Design Issues In Java and C++

Scott M. Lewandowski
scl@cs.brown.edu

Abstract

This paper analyzes two of today's most popular object-oriented languages with an emphasis on fundamental design differences between them and how these design decisions affect a programmer. Specific issues addressed include class and function friendliness, operator overloading, memory management, program documentation, array manipulation, accuracy of the object model, preprocessing, machine independence and portability. After an analysis of these issues, the author relates the differences between the languages to the suitability of each language for various programming tasks, from educational settings to real-time systems.

Design Issues in Java and C++

1. Introduction

Despite the hundreds of programming languages available to programmers on all platforms, two languages currently dominate the market for serious software development tools. Although they cannot compete with specialized languages that have been designed from the ground up to best meet the needs of specialized application programming, they clearly provide the best general solution language available today. The two languages are C++, Bjarne Stroustrop's widely accepted and until recently uncontested object oriented version of C, and Sun Microsystem's Java, the hottest trend in the computer industry which has been touted as the dominant object-oriented language of the future.

1.1 C++: The DeFacto Standard

C++ evolved as a result of the need for a language providing the efficiency and flexibility of C and the tools for organizing, structuring, and maintaining large programs that Simula offered.[1] In particular, Stroustrop was looking to develop a language providing for classes, class hierarchies, some degree of concurrency, and static type checking. He also set out to make the language generate compiled code that could be easily linked to code generated in other languages, perform similarly to BCPL, and be highly portable.[2] In large part due to his upbringing, Stroustrop designed C++ to solve programming problems in the way in which the programmer felt most comfortable; that is, C++ inherently supports a myriad of programming styles and does not lock the programmer into doing things "the right way" as defined by one individual.

1.2 Java: Improving on Success?

The birth of Java reveals a different story. Despite the widespread acceptance, availability, and use of C++, engineers at Sun Microsystems discovered that despite its strengths, C++ was not perfect for every programming task. The astounding rate of increase of the availability and use of the Internet coupled with the "complexities" of C++ indicated a need for a fresh new language.[3] With this need recognized, Sun allocated $5 million in additional funding to get Java, touted as an object-oriented Internet programming language, ready for market. Java emerged just over one year later with Sun billing it as "...C++ slightly simplified, and with libraries convenient for the Internet environment".[4] Just as Stroustrop had primary design goals when crafting C++, the

engineers at Sun had some goals. Of course, their goals were somewhat more substantial since many of the basic problems involved in the design of a C-like object-oriented language had already been solved by Stroustrop. The first of the goals was portability. Since Java was destined for use on the Internet where a wide variety of platforms are used, this was very important to its success. Second, it was to be architecture neutral; that is, the binary code that is generated by a Java compiler will run on any platform, using any microprocessor and any operating system, without modification of any kind.[5] Performance was also viewed as a critical issue, which necessitated the inclusion of support for threads. Due to the intended marketing strategy for Java (presenting it as an Internet language), security and robustness was also very important. In addition to these goals, the engineers also sought to keep the language familiar, simple, and object-oriented.[6]

1.3 Importance of Design Decisions

Seeing how the designers of Java used C++ as a base on which to build, it is not surprising that the languages are very similar. At first glance, it is very easy to mistake Java code for C++ code, or vice versa. The syntax is almost identical. However, it is the more fundamental design decisions that the creators of a language make that truly define a language. It is from these design issues that a language's strengths, limitations, and suitability for application to real-world problems is defined. Most of the basic language primitives from C++ have been retained in Java.[*] However, the focus of this paper is to deal with more "philosophical" decisions that affect the languages. After discussing some of the more significant of such issues, the suitability of each language for a variety of tasks can be accurately evaluated.

2. Design Decisions

2.1 Friendliness

One significant difference between the two languages is the mechanism through which one class can gain direct access to another class' private or protected methods and instance variables. Although many object-oriented purists feel that the inclusion of any such mechanism breaks down the object model significantly by allowing access to what has been modeled by the programmer as private implementational details, many real-world programming situations would be nearly impossible to effectively implement without this capability. The alternative would be the use of

---

[*]Notably absent, however, is the "goto" command.

cumbersome and inefficient assessor and mutator functions (the stereotypical "set/get" model). Furthermore, there are some models that cannot be accurately or realistically structured without this capability. For example, if an object-oriented model of a computer were to be constructed, it might contain classes for the microprocessor, floppy drive, memory, and input/output unit. The only one of these components that can directly interface with the memory is the microprocessor, so it might make sense (not to mention being more efficient) to allow the microprocessor to be a "friend" of the memory so that it can directly manipulate memory contents as it can in real life. Another situation that many programmers feels warrants the use of friendliness is when performance is critical. An example of such an application might be a vector and matrix math package which will be used extensively by a graphics manipulation application. The ability of a matrix to directly read the components of a vector will significantly improve execution speed, albeit at the expense of rigid object-orientation. Finally, the use of friend concepts can make a program significantly easier to read, understand, and maintain.

C++ allows for such inter-class interaction via the use of friend functions and friend classes. A class can include the declaration for a method of another class proceeded by the word *friend*. The use of this keyword allows a class to expressly grant permission of access to a single function of another class. More than one method could be declared "friend", but for obvious stylistic reasons, only those methods requiring friend status should be granted it. For situations in which an entire class should be friendly with another class, the second class could declare itself a friend of the first with the line *friend class XXX*. It is very interesting to note that this latter form (friend classes) was the only form of friendliness supported by the initial implementation of C++. Stroustrop later decided to allow friend functions since it was found to be convenient, especially for global functions (which very often serve as overloaded operator functions), and because it better maintained the object model.[7]

```
class StorageObject {
      public:
            StorageObject();
            ~StorageObject();
            int getYValue();
            void setYValue();
      private:
            int x_,y_;
            friend Manipulator::incrementX(int value);
}

class Manipulator {
      public:
            Manipulator();
            ~Manipulator();
            void incrementX(int value);
            void incrementY(int value);
}
```

Figure 1: An example of friend functions in C++. The object *StorageObject* has declared the *incrementX()* method of *Manipulator* to be a friend function. This means that *Manipulator* can directly access *x_* to increment it, but *incrementY()* must use the set/get methods of *StorageObject* to perform the same function.

