An AWK to C++ Translator

Brian W. Kernighan

Bell Laboratories
Murray Hill, NJ 07974
`bwk@research.bell-labs.com`

ABSTRACT

This paper describes an experiment to produce an AWK to C++ translator and an AWK class definition and supporting library. The intent is to generate efficient and readable C++, so that the generated code will run faster than interpreted AWK and can also be modified and extended where an AWK program could not be. The AWK class and library can also be used independently of the translator to access AWK facilities from C++ programs.

1. Introduction

An AWK program [1] is a sequence of pattern-action statements:

pattern { *action* }
pattern { *action* }
...

A *pattern* is a regular expression, numeric expression, string expression, or combination of these; an *action* is executable code similar to C. The operation of an AWK program is

```
for each input line
    for each pattern
        if the pattern matches the input line
            do the action
```

Variables in an AWK program contain string or numeric values or both according to context; there are built-in variables for useful values such as the input filename and record number. Operators work on strings or numbers, coercing the types of their operands as necessary. Arrays have arbitrary subscripts (''associative arrays''). Input is read and input lines are split into fields, called `$1`, `$2`, etc. Control flow statements are like those of C: `if-else`, `while`, `for`, `do`. There are user-defined and built-in functions for arithmetic, string, regular expression pattern-matching, and input and output to/from arbitrary files and processes.

The standard implementation of AWK is an interpreter: an AWK program is parsed into an internal representation, which is then interpreted by a set of routines.

AWK is a closed language; there is no access to libraries or to separately written code. This often forces users into contortions to do some operation that would be much more naturally expressed in some other language. A translator into C++ provides a way to extend AWK programs, by combining them with C++ code.

AWK is in some ways a seductive tool (see, for example, [2, 3, 4]) and there are numerous examples of AWK programs that grew from sensible small versions into awkward large ones. A translator provides an escape path: a program may be translated so that further development takes place in a more suitable language.

Both of these uses require that generated code be especially readable and easy to work with. The primary goal throughout the experiment has been that the translated output should be as close as possible to the original AWK input. This means that we want to define variables that have the semantics of AWK variables and use them in the natural syntax in expressions with the usual C operators, C and AWK functions, and C built-in data types. AWK syntax is close enough to C to make an exact match feasible in many places, and provide a reasonable mapping in others.

It is necessary to duplicate AWK semantics, but that is not sufficient. There is already an excellent AWK to C converter, that generates C that exactly matches the semantics of AWK. Roughly speaking, an program will run about twice as fast as the corresponding AWK program, so if one wishes only to speed up an existing AWK program, is quite satisfactory. The output of however, was never meant for human consumption, and thus it is not appropriate for augmenting or extending AWK code.

This project has several components. Their development was carried on in parallel, since they are related and activities in one area affect the other areas. I have tried to separate these as much as possible but there is still room for confusion. Here is a sketch of the pieces and events.

The *AWK interpreter* is a C program originally written in 1977 and much modified since then. For most people, the interpreter *is* AWK. The first step was to translate the interpreter into the C subset of C++ and then to make some minor changes in implementation to use C++ better. This version of the interpreter could (but does not) replace the standard C version. There was also no need to make this a C++ program but it was good practice.

The second step was to modify the interpreter so that instead of interpreting AWK programs, it translates them into C++; this program is called the *translator* and the C++ it produces is called *generated code.*

The generated code does not stand alone; it assumes that it will be compiled (by a C++ compiler) with a *header file* that contains class declarations for AWK data types and loaded with a *library* of separately compiled functions for input-output, field splitting, regular-expression matching, etc. Thus the third step is the development of this header file and library. These are written in C++.

The translator, header file, and library are interdependent, since each performs actions or provides services that the others depend on. There are often trade-offs among them, since one can do more work in one place to simplify life in another. Among the issues that can be traded off are complexity, efficiency, and readability of generated code.

2. Translation

The translator parses an AWK program creates a parse tree and walks it recursively; at each node it calls a routine that generates C++ code. The translator makes only relatively simple use of C++ facilities; its origin as a C program is very clear.

