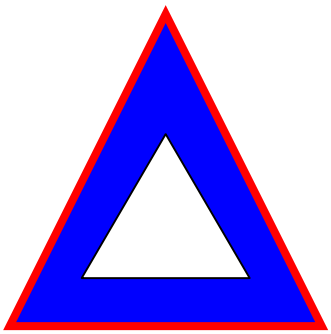


MISTRAY C

Mistray C is a coding guide for REAL programmers in all manner of programs from rockets to computer games.

This guide is guaranteed to be safe for being carried, stored or read in motor vehicles whilst stationary.

http://mistray-c.phaedsys.org/
email mistray.c@phaedsys.org

Disclaimer:-
This guide has nothing whatsoever to do with MISRA-C (www.misra.org.uk), MIRA or any other body that tries to restrict the free creativity of programmers. We totally reject any connection between this guide and any other similarly named publication. The aforementioned MISRA-C is a reactionary publication that seeks to restrict the free expression of programmers under the guise of vehicle safety and good practice. This guide does no such thing!

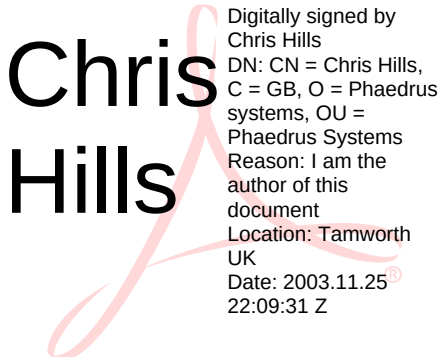
This guide is FREE in all senses in the true sprit of the free programming community. Artificial laws that restrict our programming flair and civil rights will not bind us. We proudly present this free form guide for C/C++ a free form language for free thinking programmers who will not be bound by the rules of enforced style, old thinking, law, Engineering and Physics

This guide is free for all to use and has no need of any warranty, as we are sure any reader will know exactly what to do with this guide.

This guide is guaranteed to be safe for being carried, stored or read in motor vehicles whilst stationary. This guide is worth exactly what you paid for it.

http://mistray-c.phaedsys.org

Please email any cool or radical improvements or super code fragments of pure genius for inclusion to mistray.c@phaedsys.org



MISTRAY C

Mistray C is a coding guide for REAL programmers. This is designed for sw teams that don't have time for all the sanctimonious rubbish churned out by ageing programmers who can't keep up with the modern world.

This is a coding guide designed for the fast paced world where time and costs matter. Where in reality real programmers don't have time to read lots of restrictive standards (that are only there to curb creativity) or the money to waste of over priced bug ridden tools.

We feel confident that this is the only document you will need for any project… except perhaps a copy of K&R.

Environment

Rule 1 All code shall to conform to ANSI/M$ Virtual c/C++

Any version will do they are all industry standard.

Rule 2 Code should only be written c/c++ .if VB++ is not available

Rule 3 Assembly is for geeks and should not be visible. It should be encapsulated in macros and used in line so as not to disrupt the flow of the c/C++

Rule 4 Provision should be made for run time checking. Get in some hackers to find the bugs. Provisions such as coke, pizza and burgers should be made available.

Character sets

Rule 5 Only characters and escapes that are in the OS System Character box may be used. Note Windings and zaps add 🙝🙖🙤⑨🙤🙝⓿🙕⑨ to any program. BE creative. YOUR STYLE SHOWS YOUR GENIUS.

Rule 6 Values of characters shall be restricted to any you have in a legal font. Illegal fonts should not be used.

Rule 7 Trigraffs should only be used by those who know how to use them safely. If some one can not interpret them they should not be programming.

Rule 8 Multibyte chars should only be used on &♋♓♦♎□ source.

Comments

Note: Comments like bugs should be less that 5 per K lines of code. It goes without saying that comments should be meaningful cos the are the documentation!

Rule 9 Comments should be nested with care. Badly nested comments look terrible.

```
  //This gives scope
 // for some /*fantastic *\
//layouts and lets the
//*creative spirit */
//of the programmer show though.
```

Rule 10 Sections of code should be commented out with care as it may break the rule on badly nested comments. It is noted that some code may need to be commented out as a version control. This is far more effective as it removes the need for expensive VCS programs.

