Choice of Language

MJ Rutter
mjr19@cam.ac.uk

Lent 2005

Disclaimer

I am always accused of prejudice and bias. Indeed, I am very proud of my prejudices and biases. I am usually accused of directing such things against C (so far as computer languages are concerned…).

Just for the record, I have written much more C than Fortran in the past five years, I know that there are several people in TCM whose knowledge of F90 is better than mine, and I am unaware of anyone in TCM whose knowledge of C is better than mine. This is probably how it should be.

Why?

If I were to build a bridge, I would seriously consider what material to build it out of. Also, the design of the bridge would be heavily influenced by the choice of material, and vice versa.

…to choose a language for some software, you need knowledge of several languages, and to design a piece of software successfully, you need a fairly detailed knowledge of the chosen implementation language – even if you never personally write a single line of that software.

— Stroustrup

Precision

Are you inferring that the referee was not uninterested?

Are you implying that the referee was not disinterested?

Both perfectly grammatical sentances. The first means ‘do you believe that the ref. was paying attention?’, the second ‘are you saying that the ref. was biased?’ Humans love confusing disinterested and uninterested, infer and imply.

Humans can sort out this mess from context, computers will follow precisely the directions they are given, provided that they are grammatically correct.

Evolution

Benedic, Domine, nos & dona tua. . .

Benedic, Domine, nobis & donis tuis. . .

Identical meanings, but from different centuries. Even Latin evolves.

Again, humans will work out what is going on from context, whereas computers may need more help.

Computer languages also evolve, with features becoming deprecated, and then dropped completely.

What is wanted?

Different programs are different, and therefore may suit different languages. It is not evident that a one-size-fits-all approach can deal with Doom, Word, Castep and `echo`.

If the program is at all large, a language supporting functions or subroutines will be required. If the program is under a hundred lines, this might not be an issue.

If the program requires a GUI, some form of windowing toolkit must be available. If not, not.

Execution speed may be an issue, and so might code maintance, or portability, or future-proofing. However, if the program is of the ‘write-once, run-once’ variety, maybe not.

The Compiled Languages

Fortran, C and C++ are the major compiled languages of relevance to TCM. Large projects should generally use compiled languages: there are technical reasons why they should run faster than the other sorts, and co-incidental reasons why they tend to be more stable.

First a brief discussion of these three.

Fortran

Developed by IBM for FORmula TRANslation in 1954 as a more pleasant and portable alternative to writing in assembler. It was the first ‘high-level’ language.

Major versions include Fortran IV (1961), Fortran 66, Fortran 77, Fortran 90, Fortran 95 and Fortran 2003. Fortran 66 was the first version to have an international standard.

It was not a general-purpose language, but one designed to excel at numerical work. It is now more general-purpose, but still not ideal for much outside numerics.

Throughout this document, ‘Fortran’ means Fortran 90 and later versions unless otherwise stated.

C and C++

C was developed by Dennis Ritchie in AT&T and appeared in its K&R form in 1978. It was designed to be a general-purpose language with ‘economy of expression’ and an ‘absence of restrictions’ (quotes from the preface of K&R’s ‘The C Programming Language’).

It was first standardised by ANSI in 1989. Since then, POSIX has had a go (1994), and ANSI/ISO again (1999).

C++ is firmly based on C, and was developed by Bjarne Stroustrup (also at AT&T). It appeared in 1985, and, anticipating C89, is (approximately) a superset of C89. It was standardised in by ANSI/ISO in 1998 and 2003.

Throughout this document, ‘C’ means C89 and later versions unless otherwise stated.

BASIC: Standardisation is needed

Those introduced to home computers from the 1980s will surely know BASIC. On paper there is little wrong with BASIC, save only that the common core to the language is tiny, and that all vendors of interpreters and compilers have extended the language in different fashions.

Porting code between GWBASIC (free with MS DOS 3.3) and QBASIC (free with MS DOS 5.0) was tiresome, and moving to TurboBASIC (also for MS DOS, but produced by Borland) was likely to involve another rewrite. Similarly moving from the Sinclair ZX Spectrum to the Sinclair QL would require a rewrite.

Standardise

No-one has time for such games. One wants to write code no more than once, and then be able to run it on any computer.

If there is a clear, well-defined, accepted standard, one mere follows it, ensures that someone claims to have a compiler which follows the same standard, and, if something goes wrong, one can work out whose fault it all is.

If a standard is absent, or so ill-defined that the best one can do is to discover what works by experimentation, there is nothing one can do if the behaviour suddenly changes with a new compiler release.

