The Ergonomics of Software Porting: Automatically Configuring Software to the Runtime Environment

—or—

Everything you wanted to know about your C compiler, but didn't know who to ask

Steven Pemberton CWI, Amsterdam; steven@cwi.nl

Abstract

When porting C programs to different platforms, you typically need to know many properties of the target machine, such as word length, byte order, minimum and maximum values for a range of types, and so on.

Since the introduction of ANSI C, porters have been aided by the availability of the environment enquiry header files `float.h` and `limits.h`, which give you many such values.

But what do you do if your compiler doesn't have the header files, or the values there are wrong, or you need some value that they don't supply? Where do you find out the maximum floating point value, for instance? What do you do if you are writing a portable C compiler, and you want to supply `float.h` for each machine that the C compiler runs on?

Presented here are the techniques used in a program, *enquire.c*, that calculates and prints out more details of your machine and C compiler than you'll ever want to know.

The program can be used just for interest, to see the properties of your machine, and it can be used to produce the two header files mentioned. An additional option allows you to check that your compiler reads the header files correctly (depressingly many read the minimum floating point number back as zero, or get the last few digits of floating point numbers wrong).

The Problems of Porting

Porting software to new machines poses a problem to the porter: finding which parts of the software reflect architectural features of the hardware that it runs on. Such pieces of code can exist for several reasons: bad programming — the programmer just hadn't considered the possibility; the necessity to address features of the architecture — principally for system programming; and through lack of support in the language for a portable expression of the aim.

Typical sorts of problems in C are: the existence of maximum and minimum values for certain types, whether the character type has signed values or not, the order that characters are stored in memory, most properties of floating point values, how values are converted between different types, certain properties of pointers, and differences between different implementations of C.

Having identified the problem areas, the porter has the option of producing a new version of the program that will run on the new hardware, or of changing the program so that it will run on both machines without change.

Of course, this latter option is to be preferred: it is far easier to keep both versions up to date with the latest modifications, and once you have ported a program once and made the program more portable, it is far easier to then port it to a third architecture.

For many unportable sections of code, there is a solution that will work on all machines without having to make any reference to hardware features; not dereferencing nil pointers is an obvious example of this.

But for sections that address architectural features (like byte swapping) or use unavoidably unportable features (like maxint), the only solution is to parameterise the source code on these features, such as defining a constant, or conditionally compiling different sections of code.

Any good software is designed from the start to be portable, especially with the advent of open-systems, where the likelihood is ever greater that the software will be run on a range of machines. In a sense, open systems have made the task of porting both easier and harder: easier because there are now standards that can be followed; harder because the range of target hosts is even larger, and you can never know all the systems your program will run on.

Software is often not ported or installed by the author of the software, but by someone else frequently at a site far distant from the author, so that there is little opportunity for advice. This means that the author has to make provision to make the job as obvious and straightforward as possible for the porter. The typical set up is to supply a makefile or a header file with constants that have to be changed to match the system it is to be run on before it is compiled. A problem with this is that the values to be filled in may not be obvious for the porter.

Enquire.c

The author is responsible for a large piece of software, ABC, a programming language and programming environment designed for non-expert computer users [Geurts]. The software is widely distributed to many different sites, and it is often installed by the very non-experts that the language is designed for. This means that it must be especially easy for them to install the software, since they will have had little experience with the target machine.

Even worse, there are certain parameters of the system, principally to do with properties of the floating-point representation used on the target machine, that even we found difficult to fill in when we first wrote the software. At best it meant referring to some hardware manual, at worst the manual wasn't even available.

As a consequence, we wrote a small program to determine these properties for the machine it was run on, and print them out. These values could then be filled in in the header files and makefiles, and then a set of header files could be distributed with the system for a range of machines and the installer could choose one suitable for the machine being used.

Of course, it was a small step from this program to one that generated the header file automatically, and could be distributed with the software, so that the software automatically configured itself. This also fitted in well with the aims of the project: the design of the language was on ergonomic principles, and self-installing software can be seen as an ergonomic improvement. A rule of computer ergonomics is that you should never ask the user for information that you know how to calculate; let the computer do the work, that's what it's there for.

From these simple beginnings — a program around 10,000 characters long designed for a particular package — it has evolved into a fully-fledged tool, *enquire.c*, almost 10 times larger, for automatically configuring a range of software packages.

Of course, with the advent of ANSI C [ANSI], and its environment enquiry header files `limits.h` and `float.h`, it could be argued that there is less need for a program like *enquire* now.

This would be true, except that not all compilers (by a long way) are ANSI compatible yet, and even those that are often have faults in the header files. Finally, *enquire* is used by several compiler producers for the very task of producing `limits.h` and `float.h`.