The generated code is meant to be compiled with a header file `Awk.h`, to be discussed in the next section, and loaded with a library of run-time routines. The generated code is based on this template:

```
#include "Awk.h"

// declarations of user variables and functions, if any

main()
{
        BEGIN();        // if there is a BEGIN block
        while (getline() > 0) {
                // code for pattern-action statement 1
                // code for pattern-action statement 2
                // ...
        }
        END();          // if there is an END block
}

// user function definitions, if any
```

Considerable effort has been devoted to generating readable code. For example, no redundant parentheses are produced for expressions, and no redundant braces are produced around non-compound statements. The output is piped through the C beautifier `cb` so it is properly indented. Declarations of external variables are sorted so that variables are easy to find in large programs. Default arguments and multiple declarations are used for functions like `substr` and `split` that can be called with different numbers of arguments.

There are some AWK notations that simply cannot be made to look the same in C++. For example, there is no C exponentiation operator, so `x^y` becomes `pow(x,y)`. (The alternative, overloading the C `^` exclusive operator, must be rejected because its precedence does not match the AWK precedence for exponentiation.) There is no explicit operator for concatenation in AWK; the translator generates a function call instead of overloading some operator, because there is no suitably mnemonic C operator with the right precedence. The common cases of concatenation of two or three strings are handled with `cat(s1,s2)` and `cat(s1,s2,s3)`; longer concatenations use nested function calls. Constant regular expressions must be delimited by quotes instead of slashes, and an extra layer of backslashes must be added to protect embedded backslashes.

The notation for fields is a problem, since there is no way to use '`$`' as AWK does. After some experimentation, I decided to use the variable `F` so that fields are called `F(0)`, `F(i+1)`, and so on. There are also definitions for `F0`, `F1`, etc., so that an expression like `$1 > $2` can be expressed as `F1 > F2`.

A small example will illustrate many of these points. This AWK program reads a list of numbers and prints the list with serial numbers and percentage of the total:

```
    { x[NR] = $1; sum += $1 }

END { if (sum != 0)
          for (i = 1; i <= NR; i++)
              printf("%2d %10.2f %5.1f\n", i, x[i], 100*x[i]/sum)
    }
```

The translator generates this code:

```
#include <stdio.h>
#include "Awk.h"

int     i;
double  sum;
Array   x;
void    END();

main(int argc, char *argv[])
{
    Awkinit(argc, argv);
    while (getline() > 0) {
        x[NR] = F1;
        sum += F1;
    }
    END();
}

void END()
{
    if (sum != 0)
        for (i = 1; i <= NR; i++)
            printf("%2d %10.2f %5.1f\n", i, (double) x[i], 100 * x[i] / sum);
}
```

The basic type is an `Awk`, which captures the semantics of AWK variables mentioned in Section 1: a string value, a numeric value, or both, depending on usage. In general, most variables in an AWK program would be translated into `Awk`s.

Since it is more efficient to use built-in types, however, the translator attempts to infer the simplest type that will serve for each variable. For example, since the variable `i` is used only as the index of a loop, it can be an `int`, while `sum` is a `double`, and `x` is an `Array`, a type that captures the notion of an AWK array, i.e., an indexable collection of `Awk`s. The coercion `(double)` in the `printf` is necessary to convert the array element, of type `Awk`, to a number for printing. There is no need for a coercion for `i`, however, because its type already matches; in this case, type inference leads to more readable code and potentially more efficient code.

Type inference in the translator is fairly *ad hoc*. A type is associated with each constant. Types are combined at operators to produce a result type; for instance, the result of an addition is an `int` if both operands are integer; otherwise it is `double.` Relational operators always produce `int` regardless of the type of their operands. Types of variables are set by assignment statements and also by usage in expressions and function arguments. The operands of an operator are assigned a tentative type based on the operator; for example, the expression `x+y` implies that `x` and `y` are used arithmetically. This may later be changed to `int` because the variables are only used in `int` contexts. If a numeric variable is used in an explicitly `double` context, such as `sqrt(x)`, or if no further information appears, it will become `double`. Array elements are always assumed to be of type `Awk`, as are fields, since it is too uncertain to propagate an assumed type.

