class defined in the
 * source code.
 */
bool mainClass = true;

/**
 * Stores the code for the main function, is one is defined.  This is stored
 * because special handling is required for the main function -i.e., the
 * function must be defined outside the main class in the C++ code generated
 * from this compiler.
 */
char *mainFunction = null;

/**
 * A stack for storing all the attributes of the classes being parsed.  Whenever
 * a new inner class is being parsed, initializeAttributes is pushed onto this
 * stack.  Whenever the end of an inner class is reached during a parse,
 * initializeAttributes is reassigned to whatever is on top of this stack.
 */
deque<deque<string>*> attributesStack;

/**
 * A stack for storing all the names of the classes being parsed.  Whenever a
 * new inner class is being parsed, className is pushed onto this
 * stack.  Whenever the end of an inner class is reached during a parse,
 * className is reassigned to whatever is on top of this stack.
 */
deque<string> classNameStack;

/**
 * Stores all the references that need to be removed at the end of a function
 * or before a return statement.
 */
deque<string> removeReferences;

/**
 * Stores all the variables that need to be declared at the beginning of a
 * function.
 */
deque<string> createDeclarations;

/**
 * Stores all the member variables that need to be initialized at the beginning
 * of a constructor.
 */
deque<string> *initializeAttributes = new deque<string>();

/**
 * A unique identifier that can be used for creating unique names for variables
 * that must be declared in the C++ code generated by this compiler.
 */
long int uId = 1;

/**
 * Stores all the arrays that have been declared.
 */
map<string, bool> arrayTable;

/**
 * Stores all the member variables that have been declared for the current
 * class.
 */
map<string, string> attributesTable;

/**
 * Stores all the variables that have been declared for the current function.
 */
map<string, string> declarationTable;

%}

/**
 * Structs that store information captured by the non-terminals.
 */
%union {
  char *c_ptr;
  struct s1 {char *value; char *array_length; int type;} expression_type;
  struct s2 {char *value; int type;} statement_type;
}

/**
 * The tokens.
 */
%token ADD
%token ASSIGNMENT
%token BOOLEAN
%token CHARACTER
%token CLASS
%token COMMA
%token DIVIDE
%token DOT
%token DOT_DOT
%token ELSE
%token EQUALS
%token FALSE_T
%token GREATER_THAN
%token GREATER_THAN_OR_EQUALS
%token <c_ptr> ID
%token IF
%token INCREMENT
%token INTEGER
%token <c_ptr> INTEGER_VALUE
%token LEFT_BRACKET
%token LEFT_CURLY_BRACE
%token LEFT_PARENTHESIS
%token LESS_THAN
%token LESS_THAN_OR_EQUALS
%token LOGICAL_AND
%token LOGICAL_OR
%token MULTIPLY
%token NEW
%token NOT
%token NOT_EQUALS
%token NULL_VALUE
%token PRIVATE
%token PROTECTED
%token PUBLIC
%token REAL
%token RETURN
%token <c_ptr> REAL_VALUE
%token RIGHT_BRACKET
%token RIGHT_CURLY_BRACE
%token RIGHT_PARENTHESIS
%token SEMICOLON
%token STATIC
%token STRING
%token <c_ptr> STRING_VALUE
%token SUBTRACT
%token TRUE_T
%token VOID
%token WHILE
%type <c_ptr> assignment
%type <c_ptr> access_modifiers
```

```
%type <c_ptr> condition
%type <c_ptr> conditional
%type <c_ptr> conditions
%type <c_ptr> declaration
%type <c_ptr> elseif
%type <c_ptr> elseifs
%type <expression_type> expression
%type <c_ptr> global_declaration
%type <c_ptr> opt_parameters
%type <c_ptr> opt_parameter_declarations
%type <c_ptr> parameter_declarations
%type <c_ptr> parameter_declaration
%type <c_ptr> parameters
%type <c_ptr> return_statement
%type <statement_type> statement
%type <statement_type> statements
%type <c_ptr> type
%type <expression_type> value
%type <expression_type> variable

/**
 * Precedence overriding.
 */
%left LOGICAL_OR
%left LOGICAL_AND
%left EQUALS NOT_EQUALS LESS_THAN LESS_THAN_OR_EQUALS GREATER_THAN
GREATER_THAN_OR_EQUALS
%right ASSIGNMENT
%left ADD SUBTRACT
%left MULTIPLY DIVIDE
%nonassoc UMINUS NOT
%nonassoc INCREMENT
%left DOT

%%

/**
 * Describes the syntax for an F language program.
 */
program:
  {
    printf("#include \"FLang.h\"\n");
  }
  class_definition
  {
    if (mainFunction != null) {
      printf("%s", mainFunction);
    }
  }
; // end program

/**
 * Describes the syntax for a class.
 */
class_definition:
  access_modifiers
  CLASS
  ID
  LEFT_CURLY_BRACE {
    pushGlobals();
    className = $3;
    if (!mainClass) {
      printf("%sclass %s : public Object {\n", $1, $3);
    }
    else {
      mainClass = false;
      printf("class %s : public Object {\n", $3);
    }
    printf("%s *_%s_this;\n", $3, $3);
    free($1);
    free($3);
  }
  definitions_and_declarations
  RIGHT_CURLY_BRACE {
    printf("};\n");
    popGlobals();
  }
; // end class_definition

/**
 * Describes the syntax for either class/function definitions or declarations.
 */
definitions_and_declarations:
  definitions_and_declarations
  definition_or_declaration
|
; // end definitions_and_declarations

/**
 * Describes the syntax for either class/function definitions or declarations.
 */
definition_or_declaration:
  class_definition
| function_definition
| global_declaration {
    printf("%s\n", $1);
  }
;

/**
 * Describes the syntax for a function definition.
 */
function_definition:
  access_modifiers
  type
  ID
  LEFT_PARENTHESIS
  opt_parameter_declarations
  RIGHT_PARENTHESIS
  LEFT_CURLY_BRACE
  statements
  RIGHT_CURLY_BRACE {
    if (strcmp($2, "void") == 0) {
      printf("%s%s %s(%s) {\n", $1, $2, $3, $5);
    }
    else {
      printf("%s%s *%s(%s) {\n", $1, $2, $3, $5);
    }
    for (int i = 0; i < createDeclarations.size(); i++) {
      printf("%s\n", createDeclarations[i].c_str());
    }
    createDeclarations.clear();
    printf("%s", $8.value);
    free($1);
    free($2);
    free($3);
    free($5);
    free($8.value);
    if ($8.type != RETURN_STATEMENT) {
      for (int i = 0; i < removeReferences.size(); i++) {
        printf("%s\n", removeReferences[i].c_str());
      }
    }
    removeReferences.clear();
    printf("}\n");
  }
| access_modifiers
  STATIC
  type
  ID
  LEFT_PARENTHESIS
  opt_parameter_declarations
  RIGHT_PARENTHESIS
  LEFT_CURLY_BRACE
  statements
  RIGHT_CURLY_BRACE {
    if (strcmp($4, "main") != 0) {
      printf("%sstatic %s %s(%s) {\n", $1, $3, $4, $6);
      for (int i = 0; i < createDeclarations.size(); i++) {
        printf("%s\n", createDeclarations[i].c_str());
      }
      createDeclarations.clear();
      printf("%s", $9.value);
      free($1);
      free($3);
      free($4);
      free($6);
      free($9.value);
      if ($9.type != RETURN_STATEMENT) {
        for (int i = 0; i < removeReferences.size(); i++) {
          printf("%s\n", removeReferences[i].c_str());
        }
      }
      removeReferences.clear();
      printf("}\n");
    }
    else {
      mainFunction = createString("void main($) {\n", $6, 0);
      for (int i = 0; i < createDeclarations.size(); i++) {
        mainFunction = createString("$$\n", mainFunction,
            copyString(createDeclarations[i].c_str()), 0);
      }
      createDeclarations.clear();
      mainFunction = createString("$$", mainFunction, $9.value, 0);
      if ($9.type != RETURN_STATEMENT) {
        for (int i = 0; i < removeReferences.size(); i++) {
          mainFunction = createString("$$\n", mainFunction,
              copyString(removeReferences[i].c_str()), 0);
        }
      }
      removeReferences.clear();
      mainFunction = createString("$}\n", mainFunction, 0);
      free($1);
      free($3);
      free($4);
    }
  }
| access_modifiers
  ID
  LEFT_PARENTHESIS
  opt_parameter_declarations
  RIGHT_PARENTHESIS
  LEFT_CURLY_BRACE
  statements
  RIGHT_CURLY_BRACE {
    printf("%s%s(%s) {\n", $1, $2, $4);
    printf("_%s_this = this;\n", className.c_str());
    for (int i = 0; i < initializeAttributes->size(); i++) {
      printf("%s\n", (*initializeAttributes)[i].c_str());
    }
    for (int i = 0; i < createDeclarations.size(); i++) {
      printf("%s\n", createDeclarations[i].c_str());
    }
    createDeclarations.clear();
    printf("%s", $7.value);
    free($1);
    free($2);
    free($4);
    free($7.value);
    if ($7.type != RETURN_STATEMENT) {
      for (int i = 0; i < removeReferences.size(); i++) {
        printf("%s\n", removeReferences[i].c_str());
      }
    }
    removeReferences.clear();
    printf("}\n");
  }
; // end function_definition

