COMPARATIVE ANALYSIS OF JAVA & C++
HISTORY, SIMILARITIES & DIFFERENCES,
SYNTAX AND DESIGN ISSUES

* Ahmed khan
Department of Computing and Technology, Iqra University Islamabad
ahmedkhanibt@gmail.com

Abstract— This paper discuss about Java was initially created to support network computing on embedded systems. Java was designed to be extremely portable, secure, multi-threaded and distributed, none of which were design goals for C++. Java has a syntax familiar to C programmers, but direct compatibility with C was not maintained. Java was also specifically designed to be simpler than C++, but continues to evolve above that simplification. During the decade between 1999 and 2009, especially in the part of the programming industry dedicated to enterprise solutions, “Coffee-based” languages, which rely on "virtual machines" that are familiar in Smalltalk, grew in prominence. This was a trade between performance and productivity, something that made perfect sense at the time where computing power and the need of simplified and more streamlined language that permitted not only easy adoption but a lower learning curve. There is enough similitude between the languages that the proficient C++ programmers can easily adapt to Java, that is still today in ways less complex and even in comparison more consistent in the adopted paradigms than C++. This shift in interest has however decreased, mostly due to the evolution of the languages. C++ and Java evolution has merged much of the gap about the problems and limitations of both languages, the software requirements today have also shifted and fragmented more. Now we have specific requirements for mobile, data-center and desktop computing this makes the programming language selection an even more central issue.

I. INTRODUCTION

A. HISTORY OF JAVA AND C++
C++ was written by Bjarne Stroustrup at Bell Labs during 1983-1985. C++ is an extension of C. Prior to 1983, Bjarne Stroustrup added features to C and formed what he called ‘C with Classes’. Java started to be developed in 1991 by James Gosling from Sun Microsystems and his team. The original version of Java is designed for programming home appliances. In 1994, James Gosling started to make a connection between Java and internet. The original name of Java is Oak. The new name Java was inspired by a coffee bean. “While Java is viewed as a programming language to design applications for the Internet, it is in reality a general all-purpose language which can be used independent of the Internet.

B. C++ AND JAVA FROM THEORETICAL ASPECTS
In C++, the basic data type is almost the same as C. In C++, there is one more type Booleans. A Boolean, bool, can have one of the two values true or false. A Boolean is used to express the results of logical operations. The table summarizes the basic data types and qualifiers in C++. There are 32 keywords both in C and C++. There are 30 keywords that are not in C, but they are used in C++. Java has two kinds of floating-point numbers: float and double. The default type when you write a floating-point literal is double. Table shows Java floating-point. [4][3] There are only two values available for Boolean type, true and false. Integral Types in Java is different from C++. There are five data types that can have the integral value. There are total 50 keywords in Java. Some keywords only appear in Java, such as synchronized, instanceof and strictfp.

II. MAJOR DIFFERENCES BETWEEN JAVA AND C++

A. MEMORY AND PROJECT MANAGEMENT
Java has a module mechanism called packages that groups related classes together. Classes in the same package typically can directly access one another's instance variables. Java has no .h files. All the declarations and code for a class are placed in the same .java file. Java's designers made this decision because they felt it was confusing to have to keep switching back and forth between .h and .cpp files to determine how a class operates. Typically you will only declare one class per file. The class name must match the file name Java does not require a make file. The compiler is supposed to find all the necessary .java files using a CLASSPATH environment variable and then compile the .java files if they are out of date.[5]

B. INHERITANCE
Java does not support multiple inheritances because the developers of Java felt that both the conceptual and implementation complexities and ambiguities it introduces outweigh the benefits. Java instead uses the notion of interfaces for classes that want to support more than one set of methods. Interfaces will be discussed later in the course. All methods in Java are virtual methods.

C. ENVIRONMENT ISSUES
Java does not compile its programs down to machine-level instructions as is done by C++. Instead it compiles programs into byte codes that are interpreted by a virtual machine. A virtual machine is a software program that simulates a machine
and its instruction set. Because of its byte codes, Java is architecture-neutral. In other words, a Java byte code program will run on any machine on which its virtual machine runs. This does mean that the virtual machine must be specially coded for each architectural platform. [5]Java's virtual machine provides a good deal of run-time checking to ensure that the program does not breach the security of the platform on which the program is running: array bounds checking type checking of downcasts garbage collection so that the programmer does not have to handle memory de-allocation and dangling pointers. The interpretation and runtime services come at the price of decreased efficiency. Even well coded Java programs often run two times slower than corresponding C++ programs. Just-in-time (JIT) compilers can compile byte codes down to machine instructions but the resulting programs are still less efficient than corresponding C++ programs because: 1) the programs still must provide the run-time services, and 2) a compiler can optimize a program better when it has access to the original source code than to the byte codes [2].

