



# **NetworkCentric Core Avionics**



## **C++ Coding Directives**

**Version:** 10.0  
**Date:** 11.11.2008  
**Author:** Sergio Montenegro



# C++ Coding Directives

The purposes of the coding directives are a) to achieve a uniform programming style in the whole system and b) to avoid typical programming errors. Having a uniform style helps programmers to understand code from other programmers, and to make reviews and error-searching simpler and more effective.

We formulated these coding directives from our experience with space projects. The first version of these directives was applied for the development of the BIRD Operating System. In the current version of the coding directives we have added selected recommendations from ESA (European Space Agency) , ESTEC (European Space Research and Technology Centre) and NASA (National Aeronautics and Space Administration, USA).

This coding directives describe the programming style applicable for the OS + Middleware kernel and applications tasks, including the mission dependent and mission independent tasks.

## Applicable Documents

“C and C++ Coding Standards”

Prepared by: ESA Board for Software Standardization and Control (BSSC)

Approved, March 30<sup>th</sup> 2000

M. Jones and U. Mortensen, BSSC co-chairmen

Copyright © 2000 by European Space Agency

“C++ CODING STANDARD”

Flight Software Branch – Code 582

Version 1.0 – 12/11/03 - 582-2003-004

Goddard Space Flight Center Greenbelt, Maryland

National Aeronautics and Space Administration



The controlled copy of this document is located on-line at

<http://fsw.gsfc.nasa.gov/internal/StandardsCCB>

DoxyGen

Documentation Generator

<http://www.stack.nl/~dimitri/doxygen/>

 <p>DLR</p>	<b>C++ Coding Directives</b>	 <p>NetworkCentric</p>
--	------------------------------	---



## 1. Documentation

Doxygen comments (see <http://www.stack.nl/~dimitri/doxygen/> ) will be used to document code classes, attributes and methods. These comments must always immediately precede the item they are documenting.

**Use the `/** */` or `///` commenting style for all Doxygen related comments and `/**` or `//` for any other comments.**

If a Doxygen comment can fit on one line, the format to be used is as follows:

```
/** one line comment */
int x;

/// one line comment
int x;

int x; ///< comment in the same line
```

If a Doxygen comment spans more than one line, the format to be used is like this:

```
/** changes the priority of a Thread. <p>
 * Smaller values make the thread less important
 * larger values make the thread more important
 *
 * @param newPriority the priority the thread shall have
 *     Values must be greater than <b>Zero</b>.
 */
public void setPriority (int newPriority) { ... }
```

Note the use of the HTML tags `<p>` and `<pre>`. `<p>` forces a paragraph break and `<pre> ... </pre>` specifies a block of text to be presented with a fixed font, preserving all white space.

Doxygen permits any other HTML you might need to use, but prefers that you do not use the header tags (`<h1>`, `<h2>`, etc.). You might want to use `<b> ... </b>` to bold or `<i> ... </i>` to italicize or even the `<u> ... </u>` to underline.

Note that Doxygen splits out the first line of any Doxygen comment to use for a short "table of contents". You might want to think about where you want this line division to happen. Doxygen defines this as a period followed by a space. So question marks followed by a space don't count! Nor does a period followed by a `<p>` tag!

**(Doxygen) documentation of classes must include :**

A brief synopsis

A detailed description



An example code segment using the class

An author list using the @author tag

More complex usage-examples shall be provided as an external example short program in a tutorial/test directory.

## 2. Declarations

### 2.1. Statements

"includes" statements (if used) must always appear first, then "namespace" , followed by non Doxygen comments, and Doxygen class documentation appear next in this order. The class itself comes last.

Example:

```
#include "mytools.h"
namespace Aliens {

/** this class does cool stuff
 *   @author Joe Programmer
 */
    class SpaceMouse {
        ...
    };
}
```

#### Maximal Line Length should be 120 characters

Lines should not be longer than 120 characters. Most of your code should be under 80 lines. If your code starts to get indented too far to the right, consider breaking up your code into more methods.

Reasoning: Editors and printing facilities used by most programmers can easily handle a width of 120 characters. Longer lines can be frustrating to work with for a variety of reasons including text overflow.

