SGP/4.2

Starlink Project
Starlink General Paper 4.2

Anne Charles and Jo Murray

20 June 1994

Starlink C Programming Standard

Contents

1 Introduction

C is a general-purpose programming language designed in about 1972 for use on the Unix operating system. The decisions to make the language portable and use it to rewrite Unix have caused it to become very popular, and many operating systems, compilers and general purpose applications have been coded using C. *The C programming language* by Brian Kernighan and Dennis Ritchie (second edition) is a good introductory textbook.

C combines the advantages of high-level languages such as Fortran in the realm of control structures and portability with 'machine-level' features such as pointers, increment operators, bitfields and bitwise operators. The language is especially suited to system programming because its data types and operators are closely related to the operations of most computers. However Starlink does not recommend the use of C for applications programming. There are several reasons for this, not least that Fortran provides better facilities for the manipulation of multi-dimensional data arrays and the dynamic declaration of array size. Additionally there is the problem of maintenance. It is very easy (and indeed usual) to write C code which is cryptic; and there is an ever-present temptation to write highly-condensed code. A common failing of C programmers is to write code and then collapse it into the shortest possible form assuming that anyone reading the code will be an expert. Whilst the experienced C programmer will recognise the patterns of code which result, it is recommended by Starlink that such structures are well commented for the benefit of the less experienced.

The language has no facilities to perform input and output, and lacks Fortran's large set of intrinsic functions. However all implementations are accompanied by a (more or less) standard library of functions to accomplish I/O and other tasks. These standard functions are made available to a program by specifying the modules which contain the required set of functions via an include file. For example the Standard I/O routines are contained in the file `stdio.h`, which must be explicitly `included` in the program source if the functions contained therein are to be used.

The suggestions which follow are made with maintainability, portability and efficiency in mind. Several of these recommendations are made because certain code constructs will have different effects on different C implementations. In some cases the ambiguity is resolved by the ANSI standard. However it may be some time before all C implementations meet the standard, and in any case it is preferable to avoid code which suggests more than one possibility to the human reader.

Starlink has adopted *The Elements of C Programming Style* by Jay Ranade and Alan Nash as its principal C programming standard. This book is *essential reading* for all contributors of Starlink programs written in C. The present document does not reiterate what is in the book, but contains some additions and clarifications. Where the advice in Chapter 18, Coding for Non-C Programmers, diverges from that in the main body of the book, Starlink recommends that you follow Chapter 18.

2 Coding standard

The advice which follows is grouped into five sections concerning Starlink naming conventions, programming advice, program design, machine independence, and preprocessor advice respectively. The reader is referred to SGP/16 for general advice on Starlink coding standards.

2.1 Starlink naming conventions

The following definitions are used in the rules below:

EXPORTED identifiers are the names of functions, types or preprocessor macros defined within a package and made available for public use.

INTERNAL identifiers are the names of functions, types, or preprocessor macros defined within a package for internal use only.

OPAQUE TYPE is a C type which is defined using the `typedef` keyword and whose value is hidden from the application. Manipulation of the value of the type is usually done using utility functions, *e.g.*, the Motif Widget type.

Naming Conventions – Symbolic Constants **1**

- All identifiers for symbolic constants must be in upper case and be defined as macros.
- For a given package, the names of symbolic constants which are the numerical equivalent of existing Fortran constants, both EXPORTED and INTERNAL, must be the same.
- The existing Fortran naming convention for package symbolic constants is strongly recommended:

  `PKG__NAME`

- There is no requirement to distinguish between EXPORTED and INTERNAL symbolic constants.
  - EXPORTED symbolic constants are defined in the public package header file

    `pkg.h`
  - INTERNAL symbolic constants are defined in the private package header file

    `pkg_sys.h`

Naming Conventions – Types 2

- Package-defined types must conform to the naming convention:

  ```
  PkgName
  ```

- Types which are not OPAQUE must not be pointer types.
- The global type

  ```
  StarStatus
  ```

  is defined in the Starlink header file

  ```
  star.h
  ```

  which is synonymous with `sae_par.h` and is reserved for the definition of the Starlink inherited status.