Ideally, one should do a data flow analysis to propagate type information, but instead several passes are made over the parse tree; this way, information that ‘‘flows backwards’’ can be handled, so long as the backward path is not too long. For example, in a sequence like

```
{ i = j; j = 2 * k; k = 1; x = y; y = z + 1; z /= 2 }
```

after two passes over the tree it will be concluded that `i`, `j`, and `k` are all of type `int`, while `x`, `y`, and `z` are all `double`.

3. The Awk Class

It is easy enough and quite satisfying to generate clean, clear C++ code, with all the operators in place, and no redundant parentheses or braces. It turns out to be harder to define a class that captures the behavior of AWK variables so that the clean expressions produce the expected results. In this section we will describe the `Awk` class, which is defined in the header file `Awk.h`.

The `Awk` data type keeps track of the value and state of a variable:

```
class Awk {
  private:
        double  fval;   // floating-point value if currently valid
        String  sval;   // string value if currently valid
        int     state;  // which values are currently valid
        ...
```

(The fragments of `Awk.h` presented here have been somewhat simplified to show the essence without bogging down in details.) The `state` variable holds only two bits, which are set if the numeric or string values or both are currently valid. `String` provides reference-counted strings; it is a tiny subset of the standard C++ library string package.

The next step in the class definition is constructors to create `Awk`s:

```
  public:
        Awk() : sval("")        { fval = 0.0; state = STR|NUM; }
        Awk(int i)              { fval = i; state = NUM; }
        Awk(double f)           { fval = f; state = NUM; }
        Awk(cchar *s) : sval(s) { state = STR; }
        Awk(Awk &a)             { if (a.state & STR) sval = a.sval;
                                  fval = a.fval; state = a.state; }
```

The type `cchar` is an abbreviation for `const char` here and in the sequel.

Similar functions are necessary for assignment of values to `Awk`s:

```
        Awk &operator =(int i)        { fval = i; state = NUM; return *this; }
        Awk &operator =(double f)     { fval = f; state = NUM; return *this; }
        Awk &operator =(cchar *s)     { sval = s; state = STR; return *this; }
        Awk &operator =(Awk &a)       { if (a.state & STR) sval = a.sval;
                                        fval = a.fval; state = a.state;
                                        return *this; }
```

and for increment operators like `+=` and for `++` and `--`.

There are also ‘‘conversion functions’’ for fetching the numeric and string values of `Awk` variables:

```
operator double()
        { return state&NUM ? fval : (state |= NUM, fval = atof(sval)); }
operator cchar *()
        { return state&STR ? sval : (state |= STR, sval = ftoa(fval)); }
```

These are used implicitly when calling a normal C function that expects one of these types as an argument. For example, since `sqrt` expects an argument of type `double`,

```
sqrt(awkvar)
```

is really

```
sqrt( (double) awkvar )
```

The alternative of requiring explicit casts for ''downward'' conversions from `Awk` to built-in types is unacceptable because it severely affects readability; even a few casts are undesirable.

The real complications begin with the arithmetic and relational operators. In an AWK arithmetic expression involving the operator `+`, there are five possible combinations:

```
Awk + Awk
Awk + int
int + Awk
Awk + double
double + Awk
```

Each of these produces a `double` value.

The obvious way to handle this is to overload the `+` operator as a `friend` function:

```
friend double operator +(Awk &, Awk &);
```

The arguments must be passed by reference since the function has to be able to cause the side-effect of updating the numeric state of each argument if necessary. A `friend` function is required so that the left-hand operand can be an `int` or `double`; if a member function were used, the left-hand operand would have to be an `Awk` and this would preclude expressions like `1+Awk`.