Java adopts a different approach to providing friendliness. Java classes are written in packages, which are simply code files that can contain one or more class declarations. Classes in the same package default to be friendly with each other. There is currently no way for classes in different packages to be friendly in any way, nor is there a way for a class to only be friendly to an individual method of another class. This approach leads to several problems. Most importantly is the inability for classes in different packages to be friendly. For simple application systems this restriction may not be a significant problem. In a large system, especially one being worked on by several programmers, this becomes a large problem as classes are artificially contrived to go together in a single package just to achieve the necessary level of modeling, or more likely, performance. Packages for complex classes tend to be rather unwieldy to begin with; adding another class to it compounds the problem.

The designers of both languages acknowledged the need for friendly entities to some degree or another. The approach taken by Stroustrop in the design of C++ is clearly more flexible than that offered by Java; as a matter of fact, C++'s handling of the situation is a superset of Java's abilities, since it can have friendly classes. The rationale behind Sun's decision has to be viewed as questionable, at best. Why should classes included in the same package inherently be friendly? It is possible to override this behavior, but is this not the very breakdown in the object model that the

designers of Java set out to eliminate? It is ironic that the engineers at Sun made the very same design decision that Stroustrop realized was flawed after his initial implementation. Although this shortcoming of Java is significant in its own right, since it is possible to work around it the problem is not large. Similarly, object-oriented purists programming in C++ can simply chose to ignore the powerful capabilities that properly used friend functions afford a programmer in favor of more traditional assessor and mutator methods.

2.2 Operator Overloading

When Stroustrop wrote the design specification for C++, he included the capability for operator overloading despite his initial fears that the drawbacks of this feature would outweigh the benefits. If it would be difficult to implement and nearly impossible to teach and define precisely as operator overloading was reputed to be, Stroustrop did not feel that he had the resources to provide overloading. Other detractors claimed that code using overloaded operators was inherently inefficient and more difficult to read and understand. If these claims were found to be true, then C++ would be better off without the feature than with it. On the other hand, if these concerns could be addressed, the inclusion of a facility for operator overloading could be of great utility to programmers. Many real-world problems could be elegantly solved. With operator overloading available, intuitive manipulation of complex numbers, matrixes, multi-dimensional arrays, strings, and range-checked arrays could occur; without operator overloading, solutions were possible, but they would be cumbersome and of complex syntax. Although it was certainly acknowledged that it was possible to write positively horrifying code through the misuse of operator overloading, Stroustrop preferred to focus on "how a feature can be used well" rather than "how it can be misused".[8] This will be seen as contrary to the strategy adopted by the designers of Java.

There is very little to say about Java as far as operator overloading: it simply is not supported in any way, shape, or form. Many reasons for this have been suggested, foremost amongst them that it can potentially lead to poorly written code and the generation of an inherently inefficient executable.

Overloading primitive operators so that they work with classes leads to elegant code when used correctly. There is certainly a "danger", however, in that programmers can redefine operators to do whatever they desire. However, this is an argument that can be leveled against almost any component of a flexible object oriented programming language. It is "dangerous" if a class designer foolishly names the "multiply" method for their matrix class "add" since it will lead to confusion, and misapplication of the method. Like method names, operator overloading works,

but only if the programmer chooses to follow reasonable and somewhat standardized conventions. Diehard advocators of Java still maintain that it is preferable and "more intuitive" to use descriptive method names for operations that manipulate objects, but in practice this is very tedious.[9]

The author maintains that operator overloading is an essential part of a robust programming language. Its importance is actually increased when classes are to be used by multiple programmers. This is because of the intuitiveness of working with representations of certain data types in conjunction with standard C++ operators. Since it does not provide this facility, Java classes will never reach the level of transparency to the programmer that is possible to achieve with C++ classes. A mathematician will not be able to sit down with a vector math package in Java and just infer the operators. How can it be argued that it is more convenient or intuitive to write *A.Add(B)* to add two matrices than to write *A + B*? On the other hand, there is some merit to the argument that operator overloading can lead to perverse implementations of methods. As noted, however, this is inherent with nearly every language technology. More significant claims can be made in the area of the efficiency and performance of the finished code. The inclusion of operator overloading in C++ does not mandate its use. If a programmer sees that performance suffers such that his application is no longer viable when operators are overloaded, he can structure his class so that all methods are accomplished via traditional method calls. If robustness and understandability are more important, then overloading may be the perfect solution.

The issue of operator overloading has been widely debated since its first appearance in C++ years ago and it looks like this debate will continue. In retrospect, Stroustrop feels that operator overloading has indeed been a major asset to the language, providing for consistent interfaces due to overloaded mathematical operators, simple subscripting via the use of *[]*, standardized application through the use of *()*, predictable assignment accomplished by overloading =, and standardized input/output using << and >>.[10]

2.3 Memory Management

Perhaps the most widely discussed difference between Java and C++ is the divergent memory management models that each language has adopted. The issue is so vast that it is suitable for a significant research paper of its very own. Some of the many issues affected by this one paramount difference between the two languages include array manipulation, garbage collection, efficiency, accountability, flexibility, speed of execution, and stability.

The C++ model of memory management is significantly more flexible than the scheme offered by Java. It relies on the use of the *new()* and *delete()* operators to allocate and free memory. This setup, while leaving quite a bit of control to the user, is not sufficient at all times. There are instances when a more finely-grained system of memory management is required. A typical example of such an application is a complex data structure such as a tree or a linked list. Such data structures create (and potentially delete) hundreds or perhaps even thousands of small objects throughout their existence. If a general purpose allocator is used, the overhead from the allocation and deallocation operations becomes a significant factor in the speed of program execution. Furthermore, memory fragmentation can easily occur, leading to reduced performance and low-memory situations. Another variety of system that demands fine control over the allocation of memory are real-time systems. It is frequent in such applications that memory requirements are predictable, but the acquisition of memory must be likewise predictable to ensure the appropriate degree of performance can be realized. Although rare, there are also cases where hardware or other system requirement demanded that an object reside at a specific address in memory.[11]

Objects can be placed arbitrarily in memory to accommodate hardware requirements for specific application. This undoubtedly destroys the portability of some code, but permits such things as allocating memory in the shared (as opposed to the private) memory of a multi-processor system. This mechanism allows a rich means of general resource management that provides for specialized memory handling semantics.[12] A programmer using a class providing for speed optimized memory allocation and deallocation, for example, need not even be aware of the complex memory handling that is being performed, since operator *new()* can be overloaded transparently.

