

# CS1Bh Lecture Note 27

## A Furtive Glance at C

### 27.1 Introduction

Programming language design is not a science, rather it is an art. The aim is to create a new programming language which is an improvement on previous languages, at least in some area. Languages which are non-traditional in some respect have historically not succeeded in gaining a large pool of users. This leads to there being a number of programming languages which have a form of ‘family resemblance’, with similar syntax and similar ideas. One such family includes the languages ALGOL, CPL, BCPL, B, C, C++ and Java, listed from oldest to newest. Here we compare Java (begun in 1991) with its ‘grandparent’, C, which appeared in 1973. A comparison with the earlier languages would be less useful because none is in common use (although ALGOL, originating in 1958, has been extremely influential). A comparison with C++ would be less interesting because the two languages have many similarities.

In comparing Java and C, one must remember that without C there would be no Java. First, Java is a descendant of C, and builds on decades of experience with the use of that earlier language. Second, Java directly depends on C because some of the Java run-time system is actually written in C, and we have actually made heavy use of it in this course. Aside from this historical influence on Java, C is, without doubt, a useful language in its own right. In fact, almost all of the software used in the CS1 course was written in C: the Linux operating system, KDE, even the word processing system used to produce this note. Many people have been, and still are, able to use it to produce efficient, usable applications. Thus, one should certainly not assume that Java is unquestionably better than C, just because it is more modern.

However, there are various ways in which the thirty-year-old C might now be considered to be dated, and reasons why newer languages like Java might be more suitable for the more modern methods of software engineering aimed at increasing programmer productivity. By analogy, we can return to an example from the start of the course: the group of programmers developing the Unix operating system in the early 1970s at Bell Laboratories decided to write the operating system in C at a time when the customary choice was assembly language. C was a great improvement over the use of assembler because it provided high-level control structures such as the **for** loop and the **switch** statement. It also provided data structures such as arrays, which do not exist in assembly language. In addition, programs were first processed by a compiler that could identify at least some errors, which could be removed before the program is used.

## 27.2 Java's resemblance to C

C is, in many senses, a simpler language than Java because Java contains many language constructs which C does not have. At the core of the language though, the statements look very much the same. As an example, the implementation of the factorial function in this C program could be used line for line identically in a Java program:

```

1  /* This is a C program not Java */
2  int factorial (int n) {
3      int m = 1;
4      while (n != 0) {
5          m *= n;
6          n--;
7      }
8      return m;
9  }
10 main () {
11     printf("%d\n", factorial(6));
12 }
```

The last few lines point to differences between Java and C. In particular, C has no classes or objects — it has independent *functions*, rather than methods associated with classes. The program ends with a main function, whereas Java would have a main method. Functions in C are *called* with their actual parameters (as in `factorial(6)`), whereas in Java we invoke a method on an object.

## 27.3 Compiling and running C programs

C programs are processed by a compiler to produce an executable machine code program, with a flow just as illustrated in Figure 23.3 of Lecture Note 23 (“Machines”). For example, if the above program is in a file called `factorial.c`, the GNU C compiler (`gcc`) can be invoked by the Unix shell command:

```
[adielupbj: gcc factorial.c -o factorial
```

The machine code program is then executed by using its file name, here `factorial`, as a command, preceded by the specification of the current directory:

```
[adielupbj: ./factorial
720
[adielupbj:
```

An assembly language representation of the program can be inspected by using the `-S` flag to produce a `factorial.s` file.

```
[adielugjb: gcc -S factorial.c
```

Unlike Java's byte code, which is portable from system to system, the machine code program will differ from platform to platform. For example, the version of `gcc` for an Intel x86 will produce quite different output from versions for a MIPS or a PowerPC.