A standard should be cross-platform and slow to evolve. It should be run by a committee, not a single vendor. (Committees prohibit action, which, in this case, is good.)

Ambiguous and Unhelpful Standards

The Fortran standard says of `sqrt(x)`:

‘Unless `x` is complex, its value shall be greater than or equal to zero.’

In other words, if you call `sqrt` with a real, negative argument, you have broken the standard, and subsequent behaviour is undefined.

Fortran traditionally stops with a floating point exception at this point, but it does not need to do so.

C, Standards and Libraries

Because UNIX is written in C, many C programmers fail to notice the boundaries between those functions defined by C89, and those extra functions supplied by the OS which are not part of C89 and therefore might not be available on all systems with C89 compilers.

This is most apparant in the area of file handling.

C89: `fopen`, `freopen`, `fflush`, `fclose`, `freopen`, `frename`, `fread`, `fwrite`, `fgetc`, `fputc`, `fgets`, `fgets`, `fseek`, `ftell`, `fgetpos`, `fsetpos`, `feof`, `ferror`.

Not C89: `fstat`, `fchown`, `fchmod`, `fcntl`, `flock`.

Only those functions starting with an ‘f’ are listed. There are many others.

More standard trouble

Assuming one's C system follows POSIX.1 (IEEE STD 1003.1 of 1990) then one has the `fstat` function. It is defined as returning a `stat` structure as defined in `sys/stat.h`.

What is in a `stat` structure? Well, that is not wholly standardised.

This is why building C/C++ programs on UNIX systems often requires running hideous configure scripts: the scripts need to work out which standards one's environment is supporting, and what options within those standards have been taken.

Because the Fortran standard permits less interaction wih the OS, it is much tighter, and decent Fortran usually compiles unmodified on any half-sane Fortran system.

No to Temptation

Many languages have vendor-specific extensions. These should be avoided if one wishes one's code to remain portable. Common problems included:

F90: `etime`, `dtime`, `loc`
C89: //, `inline`, `long long`, incrementing a `void*`
C++: `g++`'s used of `std:`, `<?`, `>?`

and, of course, all occurrances of `asm`.

There are other traps, such as, in Fortran, assuming unit 6 is stdout, or that `real (8)` is double precision.

Avoid such issues by compiling your code on as many different compilers as possible. This is especially important for C++.

Polylingualism

Mixed language programming is generally a bad idea, because the interface between the languages is rarely standardised. Mixing C and C++ is fine, as is F77 and F9x (mostly). Most other combinations are risky.

If mixed language programming does not make your head hurt, consider the issues of IEEE rounding mode, and IEEE exception handling.

This is a pity, because at some level mixed language programming has much to be said in its favour.

Numeric fun

Fortran and C both take the view that the language should, to some extent, be fitted to the computer. Both have single and double precision floating point types, but the only guarantee is that double has at least as much precision as single. C offers three types of integers: `short`, `int` and `long`. It says little about how long any of those might be. Thus

```
long a=1234567890;
printf("%ld\n",a);
```

may do different things depending on machine and/or compiler flags.

Standard Types

C99

`char`	8 bits
`short` and `int`	16 bits
`long`	32 bits
`long long`	64 bits (no such type in C89 or C++)
`float`	6 digits accuracy 10^{-37} to 10^{37}
`double`	10 digits accuracy 10^{-37} to 10^{37}
`long double`	10 digits accuracy 10^{-37} to 10^{37}

The above are minimum requirements for the types. The types need not be distinct.

Fortran is yet worse: no minima are specified.

C and GUIs

The C standard does not include any form of GUI, but various libraries, or toolkits, exist. Should one try:

Motif, X11, GLut, Xaw, Aqua, Gtk, glib, Qt, VC, . . .

Some of the above are free, some are commercial with free ‘clones’, some are only commercial. Some exist in multiple incompatible versions. Some exist for multiple platforms. The more one uses in a single project the less portable it will be.

If one simply wants a text-based full-screen interface, then some flavour of curses might suffice.

C vs C++

C++ is often thought of as a superset of C. It has never been precisely thus. For instance, ‘`sizeof('a')`’ will evaluate differently on a C++ and C89 compiler.

C99 introduces several featuers that C++ has never had, and is unlikely ever to gain. The complex type above is one example. C++ already has a complex template class, and is unlikely to want to gain the above intrinsic type.

C, C++ and Numerics

Neither C nor C++ were (sic) designed primarily with numeric computation in mind.

– Bjarne Stroustrup, *C++ Programming Language*, 3rd Ed.