D. POINTER, ARRAYS AND STRING
Java does not have an explicit pointer type. Instead of pointers, all references to objects— including variable assignments, arguments passed into methods, and array elements—are accomplished by using implicit references. References and pointers are essentially the same thing except that you can’t do pointer arithmetic on references (nor do you need to). References also allow structures such as linked lists to be created easily in Java without explicit pointers; simply create a linked list node with variables that point to the next and the previous node. Then, to insert items in the list, assign those variables to other node objects.

Arrays in Java are first class objects, and references to arrays and their contents are accomplished through implicit references rather than via point arithmetic. Array boundaries are strictly enforced; attempting to read past the end of an array is a compile or run-time error. As with other objects, passing an array to a method passes a reference to the original array, so changing the contents of that array reference changes the original array object. Arrays of objects are arrays of references that are not automatically initialized to contain actual objects. Using the following [2][3] Java code produces an array of type MyObject with ten elements, but that array initially contains only nulls:

```
MyObject arrayofobjs[ ] = new MyObject[10];
You must now add actual MyObject objects to that array:
for (int i=0; i < arrayofobjs.length; i++) {
    arrayofobjs[i] = new MyObject();

Java does not support multidimensional arrays as in C and C++. In Java, you must create arrays that contain other arrays. Strings in C and C++ are arrays of characters, terminated by a null character (‘\0’). To operate on and manage strings, you treat them as you would any other array, with all the inherent difficulties of keeping track of pointer arithmetic and being careful not to stray off the end of the array. Strings in Java are objects, and all methods that operate on strings can treat the string as a complete entity. Strings are not terminated by a null, nor can you accidentally overstep the end of a string (like arrays, string boundaries are strictly enforced).

E. OPERATORS AND CONTROL FLOW
Operator precedence and association behaves as it does in C. Note, however, that the new keyword (for creating a new object) binds tighter than dot notation (.), which is different behavior from C++. [2] In particular, note the following expression: new foo().bar; This expression operates as if it were written like this: (new foo()).bar; Operator overloading, as in C++, cannot be accomplished in Java. The, operator of C has been deleted. [6] The >>> operator produces an unsigned logical right shift (remember, there are no unsigned integer data types). Although the if, while, for, and do-while statements in Java are syntactically the same as they are in C and C++, there is one significant difference. The test expression for each control flow construct must return an actual boolean value (true or false). In C and C++, the expression can return an integer [3].

III. FEATURES OF JAVA

A. SIMPLE AND OBJECT-ORIENTED
Like C++, Java is an object-oriented language [1]. Object-oriented languages allow the programmer to organize a program so that it closely models the real world in structure and in the interactions among its components. This is particularly valuable in implementing applications using the graphical user interfaces popularized in the PC world by the various versions of Microsoft Windows. Writing an application using a graphical user interface tends to be much easier by use of an object-oriented language than otherwise.

B. DISTRIBUTED AND INTERPRETED
Java was built with the Internet and Web in mind. As do most other languages, Java includes prebuilt components or libraries that provide important additional capabilities beyond the language itself. However, Java’s standard libraries specifically include network-aware units that greatly facilitate writing Internet applications. Additionally, the building blocks of a particular Java application do not have to reside locally, on your desktop machine, as they do with traditional programming languages. Java is an interpreted language. This means that Java’s executable files are composed of so-called byte codes that are instructions and data relating to a hypothetical computer called the Java virtual machine. [7] Each machine that runs a Java program uses a small program, known as the Java run-time system, to execute the Java byte codes in your program. This design is what makes it possible to run the same program on a Macintosh, a Sun, and a PC.

C. ROBUST AND SECURE
A robust program is one that does not fail (at least in a catastrophic way). If you’ve used a computer for any length of time, you’re obviously aware that much modern software lacks this property. One cause of this is the complexity of modern software systems. Java contains features that make the task of writing robust software easier. In Java programs, exceptions can be detected and handled according to instructions written