/**
 * Describes the syntax for a member variable declaration.
 */
global_declaration:
  access_modifiers
  type
  ID
  SEMICOLON {
    declarationTable[$3] = $2;
    attributesTable[$3] = className;
    $$ = createString("$$ *$;", $1, $2, copyString($3), 0);
    initializeAttributes->push_back(createString("$ = null;", $3, 0));
  }
; // end global_declaration

/**
 * Describes the syntax for a function variable declaration.
 */
declaration:
  type
  ID
  SEMICOLON {
    declarationTable[$2] = $1;
    $$ = createString("$ = null;\n", copyString($2), 0);
    createDeclarations.push_back(createString("$ *$ = null;", $1, $2, 0));
  }
| type
  ID
  LEFT_BRACKET
  parameters
  RIGHT_BRACKET
  SEMICOLON {
    declarationTable[$2] = $1;
    arrayTable[$2] = true;
    $$ = createString(
        "$ = new $[_toInt($)];\n_addStrongReference($);\n_$_length = new
Integer($);\n_addStrongReference(_$_length);\n",
        copyString($2), copyString($1), $4, copyString($2), copyString($2),
        copyString($4), copyString($2), 0);
    createDeclarations.push_back(createString(
        "$ *$ = null;\nInteger *_$_length = null;", $1,
        copyString($2), copyString($2), 0));
    char *removeReference = createString(
        "_removeStrongReference($);", copyString($2), 0);
    removeReferences.push_back(removeReference);
    free(removeReference);
    removeReference = createString(
        "_removeStrongReference(_$_length);", $2, 0);
    removeReferences.push_back(removeReference);
    free(removeReference);
  }
| type
  ID
  ASSIGNMENT
  expression
  SEMICOLON {
    declarationTable[$2] = $1;
    createDeclarations.push_back(createString("$ *$ = null;", copyString($1),
      copyString($2), 0));
    if ($4.type == NON_VALUE || $4.type == NULL_VALUE) {
      $$ = createString("$ = $;\n_addStrongReference($);\n", copyString($2),
          $4.value, copyString($2), 0);
    }
    else {
      $$ = createString("$ = new $($);\n_addStrongReference($);\n",
          copyString($2), copyString($1), $4.value, copyString($2), 0);
    }
    free($1);
    char *removeReference = createString(
        "_removeStrongReference($);", $2, 0);
    removeReferences.push_back(removeReference);
    free(removeReference);
    if ($4.array_length != null) {
      free($4.array_length);
    }
  }
; // end declaration

/**
 * Describes the syntax for arithmetic expressions.
 */
expression:
  expression
  ADD
  expression {
    if ($1.type == NON_VALUE) {
      if ($3.type == NON_VALUE) {
        $$.value = createString("*$ + *$", $1.value, $3.value, 0);
      }
      else {
        $$.value = createString("*$ + $", $1.value, $3.value, 0);
      }
    }
    else {
      if ($3.type == NON_VALUE) {
        $$.value = createString("$ + *$", $1.value, $3.value, 0);
      }
      else {
        $$.value = createString("$ + $", $1.value, $3.value, 0);
      }
    }
    $$.array_length = null;
    if ($1.array_length != null) {
      $$.array_length = copyString($1.array_length);
      free($1.array_length);
    }
    if ($3.array_length != null) {
      free($3.array_length);
    }
    $$.type = INTEGER_VALUE;
  }
| expression
  SUBTRACT
  expression {
    if ($1.type == NON_VALUE) {
      if ($3.type == NON_VALUE) {
        $$.value = createString("*$ - *$", $1.value, $3.value, 0);
      }
      else {
        $$.value = createString("*$ - $", $1.value, $3.value, 0);
      }
    }
    else {
      if ($3.type == NON_VALUE) {
        $$.value = createString("$ - *$", $1.value, $3.value, 0);
      }
      else {
        $$.value = createString("$ - $", $1.value, $3.value, 0);
      }
    }
    $$.array_length = null;
    if ($1.array_length != null) {
      $$.array_length = copyString($1.array_length);
      free($1.array_length);
    }
    if ($3.array_length != null) {
      free($3.array_length);
    }
    $$.type = INTEGER_VALUE;
  }
| expression
  MULTIPLY
  expression {
    if ($1.type == NON_VALUE) {
      if ($3.type == NON_VALUE) {
        $$.value = createString("*$ * *$", $1.value, $3.value, 0);
      }
      else {
        $$.value = createString("*$ * $", $1.value, $3.value, 0);
      }
    }
    else {
      if ($3.type == NON_VALUE) {
        $$.value = createString("$ * *$", $1.value, $3.value, 0);
      }
      else {
        $$.value = createString("$ * $", $1.value, $3.value, 0);
      }
    }
    $$.array_length = null;
    if ($1.array_length != null) {
      $$.array_length = copyString($1.array_length);
      free($1.array_length);
    }
    if ($3.array_length != null) {
      free($3.array_length);
    }
    $$.type = INTEGER_VALUE;
  }
| expression
  DIVIDE
  expression {
    if ($1.type == NON_VALUE) {
      if ($3.type == NON_VALUE) {
        $$.value = createString("*$ / *$", $1.value, $3.value, 0);
      }
      else {
        $$.value = createString("*$ / $", $1.value, $3.value, 0);
      }
    }
    else {
      if ($3.type == NON_VALUE) {
        $$.value = createString("$ / *$", $1.value, $3.value, 0);
      }
      else {
        $$.value = createString("$ / $", $1.value, $3.value, 0);
      }
    }
    $$.array_length = null;
    if ($1.array_length != null) {
      $$.array_length = copyString($1.array_length);
      free($1.array_length);
    }
    if ($3.array_length != null) {
      free($3.array_length);
    }
    $$.type = INTEGER_VALUE;
  }
| SUBTRACT
  expression
  %prec UMINUS {
    if ($2.type == NON_VALUE) {
      $$.value = createString("-*$", $2.value, 0);
    }
    else {
      $$.value = createString("-$", $2.value, 0);
    }
    $$.array_length = null;
    if ($2.array_length != null) {
      $$.array_length = copyString($2.array_length);
      free($2.array_length);
    }
    $$.type = INTEGER_VALUE;
  }
| LEFT_PARENTHESIS
  expression
  RIGHT_PARENTHESIS {
    $$.value = (char*) malloc(sizeof(char) * (strlen($2.value) + 3));
    sprintf($$.value, "(%s)", $2.value);
    free($2.value);
    $$.array_length = null;
    if ($2.array_length != null) {
      $$.array_length = copyString($2.array_length);
      free($2.array_length);
    }
    $$.type = $2.type;
  }
| value {
    $$.value = createString("$", $1.value, 0);
    $$.array_length = null;
    if ($1.array_length != null) {
      $$.array_length = copyString($1.array_length);
      free($1.array_length);
    }
    $$.type = $1.type;
  }
; // end expression

/**
 * Describes the syntax for conditions.  For instance, consider the following
 * example:
 * <pre>
 * if (someVar && someOtherVar) {
 *   ...
 * }
 * </pre>
 * In this example, "someVar && someOtherVar" will be captured by this
 * non-terminal.
 */
conditions:
  conditions
  LOGICAL_OR
  conditions {
    $$ = (char*) malloc(sizeof(char) * (strlen($1) + strlen($3) + 5));
    sprintf($$, "%s || %s", $1, $3);
    free($1);
    free($3);
  }
| conditions
  LOGICAL_AND
  conditions {
    $$ = (char*) malloc(sizeof(char) * (strlen($1) + strlen($3) + 5));
    sprintf($$, "%s && %s", $1, $3);
    free($1);
    free($3);
  }
| NOT
  conditions {
    $$ = (char*) malloc(sizeof(char) * (strlen($2) + 2));
    sprintf($$, "!%s", $2);
    free($2);
  }
| LEFT_PARENTHESIS
  conditions
  RIGHT_PARENTHESIS {
    $$ = (char*) malloc(sizeof(char) * (strlen($2) + 3));
    sprintf($$, "(%s)", $2);
    free($2);
  }
| condition {
    $$ = (char*) malloc(sizeof(char) * (strlen($1) + 1));
    strcpy($$, $1);
    free($1);
  }
; // end conditions