Feature list

	F77	F90	C89	C++
Dynamic memory allocation	✗	✔	✔	✔
Complex numbers	✔	✔	C99	[1]
Array syntax[2]	✗	✔	✗	1D
Function prototypes	✗	✔	✔	✔
Pure functions	✗	F95	✗	✗
User defined types	✗	✔	✔	✔
Operator/function overloading	✗	✔	✗	✔
Public/private data/functions	✗	✔	✗	✔
Optional/default arguments	✗	✔	[3]	[3,4]
Real-time interaction	✗	✗	✔	✔
Make any UNIX library call	✗	✗	✔	✔

[1] done in two different ways, neither an intrinsic type

[2] e.g. a=a*b for element-wise operations on arrays.

[3] Variable number of arguments supported.

[4] Default trailing arguments supported. Not as flexible as F90.

Complex ‘support’

```
                                  #include<math.h>
                                  #include<complex.h>
real(kind(1d0))    :: a,b,c       double a,b,c;
complex(kind(1d0)) :: x,y,z       double complex x,y,z;

a= 0.22; b=2*sqrt(a); c=cos(b)    a= 0.22; b=2*sqrt(a); c=cos(b);
x=-0.22; y=2*sqrt(x); z=cos(y)    x=-0.22; y=2*csqrt(x); z=ccos(y);
```

C99 (right) is not quite as seamless as Fortran (left): one needs to change function names.

No transliteration

```
real(kind(1d0)), allocatable :: x(:)      double *x;

allocate(x(1000))                         x=malloc(1000*sizeof(double));
```

Not at all identical. For a start, F90's `allocate` will, by default, halt the program is the allocation fails, whereas C's `malloc` simply returns a null pointer. Therefore the C code needs an extra

```
if (x==NULL) {fprintf(stderr,"Malloc failed\n"); exit(1);}
```

An obvious and minor point. In both cases `x` can be freed, but only in C can it then be aliased with another double object.

In F90, use `allocate(x(1000),stat=i)` if you wish failed allocations to continue with an error code returned.
Use `allocate(x(0:999),stat=i)` if you want C's indexing.

Faux amis

Fortran has pointers, so does C/C++. Are they identical concepts? No. Fortran pointers are much more strongly typed, and can not be cast or aliased in the manner of `void*` and `char*` in C/C++.

Fortran both C appear to have some form of string support. Are they identical concepts? No. C strings are terminated by ASCII code zero (which therefore cannot occur within a string), whereas Fortran strings have a hidden length associated with them.

Loops

```fortran
real(kind(1d0)) :: a(:), b(:), c(:)
integer i,n
...
do i=1,n
  a(i)=b(i)+c(i)
enddo
```

```c
double *a,*b,*c;
integer i,n;
...
for (i=0;i<n;i++){
  a[i]=b[i]+c[i];
}
```

Again, very different.

Fortran’s do loops do not permit the index variable to be modified in the loop, and Fortran’s arrays are not permitted to overlap. C is enjoying its ‘absence of restrictions’ here.

Why loops matter

In Fortran, if the loop body contains no exit or goto statement, then the total loop count is known at run-time on entry.

In C, the compiler additionally has to check that the counter is not modified (either directly or via something which can legitimately point to it). This can be hard to prove in more complicated loops.

Run-time choices about loop execution are very important on SMP (and multi-core) machines. If the loop count is several million, it is worth starting threads on other cores. If it is three, it is not.

If single-core machines are the limit of your horizons, you don’t care.

Why independence matters

If independence cannot be proven, the data access pattern above must be:

```
read b(1), c(1)
calculate
write a(1)
read b(2), c(2)
calculate
write a(2)
etc.
```

If it can be proven this access pattern can be changed. An obvious, and minor, improvement is

```
read b(1), c(1)
calculate
read b(2), c(2)
write a(1), calculate
read b(3), c(3)
```

Oh no!

```
double *x,*y;

for(i=0;i<n;i++) x=pow(y,3.5);
```

Assuming `x` and `y` are proveably independent, an operation in which there are no dependencies between loop iterations?

No.

Most maths functions in C89 are not pure functions, for they modify the global variable `errno`.

`errno`

When a C maths function fails, the program often continues, but `errno` is modified to reflect the error, being set equal to `ERANGE` for an overflow or `EDOM` (domain error) for the fractional power of a negative number. If the function succeeds, `errno` is unchanged.

So `errno` must be set to reflect the error code from the last element in the array which causes an error. This massively complicates performing such an operation across multiple processors.