by the programmer, often allowing software to keep working in the face of unexpected problems. One of the potential tears of the Internet is the possibility of security breaches—viruses that infect your computer, or hackers who take advantage of a software. [4]Java has a multitude of ways for dealing with evildoers who would try to compromise your system using a Java program. Applets, which are Java programs automatically downloaded when a Web page is displayed, are subject to a number of limitations that are designed to reduce the chance that simply viewing someone’s page might result in harm to your system or data.

D. ARCHITECTURE NEUTRAL
This means, of course, that your Java program will work identically whether you run it in a Quonset hut or in a Georgetown townhouse. [5]No, wait! That’s not what it means at all. The word architecture in this phrase does not refer to the building in which you live, but to the home in which your computer program lives—in other words, the computer system. Java’s byte codes are designed to be read and interpreted—in exactly the same manner—on any computer hardware or operating system that supports a Java run-time. No translation or conversion is necessary.

E. PORTABLE AND HIGH PERFORMANCE
An early form of portability involved carrying media, for example, floppy disks, from one system to another. Portability became a much larger problem once different sorts of computers were interconnected to form the Internet. Java programs contain no implementation-dependent aspects, so the result of executing a series of Java byte codes should always be the same no matter on what system they are executed. Moreover, the Java run-time system itself, though it is written in C, is written in a way that simplifies porting the Java run-time to a new PC. A typical problem with interpreted languages is that they are somewhat less efficient than compiled languages. A program written by use of an interpreted language may run 20 to 100 times slower than the same program written by use of a compiled language. Java aims at overcoming this problem through the use of a technique known as just-in-time compilation. [6]A just-in-time compiler is an interpreter that remembers the machine code sequences it executes corresponding to the input byte codes. Studies have suggested that just-in-time compilation may make interpretation of Java byte codes almost as efficient as native execution of machine-language code.

F. MULTITHREADED AND DYNAMIC
A multithreaded system can, for example, format a floppy disk while a user surfs the Web using a browser. Multithreaded applications allow you to complete more tasks in a given time and to use a system’s resources more efficiently. However, developing multithreaded applications in C or C++ can be agony, because these languages lack standard support for operations necessary to create and control threads. Java includes support for multithreaded applications as part of its basic library. Java’s program units, classes, are loaded dynamically (when needed) by the Java run-time system. Loaded classes are then dynamically linked with existing classes to form an integrated unit. The lengthy link-and-load step required by third-generation programming languages is eliminated. [7]Thus, when parts of an application you use are updated, you don’t have to buy the latest copy. The dynamic nature of Java allows you as a developer to always have the most up-to-date version of your software available to your users.

IV. SIMILARITIES & DIFFERENCES BETWEEN JAVA & C++

<table>
<thead>
<tr>
<th>Features</th>
<th>Java</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Operator Overload</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Typedef, Define,</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Preprocessors</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Structures, Unions</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Enums</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Functions</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Goto statement</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Automatic Coercions</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Global Variables</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Templates</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Private, Protected, Public Inheritance</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Default parameters</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Garbage Collection</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Multi-thread support</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Multiple Inheritance</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Function Overload</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Internationalization</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Include of Classes</td>