Naming Conventions – Functions 3

- EXPORTED package-defined functions and function-like macros, those which evaluate their parameters only once, must conform to the naming convention:

  ```
  pkgName
  ```

- For a given package, the names of EXPORTED functions and function-like macros which are equivalent to existing EXPORTED Fortran routines must reflect their Fortran equivalents:

  ```
  Fortran:  PKG_NAME

  C:        pkgName
  ```

- INTERNAL package-defined functions and function-like macros must indicate that they are INTERNAL functions by having an underscore appended to their names:

  ```
  pkgMyname_
  ```

- The name-length restrictions applied in Fortran may be relaxed in the case of INTERNAL C functions and function-like macros.

Other macros **4**

- All identifiers for other macros must be in upper case.
- The global macro

  ```
  INT
  ```

  is defined in the standard Starlink header file

  ```
  star.h
  ```

  and is used for the declaration of integers guaranteed to have a size of 32 bits.

2.2 Programming advice

Pointer names should be clearly identified. **5**

For example, the variable `i` and the associated pointer might be declared thus:

```
int i, *i_ptr;
```

Do not include whitespace inside compound operators. **6**

The compound assignment operators in C (`+= -= *= /= %= <<= >>= &= ^= |=`) should be considered to be single tokens. In principle, the characters can be separated by whitespace (or even comments); however it is bad programming practice to do so.

Don't use the remainder operator with negative integral operands. **7**

The remainder operator `%` gives the remainder when the first operand is divided by the second. That is, `a%b` is the modulus of `a` with respect to `b`. However, if either `a` or `b` is negative the result will vary according to the way integer division is implemented on the particular machine. Of course, division by zero must also be avoided.

Declare global variables used in routines as `extern`. **8**

The use of global variables in C can lead to confusion over which variables are active in a particular program module – just as the use of COMMON variables in Fortran can be a problem. Try to declare all such variables as `extern` in the routines which use them. If this practice is adopted the reader can just look at a page of code and see a list of ALL the active variables in a module. It is recommended that:

- The global variables used by a program are defined only once in a single source file in which they are explicitly initialized. For example:

```
int counter = 0;
```

- In each function which uses an external variable defined elsewhere, use the storage class `extern` and do not supply an explicit initializer:

```
extern int counter;
```

Of course, the variable must be given the same type in all locations. The `lint` program which is usually available on Unix machines can be used to check multiple files for consistent declarations.

When using true/false variables, create a 'pseudo-type' BOOL and define constants TRUE and FALSE. 9

For example:

```
#define BOOL int
#define TRUE 1
#define FALSE 0

BOOL flag;

flag = TRUE;
if ( flag )
...
```

Do not use `long double` data type. 10

Do not use `long double` data type even with ANSI compilers. Not all support it.

Use casts to indicate where (legal) type conversions are taking place. 11

For example the function `sqrt` expects a `double` argument. To find the square root of the integer `n`, the cast below should be used:

```
sqrt ( ( double ) n )
```

This does not affect the value of `n` but provides a value of type `double` as the argument for the `sqrt` function.

Don't cast pointer types. 12

Conversion of a pointer from one type to another can cause problems if the alignment requirements for the types in question are different. If the resulting pointer is illegal, this can cause an error or the pointer may be automatically adjusted to the nearest legal address. Conversion back to the original pointer type may not recover the original pointer.

Use function prototypes. **13**

Declare all functions and the types of their parameters. This enables the compiler to detect errors in the number of parameters and their types.