/**
 * Describes the syntax for a condition.  For instance, consider the following
 * example:
 * <pre>
 * if (someVar == 2) {
 *   ...
 * }
 * </pre>
 * In this example, "someVar == 2" will be captured by this
 * non-terminal.
 */
condition:
  expression
  LESS_THAN
  expression {
    if ($1.type == NON_VALUE) {
      if ($3.type == NON_VALUE) {
        $$ = createString("*$ < *$", $1.value, $3.value, 0);
      }
      else {
        $$ = createString("*$ < $", $1.value, $3.value, 0);
      }
    }
    else {
      if ($3.type == NON_VALUE) {
        $$ = createString("$ < *$", $1.value, $3.value, 0);
      }
      else {
        $$ = createString("$ < $", $1.value, $3.value, 0);
      }
    }
    if ($1.array_length != null) {
      free($1.array_length);
    }
    if ($3.array_length != null) {
      free($3.array_length);
    }
  }
| expression
  LESS_THAN_OR_EQUALS
  expression {
    if ($1.type == NON_VALUE) {
      if ($3.type == NON_VALUE) {
        $$ = createString("*$ <= *$", $1.value, $3.value, 0);
      }
      else {
        $$ = createString("*$ <= $", $1.value, $3.value, 0);
      }
    }
    else {
      if ($3.type == NON_VALUE) {
        $$ = createString("$ <= *$", $1.value, $3.value, 0);
      }
      else {
        $$ = createString("$ <= $", $1.value, $3.value, 0);
      }
    }
    if ($1.array_length != null) {
      free($1.array_length);
    }
    if ($3.array_length != null) {
      free($3.array_length);
    }
  }
| expression
  GREATER_THAN
  expression {
    if ($1.type == NON_VALUE) {
      if ($3.type == NON_VALUE) {
        $$ = createString("*$ > *$", $1.value, $3.value, 0);
      }
      else {
        $$ = createString("*$ > $", $1.value, $3.value, 0);
      }
    }
    else {
      if ($3.type == NON_VALUE) {
        $$ = createString("$ > *$", $1.value, $3.value, 0);
      }
      else {
        $$ = createString("$ > $", $1.value, $3.value, 0);
      }
    }
    if ($1.array_length != null) {
      free($1.array_length);
    }
    if ($3.array_length != null) {
      free($3.array_length);
    }
  }
| expression
  GREATER_THAN_OR_EQUALS
  expression {
    if ($1.type == NON_VALUE) {
      if ($3.type == NON_VALUE) {
        $$ = createString("*$ >= *$", $1.value, $3.value, 0);
      }
      else {
        $$ = createString("*$ >= $", $1.value, $3.value, 0);
      }
    }
    else {
      if ($3.type == NON_VALUE) {
        $$ = createString("$ >= *$", $1.value, $3.value, 0);
      }
      else {
        $$ = createString("$ >= $", $1.value, $3.value, 0);
      }
    }
    if ($1.array_length != null) {
      free($1.array_length);
    }
    if ($3.array_length != null) {
      free($3.array_length);
    }
  }
| expression
  EQUALS
  expression {
    if ($1.type == NULL_VALUE || $3.type == NULL_VALUE) {
      $$ = createString("$ == $", $1.value, $3.value, 0);
    }
    else if ($1.type == NON_VALUE) {
      if ($3.type == NON_VALUE) {
        $$ = createString("*$ == *$", $1.value, $3.value, 0);
      }
      else {
        $$ = createString("*$ == $", $1.value, $3.value, 0);
      }
    }
    else {
      if ($3.type == NON_VALUE) {
        $$ = createString("$ == *$", $1.value, $3.value, 0);
      }
      else {
        $$ = createString("$ == $", $1.value, $3.value, 0);
      }
    }
    if ($1.array_length != null) {
      free($1.array_length);
    }
    if ($3.array_length != null) {
      free($3.array_length);
    }
  }
| expression
  NOT_EQUALS
  expression {
    if ($1.type == NULL_VALUE || $3.type == NULL_VALUE) {
      $$ = createString("$ != $", $1.value, $3.value, 0);
    }
    else if ($1.type == NON_VALUE) {
      if ($3.type == NON_VALUE) {
        $$ = createString("*$ != *$", $1.value, $3.value, 0);
      }
      else {
        $$ = createString("*$ != $", $1.value, $3.value, 0);
      }
    }
    else {
      if ($3.type == NON_VALUE) {
        $$ = createString("$ != *$", $1.value, $3.value, 0);
      }
      else {
        $$ = createString("$ != $", $1.value, $3.value, 0);
      }
    }
    if ($1.array_length != null) {
      free($1.array_length);
    }
    if ($3.array_length != null) {
      free($3.array_length);
    }
  }
; // end condition

/**
 * Describes the syntax for a type.
 */
type:
  BOOLEAN {
    $$ = (char*) malloc(sizeof(char) * (strlen("Boolean") + 1));
    strcpy($$, "Boolean");
  }
| CHARACTER {
    $$ = (char*) malloc(sizeof(char) * (strlen("Character") + 1));
    strcpy($$, "Character");
  }
| ID {
    $$ = (char*) malloc(sizeof(char) * (strlen($1) + 1));
    strcpy($$, $1);
    free($1);
  }
| INTEGER {
    $$ = (char*) malloc(sizeof(char) * (strlen("Integer") + 1));
    strcpy($$, "Integer");
  }
| REAL {
    $$ = (char*) malloc(sizeof(char) * (strlen("Real") + 1));
    strcpy($$, "Real");
  }
| STRING {
    $$ = (char*) malloc(sizeof(char) * (strlen("String") + 1));
    strcpy($$, "String");
  }
| VOID {
    $$ = (char*) malloc(sizeof(char) * (strlen("void") + 1));
    strcpy($$, "void");
  }
; // end type

/**
 * Describes the optional syntax for parameter declarations within function
 * definitions.
 */
opt_parameter_declarations:
  parameter_declarations {
    $$ = (char*) malloc(sizeof(char) * (strlen($1) + 1));
    strcpy($$, $1);
    free($1);
  }
| {
  $$ = (char*) malloc(sizeof(char));
  sprintf($$, "");
}
; // end opt_parameter_declarations

/**
 * Describes the syntax for parameter declarations within function definitions.
 */
parameter_declarations:
  parameter_declarations
  COMMA
  parameter_declaration {
    $$ = (char*) malloc(sizeof(char) * (strlen($1) + strlen($3) + 3));
    sprintf($$, "%s, %s", $1, $3);
    free($1);
    free($3);
  }
| parameter_declaration {
    $$ = (char*) malloc(sizeof(char) * (strlen($1) + 1));
    strcpy($$, $1);
    free($1);
  }
; // end parameter_declarations

/**
 * Describes the syntax for a parameter declaration within a function
 * definition.
 */
parameter_declaration:
  type
  ID {
    declarationTable[$2] = $1;
    $$ = (char*) malloc(sizeof(char) * (strlen($1) + strlen($2) + 3));
    sprintf($$, "%s *%s", $1, $2);
    free($1);
    free($2);
  }
| type
  ID
  LEFT_BRACKET
  RIGHT_BRACKET {
    declarationTable[$2] = $1;
    arrayTable[$2] = true;
    $$ = createString("$ $[], Integer *_$_length", $1, copyString($2),
        $2, 0);
  }
; // end parameter_declaration

/**
 * Describes the optional syntax for statements.
 */
statements:
  statements
  statement {
    $$.value = createString("$$\n", $1.value, $2.value, 0);
    $$.type = $2.type;
  }
| {
    $$.value = createString("", 0);
    $$.type = NO_STATEMENT;
  }
; // end statements

/**
 * Describes the syntax for a statement.
 */
statement:
  declaration {
    $$.value = (char*) malloc(sizeof(char) * (strlen($1) + 1));
    strcpy($$.value, $1);
    free($1);
    $$.type = DECLARATION_STATEMENT;
  }
| conditional {
    $$.value = (char*) malloc(sizeof(char) * (strlen($1) + 1));
    strcpy($$.value, $1);
    free($1);
    $$.type = CONDITIONAL_STATEMENT;
  }
| assignment {
    $$.value = (char*) malloc(sizeof(char) * (strlen($1) + 1));
    strcpy($$.value, $1);
    free($1);
    $$.type = ASSIGNMENT_STATEMENT;
  }
| expression
  SEMICOLON {
    $$.value = (char*) malloc(sizeof(char) * (strlen($1.value) + 2));
    sprintf($$.value, "%s;", $1.value);
    free($1.value);
    if ($1.array_length != null) {
      free($1.array_length);
    }
    $$.type = EXPRESSION_STATEMENT;
  }
| return_statement {
    $$.value = (char*) malloc(sizeof(char) * (strlen($1) + 1));
    strcpy($$.value, $1);
    free($1);
    $$.type = RETURN_STATEMENT;
  }
; // end statement

/**
 * Describes the syntax for a return statement.
 */
return_statement:
  RETURN
  SEMICOLON {
    $$ = (char*) malloc(sizeof(char) * (strlen("return;") + 1));
    strcpy($$, "return;");
    for (int i = 0; i < removeReferences.size(); i++) {
      $$ = createString("$\n$", copyString(removeReferences[i].c_str()),
          $$, 0);
    }
  }
| RETURN
  expression
  SEMICOLON {
    $$ = createString("_tmp = $;\n_addStrongReference(_tmp);", $2.value, 0);
    for (int i = 0; i < removeReferences.size(); i++) {
      $$ = createString("$\n$", $$, copyString(removeReferences[i].c_str()), 0);
    }
    $$ = createString("$\nreturn _tmp;", $$, 0);
    if ($2.array_length != null) {
      free($2.array_length);
    }
  }
; // end return_statement