**All indents are two spaces wide. The first of the braces is at the end of the previous line.**

**Use the occasional blank line within methods to break up related chunks of code. Use two blank lines in between methods.**

### 2.2. Declaration of Constants

**Symbolic constant names shall contain only capital letters.**



**Symbolic constants should be used in the code. "Magic" numbers and strings are forbidden (estec 53).**

```
const int CONTAINER_MAX_LENGTH = 4711;
```

**All symbolic constants should be provided using an "enum" or by using the "const" keyword (estec 54).**

### **Physical constants shall be qualified**

Time Constants shall be multiplied by the Time unit, eg

3 \* MILLISECONDS, or 3 \* SECONDS.

The same for metres, grams, etc. the OS shall provide the units for time.

## **2.3. Declaration of Classes**

**Each word in the class name shall begin with a capital letter followed by lowercase letters. Exceptions are acronyms, which will be all upper case.**

Examples:

```
Context
TaskContext
TargetROM
ROMTarget
```

**No two classes in the whole system shall share the same name.**

**Class member should be arranged like this:**

```
class Order {
    // attributes

    // constructors

    // methods
};
```

**(Weak recommendation) Class member variables must not be declared "public" (estec 68).**

One of the frequently quoted benefits of Object Oriented Systems is the ability to encapsulate data within an object and to hide the implementation details behind a strictly enforced interface. If the class member variables are hidden behind more widely available accessor functions then the underlying implementation of the class can be hidden from the user of the class. The implementation may change without affecting the interface itself.

Exception: variables which would have simple set and get methods shall be declared public, and the set/get methods can be eliminated.



**When possible, constructors should initialize each data member in the class to a valid default value. The object should be completely initialized (NASA 3.7.1).**

**Members should appear in an initialization list in the order in which they are declared, one per line and equally indented (NASA 3.7.2).**

**A class should always declare a constructor (estec 69).**

The programmer should always declare a constructor and explicitly initialise the member-variables appropriately. For simple assignment use "x=y" instead of x(y). This simplifies the reading of big programs.

```
class Example {  
    int a;  
public:  
    Example(int _a) { this->a = _a; }  
    // ...  
};
```

**If a class contains any virtual member function then its destructor must also be virtual (estec 71, gcc).**

## 2.4. Declaration of Variables

**Variable names shall begin with a lower case letter but any subsequent new word should start with an upper case letter. All other letters are lowercase except for acronyms which are all in uppercase.**

Examples:

```
context  
taskContext  
targetROM  
isFirstTime
```

**Boolean variables should begin with *is* or *shall*.**

**Singleton variables (singleton == only one object derived from a class) shall have the same name as the class, but the first letter shall always be lower case .**

Examples:

```
Context context;  
TaskContext taskContext;
```



**Variables should be declared just before they are used for the first time unless it will impact upon the performance of the code.**

**Variables shall not be declared within loops (NASA 3.1.6),**

This is because doing so may cause performance issues; variables should be declared within the proper scope (with the exception of loops, as noted); if a variable is only needed within a sub-block of a function or method, then it should be declared within that sub-block.

**Static declarations inside methods should be used sparingly, taking into account any re-entrance issues that may result (NASA 3.1.13).**

**Static declarations (outside of any block) shall be preceded by the keyword static.**

This is to avoid name collisions and access from outside the block. Exception: If the variable is really intended to be globally accessible from other modules too.

**Local variables are preferable to class variables, which are preferable to subsystem globals, which are preferable to system globals.**

**Variables should be initialized at declaration, if possible.**

Examples:

```
bool isFirstTime = true;
```

**Each variable shall have its own personal declaration, on its own line (estec 55, NASA 3.1.5).**

```
/*
 * These declarations are unclear
 */
int x, y, *z;    /* x,y are ints but z is pointer to int */
char* reply, c; /* reply is pointer to char, but c is char! */
```

**A const shall never be converted to a non-const (NASA 3.1.11)**

If a non-const is really needed, then the code should be restructured.

**Avoid similar names for different variables.**

Avoid names which differentiate only by capitalisation, eg timeAtStart and timeAtstart or UDPMessage and udpMessage. These are confusing and can cause programming errors.

Avoid parameters using the same name like local variables

eg

```
void constructorxx(int temperature) { this->temperatur = temperatur;}
```





The risk of misspelling is too great when coding as the above example indicates:  
(temperature <-> temperatur).