Pay careful attention to the type of the return value from library functions. **14**

For example the code fragment below is wrong because `getchar` returns an `int` not a `char`.

```
#include <stdio.h>
char c;
c = getchar ();
```

The correct way to use `getchar` is indicated in the example below:

```
#include <stdio.h>
char c;
int i;
if ( ( i = getchar () ) == EOF )
    /* end of file - do something about it */
else
    c = ( char ) i;
```

Use a function pointer type that specifies the correct return type. **15**

Functions must be correctly typed. The assumption that the default type (`int`) is sufficiently large to accommodate a pointer of any type is not correct on all computers. For example, the first function shown below will produce unpredictable results on some computers. The 'good' function illustrates the correct approach.

```
/* Bad: */
func ( double a, double b )
{
   static double c; /* a * b */
   c = a * b;
   return &c;
}

/* Good: */
double *func ( double a, double b )
{
   static double c; /* a * b */
   c = a * b;
   return &c;
}
```

Use `void *` for generic pointers, not `char *`. **16**

The ANSI C standard replaces `char *` with `void *` as the correct type for a generic pointer.

Always test the pointer returned by `malloc` *etc.* for equality with the null pointer before attempting to use it. **17**

Ignoring the possibility that `malloc` has failed can cause an access violation on a VAX and mayhem on many other systems.

Don't use a null pointer for anything other than assignment or comparison. **18**

A null pointer does not point anywhere. The effect of misusing a null pointer is unpredictable. Consider the code below:

```
ptr = NULL;
printf ( "Location 0 contains %d\n", *ptr );
```

Some C implementations permit the hardware location 0 to be read (and written), whilst others do not. Hence the code above will execute on some computers but not on others.

Don't assume a zeroed variable will be interpreted as a null pointer. **19**

Use the value NULL (which is defined as 0 in `<stdio.h>`) when assigning a null pointer. That is:

```
ptr = NULL;
```

Note that a null pointer does not necessarily result from the conversion of a floating-point zero.

Note also that the conversion of an integer zero to a floating-point type cannot be assumed to be a floating-point zero.

If the return value of a function is being discarded then cast the value to void. **20**

For example, rather than:

```
printf ( "Hello\n" );
```

use:

```
( void ) printf ( "Hello\n" );
```

2.3 Program design

No unused variables or unreachable code. **21**

Unused variables and unreachable code may occur during program development or testing, and can arise during program modification. They must not be present in operational code.

Finish a switch statement with a break statement. **22**

Adding a break statement at the end of a switch statement does not affect the program control flow, but does mean that there is less likelihood of introducing a bug if a further case is added in the future.

```
switch ( c ) {
   case 'a':
      ...
      break;
   case 'b':
      ...
      break;
}
```

Use only non-floating loop variables. **23**

The effects of cumulative rounding errors in real numbers cannot be ignored; the precision to which non-integer numbers are held must be considered and explicitly treated in the code if necessary.

Avoid changing the current loop index and range within a `for` loop. **24**

Unlike Fortran, C allows the loop index and range to be changed from within the loop control structure. For example it is permissible to alter both `i` and `n` in the body of the `for` loop below.

```
for ( i=0; i < n; i++ )
{
   ...
}
```

However this freedom should not be used unnecessarily as it leads to confusing code.

Don't use bit fields for storing integer values. **25**

This should only be done if storage space really is at a premium. Arithmetic on plain `chars` and `ints` will usually be far more efficient.

Take care with element order within structures **26**

Put the longest elements first to minimize 'padding'.

Don't add any explicit packing to structures in order to achieve optimum alignment. 27

What is optimal on one machine may not be on another. Rely on the advice in the previous item. When using bit fields, use an `int : 0` field to ensure re-alignment afterwards if necessary.

Don't use casts and `strcmp` (or `memcmp`) to compare the contents of structures for equality. 28

The padding regions within the structures may contain junk, even if the declared fields are identical.

Use assignment to copy structures (not `strcpy` or `memcpy`). 29

As indicated above, this approach avoids the problem of padding.

Use the correct method for error detection. 30

Most library routines return an indication of failure via an external variable called `errno`. The obvious way to check for an error might appear to be:

call library function
`if ( errno )`
make an error report

However there is no guarantee that the library routine will clear `errno` in the absence of an error. Initializing `errno` to zero before calling the routine does not solve the problem; `errno` may be set as a side effect of the function executing without the function actually failing!