/**
 * Describes the syntax for an assignment.
 */
assignment:
  variable
  ASSIGNMENT
  expression
  SEMICOLON {
    /*if (arrayTable[$1.value] && $1.array_length == null) {
      char *type = (char*) malloc(256 * sizeof(char));
      strcpy(type, declarationTable[$1.value].c_str());
      $$ = createString("memcpy($, $, *_$_length * sizeof($));",
          copyString($1.value), $3.value, $1.value, type, 0);
    }*/
```

```
      /*else*/ if ($1.array_length != null && strcmp($1.array_length, "1") != 0) {
        char *type = (char*) malloc(256 * sizeof(char));
        strcpy(type, declarationTable[$1.value].c_str());
        $$ = createString("memcpy($, $, $ * sizeof($));",
            $1.value, $3.value, $1.array_length, type, 0);
        $1.array_length = null;
      }
      else if ($1.array_length != null && strcmp($1.array_length, "1") == 0) {
        if ($3.array_length != null || $3.type == NON_VALUE) {
          $$ = createString("*$ = *$;", $1.value, $3.value, 0);
        }
        else {
          $$ = createString("*$ = $;", $1.value, $3.value, 0);
        }
      }
      else {
        char *dependent = getDependent($1.value);
        if (dependent != null) {
          $$ = createString(
            "_removeWeakReference($, $);\n$ = $;\n_addWeakReference($, $);",
            copyString($1.value), copyString(dependent),
            copyString($1.value), $3.value,
            $1.value, dependent, 0);
        }
        else if (attributesTable.find($1.value) != attributesTable.end()) {
          $$ = createString(
            "_removeWeakReference($, _$_this);\n$ = $;\n_addWeakReference($, _$_this);",
            copyString($1.value), copyString(attributesTable[$1.value].c_str()),
            copyString($1.value), $3.value,
            copyString($1.value), copyString(attributesTable[$1.value].c_str()),
            0);
            free($1.value);
        }
        else {
          $$ = createString(
            "_removeStrongReference($);\n$ = $;\n_addStrongReference($);",
            copyString($1.value), copyString($1.value),
            $3.value, $1.value, 0);
        }
      }
      if ($1.array_length != null) {
        free($1.array_length);
      }
      if ($3.array_length != null) {
        free($3.array_length);
      }
    }
  | variable
    ASSIGNMENT
    LEFT_PARENTHESIS
    type
    RIGHT_PARENTHESIS
    expression
    SEMICOLON {
      char *dependent = getDependent($1.value);
      if (dependent != null) {
        $$ = createString(
          "_removeWeakReference($, $);\n$ = ($*) $;\n_addWeakReference($, $);",
          copyString($1.value), copyString(dependent),
          copyString($1.value), $4, $6.value,
          copyString($1.value), dependent, 0);
        if (clearReturnVariable) {
          $$ = createString("$\n_removeStrongReference($);", $$, $1.value, 0);
          clearReturnVariable = false;
        }
        else {
          free($1.value);
        }
      }
      else if (attributesTable.find($1.value) != attributesTable.end()) {
        $$ = createString(
          "_removeWeakReference($, _$_this);\n$ = ($*) $;\n_addWeakReference($,
  _$_this);",
          copyString($1.value), copyString(attributesTable[$1.value].c_str()),
          copyString($1.value), $4, $6.value,
          copyString($1.value), copyString(attributesTable[$1.value].c_str()), 0);
        if (clearReturnVariable) {
          $$ = createString("$\n_removeStrongReference($);", $$, $1.value, 0);
          clearReturnVariable = false;
        }
        else {
          free($1.value);
        }
      }
      else {
        $$ = createString(
          "_removeStrongReference($);\n$ = ($*) $;\n_addStrongReference($);",
          copyString($1.value), copyString($1.value),
          $4, $6.value, copyString($1.value), 0);
        if (clearReturnVariable) {
          $$ = createString("$\n_removeStrongReference($);", $$, $1.value, 0);
          clearReturnVariable = false;
        }
        else {
          free($1.value);
        }
      }
      //$$ = createString("$ = ($*) $;", $1.value, $4, $6.value, 0);
      if ($1.array_length != null) {
        free($1.array_length);
      }
      if ($6.array_length != null) {
        free($6.array_length);
      }
    }
  ; // end assignment

  /**
   * Describes the syntax for a conditional, such as an if statement or while and
   * for loops.
   */
  conditional:
    WHILE
    LEFT_PARENTHESIS
    conditions
    RIGHT_PARENTHESIS
    LEFT_CURLY_BRACE
    statements
    RIGHT_CURLY_BRACE {
      $$ = createString("while ($) {\n$}", $3, $6.value, 0);
    }
  | IF
    LEFT_PARENTHESIS
    conditions
    RIGHT_PARENTHESIS
    LEFT_CURLY_BRACE
    statements
    RIGHT_CURLY_BRACE {
      $$ = createString("if ($) {\n$}", $3, $6.value, 0);
    }
  | IF
    LEFT_PARENTHESIS
    conditions
    RIGHT_PARENTHESIS
    LEFT_CURLY_BRACE
    statements
    RIGHT_CURLY_BRACE
    ELSE
    LEFT_CURLY_BRACE
    statements
    RIGHT_CURLY_BRACE {
      $$ = createString("if ($) {\n$}\nelse {\n$}", $3, $6.value, $10.value, 0);
    }
  | IF
    LEFT_PARENTHESIS
    conditions
    RIGHT_PARENTHESIS
    LEFT_CURLY_BRACE
    statements
    RIGHT_CURLY_BRACE
    elseifs
    ELSE
    LEFT_CURLY_BRACE
    statements
    RIGHT_CURLY_BRACE {
      $$ = createString("if ($) {\n$}\n$else {\n$}", $3, $6.value, $8, $11.value,
          0);
    }
  | IF
    LEFT_PARENTHESIS
    conditions
    RIGHT_PARENTHESIS
    LEFT_CURLY_BRACE
    statements
    RIGHT_CURLY_BRACE
    elseifs {
      $$ = createString("if ($) {\n$}\n$", $3, $6.value, $8, 0);
    }
  | IF
    LEFT_PARENTHESIS
    conditions
    RIGHT_PARENTHESIS
    statement {
      $$ = createString("if ($) $\n", $3, $5.value, 0);
    }
  ; // end conditional

  /**
   * Describes the syntax for else if conditionals.
   */
  elseifs:
    elseifs
    elseif {
      $$ = (char*) malloc(sizeof(char));
      strcpy($$, "");
    }
  | elseif {
      $$ = (char*) malloc(sizeof(char));
      strcpy($$, "");
    }
  ; // end elseifs

  /**
   * Describes the syntax for an else if conditional.
   */
  elseif:
    ELSE
    IF
    LEFT_PARENTHESIS
    conditions
    RIGHT_PARENTHESIS
    LEFT_CURLY_BRACE
    statements
    RIGHT_CURLY_BRACE {
      $$ = createString("else if ($) {\n$}\n", $4, $7.value, 0);
    }
  ; // end elseif

  /**
   * Describes the syntax for an access modifier.
   */
  access_modifiers:
    PRIVATE {
      $$ = (char*) malloc(sizeof(char) * (strlen("private") + 1));
      sprintf($$, "private:\n");
    }
  | PROTECTED {
      $$ = (char*) malloc(sizeof(char) * (strlen("protected") + 1));
      sprintf($$, "protected:\n");
    }
  | PUBLIC {
      $$ = (char*) malloc(sizeof(char) * (strlen("public") + 1));
      sprintf($$, "public:\n");
    }
  ; // end access_modifiers

  /**
   * Describes the syntax for a value, which is anything used as a parameter
   * in a function call or in an assignment.
   */
  value:
    variable {
      $$.value = createString("$", $1.value, 0);
      if ($1.array_length != null) {
        $$.array_length = createString("$", $1.array_length, 0);
      }
      else {
        $$.array_length = null;
      }
      $$.type = NON_VALUE;
    }
  | variable
    INCREMENT {
      $$.value = createString("(*$)++", $1.value, 0);
      if ($1.array_length != null) {
        $$.array_length = createString("$", $1.array_length, 0);
      }
      else {
        $$.array_length = null;
      }
      $$.type = NON_VALUE;
    }
  | INTEGER_VALUE {
      $$.value = (char*) malloc(sizeof(char) * (strlen($1) + 1));
      $$.array_length = null;
      strcpy($$.value, $1);
      free($1);
      $$.type = INTEGER_VALUE;
    }
  | REAL_VALUE {
      $$.value = (char*) malloc(sizeof(char) * (strlen($1) + 1));
      $$.array_length = null;
      strcpy($$.value, $1);
      free($1);
      $$.type = REAL_VALUE;
    }
  | STRING_VALUE {
      $$.value = (char*) malloc(sizeof(char) * (strlen($1) + 1));
    $$.array_length = null;
    strcpy($$.value, $1);
    free($1);
    $$.type = STRING_VALUE;
  }
| NULL_VALUE {
    $$.value = createString("null", 0);
    $$.array_length = null;
    $$.type = NULL_VALUE;
  }
; // end value

