

# Re: Universal Programming Lanaguage

---

*Source:* <http://sci.tech-archive.net/Archive/sci.logic/2006-03/msg00337.html>

---

- *From:* The Ghost In The Machine <[ewill3@xxxxxxxxxxxxxx](mailto:ewill3@xxxxxxxxxxxxxx)>
  - *Date:* Fri, 24 Mar 2006 19:30:41 GMT
- 

On Thu, 23 Mar 2006 00:00:25 -0800, edward.meinert wrote:

What work has been done in a language that allows translation from c to c++ to java , etc.

Um....there's a *\*lot\** of factors in such a work.

[1] C/C++ uses pointers; Java uses them quite differently (and hides them from everyone unless they're using something like JNI).

C and C++ allow statements such as

```
const char * str = "This is a string";
const char * p = str + 8;

printf("\">%s\<">\n", p);
```

which will print out "a string". While Java does have a substring() call, it is not clear whether the two strings share memory or not. (Not that it matters since strings are immutable in Java anyway -- mostly. See [2].)

For its part Java's pointers are much more hidden but in code such as the following:

```
class ABC { ... }
class DEF { public void callMe(ABC abc) { abc.modifyMe(); } }

ABC abc = new ABC();
DEF def = new DEF();
def.callMe(abc);
```

after return from DEF.callMe() the object in ABC has been happily modified, if modifyMe() deigns to do so.

It gets worse. In the following:

```
int x = 5;
```

## Re: Universal Programming Lanaguage

```
DEF def = new DEF();  
def.callMe2(x);
```

x does *\*NOT\** get modified, as it's passed by value; this pass by value is done for all primitive types (int, char, byte, short, long, double, boolean, float). This dichotomy in Java is a wart but not horribly fatal, but in C++ one can pass either by value or by reference regardless of whether the type is a primitive or a structure. (However, passing bitfields to something requiring a reference doesn't work, and C++ may throw a type conversion if one, say, passes a long to an int &, even if on the machine in question the long and the int are the same size.)

C++ also allows

```
int * p = 0;
```

(C also allows it but it's nonportable; one is supposed to use NULL, which is usually declared in a header file). In Java, writing

```
ABC abc = 0;
```

simply will not work unless ABC is the special type Integer, and in that case one gets (surprise surprise) an Integer whose intValue() is 0. (This as of Java 5, which has implicit boxing. Usually this is used in the context of routine calls.)

[2] Java Strings are immutable (except by cheating — it's a *\*very\** ugly hack). std::string can be modified. There are also issues regarding char \* and quoted strings; for historical reasons

```
char * p = "xyzyz";
```

is allowed, though the compiler's not all that happy about it. (The proper usage would be 'const char \*p = "xyzyz";' .)

Note that "..." is *\*not\** a std::string; an implicit conversion is available, however. Nor can std::string be converted to a char \* or const char \* without calling a cstr() method. In Java constructs such as

```
"abcde".length()  
"defgh".getBytes()  
"bdfhj".equals(x)
```

make perfect sense, especially if x can be the constant null.

[3] Java does not have operator overloading, unsigned integer types, the 'const' keyword ('final' has vaguely similar semantics), default copy constructors, assignment operators, or bit fields.

In short, if one writes

## Re: Universal Programming Lanaguage

```
ABC abc;  
DEF def;
```

in C++, one gets two different objects.

If one writes

```
ABC abc = new ABC();  
ABC def = abc;
```

in Java, abc and def refer to the exact same object.

[4] Java does not have a preprocessor. #defines such as

```
#define BEGIN {  
#define END ;}  
#define IF(x) if(x)  
#define ELSE  
#define THEN
```

will allow constructs such as

```
IF(a == 5) THEN  
BEGIN  
b = 1;  
END  
ELSE  
BEGIN  
c = 2;  
END
```

but, without a C preprocessor, will probably confuse the hell out of a converter. (There's a yearly Obfuscated C contest that encourages, erm, interesting submissions exhibiting the flexibility -- or, perhaps, the total mangleability --- of the C language, and the C preprocessor.)

[5] Java does not have globals, although it can have scoped static variables and methods, and one could dedicate a single class called something like "CGlobals" in a pinch.

[6] Java does not have printf() et al, though one can work around that by defining a printf() that takes as a first argument a format string, and a second argument an array of Object.

[7] Java templates are quite different from C++ templates; nor does Java's API map nicely to C++'s Standard Library. For starters, in Java, the interface java.util.List has a get(int) method; in C++ a std::list is implemented as a doubly-linked list and there's no elegant method by which one can get the nth item thereof, except by walking along it.

Also, Java sets, while sortable (using a java.util.Comparator or by

## Re: Universal Programming Lanaguage

declaring an object to implement Comparable<OtherType>, by writing the method `int compareTo(OtherType other)`), have a little problem if someone else modifies an object which is also in the set. C++ `std::set` does not have this problem since adding an object to a `std::set` effectively clones it (unless one declares a `std::set` which holds onto a bunch of pointers, but that's a different beast).

[8] Method pointers in C++ don't have a really good analogue in Java, although a `java.lang.reflect.Method` would probabyl do in a pinch. Ditto for field pointers.

[9] C/C++ do not have the synchronized keyword, introspection (though C++ does have some `typeof()` semantics), packages (unless one counts `makefiles` building DLLs) or implementation of multiple inheritance methods. Java does not allow multiple inheritance but it *does* allow an object to explicitly or implicitly implement multiple interfaces, each of which can declare a routine with the same signature; the object can then satisfy all of these declarations with a single implementation.

C++ doesn't even know what an interface *is*, although it does have the notion of an abstract class (using the routine `(...) = 0; construct`). However, C++ easily runs afoul of what is known as the dreaded "diamond inheritance" problem -- which is why Java does not allow multiple inheritance as such.

[10] Java's dynamic class loading is very different from the operating system's dynamically loadable libraries. (C/C++ do not implement anything in this realm.)

[11] There are a number of differences in C++'s and Java's access control. For starters, C++'s `public:`, `private:`, and `protected:` are modal, whereas in Java one is expected to prefix every declaration -- although one can also omit the prefix in Java entirely, which means only package neighbors (and, of course, oneself) can access the fields and/or methods. C++ does not have the concept of package, as such.

[12] Exception handling in Java is much more regular than in C++.

[13] In Java there's one type of cast, and it's perfectly safe (if one's willing to tangle with the `NullPointerException` and `ClassCastException`, anyway). In C++ there are multiple cast types, of varying safeness.

I'd think of more but I'm running low on time. :-) The JVM also has its issues, which I for one am going to want to study at some point.

—  
#191, ewill3@xxxxxxxxxxxxxx  
It's still legal to go .sigless.