Unfortunately, the simple solution doesn't work, because an expression like

```
Awk + int
```

is ambiguous; it could be parsed as either of

```
Awk + (Awk) int
(int) Awk + int
```

The problem is that basic types can be promoted ''up'' into `Awk`s and `Awk`s can be converted ''down'' into basic types, and there is no way to state which choice is preferred. C++ provides an elaborate sequence of rules that determines how type-matching of functions is done, but when it is finished, if there are two matched functions, the construction is ambiguous.

The problem is that in AWK all possible conversions are legal; given both upward and downward implicit conversions, the only way to capture this at compile time is to spell out all possible combinations:

```
friend double operator +(Awk &, Awk &);
friend double operator +(Awk &, int);
friend double operator +(Awk &, double);
friend double operator +(int, Awk &);
friend double operator +(double, Awk &);
// and so on for - * / %
```

All told there are 5×5 functions for arithmetic operations.

As it is for arithmetic operators, so it is for relationals, except that there are more combinations and the semantics imposed by AWK are more complicated: it is necessary to look at the state of each variable in a comparison to determine whether the comparison is numeric or string. This makes another 42 functions (6 operators, 7 type combinations). Fortunately, most of these are trivial and can be expanded in-line.

By the way, it is necessary to distinguish `int` from `double`, rather than relying on the automatic coercion that would otherwise take place. Consider the expression

```
Awkvar == 0
```

In the absence of explicit functions for `int`s, `0` can be a `double` or a `char*`, so this construction would be ambiguous.

4. Fields

The next complication is the treatment of fields. Fields in AWK are for the most part the same as ordinary variables except that they have potential side effects, and there are significant efficiency considerations since field-splitting is expensive.

Each time a new line is read, the input record `$0` is set, but it is undesirable to set `$1`, etc., until they are actually needed. In addition, if any field is assigned to, that invalidates the value of `$0`, but it is undesirable to recreate `$0` until its value is needed again. Similarly, if `$0` is explicitly assigned to, that invalidates `$1`, etc. Thus some form of lazy evaluation is called for.

To make all of this work, it is necessary to intercept every reference to any field. In the interpreter, there is type information in each variable that indicates whether the variable is a field. That requires a run-time test for each access to any variable, so it seems better in the generated C++ code to implement fields as a separate type, thus moving the test to compile-time.

Thus fields are implemented as a new type, called a `Field`. The initial try was to derive `Field` from `Awk`, so that most operations would be inherited, but this doesn't work because operations performed on derived objects may bypass the explicit assignment and conversion operations in favor of implicit ones and thus avoid the code meant to trap references to fields.

A `Field` really isn't an `Awk`, since it may never be used as a plain `Awk` without taking the side effects into account. Thus a `Field` *contains* an `Awk`:

```
class Field {   // an individual field
  private:
        Awk a;
        void rvalue(), lvalue();
        operator double() { rvalue(); return (double) a; }
  public:
        friend double operator +(Field &x, int d) { return (double) x + d; }
        Field &operator =(Field &x) { x.rvalue(); lvalue(); ... }
        // ...
};
```

Within this class definition, in every context where the value of the field will be used, a function `Field::rvalue` is called to ensure that any necessary field-building is performed. In any context where a field will be assigned to, a function `Field::lvalue` is called to build fields and to record any information about invalid state. The `lvalue` function must also be called from a few functions such as `sub` and `gsub` that can alter fields implicitly.

With one exception, there are no explicit variables of type `Field`; rather, fields are members of an array managed by a class called `Fields:`

```
class Fields {  // manages an array of Field's
  private:
        Field fields[100];
  public:
        Field &operator()(int n) { return fields[n]); }
};

Fields F;       // the fields are stored here
```

The only `Fields` operator is `()`, which is used to access an individual `Field`. Thus constructions like `F(0)`, `F(i+j)`, and so on return a reference to the corresponding `Field`.

Since a `Field` does not inherit from an `Awk`, and since a `Field` can be converted to an `Awk` and vice versa, it is again necessary to provide overloaded operators for all possible combinations of `Fields` with other types, and to add some operators to class `Awk` for combinations of `Awk` and `Field`.