The most serious limitations of C++'s memory management facilities can be observed when dealing with deallocation and destructors. C++ does not allow the *delete()* operator to be overloaded. This functionality, while having the potential to be incredibly dangerous and destructive when used improperly, would make some issues of memory management more robust and complete. For example, the current revision of C++ does not allow for a container object to do all of the memory management for all objects that it contains. The standard C++ destruction techniques must be used.

In low memory situations, C++ represents a large improvement over its predecessors. C requires programmers to check for memory exhaustion after each and every allocation request.

Programmers choosing to follow this unenforced "rule" noticed significant drops in the performance of their applications, while programmers ignoring the advice risked crashing the entire system due to lack of memory. Stroustrop set out to solve both of these problems in C++. The user is not expected to check for memory exhaustion, so a performance drop is not incurred. Yet, control is returned to the user should memory allocation fail during a library call. This is accomplished by having the constructor not execute if operator *new()* returns zero (indicating failure). C++ programmer also have the option of including a *new_handler* for their application. This is a function which is guaranteed to be called when *new()* can't find enough memory to complete. Typically, programmers will use this function to either find more resources or to report an error to the user and exit the program gracefully.[13]

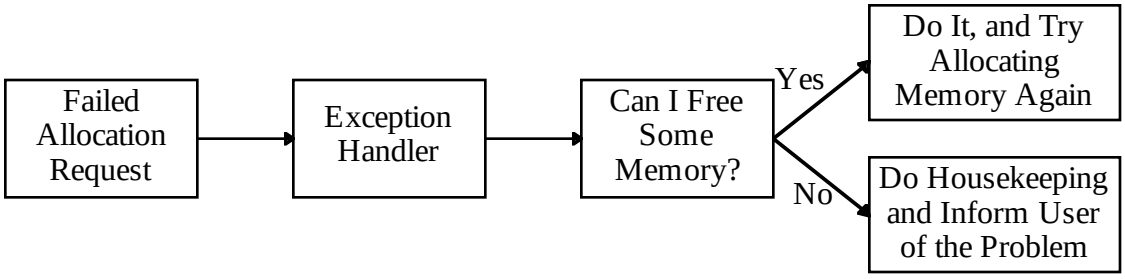

Figure 2: A typical sequence of events when appropriate memory cannot be allocated by C++. The example shows return being passed to an exception handler, which frees memory if possible, then returns control so the allocation can be tried again. When no more memory can be freed, the exception handler does the necessary housekeeping to prevent data loss and informs the user of any effects of the memory shortage.

With all of these memory issues addressed, one of the most fundamental questions remains: what should be done about garbage collection? Stroustrop "deliberately designed C++ not to rely on automatic garbage collection" because he feared "very significant space and time overheads".[14] He also cited that garbage collection would make C++ unsuitable for many of the low-level tasks that it was intended to be able to handle with ease. Stroustrop did leave the option of some form of automated memory management tool available to the user. A programmer could implement his own automatic memory management scheme. It is also possible (and some would say likely) that C++ will receive a comprehensive yet optional automatic garbage collection scheme within the next few years. Several implementations of C++ already offer this extended functionality.

Java's approach to memory management is very different from that offered by C++. While C++'s operator *new()* exists in Java, operator *delete()* is no longer present. When a programmer is done with an object and wants to delete it, he does nothing. Java has an automatic garbage collector that handles the reclamation of memory for the user.[15]

The implementation of *new* found in Java is a true operator, as it is in C++. In Java, this has a very serious implication: since operators cannot be overloaded, *new* cannot be redefined. To the application programmer this means one less thing to potentially worry about, but it also represents a total loss of control and flexibility. Obviously, the lack of a *delete* operator altogether severely limits any possibilities for more efficient or elegant deallocation schemes.

Java can respond to low memory situations as well, if not better, than C++ can. When memory is found to be unavailable, Java checks to see if any objects have zero references. If such objects exist, they are deleted immediately. If a complete pruning of all eligible objects does not free up enough memory for the task at hand, an exception is thrown like in C++. At this point, however, the programmer's only recourse is to give an error message and exit gracefully; the extra options available to a C++ programmer have already been exhausted by the Java run-time system at this point.

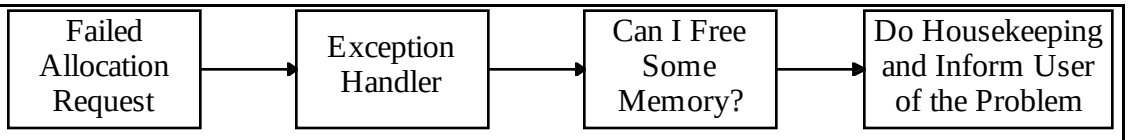

Figure 3: A typical sequence of events when appropriate memory cannot be allocated by Java. The garbage collector has already attempted to free memory, so the only course of action for the exception handler is to perform housekeeping and alert the user to the problem.

As mentioned, Java has an automatic garbage collection scheme. The system can be turned off, but there is no alternative method of reclaiming memory if this is done. Although it does alleviate some of the performance concerns associated with automatic garbage collection, it is not a viable solution for a large system that will create and release many objects since memory will soon be exhausted. Java garbage collection is accomplished via a "mark and sweep" algorithm. This algorithm calls for following all known links recursively and "marking" all found objects. All non-marked objects can then be disposed of, and memory can be compacted. This algorithm comes with potentially large costs. In particular, it means that at unpredictable times, a very large amount of processing will suddenly start-up. This effect in Java is somewhat minimized by multi-

threading, but some run-time cost is still carried.[16] Since traditional memory management schemes also mandate the eventual disposal of memory, one may wonder why automatic garbage collection is slower since it is performing the same basic operation. The answer lies in the fact that the Java garbage collector must actively seek out reclaimable memory, whereas a C++ forces the programmer to specify where it can be found. This mandate of C++ also leads to predictable delays when objects are deallocated.