## 27.4 Assistance in detecting errors

One of Java's strengths over C is that it detects and traps more errors in programs, both at compile-time and at run-time. In terms of assistance to the programmer, this is akin to the assistance that a spelling checker gives to the user of a word processor. For instance, Java systems detect *statically* that variables have not been initialised and detect *dynamically* if a program attempts to access past the end of an array (by throwing an *exception*). C systems do neither of these, since the definition of the language does not actually treat such things as being in error. As an example, the following C program will compile and run without any error messages during compilation or execution:

```

1  /* This is C, not Java */
2  main () {
3      int a[5]; /* forgot to initialise the array */
4      int i;
5
6      for (i = 0 ; i < 10 ; i++)
7          printf("%d\n" , a[i]); /* a[i], i > 4 OK in C */
8  }
```

There are two errors: forgetting to initialise the array and attempting to access elements 5, 6, 7, 8 and 9 of an array of size 5. Thus, the results of running the program are unpredictable, as one would expect.

Of course, some errors which are made by programmers can be detected and some cannot. Returning to the factorial example again, consider the negligent version shown below. Neither C nor Java would be able to detect the logical error that the programmer forgot to perform the multiplication in the loop in the body of the factorial function. However, Java will report as an error that the programmer omits the return statement which returns the *int* result from the function, whereas C will not.

```

1  /* This is a C program not Java */
2  int factorial (int n) {
3      int m = 1;
4      while (n != 0) {
5          /* multiplication has been forgotten */
6          n--;
7      }
8      /* return statement has been forgotten */
9  }
```

In fact, Java systems supply more hand-holding than this — for example, by reporting an error if there is some path through the body of the function whereby the **return** statement would not be executed. Note that all these forms of error checking do not arise from the more elaborate language constructs of Java, but rather from a stricter interpretation of how the programmer is supposed to make use of the language, intended to prevent accidents or naughtiness.

## 27.5 Object orientation

A major distinction between Java and C is that (unlike its successor C++) C has no objects. In a deep sense, there are perceived conceptual benefits in viewing a problem in terms of its component objects and the operations that might be performed on them. However, in a more practical sense, objects can be used to prevent programmers from danger through tampering with the internal implementation details. To take a simple and basic example of this point, in C, when dynamic data structures such as lists are to be built, the programmer has to explicitly allocate blocks of memory of the right size, and connect them together using *pointers*. C's pointers are essentially just memory addresses, unlike Java's references. Given this, the fact that pointers are unsigned integers means that one can add C pointers together, test them for relative ordering, and other such machine-dependent activities.

Continuing the theme of danger, the programmer does not just explicitly allocate memory space, but must also arrange to free it up again when it is no longer required. In other words, C has no garbage collection to organise this, as Java has to get rid of redundant objects. A particular difficulty that can result is the *dangling pointer* problem. Here, a part of the memory which was identified by a pointer is freed up, but the pointer continues to point to that location. Subsequent allocations can overwrite the data which was there with data of another type, but the program can still access the data by its previous type, leading to a type errors and potential chaos at run-time.

Building on the general theme of object orientation, one further reason for the popularity of Java over C is the availability of the diverse range of packages for things like graphics, mathematics and networking. This gives the programmer a head start by providing an attractive environment of ready-made, and system-independent, objects. While equivalent facilities are available in C, they typically require somewhat lower level fumbling in order to be used effectively.

## 27.6 Run-time performance

The generally higher-level approach to programming that is encouraged by Java does come at a price, in particular in terms of efficient execution. The fact that the language constructs, and general flavour, of C are quite closely mappable to typical machine instruction sets means that fast, and relatively compact, machine code programs can be generated by compilers. The latter point is still of practical relevance, particularly in *embedded systems*, where memory sizes can be very restricted. In contrast, features like the run-time error checking carried out by Java mean inherently slower execution. Also, the original model of interpretation of Java byte code meant slowness compared with executing native machine code. When direct compilation of Java to machine code is done, however, execution times can now be within range of C program equivalents.

Overall, there is a trade-off: system performance gains through continued use of C come at the price of the possibilities of hard-to-detect bugs and of some reduction in portability between machines. An experienced programmer should be able to make an informed choice, depending on particular applications and circumstances.

*Authored by Stephen Gilmore and Gordon Brebner.*

*Paul Jackson, May 6th, 2003.*