C99 does things somewhat differently, and somewhat incompatibly. In C99, the maths functions are no longer required to set `errno`, so any C89 code which reasonably assumes that they do could be in for a surprise when it meets a C99 compiler. That C99 introduced this incompatible change, with no prior warning, in the area of numerics, shows how much the C standard committee cares for numerics.

F95, Pure Functions and Array Tricks

```
where(a>0)    ! a is an array of any rank
  a=1./a
elsewhere
  a=1000
  b=0         ! b has the same shape as a
endwhere

forall(i=1:n, victim(i)%college="FITZ")
  victim(i)%mark=fudge(victim(i)%mark)
end forall
```

In Fortran, statements in a `where` construct are executed sequentially. In a `forall` construct, they may be executed in any order, and may be overlapped. Hence `forall` constructs can call only pure functions, which includes the obvious intrinsic functions.

Device independent assembler

…one of the less flattering titles given to C.

If you are faced with a single CPU which loks like a typical 1970’s microprocessor (which, for these purposes, includes a Pentium III), then C and Fortran will perform almost equally well.

If you are faced with a CPU with sophisticated prefetching abilities (e.g. a vector computer), one can expect Fortran to outperform C by a factor of between two and twenty on code like the above. This was certainly the case on the Cray Y/MP and Hitachi S3600, with the ratio being about $20\times$, and a somewhat smaller ratio on the Hitachi SR2201.

Learn your language

Last Term I was rude mostly about common misunderstandings in Fortran. So this time it is C.

Again one should say that if one is programming professionally, i.e. programming as part of one's profession and passing code one writes to others, one has a duty to know what one is doing!

C's string handling is much better than Fortran's. However, do you really understand it? The lines which are not commented are all perfectly valid and well-defined.

```
{
  char a[4]="Hello"; /* Compile-time error */
  char b[5]="Hello";
  char c[6]="Hello";
  char d[]="Hello";
  char *e="Hello";

  printf("%s\n",b);  /* Undefined run-time behaviour */
  printf("%s\n",c);
  printf("%s\n",d);
  printf("%s\n",e);

  c[0]=’Y’;
  d[0]=’Y’;
  e[0]=’Y’;          /* Undefined run-time behaviour */

  c="Bye";           /* Should give compile-time error */
  e="Bye";
}
```

Java

Java is clearly designed for writing cross-platform GUIs. It can be interpreted, ‘compiled’ to a platform-independent Java byte-code, or (rarely) compiled in the usual fashion.

Java assumes a particular ‘virtual machine’: it is up to the real host computer to provide a ‘JVM’ to simulate precisely the characteristics of this virtual computer. This leads to complete portability: any Java program run anywhere always sees precisely this virtual machine. In theory.

Unlike C and Fortran, a windowing toolkit is part of the language.

Although the syntax is C-like, it is much more restrictive, so it is harder to shoot oneself in the foot.

JVMs are extremely common: they are embedded in almost every GUI WWW browser, and also available in stand-alone forms.

Java and Maths

Java's philosophy appears to be 'carry on regardless'. The required results of converting certain double precision numbers to various integers are as below:

	`int`	`short` or `byte`
NaN	0	0
1e300	$2^{31}-1$	-1
$-$1e300	-2^{31}	0

Moral: check the floating point number first.

(Unlike C, Java defines `short` to be 16 bits, `int` to be 32 bits, and `long` to be 64 bits.
When converting to `short` or `byte`, Java converts first to `int`, then truncates.
C defines the above conversions to be implimentation dependent.)

More Java and Maths

‘Floating-point operators produce no exceptions. An operation that over-flows produces a signed infinity, [. . .] an operation that has no mathematically definite result produces NaN. All numeric operations with NaN as an operand produce NaN as a result.’

So your code will continue to run, gently accumlating NaNs, unless you explicitly check for this problem. Conversion to integer types silently loses NaNs as above. Any comparison involving a NaN is false, including `x==x`.

The only numeric runtime exceptions are integer division by zero, and integer remainder on division by zero.

Java has no complex type.

Quote from ‘The Java Language Specification, Second Edition’ (2000).

Perl

Perl is an interpreted language which will run anywhere that one can successfully compile the interpreter (written in C). The name allegedly abbreviates ‘Practical Extraction and Report Language’: it excels at string manipulations, particularly regular expressions.

It enjoys a C-like compactness, but does not have a standard. There is just one source for perl, so all the interpreters are the same, except for version number, and one prays that backwards compatibility is maintained by Larry Wall. You don’t want to use versions earlier than 5.0 though.

Perl is extremely useful for manipulating text-based input and output files.

Very common on UNIX platforms, and available for Windows too.