If Java is to stay with automatic garbage collection, the only other viable technology would be a reference counting scheme. In this scenario, the run-time engine maintains a list of the number of references that have been requested for a given object in memory. When this count falls to zero, the object is no longer in use. This has the distinct advantage of imposing a constant overhead rather than plaguing the program with "random" bursts of CPU use. The need to deal with circular references, however, adds great complexity to the system and can potentially slow it down to an unacceptable level. With this important consideration in mind, it can be argued quite easily that the engineers at Sun made a good decision to implement the mark and sweep algorithm.

Obviously such different memory management schemes have their advantages and disadvantages. The *new()* operators provided in C++ and Java are nearly identical as far as functionality in the basic case. The power and flexibility of C++ is clear though when it is realized that Java does not allow *new* to be overridden, precluding sophisticated memory management techniques critical to some applications. A nice feature of Java is automatic memory compression. Although possible in C++, this would require a custom memory management scheme. The algorithm used to do this in Java is of questionable efficiency, unfortunately. Although C++ does not afford a programmer great flexibility in regard to deallocating memory, the mere fact that the programmer can manually specify when to deallocate has many advantages, particularly in the domains of real-time and high-performance systems. Java's handling of out of memory requirements is decidedly more robust than C++'s since it handles some of what a C++ programmer would need to manually implement. Once again, however, it is not as flexible. The programmer could not create new resources with as much latitude as in C++ due to the restricted *new* operator.

It is widely agreed on that automatic garbage collection reduces programming errors. Although Stroustrop does appreciate the fact that garbage collection simplifies design and minimizes programming errors, he remains convinced that if C++ featured automatic garbage collection it would have been a failure.[17] Java has a clear advantage over C++ as far as providing a

programming environment conducive to rapid prototyping and reliable code. Garbage collection makes programming easier for programmers of all levels of ability, and has the potential to be more reliable than user-supplied memory management systems. It removes the confusion as to which object is responsible for the deletion of an object created by one object but passed as by reference to another object. On the other hand, it creates many run-time problems in the form of increased space and time requirements. Automatic garbage collection, particularly those systems relying on a mark and sweep algorithm, also has the potential for interruptions that can disrupt time-sensitive systems, such as complex GUIs , device drivers, operating system kernels, and real-time systems. The unpredictability arising from automatic garbage collection makes programming these tasks difficult. A great degree of freedom is also lost with automatic garbage collection. An example might be a function call that creates a large data set, uses it, but then wants to perform other activities. Since the function still has a reference to the data set, it cannot be destroyed. This may not be a problem, but the situation could arise such that the function had significant memory requirements and the data set might preclude the availability of the necessary resources.

Even Sun Microsystem's official literature denigrates automatic garbage collection to some degree. They even go so far as to recommend that "You should design systems to be judicious in the number of objects that they create." It is also emphasized that memory leaks can occur if a programmer is not careful to free up references when they are no longer truly needed.[18]

C++ is a capable receptor for an appropriate mechanism of automatic garbage collection. Care must be taken that the selected algorithm does not interfere with primitive C++ functions. Having optional garbage collection available would allow the programmer to make a decision based on the needs of the application at hand. It should be noted, however, that Java has no viable mechanism through which automatic garbage collection can be turned off for anything besides the most basic and short-lived programs. Many applications use their own form of automatic garbage collection. A substantial number of libraries offering this functionality already exist. The issue has received so much attention as of late it can be expected that a rich set of garbage collection routines will become publicly available, and it is also likely that an optional automatic garbage collection scheme will be incorporated into the evolving C++ standard.

2.4 Pointers and Object References

Related to the issues of memory management just examined is the inclusion of pointers in C++ but their notable absence in Java. All objects in Java are accessed by the programmer via

object references.[19] C++ gains added functionality and flexibility over Java in many ways, two of which are perhaps more significant than the rest. First, Java has no real mechanism through which pass by reference can be accomplished. Objects are passed by value result, meaning that a local copy is made in the called function, removing the distinction between call by value and call by reference. Second, Java has no support for function pointers. Although much of the functionality of function pointers can be derived through the clever use of virtual function and inheritance, some applications, notably callbacks involving GUIs and parsing functions, are much more difficult to program in Java because of this.[20] Obviously, the need for Java to make a copy of the object that has been passed has an associated performance penalty. Although it could be argued (as with almost every other performance difference between Java and C++) that this penalty is small, the sum of the small penalties incurred, particularly when repeated operations are encountered, can be quite significant. The elegance and expressiveness of pointers unfortunately comes with a very high price. This price is the numerous pointer-related bugs found in C and C++ code generated by programmers of all levels. The risk of dangling pointers, memory leaks, array under and over-runs, and failure to perform appropriate dereferencing has plagued programmers for years. Java's removal of pointers solves these problems.

2.5 Arrays

The handling of arrays in the two languages is very different. C++ does not include a built-in array object as Java does. When memory for an array is allocated in C++, the starting location of the first element in memory is remembered, and contiguous space sufficient to store all elements of the array is reserved. When a specific array element is requested, the value is just taken to be the number of bytes required to represent the object starting from the memory location corresponding to the first memory location in the array plus the number of bytes required to hold the intermediate elements. Because of the nature of this system, it is very possible to read beyond the end of an array, or to read memory preceding the array. Even more dangerous is the potential to write to this unreserved memory, which could contain other program data, the executing program itself, or even the operating system. This scheme is very fast and is quite flexible in that it allows a programmer to setup a pointer to an element in an array and then set the pointer by element relative to this initial element. For example, if a pointer P points to the third element in the array, P++ will cause P to point to the beginning of the fourth element, regardless of the size of the object that is pointed to by P.

Java treats arrays as first-class objects.[21] The most significant ramification of this is run-time bounds checking, which prevents programmers from reading or writing memory above or below the allocated memory for the array. If such an error occurs, an exception is thrown and the program can deal with the situation as appropriate. This also would allow for "fragmented" arrays, although the current implementations of Java do not support them. The use of fragmented arrays could potentially make better use of available memory by inserting individual elements into small bits of free memory. C++ arrays could not support this due to the dependency of much of the existing C++ code on physical memory layout. The fact that Java arrays are objects also leads to the inclusion of such convenient features as a means to check the number of elements in an array. In Java, multi-dimensional arrays are truly arrays of arrays, although after declaration, this is transparent to the programmer.