How Enquire Works

The aim of *enquire* is to be usable with as wide a range of compilers as possible, an extreme case of portability. The basic principle is that it must be compilable on any C compiler, under any operating system, with the minimum of user intervention. This means that amongst other things the program must be written in a lowest common denominator version of C, approximately Kernighan and Ritchie level [Kernighan], with no use of modern features like `#if` directives, ANSI standard C, or non-standard or optional libraries such as the math library, and there should be no need for a Makefile or shell or command file to run the program.

Integral Values

A basic value *enquire* produces is the maximum integer value. The essence of the process is this: increase a value until the new value is not larger than the previous value:

```
int intmax= 0, new=1;

while (new > intmax) {
    intmax= new;
    new= new*2+1;
}
```

After this loop, `intmax` has as value the largest integer value.

You will note that this code assumes the base for integers is 2, and not for instance 10. Luckily base 2 is demanded by the language definition for C for `int`, so we can safely make this assumption (which is not the case for floating point as we shall see below).

A problem with this code is that some compilers (though not many) produce overflow signals for integer arithmetic, so we have to protect the code with a trap handler:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf lab;
overflow(sig) int sig; {
    /* what to do on over/underflow */
    signal(sig, overflow);
    longjmp(lab, 1);
}
```

and then:

```
#ifdef SIGFPE
    signal(SIGFPE, overflow);
#endif
#ifdef SIGOVER
    signal(SIGOVER, overflow);
#endif
```

```
int intmax=0, new=1;
if (setjmp(lab)==0) {
    while (new > intmax) {
        intmax= new;
        new= new*2+1;
    }
}

if (setjmp(lab)!=0) {
    fprintf(stderr, "Unexpected signal\n");
    exit(1);
}
```

The `#ifdefs` for `SIGFPE` and `SIGOVER` mean that the code compiles without further intervention on machines that don't have one or other of these signals.

There is now an added complication: not all compilers have `signal` and/or `longjmp` in their libraries. This means that there has to be a compile-time flag, settable by the user, for this case:

```
#ifdef NO_SIG /* No signal() or setjmp/longjmp():
                 Dummy routines instead */
    int lab=1;
    int setjmp(lab) int lab; { return 0; }

    signal(i, p) int i, (*p)(); {}
#else
    #include <signal.h>
    #include <setjmp.h>
    jmp_buf lab;
    overflow(sig) int sig; {
     /* what to do on over/underflow */
        signal(sig, overflow);
        longjmp(lab, 1);
    }
#endif
```

The advantage of supplying dummy routines like this is that the rest of the program can remain the same, and can call `signal` and `setjmp` with dummy effects, without having to pepper the program with `#ifdef NO_SIG` everywhere they are used.

An upshot of all this is that compilers that produce overflow signals, but do not supply `signal` and `longjmp` to catch them, cannot run *enquire*. Luckily we have only come across one such defective compiler, that was later updated to include both functions.

There is a remaining problem with the code: `longjmp` on some systems restores the values of some local variables to what they were at the time of the `setjmp`. Principally variables stored in registers may be restored, and unfortunately the compiler may put variables in registers without being asked, so the value we get out of `intmax` after the loop may be zero!

There are two possible solutions. ANSI C has the keyword `volatile` that helps in this case: volatile variables are never put in registers; Kernighan and Ritchie C has the keyword `static`.

So one cure is to use the following:

```
#ifdef __STDC__ /* Then we have an ANSI standard
                   C compiler */
    #define Volatile volatile
#else
    #define Volatile static
#endif

    Volatile int intmax=0, new=1; ...
```

Another possibility is to move the `setjmp` into the loop:

```
while (new > intmax) {
    intmax= new;
    if (setjmp(lab)==0) new= new*2+1;
    else break;
}
```

A final portability problem with our tiny piece of code to calculate the maximum integer is due to Cyber machines. These have the odd property that some integer operations (such as multiplication) use 48 bits, while others (such as addition and shifting) use 64 bits. This means that replacing

```
new= new*2+1;
```

with the apparently identical

```
new= new+new+1;
```

or

```
new= new<<1+1;
```

would give a different result. Worse yet, some compilers recognise

```
new= new*2+1;
```

as a special case, and ‘optimise’ the code by replacing it with code that does

```
new= new<<1+1;
```

thus defeating our purpose.

What we have done then is to define a variable

```
int two= 2;
```

and used

```
new= new*two+1;
```

which foils the optimiser sufficiently to get it to do what we want. If optimisers get any smarter, then we would have to take more drastic action, like defining a function that returns the value 2 (similar to the drastic action described later to fool the optimiser when compiling floating point operations).

With nearly 40 lines of code necessary for an idea expressed originally in 5 lines, it is clear that portability has its price.

Floating Point

Floating point has all these problems and more. Among the new problems are that we don't know what base is used, and arithmetic doesn't always do what we expect of it.

Floating point numbers are represented as a fraction f of some fixed number of digits, in some base b (usually 2, but 16 is not unknown) and an exponent e, so that a given number is represented as $f \times b^e$. For more details on the representation and properties of floating point, see [Goldberg].

Typical values that you want to know about floating point are the base, the accuracy available (the number of digits in the fraction), the maximum and minimum values, and *epsilon*, the smallest value comparable to 1.0.

To find the base used you can use algorithms described in [Cody]. First you find a small number that adding 1.0 to leaves unchanged:

```
double a= 1.0;
do a= a+a;
while ((((a+1.0)-a)-1.0) == 0.0);
```

And then you find a small number that you can add to this number to give a different value:

```
double b= 1.0, base;
do {
    b= b+b;
    base= (a+b)-a;
} while (base == 0.0);
```

The value of `base` is then the base used.

To understand how this works, assume the base of the arithmetic we are using is 10, and that there are 3 significant digits of accuracy in the fraction.

In the first loop then, a takes the values 1, 2, 4, 8, ..., 512, 1024. The loop then stops, since with 3 digits of accuracy, 1024 is only representable as 1020, and 1020+1 will also give 1020.

In the second loop, the values b takes will depend on how the arithmetic operations round. If they round to nearest, then we will have 1, 2, 4, 8. At this point, 1020+8=1028, which will round to 1030. 1030-1020 is 10, which is the base we are searching for.

If on the other hand floating point rounds down, then 1028 will round to 1020 again, and so the loop will execute once more for b=16, giving 1020+16 = 1036 which will round to 1030 again as in the first case, giving us the base again, 10.

Once you have the base, then it is easy enough to find the number of significant digits by finding the largest power of the base that you cannot add 1.0 to:

```
ndig=0; b=1.0;
do { ndig++; b= b*base; }
while ((((b+1.0)-b)-1.0) == 0.0);
```

There are two portability problems with these pieces of code: firstly, several compilers optimise code even when you don't explicitly ask for it, and they take one look at

```
while ((((a+1.0)-a)-1.0) == 0.0);
```

and replace it with

```
while (1);
```

i.e. an infinite loop.

The golden rule of optimisation (see for instance [Aho]) is that optimisers may never replace code with code that does something else (unless the original code doesn't do anything useful anyway). We leave it to the reader to decide in this case whether the rule has been broken.

The second problem is that some compilers evaluate floating point expressions in registers that give greater accuracy than the variables of the same type. For instance in our fictitious example above, expressions might be evaluated with 5 digits accuracy, so that the value of a found would be 131070 instead of 1020. This will still find the right base, but the accuracy reported will be for expressions and not for variables, which is what we are trying to find out.

The solution for the second problem is to ensure that all intermediate results are stored in variables. Unfortunately something like the following is not enough:

```
do {
    a= a+a;
    a1= a+1.0;
    a1a= a1-a;
    a1a1= a1a-1.0;
} while (a1a1 == 0.0)
```

The reason is the same: optimisers may spot that a value is still available in a register and use that instead.

To get round this, we hide the arithmetic operations in function calls to do basic arithmetic:

```
double sum(a, b) double a, b; {
    double c;
    store(a+b, &c);
    return c;
}

store(val, var); double val, *var; {
    *var= val;
}
```

and similar for difference. Note that

```
double sum(a, b) double a, b; { return a+b; }
```

wouldn't be sufficient in the case that the compiler returned doubles in wide registers.

Luckily, using these functions also solves the first problem, since instead of

```
while ((((a+1.0)-a)-1.0) == 0.0);
```

we now write

```
while (diff(diff(sum(a, 1.0), a), 1.0) == 0.0);
```

which optimisers have a harder time optimising away.

This is theoretically insufficient for very smart optimisers that also look at the content of functions. However, we have never met optimisers that do this sort of optimisation by default: all compilers that we know of that do extensive and advanced optimisation only do so at the request of the user (via a compiler switch). If there ever comes a time that such optimisations are not suppressible, then we would have to resort to using separate compilation for the basic arithmetic functions, so that the optimiser never sees them.

Actually, there is a third problem to the two already mentioned, that shouldn't arise, but that occurs often enough in practice to make it worthwhile checking for: some compilers generate a loss-of-accuracy signal for the first loop for `a`. Loss-of-accuracy traps should never be the default (see [IEEE]), so it is difficult to know why these do occur, but in any case, the program checks for these traps, and aborts if it gets one (since it would be too much work to always account for them when they shouldn't occur anyway).

Epsilon

A useful value when using floating point is *epsilon*. There are several possible definitions for this, but a usual one is "The smallest value that can be added to 1.0 to give a different value"; in fact, this value is dependent on the rounding mode used by arithmetic, and so a generally more useful definition is: "the positive difference between 1.0 and the next larger floating point number". If we call the number according to the first definition `e1`, and according to the second definition `e2`, then `e2=(1.0+e1)-1.0`.

Calculating epsilon is actually one of the easier parts of the program: we use binary search to find `e1`, and then calculate `e2` from it. In the binary search, the range that we still have to search is bounded by the variables `bot` and `top`; the invariants are that `1.0+bot=1.0`, and `1.0+top>1.0`:

```
top= 1.0; bot= 0.0;
mid= bot+(top-bot)/2.0;
while (mid != bot && mid != top) {
    if ((mid+1.0) > 1.0) top= mid;
    else bot= mid;
    mid= bot+(top-bot)/2.0;
}
e1= top;
e2= (1.0+e1)-1.0;
```

(Of course, the real code uses things like

```
mid= sum(bot, div(diff(top, bot), 2.0));
```

)

Floating maxima

When finding the maximum integer, we had to deal with the cases of arithmetic wrap-around (effectively an arithmetic operation producing the wrong value), and possible signals for overflow.

With floating point, we have an extra problem: the possible existence of an infinite value. The problem here is that some floating point systems (principally IEEE [IEEE]) have an infinite value which is larger than all other floating point values, and yet isn't a useful numeric value.

For instance, the basic algorithm for finding the maximum exponent could be something like:

```
f_max_exp= 2; f_max= 1.0; new= base+1.0;
while (new > f_max) {
    f_max= new;
    if (setjmp(lab) == 0) new= new*base;
    else break;
    f_max_exp++;
}
```

The problem here is that if there is an infinite value, the test `new > f_max` succeeds, and we go one too many times round the loop. The easiest cure here is after multiplying by `base`, to see if dividing by `base` again gives you the original number. If not, we've reached infinity:

```
if ((new/base) != f_max) break;
```

Finding the maximum double is not a case of taking the code for integer, and replacing:

```
new= new*2+1;
```

with

```
new= new*base+(base-1.0);
```

To see why, suppose floating point has base 10, 3 digits of accuracy, and one digit of exponent (so that the largest number is .999e9). Then the algorithm will calculate for `new` 9, 99, 999,and then 9999. But this will get rounded to 10000, and further iterations will only multiply this by `base` (because adding 9 will have no effect), so that we would find a largest number of .1e9, rather than the desired .999e9. The cure is first to fill in the digits, and then find the exponent, by multiplying by `base`.

Duplication of Code

In *enquire*, a lot of code has to be duplicated: the code for short, int and long are identical except for the type of the variables used; similarly for float, double, and (where available) long double. Not only is a lot of code involved here, but maintenance of the program is difficult, since changes to a function have to be applied consistently in three different places.

This is of course an ideal situation for using macros. Unfortunately many of the functions used are larger than can be handled by most macro processors. Even the ANSI C standard says that a compiler may refuse to process macros larger than 509 characters.

The solution to this tricky problem is equally tricky: the source file is read three times by the compiler by letting the source file `#include` itself, each time with certain preprocessor symbols set to different values. Three preprocessor symbols `PASS1`, `PASS2` and `PASS3` are used to tell which pass is involved, and the preprocessor symbols are set accordingly. In outline:

```
#undef INT
#undef DOUBLE
#undef INTMAX
#undef DOUBLEMAX

#ifdef PASS1
    #define INT short
    #define DOUBLE float
    #define INTMAX shortMax
    #define DOUBLEMAX floatMax
#endif
```

```
#ifdef PASS2
    #define INT int
    #define DOUBLE double
    #define INTMAX intMax
    #define DOUBLEMAX doubleMax
#endif

#ifdef PASS3
    #define INT long
    #define INTMAX longMax
    #ifdef __STDC__
    #define DOUBLE long double
    #endif
    #define DOUBLEMAX ldoubleMax
#endif
```

Then the functions are declared as follows:

```
INT INTMAX() {
    INT a, b, c;
    /* Code to find the max short/int/long */
    ...
}

#ifdef DOUBLE
    DOUBLE DOUBLEMAX() {
        DOUBLE x, y, z;
        ...
    }
#endif
```

Then at the beginning of the file, the `PASSx` variables are initialised; `PASS` is used to indicate whether the file has already been read and it will be initially unset:

```
#ifndef PASS
#define PASS
#define PASS1
#endif
```

and then at the end of the file is preprocessor code to increment the `PASS`, and reread the file if necessary:

```
#ifdef PASS3
#undef PASS /* No more to do */
#endif
```

```
#ifdef PASS2
#undef PASS2
#define PASS3
#endif

#ifdef PASS1
#undef PASS1
#define PASS2
#endif

#ifdef PASS
#include __FILE__
#endif
```

Note that the order of the tests (`PASS3`, `PASS2`, `PASS1`) is important. The preprocessor symbol `__FILE__` is a standard symbol giving the filename; using this allows you to rename *enquire.c* to anything else without having to change the code.

Checking the Output

The output of *enquire* contains many boundary values of course, and it became clear at an early stage that not all C compilers had been fully tested with all possible boundary numbers; in particular many `printf`'s failed in one way or another to correctly print certain values out. Some produce a trap or garbage when asked to print out the maximum or minimum floating point values; some print zero when printing the smallest floating point value; some get the last few digits of any floating point number wrong, so that reading the number back would produce a different number.

Especially since the output of *enquire* is to be read back and used by other programs, it is important that the output be reliable. To this end, though also to reduce the number of reports that would come in complaining that *enquire* produces the wrong results, all output is checked for correctness by instead of using `printf`, using `sprintf` to output the value to a buffer, and then reading the value back with `sscanf`, and checking that the value read back is the same as the value printed; in outline:

```
sprintf(buf, outformat, val);
printf("%s", buf);
sscanf(buf, informat, &new);
if (new != val) printf("*** Possibly bad output\n");
```

It won't surprise you by now to know that the actual code used is more than 10 times longer than this fragment because of the need to handle problems like `sprintf` producing a trap, `sscanf` producing a trap while reading the number back, or producing an infinite value or other unusable value (like an IEEE NaN [IEEE]). The diagnostics are also more helpful than just announcing that there is a problem, and try to pinpoint the trouble.

Of course if the check fails, the fault may lie with `sscanf` and not `sprintf`: the output may have been correct after all, but at least this code increases the confidence you can have in the output, and indicates possible errors.

Checking the Compiler

The above check on the output tells you if `printf` has produced the right output, but of course, since the results are destined to be read by the C compiler, as a header file, it doesn't tell you if the compiler will read the header file correctly: for the same reasons that `scanf` can fail, so could the compiler. So the final piece of confidence building is an option that allows you to recompile the program so that it reads the `float.h` and `limits.h` files it has produced, and checks that the values it finds there are the same as the values it produces. The kernel of this is a preprocessor symbol `VERIFY`, that is set when compiling *enquire*:

```
#ifdef VERIFY
#include "limits.h"
#include "float.h"
#endif
    ...
    printf("#define FLT_RADIX %d\n", flt_radix);
#ifdef VERIFY
    if (flt_radix != FLT_RADIX)
        printf("*** Compiler has %d for above value\n",
                  FLT_RADIX);
#endif
```

This option can also be used to check that the output agrees with the values the compiler manufacturer has supplied in the two header files.

Autoconfiguring the Autoconfigurer

As mentioned before, the desire to be able to compile the program on as many C compilers as possible means that a certain lowest-common-denominator level of C has had to have been used; no `#if` directives have been used for instance.

On the other hand, there are certain features of compilers that you would still like to report on, even if those features are not available on all compilers. A good example is `long double`: not all compilers have it, but you want to be able to use it on those that do.

For many such features, you can check at compile time. `Long double` only comes with ANSI standard C compilers, and there is a preprocessor symbol for that, as demonstrated above.

But features like `unsigned short` and `unsigned long` cannot be tested for at compile-time, and so there has to be a compile-time flag to allow these parts of the code to be conditionally compiled.

Earlier versions of the program just had compile-time flags for these features, like `NO_SC` for a compiler that has no signed character type. If the user didn't know whether the compiler had this type, the answer was to try compiling the program; at each potential problem point there is a comment saying which flag to set:

```
#ifndef NO_SC
    signed char sc; /* compile with -DNO_SC if this
                       fails to compile */
        ...
```

But still, for a program that is intended to minimise the user intervention necessary when porting software, it is unfortunate to demand user intervention here.

The solution (which only works on Unix and Unix-like platforms) is to provide a shell-script that determines which features are not available. For instance (in outline):

```
echo "main(){signed char c; c=0;}" > test.c
if cc test.c
then echo " Signed char ok"
else
    CFLAGS="$CFLAGS -DNO_SC"
    echo "Signed char not accepted; using $CFLAGS"
fi
```

and then

```
cc $CFLAGS -o enquire enquire.c
```

The clever part is that the shell script is worked in along with the C program: most people have seen examples of programs that compile on both Fortran and Pascal compilers; *enquire.c* is both a C program and a shell-script. The trick is that a # at the beginning of a line is a comment symbol to the standard Unix Bourne shell, so we can use at the beginning of a file something like:

```
#ifdef NOTDEFINED
    echo Any shell code may go here
    exit 0
#endif
```

and the C compiler will ignore it; the shell will ignore the first line, and exit at the "`exit 0`" and so ignore the rest, which is just the normal C program.

So the shell code to discover the features and compile the program goes at the beginning of the file and then a call to the shell:

```
sh enquire.c
```

ensures that *enquire* configures and compiles itself.

Typical Errors

While it is difficult to anticipate bugs in compilers, and it has not been a policy to program around compiler bugs in *enquire*, where a certain bug has appeared in more than one compiler, it has been treated as a feature, and we have tried to allow for it (partly to reduce the number of reports coming in from users).

An example of such a bug is not being allowed to `#undef` something that hasn't been defined. Another problem is that not all compilers support the predefined `__FILE__` symbol, so that the line:

```
#include __FILE__
```

fails to compile. To fix this, we replaced it with:

```
#ifdef __FILE__
#include __FILE__
#else
#include "enquire.c"
#endif
```

but it then turned out that a couple of compilers couldn't cope with the line

```
#ifdef __FILE__
```

apparently (incorrectly) expanding `__FILE__` before doing the `#ifdef`.

We then bracketed the code with a symbol `BAD_CPP` which the user could set when compiling:

```
#ifdef BAD_CPP
#include "enquire.c"
#else
#ifdef __FILE__
#include __FILE__
#else
#include "enquire.c"
#endif
```

but there is still one compiler we know of that even then can't cope, since it incorrectly looks at skipped over sections and still doesn't accept "`#include __FILE__`" (there is nothing we can do about that).

As mentioned earlier, some parts of the code are bracketed with an `ifdef` on the symbol `__STDC__`; unfortunately, some bad compilers define this symbol even if they don't fully accept ANSI C (a case of wishful thinking, perhaps). Some of these compilers at least define the symbol as 0, instead of the required 1, which is testable for, but others define it as 1.

Conclusions

Enquire is a program designed to aid portability, and in a sense is a case of extreme portability: whereas most programs can depend on a set of well chosen preprocessor symbols in order to be portable, *enquire* has to do it alone, and it must do it on an unusually wide range of platforms.

C is often proclaimed as a portable language, but as this article shows, the truth is that it can only offer *post hoc* portability: you can't be sure beforehand that your program is going to compile and run on a given platform, but you have to try it first.

Enquire started out as a modest program with a specific purpose, but expanded in the face of the demands of ever more platforms, compiler bugs, and user requirements, and has ended up as a useful program, for porting as its main aim, for testing compilers, for evaluating compilers, and for generating compilers for new platforms.

As you may suspect from the exposition, *enquire* is actually much more complicated than we have presented here: we have simplified the exposition to indicate the main features, and prevent the reader from being flooded with excessive detail. The program is available from several sources: from several *ftp* servers, for instance `mcsun.eu.net` which is the repository for the latest version at any time, in directory `misc`. It is similarly available from a mail-server should you have no *ftp* facilities: send the two-line mail message "`request misc`", "`topic index`" to `info-server@hp4nl.nluug.nl` for a list of available files. One of them will be `enquire`*XX*`.c`, where *XX* is the version number. The message "`request misc`", "`topic enquire`*XX*`.c`" will send this file.

Finally, *enquire* is used by the GNU project for generating the `float.h` header file for their gcc C compiler, and can be found in the gcc distribution.

References

[Aho] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, Reading, Mass., 1986.

[ANSI] *Draft Proposed American National Standard for Information Systems — Programming Language C, ANSI Standard X3J11/88-158*, American National Standards Institute, New York, 1990.

[Cody] W.J. Cody, W. Waite, *Software Manual for the Elementary Functions,* Prentice-Hall, 1980.

[Geurts] Leo Geurts, Lambert Meertens, Steven Pemberton, *The ABC Programmer's Handbook*, Prentice Hall, 1990.

[Goldberg] David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, Vol 23, No 1, March 1991, pp. 5-48.

[IEEE] *IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, IEEE. Reprinted in SIGPLAN Notices, Vol. 22, No. 2, pp. 9-25.

[Kernighan] B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, 1978.

Appendix 1: Example Shell Output

The following is an example of the output when compiling *enquire* with “sh enquire.c” on a Sun Sparc processor:

```
Testing for needed CFLAGS ...
 Signed char not accepted; using -DNO_SC
 Unsigned char ok
 Unsigned short and long ok
 Void ok
Compiling enquire.c ...
 cc -DNO_SC enquire.c -o enquire
Producing enquire.out limits.h and float.h ...
 enquire > enquire.out
 enquire -l > limits.h
 enquire -f > float.h
Verifying the contents of limits.h and float.h ...
 cc -DVERIFY -DNO_SC enquire.c -o enquirev
 enquirev -fl > verify.out
 *** Some problems: see verify.out
Done
```

Appendix 2: Example Output

The following is an example of the sort of output you get from *enquire*. It is from `cc` on a Sun Sparc processor.

