Why C and C++ are bad. Maybe Java is better?

XXXXXXXXX

CS 553 Position Paper #1

Feb. 2003

The programming language C has been around since the 70?s. It was developed by Dennis Ritchie and Brian Kernighan at Bell Labs to help re-write the UNIX operating System. The big plus of having UNIX in C was portability. The great popularity of UNIX is in part attributable to its early portability to different architectures. This was possible because it had been written in C, a machine independent language. C++ was also developed at Bell Labs. It was developed by Bjarne Stroustrup in the 80?s in large part to bring object oriented programming to C. [1,2,3]

C helped on by its power and utility as well as its winning association with UNIX helped to become a dominant programming language. C++ leveraged C?s popularity and has also become a prevalent programming language. C and C++ have remained relevant due to their widespread use and support. They also are powerful, expressive and importantly they are fast programming languages. Their speed and ability to directly touch the memory make them an especially desired languages for systems programming. Every major operating system today makes extensive use of C. [1,2,3]

Despite their speed and utility as systems programming languages, C and C++ should be discarded for most programming tasks. I attended a conference (http://dimacs.rutgers.edu/Workshops/Software) this winter where it was postulated that:
?It is time to abandon C and C++.

It is difficult to write reliable and secure code in C and C++. Instead, any new projects should be performed in type safe programming languages such as Java, C#, Cyclone and CCured. While those languages do not eliminate security problems, they eliminate broad categories of them. Strong consideration should be given to trying to migrate security critical components of existing C and C++ applications to a type safe language.? [6]

Despite the fact that Brian Kernighan was in the audience many of the attendees agreed that it was ?time to abandon C and C++?. Although many there agreed that it was not likely to be achieved as a goal anytime soon, given the developed code base in C and C++ as well as all the programmers trained in their use.

The arguments against the widespread use of C and C++ are largely on grounds of security and reliability. Whole classes of security vulnerabilities such as buffer overflows and format string attacks are only possible in languages such as C and C++. Reliability is also a problem. Pointers are hard to get right and can lead to buggy code. Error handling is not easy to do, especially in C. The manual freeing of memory when not done right can lead to problematic memory leaks. Modern languages such as Java fix many of these problems with techniques such as exception handling, garbage collection, no pointers, sandboxing and automatic type checking.

Security

In the seminal article ?Smashing The Stack For Fun And Profit? [9] buffer overflows were presented to world in great detail. A good in depth discussion is also available in [4]. The basic idea is that when memory is allocated on the stack or the heap we have a buffer in memory. When a program, in C or C++, writes to this buffer it can overfill the buffer and overwrite other memory. The problem arises when a malicious user writes to a buffer, where the input is not sufficiently checked for size of input, and writes past the end of the buffer. In some cases on the stack, a knowledgeable malicious user can rewrite the return address of the stack frame, rerouting the control flow of the program. If the program can be rerouted to an additional location overwritten by the malicious user, control flow of the program can be returned to arbitrary code created by the attacker.

Buffer overflows are an endemic security problem in computing today. The most recent high profile internet worm, Slammer, ?exploited a buffer overflow vulnerability in computers on the Internet running Microsoft's SQL Server or MSDE 2000 (Microsoft SQL Server Desktop Engine).? [7] More statistically valid is the study done by Wagner et al. in 2000. In a look at CERT advisories over the last decade ?shows that buffer overruns account for up to 50% of today?s vulnerabilities, and this ratio seems to be increasing over time.? [8] All of these vulnerabilities would be avoided in a language like Java. ?modern safe languages like Java are immune to the problem, barring some incredibly unusual programming.? [4]

Format string attacks are another security problem that exist in C and C++. They have existed for a long time but were only recognized in 2000. [4] This attack works when the programmer writes in place of the correct:

```
fprintf (sock, ?%s?, username);
```

the incorrect but usually fine:

```
fprintf(sock, username);
```

This allows the user to add in the format of their choice. This becomes a problem with the ?%n? format string which can allow the writing of arbitrary integers onto the stack. [4]

Java is not subject to buffer overflows basically because Java will check to make sure that buffers in memory are not overwritten. Java is also not subject to format string attacks because of the way formatting is done in the String class in Java.

Java is more secure because of added features such as a sandbox and security manager. Java was designed from the ground up to be secure. Code running in the JVM from untrusted sources is placed in a sandbox by the security manager. The security manager decides what code to trust, under what circumstances. Typically applets downloaded over the web will be placed in a sandbox. Code running in the sandbox will not have many abilities most normal programs have and will be limited as to which resources it can use.

Reliability

C and C++ are hard to get right. Rutgers teaches Java as its undergraduate pedagogical language of choice. This is because of the difficulty of writing correct code in C and C++. Pointers are notoriously hard to get right. A miss pointed pointer can lead to the dreaded SEG FAULT. When a pointer is miss-placed it can point at memory that doesn?t belong to the process at hand triggering the protection mechanisms of the operating system. This will bring any process to its knees. It is also worth mentioning the relative ease in which a pointer can be given a wrong value. A simple off by one error can cause this entire problem.