/**
 * Describes the syntax for a variable.
 */
variable:
  variable
  DOT
  variable {
    char *arrayName = removeAsterisks($1.value);
    if (arrayTable[arrayName] && (strcmp($3.value, "length") == 0)) {
      $$.value = createString("_$_length", arrayName, 0);
    }
    else {
      free(arrayName);
      if (declarationTable.find($1.value) != declarationTable.end()) {
        $$.value = createString("$->$", $1.value, $3.value, 0);
      }
      else {
        $$.value = createString("$::$", $1.value, $3.value, 0);
      }
    }
    $$.array_length = null;
    $$.type = NON_VALUE;
  }
| ID {
    $$.value = createString("$", $1, 0);
    $$.array_length = null;
    $$.type = NON_VALUE;
  }
| ID
  LEFT_BRACKET
  expression
  RIGHT_BRACKET {
    $$.value = createString("&$[_toInt($)]", $1, $3.value, 0);
    $$.array_length = createString("1", 0);
    if ($3.array_length != null) {
      free($3.array_length);
    }
    $$.type = NON_VALUE;
  }
| ID
  LEFT_BRACKET
  expression
  DOT_DOT
  expression
  RIGHT_BRACKET {
    char *array_length = createString("($) - ($) + 1", $5.value,
        copyString($3.value), 0);
    $$.value = createString("&$[$]", $1, $3.value, 0);
    $$.array_length = array_length;
    if ($3.array_length != null) {
      free($3.array_length);
    }
    if ($5.array_length != null) {
      free($5.array_length);
    }
    $$.type = NON_VALUE;
  }
| ID
  LEFT_PARENTHESIS
  opt_parameters
  RIGHT_PARENTHESIS {
    $$.value = createString("$($)", $1, $3, 0);
    $$.array_length = null;
    $$.type = NON_VALUE;
    clearReturnVariable = true;
  }
| NEW
  ID
  LEFT_PARENTHESIS
  opt_parameters
  RIGHT_PARENTHESIS {
    $$.value = createString("new $($)", $2, $4, 0);
    $$.array_length = null;
    $$.type = NON_VALUE;
  }
; // end variable

/**
 * Describes the syntax for optional parameters.
 */
opt_parameters:
  parameters {
    $$ = (char*) malloc(sizeof(char) * (strlen($1) + 1));
    strcpy($$, $1);
  }
| {
    $$ = (char*) malloc(sizeof(char));
    strcpy($$, "");
  }
; // end opt_parameters

/**
 * Describes the syntax for parameters.
 */
parameters:
  parameters
  COMMA
  expression {
    $$ = createString("$, $", $1, $3.value, 0);
    if ($3.array_length != null) {
      free($3.array_length);
    }
  }
| value {
    if (arrayTable[$1.value] && $1.array_length == null) {
      $$ = createString("$, _$_length", copyString($1.value),
          $1.value, 0);
    }
    else if ($1.array_length != null && strcmp($1.array_length, "1") != 0) {
      char *lengthDeclaration = (char*) malloc(sizeof(char) * 256);
      char *uniqueName = (char*) malloc(sizeof(char) * 256);
      sprintf(uniqueName, "_array_length%ld", ++uId);
      sprintf(lengthDeclaration, "Integer *%s = new
Integer(%s);\n_addStrongReference(%s);",
          uniqueName, $1.array_length, uniqueName);
      $$ = createString("$, $", $1.value, copyString(uniqueName), 0);
      createDeclarations.push_back(lengthDeclaration);

      char *removeReference = createString(
          "_removeStrongReference($);",
          uniqueName, 0);
      removeReferences.push_back(removeReference);
      free(removeReference);
      if ($1.array_length != null) {
        free($1.array_length);
      }
      free(lengthDeclaration);
    }
    else {
      $$ = $1.value;
    }
  }
; // end parameters

%%

/**
 * Stores the current yy text.
 */
extern char *yytext;

/**
 * The input file stream.
 */
extern FILE *yyin;

/**
 * Stores the current line number being parsed.
 */
extern int lineNumber;

/**
 * Returns the length of the given format string.
 * @param format the given format string
 * @return the length of the given format string
 */
int formatLength(const char *format) {
  int count = 0;
  for (int i = 0; format[i] != 0; i++) {
    if (format[i] != '$') {
      count++;
    }
  }
  return count;
} // end formatLength()

/**
 * Returns a string with the given format and variables.  Each time the
 * character, '$', is encountered it is replaced with a given variable
 * respectively.
 * @param format the given format
 * @param ... the given variables or 0 to specify that there are no more
 * variables; the memory for each variable will automatically be de-allocated
 * by this function
 * @return a string with the given format and variables
 */
char *createString(const char *format, ...) {

  char *result;
  char *i = "";
  int totalLength = formatLength(format);
  int j = 0;
  int k = 0;
  int length = 0;
  va_list marker;

  va_start(marker, format);
  i = va_arg(marker, char*);
  while (i != 0) {
    totalLength += strlen(i);
    i = va_arg(marker, char*);
  }

  totalLength++;
  result = (char*) malloc(sizeof(char) * totalLength);

  va_start(marker, format);
  i = va_arg(marker, char*);
  while (i != 0) {
    while (format[j] != '$' && format[j] != 0) {
      result[k] = format[j];
      j++;
      k++;
    }
    j++;
    for (int l = 0; l < strlen(i); l++) {
      result[k] = i[l];
      k++;
    }
    i = va_arg(marker, char*);
  }
  while (format[j] != 0) {
    result[k] = format[j];
    j++;
    k++;
  }
  result[k] = 0;

  va_start(marker, format);
  i = va_arg(marker, char*);
  while (i != 0) {
    free(i);
    i = va_arg(marker, char*);
  }
  va_end(marker);

  return result;
} // end createString()

/**
 * Returns the given type with all the asterisk characters removed.
 * @param type the given type
 */
char *removeAsterisks(char* type) {
  char *newType = null;
  int i = 0;
  while (type[i] == '*') {
    i++;
  }
  newType = (char*) malloc(sizeof(char) * (strlen(&type[i]) + 1));
  strcpy(newType, &type[i]);
  return newType;
} // end removeAsterisks()

/**
 * Returns a copy of the given string.
 * @param string the given string
 * @return a copy of the given string
 */
char *copyString(const char* string) {
  char *copy = (char*) malloc(sizeof(char) * (strlen(string) + 1));
  strcpy(copy, string);
  return copy;
} // end copyString()

/**
```

```
 * Returns everything to the left of the "->" operator in the given string.
 * @param string the given string
 * @return everything to the left of the "->" operator in the given string
 */
char *getDependent(const char* string) {
  int length = strlen(string);
  int i = 0;
  for (i = length - 1; i > -1 && string[i] != '-'; i--);
  if (i > -1) {
    char *dependent = (char*) malloc(sizeof(char) * (i+1));
    strncpy(dependent, string, i);
    dependent[i] = 0;
    return dependent;
  }
  return null;
} // end getDependent()

/**
 * Sets all the global variables to the last elements on their respective
 * stacks.
 */
void popGlobals() {
  delete initializeAttributes;
  initializeAttributes = attributesStack.back();
  attributesStack.pop_back();
  className = classNameStack.back();
  classNameStack.pop_back();
} // end popGlobals()

/**
 * Pushes all the global variables, such as className and initializeAttributes,
 * onto their respective stacks and reinitializes them.
 */
void pushGlobals() {
  classNameStack.push_back(className);
  attributesStack.push_back(initializeAttributes);
  initializeAttributes = new deque<string>();
} // end pushGlobals()

/**
 * The main entry point to this compiler.
 */
main() {
  do {
    yyparse();
  }
  while (!feof(yyin));
} // end main()

/**
 * Prints out parse exceptions.
 */
int yyerror(char *s) {
  fprintf(stderr, "%s: Encountered '%s' at line, %d.\n", s, yytext, lineNumber);
} // end yerror()
```

# APPENDIX D: FLANG.H SOURCE CODE

```
/*****************************************************************************
 *                                                                           *
 *                             Stephen Hoskins                               *
 *                               G-00337381                                  *
 *                          George Mason University                          *
 *                                  CS707                                    *
 *                               Summer 2007                                 *
 *                              Thesis Project                               *
 *                                                                           *
 *****************************************************************************/

#ifndef INTEGER_H
#define INTEGER_H

/**
 * This file includes everything that is needed to make an F language project
 * compile in Microsoft Visual Studio 2005.
 */

#include "Integer.h"
#include "FUtil.h"

#endif
```

# APPENDIX E: INTEGER.H SOURCE CODE

```
/****************************************************************************
 *                                                                          *
 *                           Stephen Hoskins                                *
 *                             G-00337381                                   *
 *                       George Mason University                            *
 *                                CS707                                     *
 *                             Summer 2007                                  *
 *                            Thesis Project                                *
 *                                                                          *
 ****************************************************************************/

#include "Object.h"

/**
 * An implementation of an object that stores integer information.
 * @author Stephen Hoskins, George Mason University
 * @version 1.0, 09/05/07
 */
class Integer : public Object {

public:

  /**
   * Stores the integer value.
   */
  int _integer;

