Programming in Franglais

Rodney Bates, Wichita State University

When I was studying French in high school, we students often spoke “Franglais”: French grammar and words where we knew them, English inserted where our command of French failed us. It was pretty awful, and the teacher did not think highly of it. But we could communicate haltingly because we all had about the same levels of knowledge of the respective languages.

Today, there is a kind of programmer’s Franglais that is all too pervasive. Those who are old enough will remember the pitched controversy in the late 1960s and early 1970s over whether compilers, operating systems, and other systems programs should be written in assembly code or a high-level language. The prime argument for the languages was that their higher-level computational model allowed far more function to be coded in the same amount of development time.

I was a slow adopter on this one, until the emergence of languages with type systems rich enough to support complex linked data structures and memory management, which pushed me over that line. Unfortunately, after three decades, we still aren’t really there. The problem is that in the most popular languages, C and C++, there is lots of undefined and undefinable behavior, because, in reality, programming has to be done in a confusion of two, quite different computational models.

UNDEFINED, BUT STILL WITHIN THE LANGUAGE

First, let’s look at a milder example of undefined behavior that nevertheless does not force us outside the high-level language model. Most languages, Java being a notable exception, do not define the order in which the actual parameters of a call are evaluated, leaving this up to the implementer. Normally, the same also goes for the order of evaluation of the operands of binary operators. The standard justification is that, when a code generator is right on the borderline of running short of registers, it may be able to avoid storing and reloading one, if it can reorder the evaluation.

This can make a difference in the behavior of the program. For example, suppose I write

```
f(g(),y);
```

If execution of `g` changes `y`, it will mean the compiler’s

Six of one,

HALF A DOZEN

OF D’AUTRE.

choice of whether to evaluate `g()` before or after `y` will alter the parameters received by `f`, and probably

`f`’s result as well. Programmers are on their own recognition not to write code that depends on this order. Otherwise, they will get no error message, either at compile time or runtime.

It is possible that the program will work—or appear to work—as desired, but suddenly break after a seemingly unrelated change elsewhere in the code, which alters the compiler’s usage of registers. Bugs like this don’t occur especially often, but when one does, it often takes such an inordinate amount of time to discover, diagnose, and fix, that the effect on a project is out of proportion to its seeming simplicity.

This is not ideal programming language design, but there are at least a couple of mitigating circumstances. All of the possible behaviors that might ensue can be enumerated and understood by a programmer, using solely the concepts of the high-level language. The number of permutations of evaluation order goes up factorially with the number of parameters, of course, but each can be understood in a reasonable way, if you take the time.

It would be perfectly easy for the language designers to eliminate the problem by just defining the evaluation order, probably left-to-right. That would add only two or three sentences to the language definition. It would also be quite easy for a compiler writer to comply—in fact, easier than exploiting an undefined evaluation order.

An alternative that is not feasible would be to leave the order of evaluation undefined, but expect the compiler to detect code whose behavior depends on the actual order chosen. This is a classic undecidable problem, so there would always be infinitely many coding situations where a compiler could not be sure. The best it could do would be to generate an error whenever it couldn’t be sure that there was no problem.

UNDEFINED, AND FORCING INTO FRANGLAIS

But there is a far more serious kind of undefinedness.

Continued on page 78

Continued from page 80

There are large areas in C and C++ that are not only undefined, but simply can't be defined. Both defining and understanding them requires shifting to an assembly programmer's model of data. There, data is a big linear array of storage units, all having the same size and set of possible values. Each type in the language has a different way of encoding its values into the bits and storage units at the machine level. Some of these encodings could be defined by the language (though they seldom are), but some aspects are unavoidably machine-dependent.

The gap between a language's model and the machine-level model gets a lot wider when you start making memory accesses that spill over the edges of objects. In every high-level language, data is divided into what I will call "complete objects." By this, I mean objects that are not fields of structs or classes, elements of arrays, or other subcomponents of some containing object. I include local and global variables, as well as heap-allocated objects. Each complete object is independent of all the others. It is meaningless to access any data beyond its boundaries.

Unsafe languages, however, allow this to happen very easily—for example, by casting a pointer to one type of object into a pointer to a different-size object, or, of course, to an array-bounds error. To define this, you have to resort to the assembly programmer's model of data as occupying a big, linear, homogeneous array of fixed-size units. Objects in the high-level language model are laid out next to each other in memory in the machine model. Machine code and anonymous constants get mixed in there, along with various things the programmer didn't explicitly code—for example, stack linkages, register spill areas, etc. Linked-in libraries contribute their own stuff in all of the above categories.

To make sense of what happens when you go outside the bounds of an object, you have to know the placement of all these things. It happens by a complex conspiracy involving the compiler, linker, loader, heap manager, all source code, including linked-in libraries, and perhaps some other things such as the operating system's memory management and protection scheme. This is all unavoidably highly dependent on the hardware and software of the platform. It is not only undefined in C and C++, but also impossible to define using the data model of the language.

This is the Franglais of programming. It's a hodge-podge of high-level and assembly-level concepts. And it makes a lie of the claim I referred to at the beginning, that programming in high-level languages allows you to get away from the machine-level model of computation.

HOW FAR HAVE WE COME?

C did make progress in some areas. If you disregard pointer arithmetic, array subscripting, and related things such as the way strings are represented and the way heap allocation has to be done, most of the remaining operations in C are comprehensible in terms of its own type system. They are higher-level in that the programmer need not think about registers or address modes. The expression operators are mostly machine-independent, disregarding the apparently intentional relegation of the modulo ("%") operator to the belief that speed is more important than predictable answers.

The standards for C and C++ have attempted to deal with the areas that can't be defined in the language's own model by what I call "affirmative undefinedness." Since defining them is hopeless, the reference manuals instead enumerate at great length all the things a programmer could write that are undefined. "Undefined" is a technical term that in essence means if you do it, you will neither get behavior you can depend on nor an error message, either at compile time or runtime.

The undefined areas are a huge and gerrymandered territory for a mere mortal programmer to have to remember to stay out of through discipline alone.

We can do better. A wide range of languages, starting about the same time as C, provide mostly type-safe facilities that only rarely need to be circumvented, and make these work across separate compilations. Several variants of Pascal, Modula, Ada, Oberon, and Java are all examples. They largely fulfill the promise that high-level languages allow the programmer to get out of the machine-language data model. But unfortunately, programming in the most popular languages is like having to communicate with someone who is fully conversant in neither English nor French, but only Franglais. Programmers must be able to think simultaneously in two, very different models, in order to understand the code they are working on. ☹

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

RODNEY BATES (rodney.bates@wichita.edu) has never done a paid day's work in electrical engineering, for which he was trained. Instead, he has worked 33 years in computer software, two-thirds in industry, the rest in academia. He has been involved mostly with operating systems, compilers, and as a resident programming language lawyer. He is an assistant professor and graduate coordinator in the computer science department at Wichita State University.

© 2004 ACM 1542-7730/04/1100 \$5.00