### Variable names should not begin with "\_".

This is to avoid confusion with names reserved for macros in the programme or file definition section (#define ....), and for parameters for these classes.

Example:

```
class Example {
    long time;
    Example(long _time) {
        time = _time;
    }
};
```

### Variables in macros shall begin with “\_”

Conversely, all macros (especially those declared in #define....) should begin with “\_” to clearly identify them as such.

Example:

```
#define MIN(_a, _b)    ( (_a) < (_b)? (_a) : (_b) )
```

## 2.5. Declaration of Functions

**The first (or only) word in a function shall begin with a lower case letter. Any subsequent words shall be begin with capitals (as with variables).**

**Each function shall be defined using the following layout (doxygen, different from estec 65):**

```
/**
 * General description of exampleFunction()
 * @param alice Description of parameter alice
 * @param bob   Description of parameter bob
 * @return      Description of return value
 */
ReturnType exampleFunction(ParamType1 alice, ParamType2 bob) {
    /* implementation */
}
```



**Methods that set object state must use the prefix set in the method name. Methods that retrieve a boolean object state must use the prefix is/shall in the method name. All other methods that retrieve object state must use the prefix get on the method name .**

Examples:

```
setEnabled()  
getName()  
isEnabled()
```

**The name of a function shall be an imperative (do some thing) or in question form (bool functions) for example:**

```
activateMotor()  
deactivateTransmitter()  
isMoterOn()
```

**User-defined types shall be passed by reference. Unless the intent of the method is to modify an argument, the argument in the method signature should be declared as a reference to a const object (NASA 3.4.10).**

Note the following example, in which X is a user-defined type:

```
// Not Recommended  
MyClass::MethodA(X MyArg); // object passed by value  
MyClass::MethodB(X* pMyArg); // object passed by pointer  
  
// Acceptable  
MyClass::MethodC(const X& MyArg); // MyArg NOT to be modified  
MyClass::MethodD(X& MyArg); // MyArg to be modified
```

**Code shall never return a reference or pointer to a local variable (NASA 3.4.11).**

**The formal arguments of methods shall have names, and the same names should be used for both the method declaration and the method definition (NASA 3.6.5).**

**When declaring methods, the leading parenthesis and the first argument (if any) should be written on the same line as the method name. If space permits, other arguments and the closing parenthesis may also be written on the same line as the method name. Otherwise, each additional argument is to be written on a separate line (with the closing parenthesis directly after the last argument) (NASA 3.6.6).**

```
ReturnType exampleFunction(ParamType1 alice, ParamType2 bob) {  
    /* implementation */  
}  
  
// or  
ReturnType exampleFunction(ParamType1 alice,  
                           ParamType2 bob) {  
    /* implementation */  
}
```



```
}
```

**Method length, including white space and comments, should not exceed 100 lines (NASA 3.6.7).**

## 2.6. General Member Function Guidelines

**Member functions, which do not alter the state of an object, shall be declared “const” (estec 81).**

If a member function does not affect the internal state of an object, i.e. provides information about an object or calculates something derived from the contents of the object, then the member function should be declared as “const”. This allows the compiler to issue error messages if the programmer subsequently tries to modify the object from within the function.

eg.

```
int getPriority() const { .... }
```

**Public member functions may not return non-cost references or pointers to member-variables of an object (estec 82).**

This rule states that non-member variables may be public so that access to them can be controlled. The programmer must ensure that there are no ways of changing these member variables without using the intended interface. If a member function returns a non-const reference or pointer to a member variable the programmer may inadvertently change the value of a member variable through using this reference or pointer. As there are other routines which are used to limit the access to the member variables this is probably not what the class designer intended. Such indirect access to member variables can result in corruption if other member variables are interdependent on the one being changed.

**Do not override operators (estec 94).**

## 3. Flow Control

### 3.1. Flow Control

**The flow control primitives if, else, while, for, and do should be followed by a block, even if it is an empty block (NASA 3.9.1).**

Occasionally, all tasks to be completed in a loop may be easily written on just one line within the loop statement. It may be obvious to conclude the statement with a semicolon at the end of the line, however this may lead to a misunderstanding if, when reading the code, the semicolon is missed. Instead of a semicolon then, an empty block should be placed after the statement to clarify completely what the code is doing. If an empty block is used, a comment



indicating that the block is intentionally left blank should be used to indicate that the code is complete.