It is also necessary to deal with the AWK built-in variable `NF`, which records the current number of fields. If `NF` is referred to, the fields must be computed, or at least counted. The easiest way to do this turned out to be to make `NF` a variable of type `Field` and to add a bit of special code in the `lvalue` and `rvalue` routines to handle it correctly. (The thought of adding another type and another 75 functions to manage it was more than I could bear.)

5. Arrays

AWK arrays are implemented as a separate class `Array`. Thus `Awk v[10]` is not an AWK array in the traditional sense, but `Array v` is. The specific implementation of an array doesn't matter here; the only visible operations are subscripting to implement associative arrays, membership test, and element deletion:

```
class Array {
  private:
        // standard hash table here
        Awk &lookup(cchar *);   // install if not found
  public:
        Array();

        Awk &operator [](Awk &s)        { return lookup(s); }
        Awk &operator [](Field &s)      { return lookup(s); }
        Awk &operator [](cchar *s)      { return lookup(s); }
        Awk &operator [](double f)      { return lookup(ftoa(f)); }
        Awk &operator [](int i)         { return lookup(ftoa(i)); }
        // etc.
};
```

Again, an explicit rule is needed for `int`; otherwise, `x[0]` is ambiguous since `0` is a `double` and a `char*`.

6. Print Statements

The treatment of the AWK `print` and `printf` statements present some interesting problems. In AWK, one writes

```
print e1, e2, e3        # print values of 3 expressions
print e4 > e5           # print e4, redirecting into file e5
print e6, e7 | e8       # print e6 and e7, piping output into e8
```

The `e`s are arbitrary expressions. The goal is to generate code that looks as much as possible like this, something that is natural for human readers and reasonably efficient. The translation has to be legal C++ and provide AWK semantics when executed: values are separated by the value of the variable `OFS` (usually a space) and an `ORS` (usually a newline) is added at the end.

One solution is to define a print function that takes a fixed (large) number of arguments of a single type, a ‘‘print arg’’. Constructors are defined that make a print arg from each kind of object that will be printed. There is a default value that marks the end of the real arguments. Default arguments are used to convert calls into the full-length list; short versions are provided for common cases.

```
class Prarg {
        int t;
        union {
                cchar   *sval;
                int     ival;
                double  dval;
        };
  public:
        Prarg()                  { t = 'v'; ival = 0; }  // void marks the end
        Prarg(int n)             { t = 'i'; ival = n; }
        Prarg(double d)          { t = 'd'; dval = d; }
        Prarg(cchar *s)          { t = 's'; sval = s; }
        Prarg(Awk &a)            { t = 's'; sval = a; }
        Prarg(Field &a)          { a.rvalue(); t = 's'; sval = a; }
};

extern  Prarg   Prarg0;          // will mark end of list

void    print(const Prarg& = Prarg0, const Prarg& = Prarg0,  // etc.
              const Prarg& = Prarg0, const Prarg& = Prarg0);
```

Now `print` statements look exactly the same as they do in AWK, except that the list of expressions must be parenthesized in all cases and there is an upper limit on the number of arguments.

Redirection is handled by a separate class:

```
class Redir {
        char *buf;
  public:
        Redir(char *s)  { buf = s; }

        friend void operator >(const Redir &r, cchar *f);
        friend void operator |(const Redir &r, cchar *f);
};
```

A function named `Fprint`, which is analogous to `print`, creates a string that can be printed by `Redir::operator >(Fprint,filename)` or `operator |(Fprint,filename)`. This permits translation of an AWK statement like

```
print e1, e2, e3 > e4
```

into

```
Fprint(e1, e2, e3) > e4;
```

What about `printf` and `sprintf`? Here, the first argument is scanned for format conversion characters that are used to infer the type of each expression in the list. If the type of the expression doesn't match the conversion character, a cast is generated, avoiding redundant parentheses if possible. So, for example, the AWK statement

```
printf("%s %d %f\n", $1, $2+1, 123.4)
```

generates

```
printf("%s %d %f\n", (cchar *) F1, (int)(F2 + 1), 123.4);
```

The casts are unattractive but there seems to be no better solution.