The designers of Java missed a good opportunity to fix some one of the proverbial shortcomings of C++ arrays. They did catch the major error (bounds checking), but they did not "fix" the indexing mechanism. All array indexes still start with zero, and end with size minus one. A scheme more similar to Pascal, where the start and end identifiers could be anything (even characters and negative numbers) would have introduced some degree of inefficiency, but would have made programming much easier; a programmer could use array indexes that suited the task at hand. Java's mechanism for managing arrays is much more refined and safe. The flexibility lost from C++ can be easily regained by creating wrapper classes around the standard Java array object. The same could be said about C++, though. A programmer could (as many have already done) write an array class that supports bounds checking in the same manner that Java does, "fixes" the indexing problem, and for that matter do anything else seen fit. One thing that Java could never regain, no matter how the wrapper was designed, is the speed advantage. The overhead for bounds checking in Java is always present. A C++ the programmer has the option of using a simple pre-written library function to derive all of the functionality of Java arrays if it is desired, but a Java programmer has no way of gaining the advantage that C++ arrays clearly claim, that of speed of execution.

2.6 Preprocessing and Header Files

Having made an educated choice as to which programming language to use, a programmer is left to deal with several other language elements. One which certainly affects a programmer, particularly in today's group programming paradigm, is the presence or absence of a preprocessing unit. C++ implementations contain preprocessing units that provide services such

as conditional compilation, inclusion of named files, and macro substitution.[22] The least commonly used of these is the macro expansion capability. This allows a programmer to make a declaration in the form *#define identifier token-string* and have every subsequent instance of *identifier* replaced by *token-string*. Although widely used, it is not the preferable way to accomplish most programming goals, and is consequently not considered good style. Any numeric substitutions would be better represented by constants, and any short functions should be inlined functions, global in scope if absolutely necessary. The most common use for preprocessor directives is to *#include* files. Nearly every C++ program keeps its class declarations in a file separate from its method declarations. The preprocessor replaces the line containing the include statement with the complete file contents. An advantage of this setup is the ability to completely separate a class' interface and its implementation. This is especially useful for commercial libraries, where developers may want to ship header files that users can reference for documentation and structural information, but that keeps the code itself secret. Programmers can also split code into more manageable pieces. For example, each of several very long methods could be its own file, which are then included in one "master file". The use of the *#include* directive to reference a class' declaration is so much a part of C++ coding practice that many newcomers to the language think that it is a requirement. Conditional compilation, a little used feature of the language, is perhaps the most interesting of all preprocessor directives. In addition to its typical use at the beginning of a header file to prevent multiple inclusion, conditional compilation allows a programmer to maintain separate versions of software within the same file. For example, a programmer writing a program for distribution on the Internet may wish to give users a save-disabled demo version. He could enclose all blocks of code related to saving and loading files with *#ifndef demo_version ... #endif*. With this done, a full version of the program could be compiled per usual. If he wanted a demo version generated, he need only specify *#define demo_version* at the beginning of the first file that the compiler is scheduled to examine. There are other uses for the preprocessor in C++, but these are the primary applications.

```
/*****************************************************************
XFMath.H

Header File Declaring Matrix and Vector Operations
*****************************************************************/
#ifndef XFMATH_H
#define XFMATH_H


< HEADER FILE CONTENTS HERE >


#endif  /* XFMATH_H */
```

Figure 4: Using preprocessor directives to prevent multiple inclusion of a header file.

Java has completely done away with the notion of preprocessing, and subsequently header files. Macros are no longer possible, nor is conditional compilation. Sun's Java whitepaper suggests constants as an alternative to some macros and to *#define*, but offers no suggestion as to how to replace the lost feature of conditional compilation. Inclusion of files is convenient for large projects, but its absence does not limit possibilities. It does introduce one question, however: how is one class to know of the existence of another class? In Java, this is accomplished with the *import* command, which is essentially a way to tell the compiler to go examine the referenced class so its methods are known.

With one exception, Java's break from the preprocessor paradigm is wonderful. The exception is the inability to keep an interface completely separate from code. This issue is addressed to some degree through Java's primitive (yet existent) documentation features. The benefits of removing the preprocessor are many, though. No real functionality is lost by not having macros, since as discussed above they can be replaced by constants or functions. In regard to preprocessor macros, even Stroustrop warns "Don't use them if you don't have to". He goes on to cite that most macros demonstrate a flaw in the programmer, his program, or the programming language and that many programming tools will lose functionality when subjected to code laden with preprocessing directives.[23] Conditional compilation, while sometimes useful, is rarely implemented correctly, except in the case of preventing multiple inclusion of a header file. Most usage of the conditional compilation capabilities of C++ was to account for different versions of code necessary for different machines, a problem solved far more elegantly by Java. By removing preprocessing, Java is made more context-free. With preprocessing removed, Java code is considerably more readily read and understood, and consequently more readily re-used in a

faster and more simple manner.[24] A programmer attempting to understand a large program in Java has no need to sift through all of the header files that would be "required reading" in C++.

2.7 Automated Program Documentation

As mentioned, Java's lack of header files means no documentation for classes in the form of header files. Java does offer a dedicated language mechanism for program documentation, whereas C++ has no such facility.

Java comments beginning with /** instead of the more traditional // or /* inherited from C++ are picked up by javadoc, an automatic documentation generator. Based on the idea of literate programming introduced by Donald Knuth, the javadoc utility parses the specially marked comments, and formats the text contained within into a special set of HTML tags. A programmer should include the version number, author, and necessary cross-references to other code, a description of what is being commented (the class, instance variable, etc.) and any other pertinent comments. HTML tags can be used within comments, which theoretically allows program documentation to link to world-wide web sites and contain diagrams. The documentation generated will show a chain of class inheritance and all of the public fields in a class. Although many people consider this form of documentation difficult to work with since it requires a web browser to read, this mechanism does always generate accurate and complete documentation.[25]

```
/**
 * A button which sets the message of a Text to "Bye!" when clicked on.
 *
 * @version   Wed Aug 14 22:40:08 1996
 * @author dpm
 */

public class ByeButton extends GP.Components.Buttons.Push {
```

Figure 5: An example of a comment set-up for parsing by javadoc. This simple comment just gives a description of the class *ByeButton* and its author and version information. It should be noted that the javadoc parser will automatically match the comment describing the class with the class declaration.