Cleaning up memory can be difficult in C and C++. Every piece of memory allocated on the heap has to be created, tracked and eventually destroyed. Even the smallest piece of memory that is not destroyed when done can create a potentially devastating memory leak for a long running application. It can be difficult to keep track of all memory allocated making the complexity of writing a correct and non-leaking program that much more difficult.

Java is much easier to get right. Java has no pointers. Java has references which can point to objects but not to arbitrary memory. Java has exceptions implemented well. Instead of crashing when an exceptional condition occurs, Java can catch and handle the condition. It has garbage collection. Memory allocated on the heap is cleaned up when there are no references reaming to it. It is handled transparently by the JVM. No added complexity for the programmer to handle worrying about the allocated memory. Also no insidious memory leaks to worry about. It is a mature modern language built from the ground up to bypass many of the problems that C and C++ have. Code written in Java will therefore be more relaible.

An added bonus of this is, that without pointers, potential memory leaks and with quality exceptions, it can be faster to code an equivalent application in Java. Not only reliability and security are addressed but speed of development can be positively effected by the above.

Portability

We cannot forget the most touted benefit of Java, its portability. Since the JVM has been implemented on every major existing platform (and some not so major). Java can theoretically be run in its compiled form on any platform. In practice this turns out to not be entirely true. There are some system dependencies but, for the most part, many applications are indeed platform independent.

Performance

The downside to java is of course the performance issue. You have to deal with the Java Virtual Machine (JVM). This introduces performance penalties in a variety of ways. The code is only compiled into an intermediate step which means there has to be some real time interpretation at runtime. Also you have the fairly substantial memory footprint of the JVM to run for any Java program.

The common wisdom is that C and C++ blow java out of the water in terms of performance. This is true anecdotally but looking at some experimental results makes this seem not as clear. The results from [11], a comparative study of C and Java, point to some advantages of C over Java but these difference appear to have as much to do with the different compilers as with which language is used. Even in the same language there are widely varying results based on the compiler. The variance in results seem to be possibly more compiler dependent then language dependent.

So we can see that Java has a penalty in terms of performance versus C and C++. Is this performance really a necessity? The rapid trend of upwards computer performance tells us that we can potentially trade some performance for other benefits. Recent CPU speeds have led people to speculate that for the large desktop market there are not many applications that need the speed available in the high end machines. Also CPU clock speed doubles every 18 months where the performance disparity between Java and C or Java and C++ stays constant. This gives us performance to utilize and trade for a better (but slower) language such as java. The above mentioned benefits seem to outweigh the performance gap for most applications.

Another issue that ameliorates the performance gap is that Java?s performance can be improved by compiling it into native code. Some of Java?s benefits, notably the virtual machine in which it runs and the sandbox that the JVM provides is lost. If compiled directly into native machine code the entire performance penalty due to interpretation by the JVM is gone. Projects such as GCJ purport to solve this problem for us. It can compile Java source code directly to native machine code. [10] It is unclear why most server side Java that I am aware of does not use this type of compiler. You can gain almost all the benefits of Java (and since it is server side, portability is not the largest concern) but yet most server side Java seems to use the traditional JVM. This may be due to a lack of maturity in the GCJ compiler or may only signify an ignorance of this breed of compiler.

Conclusion

C and C++ are bad in terms of security and reliability. Java solves many of these issues as well as bringing us some new features. There is a trade off in terms of performance. This performance gap is mitigated however by quickly increasing computer performance and the ability to compile Java into native code and side step the JVM. Java is the clear winner.

There are still places for C and C++. Operating systems will still need the direct access to memory that these languages provide. It is hard to imagine commercial operating systems being written in Java, for instance, any time soon. Also applications that still demand cutting edge speed, such as scientific applications and graphical engines will still need the edge that C and C++ will bring to the table. Finally C is currently a compilation target for some languages (in certain situation PERL compiles into C for instance) and likely to remain as such.

Despite any of the above arguments against C and C++ we are unlikely to see them disappearing any time soon. The code base is too massive to contemplate its replacement. If we still see COBOL code being maintained in the 21st century, how long may we still see C code being maintained? Also many of today?s developers are strong proponents of C and C++ and will be loathe to see them go. Since we seem to be stuck with them, the obvious solution is to write better, more reliability and security conscious code in C and C++ or attempt to get more applications written in these languages, such as Java, that save us for the above mentioned troubles. Easier said then done, but that is a topic for a separate paper.

References

[1] http://www.research.att.com/~bs/C++.html
[2] http://cm.bell-labs.com/cm/cs/who/dmr/
[3] The C Programming Language, Kernighan and Ritchie
[4] Building Secure Software, Viega and McGraw
[5] Java In a Nutshell, Flanagan
[6] http://www.cigital.com/ssw/presentations/pugh.txt
[7] http://www.cs.berkeley.edu/~nweaver/sapphire
[8] http://www.simovits.com/archive/ndss_039.pdf
[9] http://www.phrack-dont-give-a-shit-about-dmca.org/show.php?p=49&a=14
[10] http://gcc.gnu.org/java/ [11] http://www.aceshardware.com/Spades/read.php?article_id=153