<td>#import</td>
<td>#include</td>
</tr>
</tbody>
</table>
| Comments            | "//", "/* */,/** | "//","/

V. CODE COMPARISON OF C++ AND JAVA

1) Main Function
- C++
  // free-floating function
  int main(int argc, char* argv[])
  {
    printf( "Hello, world" );
  }
- Java
  class HelloWorld
  {
    public static void main(String args[])
    {
      System.out.println( "Hello, World" );
    }
  }

2) Compiling
- C++
g++ foo.cc -o outfile
// run with
./outfile

• Java
// compile classes in foo.java to <classname>.class
javac foo.java
// run by invoking static main method in <classname>
java <classname>

3) Class Declarations
• C++
class Bar {};
• Java
class Bar {};

4) Comments
Same in both languages (// and /* */ both work)

5) Method Declarations
Same, [8] except that in Java, must always be part of a class, and may prefix with public/private/protected.

6) Constructors and destructors
Constructor has same syntax in both (name of the class), Java has no exact equivalent of the destructor.

7) Static member functions and variables
Same as method declarations, but Java provides static initialization blocks to initialize static variables (instead of putting a definition in a source code file):
class Foo
{
    static private int x;
    // static initialization block
    { x = 5; }
}

8) Scoping static methods and namespaces
• C++
If you have a class and wish to refer to a static method, you use the form Class::method.
class MyClass
{
    public:
        void foo();
        void bar();
};
// now it's used like this
MyClass::doStuff();

• Java
You always work with references (which are similar to pointers—see the next section), so you always use a dot:
myClass x = new MyClass();
x.my_field; // ok

9) Object declarations
• C++
// on the stack
myClass x;
// or on the heap
myClass *x = new MyClass;
• Java
// always allocated on the heap (also, always need parens for constructor)
myClass x = new MyClass();

10) Accessing fields of objects
• C++
If you're using a stack-based object, you access its fields with a dot:
myClass x;
x.my_field; // ok
But you use the arrow operator (->) to access fields of a class when working with a pointer:
myClass x = new MyClass;x->my_field; // ok
• Java
You always work with references (which are similar to pointers—see the next section), so you always use a dot:
myClass x = new MyClass();
x.my_field; // ok

11) References vs. pointers
• C++
    int bar = 7, qux = 6;
    int& foo = bar;
• Java
// references are mutable and store addresses only to objects; there are
    // no raw pointers
    myClass x;
    x.foo(); // error, x is a null ```pointer```
    // note that you always use . to access a field

12) Inheritance
• C++
class Foo : public Bar
{ ... };
• Java
class Foo extends Bar
{ ... };

13) Protection levels (abstraction barriers)
• C++
    public:
        void foo();
        void bar();
• Java
    public void foo();
    public void bar();

14) Virtual functions
• **C++**
  virtual int foo(); // or, non-virtually as simply int foo();
• **Java**
  functions are virtual by default; use final to prevent overriding
  int foo(); // or, final int foo();

15) *Abstract classes*
• **C++**
  class Bar { public: virtual void foo() = 0; };
• **Java**
  abstract class Bar { public abstract void foo(); }
  interface Bar { public void foo(); }
  class Chocolate implements Bar{
    public void foo() { /* do something */ }
  }

16) *Memory management*
Roughly the same--new allocates, but no delete in Java since it has garbage collection.

17) *NULL vs. null*
• **C++**
  // initialize pointer to NULL
  int *x = NULL;
• **Java**
  // the compiler will catch the use of uninitialized references, but
  // need to initialize a reference so it's known to be invalid,
  assign null
  myClass x = null;

18) *Booleans*
Java is a bit more verbose: you must write boolean instead of merely bool.
• **C++**
  bool foo;
• **Java**
  boolean foo;

19) *Constness*
• **C++**
  const int x = 7;
• **Java**
  final int x = 7;

20) *Throw Spec*
• **C++**
  int foo() throw (IOException)
• **Java**
  int foo() throws IOException

21) *Arrays*
• **C++**
  int x[10];
  int *x = new x[10];delete[] x;
• **Java**
  int[] x = new int[10];

22) *Collections and Iteration*
• **C++**
  Iterators are members of classes. The start of a range is
  <container>.begin(), and the end is <container>.end(). Advance
  using ++ operator, and access using *.
  vector myVec;
  for ( vector<int>::iterator itr = myVec.begin();
    itr != myVec.end();
    ++itr )
    { cout << *itr; }
• **Java**
  Iterator is just an interface. The start of the range is
  <collection>.iterator, and you check to see if you're at the end
  with itr.hasNext(). You get the next element using itr.next() (a
  combination of using ++ and * in C++).
  ArrayList myArrayList = new ArrayList();
  Iterator itr = myArrayList.iterator();
  while ( itr.hasNext() )
    { System.out.println( itr.next() ); }