  /**
   * Creates a new <code>Integer</code> object.
   */
  Integer() : Object() {
    _integer = 0;
  } // end default constructor

  /**
   * A copy constructor for the given integer.
   * @param integer the given integer
   */
  Integer(const Integer *integer) : Object() {
    this->_integer = integer->_integer;
  } // end constructor

  /**
   * A copy constructor for the given integer.
   * @param integer the given integer
   */
  Integer(const int &integer) : Object() {
    this->_integer = integer;
  } // end typecasting

  /**
   * Overloads the '+' operator to support addition.
   * @param rhs the variable that is being added to this instance of an
   * Integer object.
   */
  int operator+(const int &rhs) {
    return this->_integer + rhs;
  } // end operator+

  /**
   * Overloads the '-' operator to support subtraction.
   * @param rhs the variable that is being subtracted from this instance of an
   * Integer object.
   */
  int operator-(const int &rhs) {
    return this->_integer - rhs;
  } // end operator-

  /**
   * Overloads the '*' operator to support multiplication.
   * @param rhs the variable that is being multiplied to this instance of an
   * Integer object.
   */
  int operator*(const int &rhs) {
    return this->_integer * rhs;
  } // end operator*

  /**
   * Overloads the '/' operator to support division.
   * @param rhs the variable that is dividing this instance of an
   * Integer object.
   */
  int operator/(const int &rhs) {
    return this->_integer / rhs;
  } // end operator/

  /**
   * Overloads the '<' operator to support less-than comparison.
   * @param rhs the variable that is being compared with this instance of an
   * Integer object.
   */
  bool operator<(const int &rhs) {
    return this->_integer < rhs;
  } // end operator<

  /**
   * Overloads the '<=' operator to support less-than-or-equal-to comparison.
   * @param rhs the variable that is being compared with this instance of an
   * Integer object.
   */
  bool operator<=(const int &rhs) {
    return this->_integer <= rhs;
  } // end operator<=

  /**
   * Overloads the '>' operator to support greater-than comparison.
   * @param rhs the variable that is being compared with this instance of an
   * Integer object.
   */
  bool operator>(const int &rhs) {
    return this->_integer > rhs;
  } // end operator>

  /**
   * Overloads the '>=' operator to support greater-than-or-equal-to comparison.
   * @param rhs the variable that is being compared with this instance of an
   * Integer object.
   */
  bool operator>=(const int &rhs) {
    return this->_integer >= rhs;
  } // end operator>=

  /**
   * Overloads the '==' operator to support equal-to comparison.
   * @param rhs the variable that is being compared with this instance of an
   * Integer object.
   */
  bool operator==(const int &rhs) {
    return this->_integer == rhs;
  } // end operator==

  /**
   * Overloads the '!=' operator to support not-equal-to comparison.
   * @param rhs the variable that is being compared with this instance of an
   * Integer object.
   */
  bool operator!=(const int &rhs) {
    return this->_integer != rhs;
  } // end operator!=

  /**
   * Overloads the left-hand-side '++' operator to support the increment
   * operator
   */
  int &operator++() {
    return ++(this->_integer);
  } // end operator++

  /**
   * Overloads the right-hand-side '++' operator to support the increment
   * operator
   */
  int operator++(int i) {
    int ans = this->_integer;
    this->_integer++;
    return ans;
  } // end operator++

  /**
   * Overloads the '=' operator to support assignment.
   * @param integer the variable that is being assigned to this instance of an
   * Integer object.
   */
  Integer &operator=(const int &integer) {
    this->_integer = integer;
    return *this;
  } // end operator=

  /**
   * Overloads the '+' operator to support addition.
   * @param rhs the variable that is being added to this instance of an
   * Integer object.
   */
  int operator+(const Integer &rhs) {
    return this->_integer + rhs._integer;
  } // end operator+

  /**
   * Overloads the '-' operator to support subtraction.
   * @param rhs the variable that is being subtracted from this instance of an
   * Integer object.
   */
  int operator-(const Integer &rhs) {
    return this->_integer - rhs._integer;
  } // end operator-

  /**
   * Overloads the '*' operator to support multiplication.
   * @param rhs the variable that is being multiplied from this instance of an
   * Integer object.
   */
  int operator*(const Integer &rhs) {
    return this->_integer * rhs._integer;
  } // end operator*
```

```
/**
 * Overloads the '/' operator to support division.
 * @param rhs the variable that is dividing this instance of an
 * Integer object.
 */
int operator/(const Integer &rhs) {
  return this->_integer / rhs._integer;
} // end operator/

/**
 * Overloads the '<' operator to support less-than comparison.
 * @param rhs the variable that is being compared with this instance of an
 * Integer object.
 */
bool operator<(const Integer &rhs) {
  return this->_integer < rhs._integer;
} // end operator<

/**
 * Overloads the '<=' operator to support less-than-or-equal-to comparison.
 * @param rhs the variable that is being compared with this instance of an
 * Integer object.
 */
bool operator<=(const Integer &rhs) {
  return this->_integer <= rhs._integer;
} // end operator<=

/**
 * Overloads the '>' operator to support greater-than comparison.
 * @param rhs the variable that is being compared with this instance of an
 * Integer object.
 */
bool operator>(const Integer &rhs) {
  return this->_integer > rhs._integer;
} // end operator>

/**
 * Overloads the '>' operator to support greater-than-or-equal-to comparison.
 * @param rhs the variable that is being compared with this instance of an
 * Integer object.
 */
bool operator>=(const Integer &rhs) {
  return this->_integer >= rhs._integer;
} // end operator>=

/**
 * Overloads the '==' operator to support equal-to comparison.
 * @param rhs the variable that is being compared with this instance of an
 * Integer object.
 */
bool operator==(const Integer &rhs) {
  return this->_integer == rhs._integer;
} // end operator==

/**
 * Overloads the '!=' operator to support not-equal-to comparison.
 * @param rhs the variable that is being compared with this instance of an
 * Integer object.
 */
bool operator!=(const Integer &rhs) {
  return this->_integer != rhs._integer;
} // end operator!=

/**
 * Overloads the '=' operator to support assignment.
 * @param rhs the variable that is being assigned to this instance of an
 * Integer object.
```

```
    */
   Integer &operator=(const Integer &integer) {
     this->_integer = integer._integer;
     return *this;
   } // end operator=

 }; // end class Integer

 /**
  * Overloads the '+' operator to support addition.
  * @param rhs the variable that is being added to this instance of an
  * Integer object.
  */
 int operator+(const int &lhs, const Integer &rhs) {
   return lhs + rhs._integer;
 } // end operator+

 /**
  * Overloads the '-' operator to support subtraction.
  * @param rhs the variable that is being subtracted from this instance of an
  * Integer object.
  */
 int operator-(const int &lhs, const Integer &rhs) {
   return lhs - rhs._integer;
 } // end operator-

 /**
  * Overloads the '*' operator to support multiplication.
  * @param rhs the variable that is being multiplied to this instance of an
  * Integer object.
  */
 int operator*(const int &lhs, const Integer &rhs) {
   return lhs * rhs._integer;
 } // end operator*

 /**
  * Overloads the '/' operator to support division.
  * @param rhs the variable that is dividing this instance of an
  * Integer object.
  */
 int operator/(const int &lhs, const Integer &rhs) {
   return lhs / rhs._integer;
 } // end operator/

 /**
  * Overloads the '<' operator to support less-than comparison.
  * @param rhs the variable that is being compared to this instance of an
  * Integer object.
  */
 bool operator<(const int &lhs, const Integer &rhs) {
   return lhs < rhs._integer;
 } // end operator<

 /**
  * Overloads the '<=' operator to support less-than-or-equal-to comparison.
  * @param rhs the variable that is being compared to this instance of an
  * Integer object.
  */
 bool operator<=(const int &lhs, const Integer &rhs) {
   return lhs <= rhs._integer;
 } // end operator<=

 /**
  * Overloads the '>' operator to support greater-than comparison.
  * @param rhs the variable that is being compared to this instance of an
  * Integer object.
  */
```

```
bool operator>(const int &lhs, const Integer &rhs) {
  return lhs > rhs._integer;
} // end operator>

/**
 * Overloads the '>=' operator to support greater-than-or-equal-to comparison.
 * @param rhs the variable that is being compared to this instance of an
 * Integer object.
 */
bool operator>=(const int &lhs, const Integer &rhs) {
  return lhs >= rhs._integer;
} // end operator>=

/**
 * Overloads the '==' operator to support equal-to comparison.
 * @param rhs the variable that is being compared to this instance of an
 * Integer object.
 */
bool operator==(const int &lhs, const Integer &rhs) {
  return lhs == rhs._integer;
} // end operator==

/**
 * Overloads the '!=' operator to support not-equal-to comparison.
 * @param rhs the variable that is being compared to this instance of an
 * Integer object.
 */
bool operator!=(const int &lhs, const Integer &rhs) {
  return lhs != rhs._integer;
} // end operator!=