The correct method is to test the value returned by the function for an error indication before examining `errno` to find the nature of the error. For example:

```
status = func ( a, b );
if ( status )
{
(examine errno)
}
```

See SSN/4, Section 8, for advice on making a suitable error report compliant with Starlink error-reporting conventions.

Adopt a particular order for parameters in functions. 31

It is recommended that the following order is used for parameters in functions:

(1) supplied and unchanged,

(2) supplied and changed,

(3) returned, and

(4) status return.

Functions should return a parameter containing a status value indicating success or failure. **32**

As mentioned above, global status should be the final item in a function parameter list. It should be a pointer to an integer type, and it is recommended that it is implemented as shown in the example below:

```
#include "star.h"
int function fun ( int p, int q, StarStatus *status )
{
   ...
}
```

This ensures that the status value is inherited from and can be returned to the calling module so that appropriate action can be taken if an error is detected.

When passing an array to a function pass the size of the array too. **33**

A function which is passed an array (particularly a character string) should also be passed the size of the array. The size should be used to ensure that the function does not overwrite the end of the array. Failure to adopt this practice can lead to bugs which are very difficult to track down. Note that unlike in Fortran you cannot use a function parameter to declare the size of a passed array; only a pre-defined constant can be used.

Use only one return statement in a function. **34**

Generally a single return at the end of a function is recommended. However it is permissible to have an extra return after an initial status check if such a check is performed.

Don't use the `gets` function. **35**

It is impossible to ensure that the string being read doesn't over-run the input buffer and overwrite other areas of memory.

Use the `sizeof` function explicitly where appropriate. **36**

For clarity and portability, it is recommended that the size of objects be obtained using the `sizeof` function. For example, use:

```
pntr = ( struct ABC * ) malloc ( sizeof ( struct ABC ) );
```

Do not use:

```
#define SIZEABC ...
   ...
pntr = malloc ( SIZEABC );
```

Beware using standard library pseudo-random number generators. **37**

The range of the numbers generated will be machine-dependent. ANSI compilers limit the upper bound of the range of numbers generated to `RAND_MAX`. This number can be used to normalize the distribution, although problems with the resolution of the resulting number distribution may remain.

2.4 Machine independence

Don't assume an order of evaluation as specified by brackets will be followed. **38**

Many pre-ANSI C compilers assume that the binary operators `+, *, &, ^, |` are both associative and commutative, (in the case where all variables have the same type). Hence it may rearrange the expression `( a + b ) + ( c + d )` to `( a + d ) + ( b + c )`.

This rearrangement may cause problems if, for example, the order specified was chosen to avoid the possibility of overflow. In cases where such considerations apply it is necessary to use assignment to temporary variables to force the desired evaluation order.

This freedom to regroup expressions has been removed by the new ANSI standard, in recognition of the reality that floating-point arithmetic is not in general associative and commutative. Hence the above advice will become obsolete as ANSI C-compliant compilers come into general use.

Avoid the more obscure parts of the language. **39**

Different C implementations are very likely to vary from each other in obscure areas. Keeping to the well-known parts of the language will reduce a program's dependence on the implementation getting everything correct.

Don't assume a particular encoding system is in use. **40**

Each standard character in C must have a distinct, positive integer encoding. **However this need not be accomplished using the ASCII encoding system.** Many C programs rely on the assumption that letters of the alphabet, (or the digits 0–9) are coded contiguously as is the case with ASCII. For example, the test below might be used to check if a character is a capital letter:

```
c >= 'A' && c <= 'Z'
```

This code will work with ASCII, but not with the EBCDIC character set.