7. Library

Most of the run-time library comprises either functions necessary to implement the operators, or transliterations of functions from the interpreter for regular-expression matching, input and output, field splitting, and so on. Most of these are much the same, although there are some changes in interfaces. Field splitting is probably the most different, since it now has to interface to the different field-handling implementation described above. The library also includes definitions for built-in variables like `NR` and a routine `Awkinit` to set up the `ARGC` and `ARGV` variables.

One problem arising with the library echoes a previous problem. Consider defining the AWK built-in function `length`, which returns the number of characters in the string value of a variable. The obvious implementation is

```
inline int length(const Awk &a) { return strlen(a); }
```

Thus `length` may be called with any type; a constructor will convert this to an `Awk`, and `operator const char*` will be called implicitly to coerce the string value for `strlen`.

One drawback is that calling a constructor is more costly than might be expected; in fact, even though my constructors are all declared `inline`, some are not inlined because they are too complex. The alternative implementation is again to write out every possible type explicitly. This obviates the problems of constructors and unimplemented features, but it generalizes poorly to functions that have more than one argument, such as `cat` or `substr`.

8. Status

Most of AWK can be handled properly. There are a handful of known bugs and constructions that may never work; some of these are intrinsic to the way that I have made trade-offs.

Type inference creates some problems. Consider the program

```
$1 > $2 { i = NR }
END { print i }
```

This generates

```
int     i;
void    END();

main(int argc, char *argv[])
{
        Awkinit(argc, argv);
        while (getline() > 0) {
                if (F1 > F2)
                        i = NR;
        }
        END();
}

void END()
{
        print(i);
}
```

Notice that the type of `i` has been inferred as `int`. Suppose, however, that `$1` is never greater than `$2`. In the interpreter, the variable `i` will have a null value, so the output will be null. In the compiled code, however, since `i` is an `int`, the output will be a literal `0`. This is an example of a trade-off. Is it better to do type inference and get this one wrong, or not to do it and produce less readable code that runs more slowly?

There are minor problems with name clashes. For example, the standard version of `rand` has different properties from the AWK version, which uses the name `Arand`. There are similar problems with `system`, `sprintf`, C++ keywords, and probably others that I haven't stumbled into yet.

The header file `Awk.h` is 625 lines long and the library is 1275, including comments and some debugging code but excluding regular expression matching. The translator is 3700 lines. For comparison, the interpreter in C is 4900 lines.

Performance is mixed, and it is difficult to decide which comparisons are most representative. On some test cases, the compiled code is significantly faster than the interpreter, while on some others, it is somewhat slower. For example, the prototypical AWK program is to compute a word-frequency count:

```
{ for (i = 1; i <= NF; i++) count[$i]++ }
END { for (i in count) printf("%4d %s\n", count[i], i) }
```

The code generated for this is

```
#include <stdio.h>
#include "Awk.h"

Array   count;
Index   i;
void    END();

main(int argc, char *argv[])
{
        Awkinit(argc, argv);
        while (getline() > 0) {
                for (i = 1; i <= NF; i++)
                        count[F(i)]++;
        }
        END();
}

void END()
{
        For (i, count)
            printf("%4d %s\n", (int) count[i], (char *) i);
}
```

This can also be expressed in C++ with standard libraries:

```
#include <String.h>
#include <Map.h>
#include <stream.h>

Mapdeclare(String,int)
Mapimplement(String,int)

main()
{
        Map(String,int) count;
        String word;

        while (cin >> word)
                count[word]++;
        Mapiter(String,int) p (count);
        while (++p)
                cout << dec(p.value(),4) << " " << p.key() << "\n";
}
```

The following table shows running times for four implementations, on an input file of 320,000 bytes with about 600 distinct words:

`Map` class	16.9 sec.
AWK interpreter	11.5
AWK-C++	8.1
AWKCC	5.7

On another test, a large program that includes representative AWK statements, the results are less favorable:

AWK interpreter	117 sec.
AWK-C++	71
AWKCC	39

Although it might appear that is uniformly faster, this is not true. Computing values of Ackermann’s function up to Ack(3,5) shows quite a different picture:

AWK interpreter	21.1 sec.
AWK-C++	1.8
AWKCC	44.5

This stress test of the function calling mechanism appears to show the AWK-C++ translator in its best light.