23) *Data Abstraction; Separation of Interface and Implementation*
• **C++**
  /*
  * This program calculates and prints
  * a fibonacci number, using a
  * recursive function to do the
  * calculation. It also prints out
  * the number of times the function
  * was called.
  */
  #include <iostream>
  using namespace std;
  int fibCallsCounter;
  /* Calculate the nth fibonacci
  * number. Increment fibCallsCounter
  * every time it is called.
  */
  int fib(int n)
  { fibCallsCounter ++;
    if (n <= 2)
      return 1;
    else
      return fib(n-1) + fib(n-2);
  }
  int main(int argc, char * argv[])
  { while (true)
{ 
    int N;
    fibCallsCounter = 0;
    cout << "Desired value of N - " << "0 to quit: ";
    cin >> N;
    if (N == 0)
        break;
    cout << "The" << N << "th fibonacci number is "
        << fib(N) << endl;
}

24) Namespaces and Packages
• C++
    file person.h:
    /* This class represents a Person */
    #include <string>
    using std::string;
    class Person
    {
    public:
        // Constructor
        Person(string name, int age);
        // Mutator
        void setAge(int age);
        // Accessors
        string getName() const;
        int getAge() const;
        // Print out full information
        void print() const;
        private:
            string _name;
            int _age;
    }; // *** NOTE TO THE READER: THIS SEMICOLON IS _REALLY_ IMPORTANT

    file person.cc:
    /* Implementation for class Person */
    #include "person.h"
    #include <iostream>
    using std::cout;
    Person::Person(string name, int age)
    : _name(name), _age(age)
    { }
    void Person::setAge(int age)
    { _age = age; }
    string Person::getName() const
    { return _name; }
    int Person::getAge() const
    { return _age; }
    void Person::print() const
    { cout << "Name: " << _name << "Age: " << _age; }

• Java
    file Person.java:
    /* This class represents a Person */
    public class Person
    {
Ahmed khan | Code Comparison of C++ and Java

/** Constructor
* @param name the person's name
* @param age the person's age
*/
public Person(String name, int age) {
  this.name = name;
  this.age = age;
}

/** Mutator for age
* @param age the new age
*/
public void setAge(int age) {
  this.age = age;
}

public String getName() {
  return name;
}

public int getAge() {
  return age;
}

/** Print out full information */
public void print() {
  System.out.print("Name: " + name + " Age: " + age);
}

private String name;
private int age;

25) Inheritance
  • C++
    /* This class represents a Student */
    #include "person.h"
    class Student : public Person {
      public:
        // Constructor
        Student(string name, int age, string major);
        // Mutator
        void setMajor(string major);
        // Accessors
        string getMajor() const;
        virtual void print() const;
      private:
        string _major;
    };

    file student.cc:
    /* Implementation for class Student */
    #include "student.h"
    #include <iostream>

    Student::Student(string name, int age, string major):
      Person(name, age), _major(major) {};
    void Student::setMajor(string major) {
      _major = major;
    }
    string Student::getMajor() const {
      return _major;
    }
    void Student::print() const {
      Person::print();
      std::cout << " Major: " << _major;
    }

  • Java
    file Student.java:
    /* This class represents a Student */
    import Person;
    public class Student extends Person {
      /** Constructor
       * @param name the student's name
       * @param age the student's age
       * @param major the student's major
       */
      public Student(String name, int age, String major) {
        super(name, age);
        this.major = major;
      }
      /** Mutator for major
       * @param major the new major
       */
      public void setMajor(String major) {
        this.major = major;
      }
      /** Accessor for major
       * @return the student's major
       */
      public String getMajor() {
        return major;
      }
      /** Print out full information */
      public void print() {
        super.print();
        System.out.print(" Major: " + major);
      }
    }
26) Static and Dynamic Binding of Methods

- **C++**
  
  ```cpp
  A* a = new B();
a -> yyy();
  ```

- **Java**
  
  ```java
  A a = new B();
a.yyy();
  ```

27) Storage Allocation Model

- **C++**
  
  ```cpp
  void someRoutine()
  {
    SomeClass * p = new SomeClass();
    // Code that uses p
    // Now suppose the object we created
    // is no longer being used
    // Free up the space allocated by new
    delete p;
  }
  ```

- **Java**
  
  ```java
  class Foo {
  public:
    Foo(char ch, int code)
    { this.ch = ch;
      this.code = code;
    }
    ... // STATE OF MEMORY SHOWN HERE
  }
  ```

28) Value, Pointer, and Reference Parameters to Functions/Methods in C++

- **C++**
  
  ```cpp
  void addOne(int x) // Add one to parameter
  { // Doesn't work!
    x ++;
  }
  ```