This automatic documentation tool is wonderful if used by the programmer. It can be somewhat cumbersome, especially since everything must be formatted and tokenized very carefully for readable documentation to be generated. But the features far outweigh the drawbacks. Linking to other web pages is useful, and diagrams of data structures is a potentially useful tool. Automatic generation of class hierarchies is also a nice feature. Perhaps most important is the fact

that only public fields are displayed in the documentation; this is something that is impossible in C++. Private fields can be documented by the simple addition of the meta-HTML tags defined by Sun. The inclusion of this mechanism eclipses almost all of the lost functionality from not having header files.

2.8 Portability and Machine Independence

When Stroustrop to designed C++, he had portability in mind as a fairly significant goal.[26] However, Stroustrop's notion of portability (which was certainly influenced by the time) was that code that compiled on one system could be copied to another system and recompiled without incident. This notion is almost taken for granted today. C++ meets this goal, except in cases where hardware specific code is used, or when libraries available for one platform are not available for another. For the most part, however, one version of a C++ program can be easily prepared for distribution on multiple platforms.

One of the major design goals of the Java team was to provide a machine-independent and portable language. The definition of portable suggested by Sun's design team is much more ambitious than Stroustrop's was. To the Java design team, portability implied heterogeneous systems, different window libraries, and different network features. Java not only provides traditional portability, but it also offers identical functionality on multiple platforms. This is in large part due to the fact that decisions normally left to a compiler designer are mandated by the Java specification. Some of these issues include the number of bits dedicated to various data types, the order of evaluation of expressions, and the inclusion of support for Unicode characters permitting world-wide text storage. This notion of portability mandates that the same set of library calls be available on all platforms.[27] This idea is even more interesting when it is considered that all of this can be done without recompiling any code. That is because the Java compiler generates architecture neutral bytecodes that can be executed on any machine. This is currently accomplished via a Java virtual machine implemented on various platforms, but microprocessors capable of natively executing Java code are in design phases. The virtual machine, which is custom designed to meet the specifications of the specific microprocessor at hand, translates the Java bytecodes and then executes them.

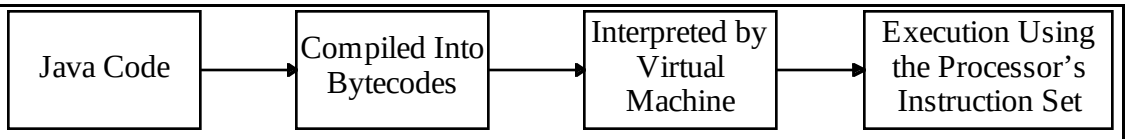

Figure 6: The Java model of program execution. After a Java program is written by a programmer, a compiler compiles it into machine-independent bytecodes. When the program is executed, the virtual machine interprets these bytecodes and converts them to instructions native to the processor being used at the time. These native instructions are then executed.

C++'s level of portability is dwarfed by Java's comprehensive treatment of the issue. Java programs can be written and debugged once, and a single version can be shipped that will run on any computer supporting a Java virtual machine. This is quite an advantage for a software developer. Unfortunately, these benefits come with some tradeoff. In this case, speed is a major area affected. By not allowing the compiler to take advantage of architectural design characteristics, full speed of execution may never be realized on some platform. In addition, the translation that must be performed by the virtual machine is not without performance penalty. Although not as pronounced as more traditional interpreted languages, Java does suffer from the same performance problems that plague languages such as BASIC. The need to keep all code portable also precludes the use of machine-specific features and capabilities; all machines must support something for it to become part of Java.

C++ will likely become more portable as the standard evolves, but not in the sense that Java is portable. It will essentially be a more complete realization of Stroustrop's original goal. Machine independence will never be realized by a C++ standard even closely resembling the currently evolving standard; so much of the standard is based on the fact that code need not be machine independent that this would be a completely unrealistic goal.

2.9 Accuracy of the Object Model

A final issue for analysis is the degree of object-orientation achievable through each language. Both support the standard object-oriented features such as classes, inheritance, virtual functions, polymorphism, encapsulation, abstraction, etc. With this being the case, it is smaller differences between the languages which set them apart in this domain. As a matter of fact, aside from the issues raised in the discussion comparing the handling of friendship in the two languages, their support for object-orientation is nearly identical. One could argue that Java better provides for object-orientation by removing header files, allowing the programmer's representation of an object

to be one physical code unit, as a logical representation of an object would be. Java certainly enforces a more strict adherence to object-oriented design principles by removing some of the "loopholes" present in C++, but one could argue that the loopholes in C++ are minor and serve to provide more accurate (although not necessarily more object-oriented) models than are possible in Java. In this domain, the author would suggest that the two languages are equally capable.

3. Applications

Thus far, this paper has examined several of the most significant design decisions made by Stroustrop and the team at Sun Microsystems. Since a programmer cannot choose bits and pieces of each language to form the "ultimate" language, a decision must be made as to which language to choose. This decision depends on the task at hand.

3.1 Internet Programming

Of little debate is that Java is the clear language of choice for writing Internet applications. When developing for the Internet, a programmer has no idea what platform a user will be using; yet there is the desire to make an application equally available to all users, with identical functionality. Furthermore, the more transparent that this is, the more likely it is that an application will survive. Java quite nicely fulfills these goals through its rich form of portability and its machine independent code. The convenience and expressiveness of a language is irrelevant if the product generated by the compiler is not available to users. As the only language meeting these paramount goals, Java must be designated the language of choice. Inherent restrictions in C++ will prevent it from ever meeting these criteria.