```
Produced by enquire version 4.4(pre), CWI, Amsterdam
Compiled without signed char
Compiler does not claim to be ANSI C

Compiler names are at least 64 chars long
Preprocessor names are at least 64 long

SIZES
char = 8 bits, signed
short=16 int=32 long=32 float=32 double=64 bits
char*=32 bits
int* =32 bits
func*=32 bits
Type size_t is signed int/long
```

```
ALIGNMENTS
char=1 short=2 int=4 long=4
float=4 double=8
char*=4 int*=4 func*=4

CHARACTER ORDER
short: AB
int: ABCD
long: ABCD

PROPERTIES OF POINTERS
Char and int pointer formats seem identical
Char and function pointer formats seem identical
Strings are not shared
Type ptrdiff_t is signed int/long

PROPERTIES OF INTEGRAL TYPES
Overflow of a short does not generate a trap
Maximum short = 32767 (= 2**15-1)
Minimum short = -32768
Overflow of an int does not generate a trap
Maximum int = 2147483647 (= 2**31-1)
Minimum int = -2147483648
Overflow of a long does not generate a trap
Maximum long = 2147483647 (= 2**31-1)
Minimum long = -2147483648
Maximum unsigned short = 65535
Maximum unsigned int = 4294967295
Maximum unsigned long = 4294967295

PROMOTIONS
unsigned short promotes to unsigned int/long
long+unsigned gives unsigned int/long

PROPERTIES OF FLOAT
Base = 2
Significant base digits = 24 (= at least 6 decimal digits)
Arithmetic rounds towards nearest
 Tie breaking rounds to even
Smallest x such that 1.0-base**x != 1.0 = -24
Smallest x such that 1.0-x != 1.0 = 2.98023259e-08
Smallest x such that 1.0+base**x != 1.0 = -23
Smallest x such that 1.0+x != 1.0 = 5.96046519e-08
(Above number + 1.0) - 1.0 = 1.19209290e-07
Number of bits used for exponent = 8
Minimum normalised exponent = -126
Minimum normalised positive number = 1.17549435e-38
The smallest numbers are not kept normalised
Smallest unnormalised positive number = 1.40129846e-45
```

```
There is an 'infinite' value
Maximum exponent = 128
Maximum number = 3.40282347e+38
Arithmetic uses a hidden bit
It looks like single length IEEE format

PROPERTIES OF DOUBLE
Base = 2
Significant base digits = 53 (= at least 15 decimal digits)
Arithmetic rounds towards nearest
 Tie breaking rounds to even
Smallest x such that 1.0-base**x != 1.0 = -53
Smallest x such that 1.0-x != 1.0 = 5.5511151231257839e-17
Smallest x such that 1.0+base**x != 1.0 = -52
Smallest x such that 1.0+x != 1.0 = 1.1102230246251568e-16
(Above number + 1.0) - 1.0 = 2.2204460492503131e-16
Number of bits used for exponent = 11
Minimum normalised exponent = -1022
Minimum normalised positive number = 2.2250738585072014e-308
The smallest numbers are not kept normalised
Smallest unnormalised positive number = 4.9406564584124654e-324
There is an 'infinite' value
Maximum exponent = 1024
Maximum number = 1.7976931348623157e+308
Arithmetic uses a hidden bit
It looks like double length IEEE format

Float expressions are evaluated in double precision
Double expressions are evaluated in double precision
Memory mallocatable ~= 14376 Kbytes
```

Appendix 3: Sample limits.h File

```
/* limits.h */
/* Produced by enquire version 4.4(pre), CWI, Amsterdam */

 /* Number of bits in a storage unit */
#define CHAR_BIT 8
 /* Maximum char */
#define CHAR_MAX 127
 /* Minimum char */
#define CHAR_MIN (-128)
 /* Maximum signed char */
#define SCHAR_MAX 127
 /* Minimum signed char */
#define SCHAR_MIN (-128)
 /* Maximum unsigned char (minimum is always 0) */
#define UCHAR_MAX 255
 /* Maximum short */
```

```
#define SHRT_MAX 32767
 /* Minimum short */
#define SHRT_MIN (-32768)
 /* Maximum int */
#define INT_MAX 2147483647
 /* Minimum int */
#define INT_MIN (-2147483647-1)
 /* Maximum long */
#define LONG_MAX 2147483647L
 /* Minimum long */
#define LONG_MIN (-2147483647L-1L)
 /* Maximum unsigned short (minimum is always 0) */
#define USHRT_MAX 65535
 /* Maximum unsigned int (minimum is always 0) */
#define UINT_MAX 4294967295
 /* Maximum unsigned long (minimum is always 0) */
#define ULONG_MAX 4294967295L
```

Appendix 4: Sample float.h File