  ```cpp
  int i = 42;
  addOne(i);
  cout << i << endl;
  ```

  ```cpp
  void addOne(int * x) // Add one to parameter
  { // This one works!
    (* x) ++;
  }
  ```

  ```cpp
  int i = 42;
  addOne(& i);
  cout << i << endl;
  ```

  ```cpp
  void foo(int x)
  { }
  int i = 42;
  foo(i);
  cout << i << endl;
  ```

- **Java**
  
  ```java
  void foo(int x)
  { }
  int i = 42;
  foo(i);
  System.out.println(i);
  ```

29) Const/final variables

```java
static int a = 1;
int [] c = new int[3]; // Close, but
// not identical
public static void bar()
{
  // Java couldn't declare b
c[2] = 3;
  int [] d = new int[4];
d[1] = 4;
  // No java equivalent
  Foo f = new Foo('B', 66);
}
```
VI. JAVA DESIGN ISSUES

Not strongly-typed at compile time casts are checked at runtime. Therefore a system design can only be checked by run-time testing. This means that design problems may not be seen until too late, and they may not show the big picture. If there are an almost infinite number of paths through the code - for instance, a converter, compiler, or virtual machine - then the design flaws may never be seen during testing. All objects are passed by reference, therefore developers must do a great deal of explicit cloning in order to ensure that object references are not being shared. Developers must manually clone an object if they want to get a copy of an argument that they can safely change. Collections (e.g. lists) are collections of object references, so a method that returns a collection needs to manually clone each item in the collection (note that normal collection cloning just creates a copy of the same references). Developers need to trust that methods do this properly although there is no way to tell without looking at the implementation. Java programmers often believe that there are no pointers in Java, but forget that everything apart from a base type (such as an int) is a reference. It is far too easy to change an object and inadvertently affect some other code that you didn't know was sharing the same object. Also, this type of problem is very difficult to debug - it just looks like random behavior. Developers do not know whether a method will change the values of its arguments unless they look at the implementation of that method. This type of problem is also difficult to debug because coders rarely expect a method to change its arguments. There are no collections of particular types of objects - there are just collections of references to Objects. Collections will normally contain references to instances of a particular class or its subclasses, but nothing about a collection's declaration says what it will contain. Developers who take objects from these collections must cast to the expected type, but the compiler is unable to check the validity of the cast until run-time. Again, a developer can only know what a method will do to a collection by looking at the implementation. It is possible to create your own subclassed containers which are specialized for a particular type, but this leads to a lot of repetitive code, and it is not possible to change the return type of an overridden method.

VII. C++ DESIGN ISSUES

As a basic design premise, consider using references instead of pointers. A quick example for comparison:

```cpp
define Print(int x):
    x = x + 1;
```

```cpp
define Print(int& x):
    x = x + 1;
```

The `Ptr` function and the `Ref` function generate exactly the same machine language. The advantages of the Ref function:

1. **Limit Exception Handling**
   Exceptions are a great way to deal with unexpected errors. But they're expensive. Scott Meyers notes that throwing an exception is about "three orders of magnitude slower" than a normal return. Programs using exceptions are about 5-10% larger and 5-10% slower.” There are a couple of options. Avoiding exceptions means not using try, throw or catch in your code or in library code. [2][7] Use operator new with the no-throw specification. Turn off exception handling in the compiler itself. I believe the judicious use of exceptions is the best solution. Limit try blocks to a few key places and make use of the throw().

2. **Avoid Runtime Type Identification**
   Runtime type identification allows you to programmatically get information about objects and classes at runtime. For instance, given a pointer to a base class, [8][6] you can use RTTI to determine exactly which type of class the object really is. The `dynamic_cast` operator relies on RTTI to perform the proper casting of objects.

3. **Prefer stdio to iostream (printf vs. cout)**
   C++ stream I/O is a very flexible and safe method of doing input and output. It allows you to define specific output formats for your own objects. If an object doesn't support output, the compiler will tell you. Our old friend printf, on the other hand, is not very safe.

```cpp
int x;
void Ref(const int& p) { x += p; }
```

```cpp
int x;
void Ptr(const int* p) { x += *p; }
```
cout << 'a' << ' ' << 1234 << ' ' <<
setiosflags(ios::showpoint)
<< setiosflags(ios::fixed) << 1234.5678 << ' ' <<
setiosflags(ios::hex) << &i << ' ' << "abcd" << 'n';
printf("%c %d %f %p %sn", 'a', 1234, 1234.5678, &i,
"abcd");

4) Evaluate Alternative Libraries
It can pay big dividends to consider alternative libraries,
whether they're 3D, graphics, mathematical or I/O libraries. The
libraries that came with your compiler are probably not the
most efficient code available.

VIII. CONCLUSION
Java is a fun language. C++ programmers should have a
relatively easy time learning it, and will find that they enjoy
using it. I have noted a few problems with the language in the
above paper, but I don’t consider these to be very critical
issues. Moreover, I don’t know of any language that doesn’t
have such problems. Language design always involves trade-
offs that displease someone. I look forward to writing lots of
interesting Java applications. I also look forward to watching
how the language evolves from this point onward. I expect to
see some changes in the next few years.

IX. REFERENCES