Identifiers

Rule 11 Identifier names should be meaningful For example

This_variable_is_the_loop_counter_start_value
This_variable_is_the_lopp_counter_end_value

Compilers that cannot handle this are buggy.

Rule 12 No two identifiers should have the same name in the same function space.

Keep them different for example :- 0O0 is not the same as 000! But a standard naming convention helps eg “temp”, “speed”, “value” all help readability for novices who don’t really understand real programming.

Types

Rule 13 Hiding types is a waste of time and only confuses. Everyone knows what size an int is! The used of type def should be kept to a minimum.

Rule 14 To save key strokes the “unsigned” can be dropped. Compilers default to it anyway for ints and chars.

Rule 15 Floating point is REAL maths and should always be used where possible.

Rule 16 To save much time and space get to know your FP implementation so you can access the bits directly. This should only be attempted by real programmers. If you can’t hack the FP bit layout you shouldn’t be programming!

Rule 17 Typedefs can be reused. It simply over rides the previous typedef. If the compiler complains the warning level needs to be adjusted.

Constants

Rule 18 Compilers can handle all manner of numeric constants and these can be used in the most appropriate base even in the same line. Binary, octal, hex and decimal it saves on conversion time and memory.

Rule 19 Octal constants are smaller than hex and should be used if you need to save space.

Declarations and Definitions

Rule 20 Identifier functions can be declared on the fly when they are needed. Good compilers will fill in the blanks

Rule 21 Because of the c/C++ scope rules you can reuse names inside {} blocks. This saves memory and increases speed. It is encapsulation and therefore good OOP.

Rule 22 Global scope reduces the need for header files and parameter passing. Both parameter passing and headers are a source of errors and should be avoided. Besides parameter passing wastes stack space..

Rule 23 Static can be used on both functions and variable for different things which is confusing. Therefore it should not be used for either.

Rule 24 Rules for linkage are complicated and are therefore best left to the compiler and linker. They will complain if things have the wrong linkage It saves the programmer having to worry.

Rule 25 The best way of using a variable in multiple files is via an extern definition in all files. It saves having confusing header files

Rule 26 Due to the weak type checking of c/C++ conversions are possible simply by giving an identifier different types. A bit like a union which is the slower way of doing this.

Rule 27 External objects may be declared in all files. The compiler is intelligent and will work out the best routing.

Rule 28 The register key work is an essential aid to the programmer. This can be used to optimise the C in ways the compiler hasn't thought of. This is really only for experts, which is why the old has-been's try and ban it. .

Rule 29 Tags are a nuisance. For data hiding (ie good oo) tags use and declaration are not required to be the same.

Initialisation

Rule 30 The compiler usually initialises automatic variables. However as you are about to use them in the function it only wastes time and space to initialise them before use. Besides it is a useful debugging aid.

Rule 31 Braces can be used to make the layout of arrays and structures look really cool. Most experienced programmers have their own recognisable style.

Rule 32 In Emuns the = can be used to create customisable sequences. This means you can do it right. They don't have to in normal (boring) order. Again many programmers can be identified by their style. Be radical.

```
Enum lights= { green, red =3, cyan, blue, orange = 2, magenta = 7}
```

Operators

Rule 33 The clever use of && and II in compound statements can do some wild things. With care you can get these lines to trip out faster saving a lot of time. Most rules on && and | were written by boring old Cobol programmers. You can ignore these.

Rule 34 Hardware engineers base everything on && & II gates. You can encode a whole function in one macro line using && and || (with some? operators as well).

Rule 35 Bool can be rigged in C so that you can get True/False from some complex equations making compact if, case and while expressions that save time and space.

Rule 36 Remember that and is and and or is or. The compiler can usually tell by context the difference between && & & & II & I & ~ &!

Rule 37 Most compilers correctly handle bit sifts on signed ints by default.

Rule 38 Compilers wrap when the left or right shift is more than the size of the int.

Rule 39 All values should be inverted by using "-"

Rule 40 sizeof should be used to get the final size of most equations to determine storage size required for dynamic storage.

Rule 41 Most modern compilers and MCU have decent FP maths units therefore signed and unsigned values can be mixed in multiplication and division. (If not get a decent CPU)

Rule 42 Comas can be used to add additional items to a "for" loop to keep things in synch in a loop.