```
/* float.h */
/* Produced by enquire version 4.4(pre), CWI, Amsterdam */

 /* Radix of exponent representation */
#define FLT_RADIX 2
 /* Number of base-FLT_RADIX digits in the significand of a
  float */
#define FLT_MANT_DIG 24
 /* Number of decimal digits of precision in a float */
#define FLT_DIG 6
 /* Addition rounds to 0: zero, 1: nearest, 2: +inf, 3: -inf, -
  1: unknown */
#define FLT_ROUNDS 1
 /* Difference between 1.0 and the minimum float greater than
  1.0 */
#define FLT_EPSILON ((float)1.19209290e-07)
 /* Minimum int x such that FLT_RADIX**(x-1) is a normalised
  float */
#define FLT_MIN_EXP (-125)
 /* Minimum normalised float */
#define FLT_MIN ((float)1.17549435e-38)
 /* Minimum int x such that 10**x is a normalised float */
#define FLT_MIN_10_EXP (-37)
 /* Maximum int x such that FLT_RADIX**(x-1) is a representable
  float */
#define FLT_MAX_EXP 128
 /* Maximum float */
#define FLT_MAX ((float)3.40282347e+38)
 /* Maximum int x such that 10**x is a representable float */
```

```
#define FLT_MAX_10_EXP 38

 /* Number of base-FLT_RADIX digits in the significand of a dou-
   ble */
#define DBL_MANT_DIG 53
 /* Number of decimal digits of precision in a double */
#define DBL_DIG 15
 /* Difference between 1.0 and the minimum double greater than
   1.0 */
#define DBL_EPSILON 2.2204460492503131e-16
 /* Minimum int x such that FLT_RADIX**(x-1) is a normalised
   double */
#define DBL_MIN_EXP (-1021)
 /* Minimum normalised double */
#define DBL_MIN 2.2250738585072014e-308
 /* Minimum int x such that 10**x is a normalised double */
#define DBL_MIN_10_EXP (-307)
 /* Maximum int x such that FLT_RADIX**(x-1) is a representable
   double */
#define DBL_MAX_EXP 1024
 /* Maximum double */
#define DBL_MAX 1.7976931348623157e+308
 /* Maximum int x such that 10**x is a representable double */
#define DBL_MAX_10_EXP 308
```

Appendix 5: Example Checks

The C standard specifies that all floating-point arithmetic must round in the same way:

```
/* *** WARNING: double arithmetic rounds differently (1) from
   float */
```

Printf fails to print the minimum float correctly:

```
#define FLT_MIN ((float)0.00000001e-38)
/* *** WARNING: Possibly bad output from printf above */
/* expected value around 2.93873588e-39, bit pattern:
      00000000 10000000 00000000 00000000 */
/* sscanf gave 1.15792088e+31, bit pattern:
      01110100 00010010 00100110 01110001 */
/* difference= -1.15792088e+31 */
```

Printf outputs the correct value, scanf reads it back wrongly:

```
#define DBL_EPSILON 2.77555756156289140e-17
/* *** WARNING: Possibly bad output from printf above */
/* expected value around 2.77555756156289140e-17, bit pattern:
      00100101 00000000 00000000 00000000 00000000 00000000
     00000000 00000000 */
/* sscanf gave 2.77555756156289120e-17, bit pattern:
```

```
        00100100 11111111 11111111 11111111 11111111 11111111
        11111111 11111101 */
/* difference= 1.54074395550978870e-33 */
```

Scanf fails completely to read back the maximum double.

```
#define DBL_MAX 1.7976931348623455e+308
/* *** WARNING: sscanf returned an unusable number */
/* scanning: 1.7976931348623455e+308 with format: %le */
```

Scanf reads the value back incorrectly, but printf prints *that* in its turn incorrectly in the warning message, so that it appears to be correct (two wrongs *can* make a right after all!):

```
#define DBL_EPSILON 2.77555756156289140e-17
/* *** WARNING: Possibly bad output from printf above */
/* expected value around 2.77555756156289140e-17, bit pattern:
        00000000 00100101 00000000 00000000 00000000 0000000
        00000000 00000000 */
/* sscanf gave 2.77555756156289140e-17, bit pattern:
        00000000 00100101 00000000 00000000 00000000 00000000
        00000001 00000000 */
/* difference= -7.70371977754894340e-34 */
```

The following checks come from the verify phase.

The cast doesn't convert the value to a float, but leaves it as a double (with too many digits of precision):

```
#define FLT_EPSILON ((float)1.19209290e-07)
/* *** WARNING: the cast didn't work */
```

Printf prints the wrong value:

```
#define FLT_MIN ((float)0.00000001e-38)
/* *** Verify failed for above #define!
        Compiler has 0.00000000e+00 for value */
```

Checking an existing limits.h file, which contains the wrong value:

```
#define CHAR_MIN (-128)
/* *** Verify failed for above #define!
        Compiler has -127 for value */
```

Similarly:

```
#define SHRT_MIN (-32767-1)
/* *** Verify failed for above #define!
        Compiler has -32767 for value */
```

The compiler fails to read the minimum float back correctly:

```
#define FLT_MIN ((float)1.17549435e-38)
/* *** Verify failed for above #define!
        Compiler has 0.00000000e+00 for value */
```