The programmer must use the functions supplied with the standard header

```
ctype.h
```

This contains functions for testing and converting characters. For example, the function

```
isupper ( (int) c )
```

is a portable replacement for the code above. Similarly, the test of whether a character is a decimal digit should be accomplished with the standard function

```
isdigit ( (int) c )
```

not with the test:

```
c >= '0' && c <= '9'
```

Initialize all automatic variables before use. **41**

If local variables are not set they may contain junk.

Always explicitly declare `signed` or `unsigned` when using characters to store numerical (*i.e.* non-character) data. **42**

This is important because the default `char` type may be signed or unsigned depending on the implementation.

Don't assume more bits or bytes in a data type than the standard requires. **43**

Also don't write code which will fail if there are more bits or bytes present than the standard requires.

Don't write hex or octal constants which assume the word-length of a machine. **44**

For example use ~ 0, not `0Xffffffff`.

Don't make assumptions about the order or degree of packing of bit fields. **45**

Both the order and packing of bit fields are machine-dependent, so assumptions regarding them will result in non-portable code.

Don't write C structures to files which must be read on another machine. **46**

Details of structure packing are machine-dependent and should not be allowed to appear in files which must be portable.

Restrict use of library functions to those which are generally available. **47**

Avoid the use of library functions which are only implemented on some platforms. Use ANSI C Standard or POSIX functions.

Use the Starlink machine-independent macros for Fortran-to-C interfaces. **48**

These are documented in SGP/5.

2.5 Preprocessor advice

Set out pre-processor directives correctly. **49**

Start all pre-processor directives on column 1 and do not have any white space between the # and the keyword – some compilers insist on these requirements.

Limit use of preprocessor commands. **50**

The use of preprocessor commands should be limited to conditional statements and the inclusion of symbolic constants. In the case of portable interfaces it is sometimes necessary to define macros so that a single version of the main body of the C code can be maintained, with any architectural differences confined to header files.

Use only the generally availabile pre-processor directives. **51**

For example `#debug, #eject, #list, #module, and #section` are not available on every platform.

Restrict the contents of personal header files. **52**

The main purpose of header files is to encapsulate function prototypes and package-dependent global constants, global variables and global structures. Header files must *not* be used extensively for conditional compilation. It is recommended that the number of personal header files used be kept to a minimum.

Define macros only once. **53**

Don't rely on defining a macro at one point and then re-defining its value at another point. Define any macro only once, otherwise later changes may be made at the wrong point and may not be used. Also, if the second definition of the macro is inadvertently omitted (perhaps an include file may have been left out) you may use the wrong value unknowingly, whereas if the macro is defined only once, omitting this definition accidentally will probably produce a compiler error and the mistake will be spotted.

Test if macros have already been defined before defining them. **54**

A header file which defines a macro should test whether it has been previously defined so that the macro is not interpreted twice if it has actually been included twice. For example:

```
#ifndef header_read
#define header_read 1

  ...

#endif
```

This is a requirement for ANSI C header files and should also be applied to personal files.

Don't use reserved words as names of preprocessor macros. **55**

The following identifiers are reserved and must not be used as program identifiers:

```
auto, break, case, char, const, continue, default, do, double,
else, enum, extern, float, for, goto, if, int, long, register,
return, short, signed, sizeof, static, struct, switch, typedef,
union, unsigned, void, volatile, while.
```

Although not forbidden by the standard it is bad programming practice to use these words as preprocessor macro names.

Don't put directory names explicitly into `#include` statements. **56**

Rather than specifying directory names, use the compiler options to search the relevant directories.

Use the standard syntax for include files. **57**

Files should be included with the standard syntax. To search for the file beginning where the source program was found, use

```
#include "filename"
```

To search for the file in the system directories, use

```
#include <filename>
```

The VAX C syntax

```
#include filename
```

extracts the header file from a VMS text library. This is non-portable.

3 Acknowledgements

Thanks to Peter Allan, Chris Clayton, Malcolm Currie, Brian McIlwrath, Paul Rees, Keith Shortridge, Dave Terrett, Pat Wallace and Rodney Warren-Smith for their help in compiling this document.