Conversions

Rule 43 The compiler will tell you when there is a loss of precision though usually integer promotion will take care of most situations automatically.

Rule 44 If in doubt cast everything. You can't have too many casts. OK We lied. Actually you don't need ***any*** casts. Modern compilers automatically do the correct casting.

Rule 45 Pointers can be cast for manipulation to avoid some confusing types of pointer arithmetic. You just cast them back to pointers afterwards.

Expressions

Rule 46 Compilers always read lines from left to right.

Rule 47 Apart from left to right always use the C precedence rules. All compilers implement these in a standard way.

Rule 48 Compilers automatically compensate for mixed precision arithmetic.

Rule 49 Zero is absolute and available on all systems. 0 == False.

Rule 50 There is a myth that 0.0 != 0 Zero is Zero.

Rule 51 When integers wrap round the compilers compensate. Thus integrity is maintained.

Control Flow

Rule 52 Version control may require the storage of code in files. This code should be made unreachable.

Rule 53 Lines with no effect can be used to slow things down rather than using interrupts or timers. This is a simple and easy way of slowing a program like a no op without complex timers that take much code and are difficult to adjust.

Rule 54 Null statements should be a ; on the end of a preceding line to save space. Null lines are just that and can be ignored.

Rule 55 Use meaningful names for labels for goto.

Rule 56 Goto should be used to avoid complex nesting of control statements and for fast exit from deeply nested loops for speed and efficiency. Otherwise see rule 122.

Rule 57 Continue is a useful construct that should be used to reflect the real world of embedded systems.

Rule 58 Break can be used as a quick exit for loops. Invaluable in hard real time systems. Otherwise see rule 122.

Rule 59 In single line if and else statements the use of {} is not required as it slows the compiler and the program. Resultant code will execute faster if the compiler does not have to evaluate the {}.

Rule 60 All if else if statements don't need a final else.

Rule 61 Break statements are only required for where you don't want to drop through cases.

Rule 62 Default on a switch statement is only required where there is a default state. Anything else is a through back to the bad old days of Algol and Pascal.

Rule 63 Switch statements can be used for true/false situations because they are faster than if else statements and it leaves room for expansion.

Rule 64 Switch statements do not need to have a hit. Therefore switch statements with goto's can be used to filter incoming data. Ie what gets past the switch filters is dealt with in the following code.

Rule 65 Loop counters can work with floating point numbers because compilers will round accordingly.

Rule 66 For speed "for" control statements can be used to initialise variables later used in the loop.

Rule 67 Calculations in the loop can change the loop counter rather than a flag. This saves time and variables.

Functions

Rule 68 For data hiding and encapsulation functions can be declared in a {} block.

Rule 69 Functions with variable arguments are a great way of getting C++ style templates into C. It also means that a single standard function can be the whole API… see stdarg.h for examples

Rule 70 Functions can call themselves (stack space permitting). Some really cool space saving techniques, useful for embedded systems, can be employed here.

Rule 71 Prototypes are not mandatory they weren't in K&R so they are not really needed. They only cause problems when a function needs to be edited and introduce errors in at compile time.

Rule 72 As C is not strongly typed you can save time and space by doing conversions across a function call. All compilers should understand that you want the bottom 8 bits when you pass an Int or long to a char parameter.

Rule 73 Identifiers are required for important parameters. They are not required for the obvious parameters.

Rule 74 Definitions and declarations do not actually have to be the same. This helps when interfacing between your code and some silly restrictive naming convention used in some libraries or indeed other teams code.

Rule 75 Return types are not "required" and only take up space and time. The compiler automatically assumes int so this does not require stating. If you wanted to type meaningless stuff you should have been a secretary.

Rule 76 As a parameter void is literally that and therefore pointless. It is not required in functions so why type it?

Rule 77 Void can be used to pass types when you are not sure what they will be. It defaults to int anyway.

Rule 78 If you understand your compiler and the stack it uses you can pass more parameters than are declared. **Real programmers know all the undocumented parts of the compiler.**

Rule 79 Void return types are normally guaranteed to be an INT when returned from a function call. Except in buggy compilers.

Rule 80 Void is a useful parameter as the compiler will usually accept an int see rule 78. This separates the old men from the cool boys as you have to know your compiler implementation for this one.