More Perl

Some interpreted languages, such as perl and python, have as much structure as the older compiled languages. Both support functions and libraries (modules). Perl was first released in 1987, and python in 1991, so both are a reasonable age, although perl is arguably more stable.

Two features supported by these scripting languages (and also `awk`), but not by traditional compiled languages, are the use of regular expressions as intrinsic functions, and the use of hashed arrays. The latter is an array indexed by (unique) strings, rather than consecutive integers. Yes, an access overhead is implied.

```
#!/usr/local/shared/bin/perl    #!/usr/local/shared/bin/python
                                atw={}
$atw{"H"}=1; $atw{"He"}=4       atw["H"]=1; atw["He"]=4
$atw{"Li"}=7; $atw{"Be"}=8      atw["Li"]=7; atw["Be"]=8

print $atw{"Li"}."\n"           print atw["Li"]
```

Tcl/Tk

Another scripting language, but one rapidly losing popularity. Unlike perl, windowing features are inbuilt. In can also interface closely with C, with mixed Tcl/Tk - C programs possible. The XCrysDen crystal viewer installed in TCM is one example of this.

Pure Tcl/Tk scripts include the `tkdiff` and `tkinfo` commands in TCM.

The Tk toolkit also interfaces with python.

Scripting languages are not the poor relations of compiled languages, at least in respect of features and structure (python claims to be OO), that they used to be. They are when it comes to fast numerics though.

Bourne Shell

The original UNIX batch scripting language. Always, always there (in UNIX), and remarkably flexible. The Bourne shell (named after Stephen Bourne) itself does little, except provide variables and control structures. However, it can call any UNIX command, and swallow text output and error codes from them.

Excellent / essential for UNIX job control, such as scripts submitted to queueing systems etc. Dozens of Bourne shell scripts are run by every UNIX system as it boots, and it is the only shell that a UNIX system is required to provide.

The Bourne shell permits function definitions, but is executed in a single pass: function definitions must occur before their first use. That variables are untyped and function calls not even checked for number of arguments should put you off writing anything substantial in this language.

The bash shell is an extension to the Bourne shell. It is slightly less common, and, if one must be lazy and use its extra features, do ensure that the resulting script starts

```
#! /bin/bash
```

not `#!/bin/sh` (which will mark you out as an ignorant Linux abuser).

Decisions, Decisions

One should consider carefully what language features are necessary for one’s project, and use the result to suggest (and eliminate) possible languages. For instance:

Need speed? Eliminate wholly interpreted languages (UNIX shells), and frown on semi-interpreted languages (perl, python, Java).

Need a GUI? No to Fortran, yes to Java.

Need aggressive optimisation of numeric code for exotic architecture? Fortran.

Code longer than 200 lines? You need functions/subroutines. Therefore no to the C shells.

Code longer than 2000 lines? You need argument checking on function calls: no to the Bourne shell.

Not the Last Word

Just because you have chosen the correct language, it is not necessarily the case that the resulting style is correct. Fortran does not force one to declare variables or to use subroutines, although one would be foolish not to do so. C++ does not force one to use features not present in C, but, again, feeding C to a C++ compiler is to miss the point.

All languages permit comments. None requires them.

Most languages permit some form of global variable, but these should be used sparingly.

(Almost) all languages permit indentation. Python is one of the few which requires it.

In many cases this precludes automatic translation: yes, one can write C++ in the style of F77, and one can imagine something which could automatically translate F77 to this form of C++, but the result would have none of the advantages of C++, and all the disadvantages of both languages.

Never the Last Word

For long-lasting projects, choice of language is an issue which may need revisiting. Afterall, F90 was not an option before 1990 (or, arguably for a few years thereafter, until compilers were common, stable and decently optimising). However, today (2005), both F90 and F95 are reasonable choices in some circumstances.

Unfortunately, if one’s choice of language changes, one may need to rewrite everything. Fortunately you kept your code need, compact, and easy to understand. Also fortunately this is unlikely to be necessary more than once per decade.

If the migration is more minor, e.g. F90 to F95, of course a re-write is unlikely to be required.

KISS

Notwithstanding the above remarks about feeding C to C++ compilers, gargantuan structures of intricate baroque beauty may be appropriate for certain forms of civil building (preferably not maintained at my expense), but are unlikely to be appropriate models for coding. The simpler one’s style, the more likely the compiler is to optimise it well, and the less likely the compiler is to make mistakes.

Yes, define datatypes and classes where such things are clearly fundamental to the code. But not simply to demonstrate that you can turn a ten line problem into a two hundred line obfuscation.