/**
 * Returns true if the given left-hand-side variable is less than the given
 * right-hand-side variable; false otherwise.
 * @param lhs the given left-hand-side variable
 * @param rhs the given right-hand-side variable
 * @return true if the given left-hand-side variable is less than the given
 * right-hand-side variable; false otherwise
 */
bool compare(const Integer *lhs, const int &rhs) {
  return lhs->_integer < rhs;
} // end compare

/**
 * Returns the given reference to an Integer object as an int.
 * @param integer the given reference to an Integer object
 * @return the given reference to an Integer object as an int
 */
int _toInt(Integer *integer) {
  return integer->_integer;
} // end _toInt()

/**
 * Returns the given Integer object as an int.
 * @param integer the given reference to an Integer object
 * @return the given Integer object as an int
 */
int _toInt(Integer integer) {
  return integer._integer;
} // end _toInt()

/**
 * Returns the given int as an int (in case this accidentally gets called on an
 * int in the C++ generated from the F compiler.
 * @param integer the given int
 * @return the given int as an int (in case this accidentally gets called on an
 * int in the C++ generated from the F compiler
 */
int _toInt(int integer) {
  return integer;
} // end _toInt()
```

# APPENDIX F: FUTIL.H SOURCE CODE

```
/*****************************************************************************
 *                                                                           *
 *                             Stephen Hoskins                               *
 *                               G-00337381                                  *
 *                          George Mason University                          *
 *                                  CS707                                    *
 *                               Summer 2007                                 *
 *                              Thesis Project                               *
 *                                                                           *
 *****************************************************************************/

#ifndef FUTIL_H
#define FUTIL_H

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

/**
 * Prints the given integer to standard out.
 * @param integer the given integer
 */
Object *println(Integer *integer) {
  printf("%d\n", integer->_integer);
  return 0;
} // end println()

/**
 * Copies a block of memory from the given source to the given destination with
 * the given length.
 * @param dest the given destination
 * @param destLength the given destination length
 * @param src the given source
 * @param integer the given integer
 */
void memcpy(Integer *dest, Integer *destLength, Integer *src) {
  memcpy(dest, src, sizeof(Integer) * destLength->_integer);
} // end memcpy()

#endif
```

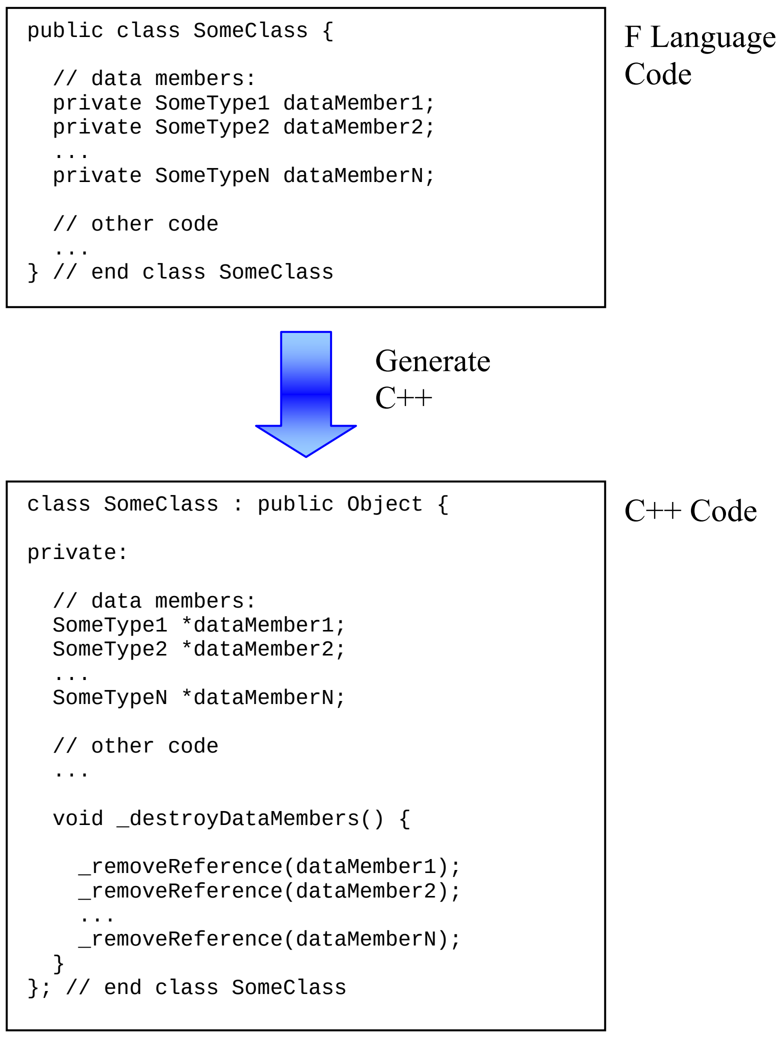
# APPENDIX G: OBJECT.H SOURCE CODE

```
/****************************************************************************
 *                                                                          *
 *                              Stephen Hoskins                             *
 *                                G-00337381                                *
 *                          George Mason University                         *
 *                                   CS707                                  *
 *                                Summer 2007                               *
 *                               Thesis Project                             *
 *                                                                          *
 ****************************************************************************/

#ifndef OBJECT_H
#define OBJECT_H

#include <hash_map>

using namespace std;
using namespace stdext;

#ifndef null
#define null 0
#endif

/**
 * The base class that all other classes inherit from in the F language.
 * @author Stephen Hoskins, George Mason University
 * @version 1.0, 08/17/07
 */
class Object {

public:

  /**
   * Used by hash_map class for object comparisons and hashing.
   * @author Stephen Hoskins, George Mason University
   * @version 1.0, 11/21/07
   */
  class ObjectHasher : public hash_compare<Object*> {

  public:

    /**
     * Hash function for the given object.
     * @param o the given object
     */
    size_t operator() (const Object *o) const {

      return (size_t) o;
    } // end operator()

    /**
     * Function for comparing the given objects.
     * @param o1 the first given object being compared
     * @param o2 the second given object being compared
     */
    bool operator() (const Object *o1, const Object *o2) const {

      return o1 < o2;
    } // end operator()

  }; // end class ObjectHasher
  /**
   * Stores the number of direct strong references referencing this object.
   */
  int _numStrongReferences;

  /**
   * Stores all the objects that have weak references to this object.
   */
  hash_map<Object*, int, ObjectHasher> _weakReferences;

  /**
   * Stores all the data members of this class.
   */
  hash_map<Object*, int, ObjectHasher> _dataMembers;

  /**
   * Creates a new Object.
   */
  Object() {
    _numStrongReferences = 0;
  } // end default constructor

  /**
   * This must be changed to a breadth-first search algorithm.
   */
  /**
   * Determines if this object has any strong references either directly or
   * indirectly.
   * @param visitedReferences stores all the objects that are visited from
   * recursive calls made to _hasStrongReference().
   */
  bool _hasStrongReference(
      hash_map<Object*, bool, ObjectHasher> *visitedReferences) {
    if (visitedReferences->find(this) != visitedReferences->end()) return false;
    if (_numStrongReferences > 0) return true;
    (*visitedReferences)[this] = true;
    hash_map<Object*, int, ObjectHasher>::iterator iter;
    for (iter = _weakReferences.begin(); iter != _weakReferences.end(); iter++)
    {
      if (iter->first->_hasStrongReference(visitedReferences)) {
        return true;
      }
    }
    visitedReferences->erase(this);
    return false;
  } // end _hasStrongReference()
}; // end class Object

/**
 * Stores that the given object has a weak reference to it by the given
 * dependent.
 * @param object the given object
 * @param dependent the given dependent
 */
void _addWeakReference(Object *object, Object *dependent) {

  if (object == null || dependent == null) return;

  if (object->_weakReferences.find(dependent) ==
      object->_weakReferences.end())
  {
    object->_weakReferences[dependent] = 0;
  }
  object->_weakReferences[dependent] = object->_weakReferences[dependent] + 1;

  if (dependent->_dataMembers.find(object) ==
      dependent->_dataMembers.end())
  {
    dependent->_dataMembers[object] = 0;
  }
  dependent->_dataMembers[object] = dependent->_dataMembers[object] + 1;
} // end _addWeakReference()

typedef Object::ObjectHasher ObjectHasher;

/**
 * Stores that the given object no longer has a weak reference to it by the
 * given dependent.  NOTE: Don't use this function.  Call the other
 * _removeWeakReference() function instead.
 * @param object the given object
 * @param dependent the given dependent
 * @param visitedReferences stores all the objects that are visited from
 * recursive calls made to _removeWeakReference() and _hasStrongReference()
 * @param deletedReferences stores all the objects that are deleted from
 * recursive calls made to _removeWeakReference()
 */
void _removeWeakReference(Object *object, Object *dependent,
    hash_map<Object*, bool, ObjectHasher> *visitedReferences,
    hash_map<Object*, bool, ObjectHasher> *deletedReferences) {

  if (object == null || dependent == null) return;

  int numReferences =
    dependent->_dataMembers[object] = dependent->_dataMembers[object] - 1;
  if (numReferences <= 0) {
    dependent->_dataMembers.erase(object);
  }

  numReferences =
    object->_weakReferences[dependent] = object->_weakReferences[dependent] - 1;
  if (numReferences <= 0) {
    object->_weakReferences.erase(dependent);
    if (!object->_hasStrongReference(visitedReferences)) {
      (*deletedReferences)[object] = true;
      hash_map<Object*, int, ObjectHasher>::iterator iter =
        object->_dataMembers.begin();
      while (iter != object->_dataMembers.end()) {
        if (deletedReferences->find(iter->first) ==
            deletedReferences->end()) {
          _removeWeakReference(iter->first, object, visitedReferences,
              deletedReferences);
        }
        else {
          object->_dataMembers.erase(iter->first);
        }
        iter = object->_dataMembers.begin();
      }
      delete object;
    }
  }
} // end _removeWeakReference()