Rule 81 Const confuses everyone and should not be used in a function call. Later it will stop you modifying the parameter in the next release. More trouble than it's worth.

Rule 82 The single point of exit rule is a ridiculous myth. Multiple exits are sensible this is why the **Return** key word is there.

Rule 83 You can use any return type in multiple return functions as long as the return type is smaller (IE to fit in to) the declared type. Therefore it makes sense to declare return types as int or long.

Rule 84 If the return type is void then a return expression may be required to fit the type into the void (int) ie

```
FUNC()

    INT,A,B,C;

    RETURN ((INT)(A+B/C))
```

Rule 85 Functions with no parameters can be used without () This saves time and space when typing and also in execution.

Rule 86 You should test error information that is returned if you know there is a likelihood of failure. Things like printf never fail so it is a waste of time.

Pre-processor directives

Rule 87 Include statements are simply text replacement and can be used creatively to include all manner of data including *.c files. This makes for good modular or OO style programming

Rule 88 Include file names should be meaningful. These days the OS handles many more fonts that it used to. You can be distinctive with the team header files eg #include <DRIVERS-#HOMBOY#-USB.&DRIVE> Be radical. Compilers these days can handle it even if the reactionary managers can't!

Rule 89 As most compiler IDE's have full project information in them you can use either "" or <> for include files as the compiler isn't stupid and will pick up the correct file. Besides on one can remember which is which with "" and <> so the compiler check both paths anyway.

Rule 90 Macros are extremely powerful and should be used to encapsulate special tricks. Many conversions, filters and functions can be constructed. They can even be used to improve c/c++ for example

```
#define  IF      if(
```

```
#define THEN  ){
```

saves much typing and makes the code more readable.

Rule 91 Macros and defines can be overridden to give C++ like overloading within {} blocks.

Rule 92 Macros can be undef'ed when no longer needed see rule 91

Rule 93 Macros should be used in preference to functions. They mimic the inline functions in C++ and are faster than functions.

Rule 94 Macros may be called with some or all of their arguments depending on implementation. Good compilers do not require all the arguments. They automatically default it ints (with 0 value)

Rule 95 Using pre processor commands in Macros can produce some really effective and obfuscated code. This technique is useful for hiding various techniques from management. Ideal for contractors.

Rule 96 Extending rule 95 multiple macros can be joined arbitrarily. Thus lines can be constructed from several macros making one apparently normal line actually result in something very different to the compiler. Freaks the hell out of testers!

Rule 97 Macros can use any identifiers. As macros are text replacement they can use identifiers that may not be declared or initialised until later. Thus macros can be used with local variables inside blocks

Rule 98 Macro definitions # and ## may be nested. Check your documentation for the compiler. This is a n aspect of programming that is derided by the old school because they never understood it.

Rule 99 #Prgama is the method of using extensions in the compiler. As 99% of these are the same across all compilers everyone knows what they are so the can be used freely. People who don't understand these should not be programming.

Rule 100 The #define pre-processor command has a wide range of uses. It is only the novice who sticks with the two most common methods.

Rule 101 Pointer arithmetic is fast and saves time. Programmers who not understand it should do something else like test or maintenance programming.

Rule 102 Pointer indirection saves space. The more you use the more space you save and the faster the program runs. It is normal to have a prize for the highest number of levels of indirection on a project. This is a young man's game. Go for it!

Rule 103 As pointers are always the same size no matter what (or where) the type they point to comparisons are easy and cut out a lot of boring messing about to get the same answer. This can speed you programs no end.

Rule 104 Using arithmetic on pointer to functions it is possible to simulate c++ classes and objects in C! Therefore pointers should not be constants. (that is an urban myth put about by old Cobol programmers)

Rule 105 As pointers can point to anything and C is not strongly typed the same pointer can point to different types of the same function there by creating C++ style templates in C

Rule 106 As compilers only do what is needed in some implementations ie they leave memory until they have to use it again, you can pass an address of an automatic variable and use it out side the function as long as you do not call too many other functions first.

Rule 107 The NULL pointer is * 0 so it can be used. The myth that you can't use it springs from the early days when people did not really understand C compilers. Besides compilers have got a lot better in the last few years and handle it properly now.

Structures and Unions

Rule 108 Structures are a defined space. It is up to the programmer to define the contents. As C is not strongly typed the contents do not need to be full specified at the beginning leaving room for changes later.