3.2 Real-Time and Performance Critical Applications

Developers working on real-time and performance critical applications need a language that makes as few sacrifices to speed as possible and gives them as much control as possible over all phases of execution. C++ best meets these goals. The availability of a full suite of options for inter-class friendliness serve to improve performance, albeit at the expense of program design. Some performance penalty would be incurred by the programmer choosing to use overloaded operators, but their presence in C++ does not mandate their use, and the performance when handling a non-overloaded operator does not suffer. One of the most significant areas differentiating the performance of the two systems is memory management. The fine level of control afforded to a C++ programmer may well be necessary for a successful real-time or

performance critical application. In performance and real-time applications it is sometimes required that objects be located in certain blocks of memory for maximum speed of execution to be attained. This is possible in C++, but Java provides no facility for this since it would have sacrificed the overriding goals of portability and neutrality. The option to delete objects no longer needed when desired is also important. A C++ programmer can designate the optimal time for objects to be destroyed, such as while waiting for a server response, or between time-critical tasks. Because of this, predictable levels of code execution can be constructed. When programming in Java, a programmer must account for the fact that garbage collection could start up in the middle of a time-sensitive task. In applications truly striving for the ultimate in performance, the mere fact that Java does automatic garbage collection, and hence relies on an algorithm to identify unused memory instead of getting this information from the user, is a drawback, even if the unpredictability of the operation is not of great significance. The automated mode of memory management that is used by Java does reduce programming bugs and is certainly more convenient. However, just as in a high performance automobile in which ride comfort is sacrificed for speed, C++ sacrifices some of the convenience for sheer power. On a similar note, the option to quickly pass a pointer to an object instead of generating a copy of the object has performance advantages; in this way too, C++ has a clear speed advantage. Accessing arrays in Java has already been shown to be more robust but less efficient. The added overhead from bounds checking may be unacceptable for many performance-critical applications. The issue of predictability also arises here, since the delay in bounds checking is not directly proportional to array size. The availability, or lack thereof, of a preprocessing mechanism is of no real performance concern. The few speed optimizing things that can be done through the use of the preprocessor (namely inlined functions) can be done in other ways in both languages.

Java's largest asset, its portability and architecture neutrality, turns out to be its biggest flaw when looking at its suitability for performance-critical applications. C++ affords a programmer many means through which to exploit the full power of the hardware available. The compiler designer also has many ways of taking full advantage of the design attributes of the hardware at hand to get maximum performance. In addition to removing much of the freedom and flexibility from the programmer, Java also removes the freedom from the compiler designer by mandating that standardized bytecodes are generated. The very means by which Java programs are run is inefficient; the fact that the bytecodes need be translated at run-time is clearly not acceptable when performance is the number criterion for evaluation. The minimized safety net provided by C++

and the option to deal with hardware specific details put C++ far ahead of Java for programming real-time and performance-critical applications in which raw speed and predictability are the most important factors.

3.3 Large Systems and Team Programming

Programmers working on large systems need a language that offers the best blend of expressiveness, maintainability, and convenience for multiple-programmer teams. As a system gets larger and more complex, bugs become more difficult to find and fix, so the role of the language in keeping the number of bugs low to begin with it very important. The handling of friendship should not be a true issue here. There are ways to circumvent the traditional friendship paradigms in Java, and C++ provides them all natively. In a large system, it may well be better to use assessor and mutator functions for cleanliness and to ensure maximum compatibility between program components. Although operator overloading can certainly enhance the programming experience of a programmer working with a library, the complexities of code maintenance introduced by the use of operator overloading are significant. Because of this, operator overloading must be looked upon as a pitfall, since most time spent with code is understanding and maintaining it. The time saved learning the class library is minimal compared to the time spent tracking down bugs and code dependencies later on in the software life cycle.

For large and complex systems, Java's automatic memory management features are most welcome. They eliminate the need for programmers to coordinate who is responsible to dispose of "shared" objects and can quickly eliminate many of the programming bugs that make their way into projects of all size, most notably large ones. In a large system there are many things that can go wrong; a simplified memory management model removes many of these from the immediate concern of the application programmer. Although time lost to garbage collection increases faster than the increase in program size, this should still not be an issue, particularly due to the fact that large system will likely have several threads executing simultaneously, minimizing the apparent speed loss to the user. On a similar note, the absence of pointers in Java eliminate an entire class of programming flaws and lead to more stable and easily maintained code. The inherent protection that Java offers when dealing with arrays is also very nice for large application development, when once again, the goal is minimizing room for programmer error. This benefit is minor, however, in light of the number of array classes available for C++ providing the same degree of safety that Java offers.

The absence of a preprocessor puts Java at a large advantage to C++ for large systems development, especially in the maintenance phase of the life cycle. By removing context sensitivity from as many parts of the program as possible, a programmer can sit down with a given block of code and confidently analyze its methodology without worrying about declarations in header files, or about any macro expansion that may be occurring. Programming tools are also likely to be more widely used for large system development, so the elimination of the risk of having preprocessor directives undermine these utilities is good. Java's method of program documentation generation is certainly more robust than the use of C++ header files. It allows the programmer the full flexibility that is allowed in C++ header files, while providing consistency and being automatically updated (for most features). Up to date and standardized documentation is clearly important, and Java allows it to be written as the program is written in a relatively unobtrusive manner. For large systems, object modeling may also be considered essential for the program to be understandable. Both languages do an equally good job in that area.

In all of the areas discussed, Java is as good as C++, and in most areas it surpasses C++. Java eliminates may sources of bugs in C++ programs without adding any Java-specific bugs. This robustness speeds development and makes maintenance easier.

3.4 Educational Environments

As computers become prevalent in society, the need to teach programming continues to rise. It is difficult to precisely define what makes a programming language good for educational purposes. This is largely due to divergent pedagogical theories. Some theories would advocate that learning by experience (or by failure) is preferred because it forces a student to become intimate with the problems inherent in solving the problem at hand. Others would argue that this technique discourages students and that starting out at whatever the task at hand is should be simple and rewarding. Depending on which of these schools of thought is followed, either C++ or Java could be given the nod as being the preferred introductory language. Both appropriately teach the use of object-oriented language fundamentals, which is perhaps the most significant concern.

The fact that Java classes in the same package default to be friendly could be a problem in an introductory programming setting. Care must be taken that each class be in its own package so that students do not unknowingly violate the object model. C++ friendship will not be realized by a student accidentally, so need not be accounted for. Although it is unlikely that it would directly affect a student, students could be affected by the use, or lack thereof, of overloaded operators in libraries that they use. Once again, it is an issue of ease of learning or learning by trial and error.