9. Observations

The job has turned out to be harder than expected, even allowing for my learning curve and the fact that it has been an oft-interrupted back-burner project. In part this is because it is difficult to match exactly an existing program, warts and all, in a new medium; the task would be far easier if I were free to adjust the problem to the solution, rather than being constrained in all directions.

I spent too much time stumbling around trying to get the overloaded operators right. In retrospect, it is quite trivial, but I kept hoping for some alternative to writing out all possible combinations of operands and operators. This would of course be easier if one needed conversions only in one direction, which is the only situation that textbooks typically mention. It may also be easier with templates, but I have not studied the issue.

There was a similar problem with fields, and a lot of trouble getting the semantics exactly the same as the interpreter. Until the `lvalue` and `rvalue` functions were properly in place (in all places!) this just didn't work.

Reference parameters must be carefully thought through so that one does not incur unnecessary overhead, create unwanted temporaries (current C++ implementations warn of this), or inadvertently modify something that should be untouched. The meaning of `const` for reference arguments is in the hands of the implementer; it is quite possible and sometimes desirable to change the value of a `const` object. For example, updating the string or numeric state of an `Awk` does not change its value as far as the rest of the program is concerned, so this is done even for `const` reference arguments.

One must be careful to respect the levels of abstraction when one is building a data type. Several times I inadvertently used an operation on a type from deep within its implementation; this usually caused an infinite recursion. A typical example is using the assignment operator for a type in some function that is indirectly part of the implementation of that type.

Although most of the bugs I encountered were of my own making, I also uncovered perhaps a dozen bugs in C++ (using a development version of cfront) and a handful of C compiler bugs. In general, these were in areas where I was pushing hard on type matching, operator overloading, and conversion functions, exercising them in unusual ways.

C++ has major advantages. Type checking finds lots of errors early in the game that would be terrible to find in a conventional C program. Type-safe linkage extends this checking to separately compiled routines. Operator and function-name overloading are often a help; they are obviously mandatory for an exercise like this one.

C++ diagnostics are very good, pinpointing errors and often suggesting correct code. Compilation and loading are slow, but not a serious problem, at least on a fast machine.

As others have observed, C++ is not a panacea: one can make many of the old mistakes and some interesting new ones as well. In particular, because the meaning of names and even operators depends so much on context, it is harder to see what is going on in ordinary expressions—more work is required to trace through the meaning of an expression.

Nevertheless, the experience has been positive and instructive. C++ made it possible to undertake a project that would have been infeasible in most other languages.

Acknowledgements

I am deeply indebted to Georges Gonthier, Andy Koenig, Doug McIlroy, Bjarne Stroustrup, and Chris Van Wyk, who spent many hours patiently listening to problems and suggesting ways to attack them. I am also grateful to Stan Lippman for bug fixes and enhancements, and to Jonathan Shopiro for some significantly more efficient code. Jon Bentley, Doug McIlroy, Peter Nelson, Rob Pike, Jonathan Shopiro, Bjarne Stroustrup, and Chris Van Wyk made helpful comments on this paper.

References

1. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley (1988).
2. J. L. Bentley, *Programming Pearls*, Addison-Wesley, Reading, Mass. (1986).
3. H. Spencer, ‘‘AWK as a Major Systems Programming Language,’’ *USENIX Winter ′91 Proceedings* (January, 1991).
4. C. J. Van Wyk, ‘‘AWK as Glue for Programs,’’ *Software — Practice and Experience* **16**(4), pp. 369-388 (1986).
5. J. C. Ramming, *AWKCC: An AWK-to-C Translator*, AT&T Bell Laboratories internal memorandum (1988).