/**
 * Stores that the given object no longer has a weak reference to it by the
 * given dependent.
 * @param object the given object
 * @param dependent the given dependent
 */
void _removeWeakReference(Object *object, Object *dependent) {
  _removeWeakReference(object, dependent, new hash_map<Object*, bool,
      ObjectHasher>(), new hash_map<Object*, bool, ObjectHasher>());
} // end _removeWeakReference()

/**
```

```
 * Updates the number of strong references that exist for the given object.
 * @param object the given object
 */
void _addStrongReference(Object *object) {

  if (object == null) return;
  object->_numStrongReferences++;
} // end _addStrongReference()

/**
 * Updates the number of strong references that exist for the given object.
 * NOTE: Don't use this function.  Call the other _removeStrongReference()
 * function instead.
 * @param object the given object
 * @param visitedReferences stores all the objects that are visited from
 * recursive calls made to _removeWeakReference() and _hasStrongReference()
 * @param deletedReferences stores all the objects that are deleted from
 * recursive calls made to _removeWeakReference() and _hasStrongReference()
 */
void _removeStrongReference(Object *object,
    hash_map<Object*, bool, ObjectHasher> *visitedReferences,
    hash_map<Object*, bool, ObjectHasher> *deletedReferences) {

  if (object == null) return;
  object->_numStrongReferences--;
  if (object->_numStrongReferences == 0) {
    if (!object->_hasStrongReference(visitedReferences)) {
      (*deletedReferences)[object] = true;
      hash_map<Object*, int, ObjectHasher>::iterator iter =
        object->_dataMembers.begin();
      while (iter != object->_dataMembers.end()) {
        if (deletedReferences->find(iter->first) ==
            deletedReferences->end()) {
          _removeWeakReference(iter->first, object, visitedReferences,
              deletedReferences);
        }
        else {
          object->_dataMembers.erase(iter->first);
        }
        iter = object->_dataMembers.begin();
      }
      delete object;
    }
  }
} // end _removeStrongReference()

/**
 * Updates the number of strong references that exist for the given object.
 * @param object the given object
 */
void _removeStrongReference(Object *object) {
  _removeStrongReference(object, new hash_map<Object*, bool, ObjectHasher>(),
      new hash_map<Object*, bool, ObjectHasher>());
} // end _removeStrongReference()

/**
 * A general use variable for storing temporary information.
 */
Object *_tmp = null;

#endif
```

# APPENDIX H: A BETTER SOLUTION TO DE-ALLOCATING MEMBER VARIABLES

The solution for de-allocating member variables of objects during garbage collection in this paper has been that data members should be stored in a hash map and then subsequently retrieved from this hash map and destroyed when it is time to perform garbage collection. However, there is clearly a more efficient way to do this than using hash maps. In particular, since the F language compiler parses data members of classes when generating C++, the F language compiler can use this information to produce C++ that handles the destruction of these data members explicitly. An example of how the F language compiler can do this is shown in Figure 39.



```
public class SomeClass {

  // data members:
  private SomeType1 dataMember1;
  private SomeType2 dataMember2;
  ...
  private SomeTypeN dataMemberN;

  // other code
  ...
} // end class SomeClass
```

```
class SomeClass : public Object {

private:

  // data members:
  SomeType1 *dataMember1;
  SomeType2 *dataMember2;
  ...
  SomeTypeN *dataMemberN;

  // other code
  ...

  void _destroyDataMembers() {

    _removeReference(dataMember1);
    _removeReference(dataMember2);
    ...
    _removeReference(dataMemberN);
  }
}; // end class SomeClass
```

Figure 39 – F Compiler Producing C++ that Destroys Data Members Explicitly

As can be seen from Figure 39, the F language compiler can produce a member function, _destroyDataMembers, for each class written in F in the generated C++ code, which makes explicit calls to the function, _removeReference, for each data member that exists in the class. This means that the function, _removeReference, would have to be modified to make use of this member function. The modified version of the function, _removeReference, is shown in Figure 40.

```
/**
 * Updates the number of references that exist for the given object.
 * @param object the given object
 */
void _removeReference(Object *object) {

  if (object == NULL) return;
  object->_numReferences--;
  if (object->_numReferences == 0) {
    object->_destroyDataMembers();
    delete object;
  }
} // end _removeReference()
```

Figure 40 – Modified Version of the Function, _removeReference, that Makes Use of the Member Function, _destroyDataMembers

As can be seen in Figure 40, the function, _removeReference, assumes that every object has a member function called _destroyDataMembers. This means that the function, _destroyDataMembers, will have to be implemented as a virtual function in the base class, Object. Figure 41 shows how the member function, _destroyDataMembers, would be implemented in the base class, Object.

```
class Object {

public:

  int _numReferences;

  // virtual function
  virtual void _destroyDataMembers() {
  } // end _destroyDataMembers()
```

```
    // other code
    ...
  }; // end class Object
```

Figure 41 – An Implementation of the Function, _destroyDataMembers, in the Base Object Class

Using this implementation for destroying data members would be much more efficient. For one thing, a hash map would not need to be updated every time there is an assignment statement that deals with data members. Secondly, this implementation would not have to deal with any of the overhead associated with using hash maps. Therefore, this implementation would be much more ideal for destroying data members when performing garbage collection.

# LIST OF REFERENCES

## LIST OF REFERENCES

Archer, T. and Whitechapel, A., 2002, *Inside C#*, 2nd Edition, Microsoft Press.

Azatchi, H., Levanoni, Y., Paz, H., and Petrank, E., 2003, An On-The-Fly Mark and Sweep Garbage Collector Based on Sliding Views, *Conference on Object Oriented Programming Systems Languages and Applications: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, Inc., p. 269-281.

Boehm, Hans-J., 2004, The Space Cost of Lazy Reference Counting, *Annual Symposium on Principles of Programming Languages: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, Inc., p. 210-219.

Bollella, G. (Editor), Brosgol, B., Gosling, J., Dibble, P., Furr, S., and Turnbull, M., 2000, *The Real-Time Specification for Java,* Addison Wesley Longman; 1st edition.

Colvin, G. and Dawes, B., 1999, *Smart Pointers*. Boost Software, http://www.boost.org/libs/smart_ptr/smart_ptr.htm, (Accessed: November 26, 2007).

Davison, A., 2005, *Killer Game Programming in Java*. O'Reilly, p. 10-11 in Appendix C.

Dibble, P.C., 2002, *Real-Time Java Platform Programming*. Prentice Hall PTR; 1st edition.

Digital Mars, 2007, *Overview – D Programming Language 2.0*, http://www.digitalmars.com/d/overview.html, (Accessed: November 27, 2007).

Eyre-Todd, R.A., 1993, The Detection of Dangling References in C++ Programs, *ACM Letters on Programming Languages and Systems (LOPLAS)*, ACM, Inc., p. 127-134.

Goetz, B., 2003, *Java Theory and Practice: A brief history of garbage collection*, IBM, http://www.ibm.com/developerworks/java/library/j-jtp10283/ (Accessed: November 24, 2007).

Hamilton, J., 1997, Montana Smart Pointers: They’re Smart and They’re Pointers, *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) – Volume 3,* USENIX Association, p. 2-2.

Huelsbergen, L. and Larus, J.R., 1993, A Concurrent Copying Garbage Collector for Languages that Distinguish (Im)mutable Data, *Principles and Practice of Parallel Programming: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* ACM, Inc., p. 73-82.

Morgan, T.P., 2006, *Evans Data Cases Programming Language Popularity,* Guild Companies, Inc., http://www.itjungle.com/tug/tug121406-story03.html (Accessed: November 24, 2007)

Shaffer, C.A., 1997, *A Practical Introduction to Data Structures and Algorithm Analysis*. Prentice-Hall, Inc., p. 321-325.

Weinstein, B., 2001, *Got Any Java Programmers?* CNET Networks, Inc., http://articles.techrepublic.com.com/5100-10878-1048401.html, (Accessed: November 24, 2007).

Wellings, A., 2004, *Concurrent and Real-Time Programming in Java,* Wiley.

# CURRICULUM VITAE

Stephen Hoskins graduated from Lake Braddock High School, Burke, Virginia, in 1998. He received his Bachelor of Science in Computer Science and Mathematics from Virginia Polytechnic Institute and State University in 2003. He was employed as a software developer at Trident Systems, Inc. in Fairfax City, Virginia for seven years.