Rule 109 In embedded systems you should learn the packing method. Thus you can overlap structures and unions to save space and increase speed. If

you can have the same variable in 3 structure it save space and time moving it between them

Rule 110 Unions are a great way of accessing parts of floating point numbers. A simple method that saves lots of time.

Rule 111 Bit fields can be of any type. This is an alternate method to rule 110 for accessing bits in floats. It's personal preference.

Rule 112 bits are bits. They can have any amount of padding but al bits are unsigned and assumed to be signed positive.

Rule 113 In line with 109, and 110 direct access to bits in a structure can save time with addition naming of bits that may only be required one or twice in a program.

Standard Libraries

Rule 114 C Library key words can be over ridden as per C++. For example __file__ __Line__ etc . These can be used for many functions as the code will not be debugged they will not be needed in testing.

Rule 115 Some of the standard library functions are a bit lame so overload them C++ style and do your own. Printf is a good candidate for this

Rule 116 Libraires are outside the scope of this document. They should be compiled without debug information to save space and increase speed (and written in assembler) and therefore cannot be tested.

Rule 117 Libraries always handle out of range values without crashing. So there is no need to validating inputs. It only wastes time and space repeating it in your app. Leave it to the testers it's what they are there for.

Rule 118 Dynamic memory is essential in embedded system. This permits memory to be reused for various functions. Only storage that is actually needed is malloc'd.

Rule 119 Library functions return an "errrorno" this should be used (if required) for return values on critical library functions.

Rule 120 Offset has a lot of uses in portability. It is also a great way of speeding up access to specific bits. Another method for getting at parts of floats.

Rule 121Set local can be used to individualise programs. It is very useful for automatically customising etc. This is a really cool feature that should be implemented to impress the customer. Getting . and , confused locally in not safe. Your app will know where it's at!

Rule 122 S*etjmp* and *longjmp* should only be used by people who know what they are doing. These functions are extremely useful for breaking out of complex and nested algorithms. They are also useful for HW interfacing and quickly calling code without a function call saving time and space.

Rule 123 Signal.h is a quick way of handling signalling functions. Because they are "implementation defined" it means the compiler will take care of all the specific interfacing. Thus is can be relied upon to work the same in all decent compilers.

Rule 124 Stdio.h is the programmers friend. It is one of the best tools for debugging. Ie printf and scanf. It is also a solid way of access data files.

Rule 125 atof, atoi and atol are standard macros that will save you writing your own. They should be used unless you need something special in which case printf may be better.

Rule 126 stdlib is just that, the Standard Library so it's functions should be supported by all compilers. However real programmers write their own custom libraries.

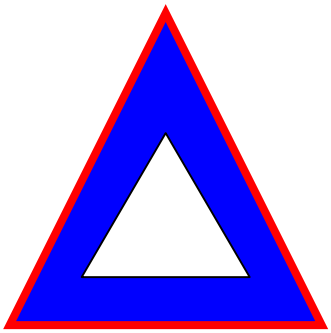
Rule 127 Time.h functions are a bit lame so use your own. Chinese and Arabic times are good. Year of the monkey and a lunar calendar!

References

C Programming Language K&R

Annotated C Hugebert Schildst

ANSI C (N289 committee draft is best as it is free and a text file)



MISTRAY C

Mistray C is a coding guide for REAL programmers in all manner of programs from rockets to computer games.

Disclaimer:-

This guide has nothing whatsoever to do with MISRA-C, MIRA or any other body that tries to restrict the free creativity of programmers. We totally reject any connection between this guide and any other similarly named publication. The aforementioned MISRA-C is a reactionary publication that seeks to restrict the free expression of programmers under the guise of vehicle safety and good practice. This guide does no such thing!

This guide is FREE in all senses in the true sprit of the free programming community. Artificial laws that restrict our programming flair and civil rights will not bind us. We proudly present this free form guide for C/C++ a free form language for free thinking programmers who will not be bound by the rules of enforced style, old thinking, law, Engineering and Physics

This guide is free for all to use and has no need of any warranty, as we are sure any reader will know exactly what to do with this guide.

This guide is guaranteed to be safe for being carried, stored or read in motor vehicles whilst stationary. This guide is worth exactly what you paid for it.

http://mistray-c.phaedsys.org