Overloaded operators found in C++ have the potential to make libraries much more intuitive to use, whereas the need for dedicated methods in Java could push a student to read and understand program documentation. In regard to memory management, once again the pedagogical model adopted is of paramount importance. If the instructor wants students to get off to a quick start programming, Java's memory management is the tool of choice. Students need not worry about "low level" implementational details; they can focus their efforts at high-level design and coding fundamentals. In contrast, learning memory management is a skill that could certainly be valuable for later programming. Understand the complexities of how and when memory is allocated and deallocated, and its effect on your code is important. Some would even argue that an appreciation of automatic memory management can not be acquired without having experienced manually managed memory environments. Likewise with arrays and pointers. Java protects the student from potentially difficult to find and frustrating errors, but enduring these things in C++ perhaps better teaches a student to be cognizant of array bounds. It is certainly more detrimental to witness a system crash in C++ due to neglecting array bounds than to see an exception warning in Java.

In this author's opinion, the absence of preprocessor files from Java is of no true educational benefit. The only use that preprocessing would get in an introductory programming course is for header files, and for the programs written in most introductory classes, the use of a separate header file is of questionable utility. The automatic documentation features in Java are not unlike the other features mentioned. It forces a student to do things the right way and makes programming somewhat more efficient, but having to manually comment C++ code can be a good way of learning what the salient features of a particular implementation or design are. Java's machine independence and portability could never be seen as a drawback to Java in an educational forum, and in many cases it is a welcome feature since most universities have mix of computers, ranging from workstations to Macintosh and IBM compatible computers owned by students and maintained in smaller computer clusters.

As stated, either Java or C++ could be chosen as the programming language of choice for beginners. The "hand-holding" that Java does is nice and certainly will save students much frustration by preventing errors that even experienced C++ programmers make. However, this author is of the opinion than negotiating these pitfalls is of great educational value, and would therefore advocate the use of C++.

3.5 General Programming

The general question is finally reached: which language should be chosen for "ordinary, everyday programming"? This author would find it necessary to recommend the use of Java. With the advancing speed of hardware the added performance that it is possible to squeeze out of C++ is becoming more and more negligible. The advent of Java bytecode microprocessors should close the performance gap even more by eliminating the virtual machine; they may even reverse the results of the current performance benchmarks. Today's most prominent computing environment is the network environment, particularly the Internet. Java has already been seen to be the only language reasonable for development of application to be executed on such a diverse range of systems. Although C++ can be "patched" through the use of libraries to provide most of what Java offers, it is preferable to use Java in its almost native form. Java speeds program development, reduces bugs, and increases code maintainability and reusability. Most applications do not require the flexibility that C++ offers, so why should the robustness of Java be ignored? For normal programming, this author would offer that the biggest shortcoming of Java is the lack of operator overloading. Although this issue can be worked around, the resulting system will not offer the robustness that a corresponding implementation in C++ could offer. Many features in Java, such as automatic program documentation, serve to make the task of the programmer more simple and enjoyable. There really is not a great deal of explanation needed here. Java is a more "safe" language, and the benefits of C++ over Java as outlined in this paper are in the areas of performance and flexibility, something demanded by few programming tasks.

4. Conclusion

The Java language has improved on C++ in a variety of ways. It has sacrificed speed and flexibility for stability, ease of use, and optimal performance in a networked computing environment. Although not suitable for all programming tasks, these changes make Java more suitable for general programming than C++. The forthcoming class libraries to be released by Sun will further extend the options that Java affords a programmer and provide for even more robust applications than are possible right now. Java Bytecode CPUs also lie ahead. When they become commercially available, the speed of execution of Java will increase tremendously as the overhead of the interpreter will be eliminated, and most of the advantages of C++ over Java will be drastically reduced, if not eliminated. In the author's opinion, the most significant shortcoming of Java is the absence of a facility for overloading operators; this limitation will likely exist forever.

For “obvious reasons” Sun Microsystems suggests that Java has “leapfrogged” C++ and provides a simple language that “will undoubtedly replace it [C++]”.[28] This author agrees wholeheartedly.

Works Cited

Arnold, Ken and James Gosling. *The Java Programming Language*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1996.

Daconta, Michael C. *Java for C/C++ Programmers.* New York: John Wiley & Sons, Inc., 1996.

Gosling, James and Henry McGilton. *The Java Language Environment: A White Paper*. Mountain View, California: Sun Microsystems, Inc., 1995.

Stroustrop, Bjarne. *The C++ Programming Language*. Reading, Massachusetts: Addison Wesley Publishing Company, 1991.

Stroustrop, Bjarne. *The Design and Evolution of C++*. Reading, Massachusetts: Addison Wesley Publishing Company, 1994.

van der Linder, Peter. *Just Java.* Mountain View, California: Sunsoft Press, 1996.

Reference Notes

[1]Bjarne Stroustrop, *The Design and Evolution of C++* (Reading, Massachusetts: Addison-Wesley Publishing Company, 1994), 1.

[2]Ibid., 21.

[3]Peter van der Linder, *Just Java* (Mountain View, California: Sunsoft Press, 1996), xviii.

[4]Ibid., 21.

[5]Ibid., 22.

[6]Michael C. Daconta, *Java for C/C++ Programmers* (New York: John Wiley & Sons, Inc., 1996), 2-4.

[7]Stroustrop, *Design*, 53.

[8]Ibid., 78.

[9]Daconta, 137.

[10]Stroustrop, *Design*, 79.

[11]Ibid., 212.

[12]Ibid., 215.

[13]Bjarne Stroustrop, *The C++ Programming Language* (Reading, Massachusetts: Addison-Wesley Publishing Company, 1991), 308-314; Stroustrop, *Design*, 218-219.

[14]Stroustrop, *Design*, 219.

[15]Ken Arnold and James Gosling, *The Java Programming Language* (Reading, Massachusetts: Addison-Wesley Publishing Company, 1996), 11.

[16]van der Linder, 181.

[17]Stroustrop, *Design*, 219.

[18]Arnold and Gosling, 46.

[19]Ibid., 9.

[20]Daconta, 63-65.

[21]James Gosling and Henry McGilton, *The Java Language Environment: A White Paper* (Mountain View, California: Sun Microsystems, Inc., 1995), 17.

[22]Stroustrop, *C++ Language*, 606.

[23]Ibid., 138.

[24]Gosling and McGilton, 29.

[25]van der Linden, 80-81.

[26]Stroustrop, *Design*, 26.

[27]van der Linder, 22.

[28]Ibid., 77.