**Nesting levels greater than six (6) deep should not be used (NASA 3.9.4).**

If more than 6 levels of nesting are required, consideration should be given to dividing the method or function into smaller ones.

**The ternary conditional operator (?:) shall not be nested (NASA 3.9.5).**

**(Weak recommendation) Only one exit point (return) should be used in a single method (NASA 3.9.6).**

If this recommendation is violated, then the different exit points should be marked VERY clearly.

**The code following a case label shall be terminated by a break statement. If several case statements execute the same block of code, i.e., fall-throughs, they must be clearly commented (NASA 3.9.7).**

**A switch statement shall always contain a default case that handles unexpected cases (NASA 3.9.8).**

**(Weak recommendation) breaks shall only be used in switch statements, and nowhere else (NASA 3.9.9).**

If this recommendation is violated, then the different exit points should be marked VERY clearly.

**(Weak recommendation) The continue statement shall never be used (NASA 3.9.13).**

If this recommendation is violated, then the different exit points should be marked VERY clearly.

**goto shall never be used (NASA 3.9.10).**

**unsigned shall be used only for variables that cannot have negative values (NASA 3.9.11).**

Variables representing size or length are typical candidates for unsigned declarations

**Inclusive lower limits and exclusive upper limits shall be used (NASA 3.9.12).**

Instead of saying that  $x$  is in the interval  $x \geq 23$  and  $x \leq 42$ , use the limits  $x \geq 23$  and  $x < 43$ . The following important claims then apply:

1. The size of the interval between the limits is the difference between the limits.
2. The limits are equal if the interval is empty.



3. The upper limit is never less than the lower limit.

## 3.2. Conditional Expressions

**Logical operators shall not be used on non-Boolean values (NASA 3.9.14):**

Not Acceptable:

```
int x;  
// code that modifies x  
if (!x) {  
    // do something  
}
```

Acceptable:

```
int x;  
// code that modifies x  
if (x == 0) {  
    // do something  
}
```

**Conditional expressions must always compare against an explicit value (estec 92).**

The traditional C idiom has always favoured brevity in code and this is extended to the implicit comparison of values against zero, which is also used to denote false. In some cases this can be counter-intuitive and the programmer reading the code must expend some effort to understand what is actually happening.

The other area where this can be problematic is when dealing with the return codes from functions. In many cases there is a range of return codes, which may indicate different levels of success or failure of the function and not just zero (false) and non-zero (true).

Different routines are not consistent in their use of return codes to indicate success and failure. Many routines, which deal with pointers, return NULL (i.e. zero) to indicate failure, while many others use a negative number to indicate failure. The common string comparison routine strcmp() uses zero to indicate equality!

Routines, which use enumeration as the return value, can suffer when a new member is added to the enum, which may cause the renumbering of the existing values, and invalidate any implicit comparisons.

The programmer is therefore encouraged to use explicit comparisons in the code so that the meaning is clear, and to safeguard against changes in the underlying implementation of functions and their return types.

```
if (function(parameter) == SUCCESSFUL)
```

is guaranteed to continue working as expected even if the actual value associated with the return code SUCCESSFUL is ever changed. It doesn't matter whether function() returns a negative number, zero, or a positive number as long as SUCCESSFUL has the appropriate value. The symbolic name also helps to clarify the code. On the other hand, the following



fragment is unclear and will not continue to work correctly if the return value is changed to or from zero.

```
if (function(parameter))
```

Note also that the language standards define zero as meaning false. They do not define a particular non-zero number to mean true. It is theoretically possible to fall foul of a compiler if a symbolic constant for true is defined to be a particular value. For example if MyFalse is declared to be 0 and MyTrue to be 1 and if value is 2 then both value != MyFalse and value != MyTrue are true at the same time. This does not lead to intuitive code!

Do not compare Boolean values with true, eg

```
if(boolx == true) // WRONG
```

use

```
if(boolx)
```

because “true” is only one value, but there are 4000000000 values of boolx which may be true.

## 4. Expressions

### 4.1. Casting

**(weak recommendation) The programmer could use the new C++ cast operators rather than the traditional C cast (estec 91, NASA 3.1.12).**

The traditional C method of casting simple variables and pointers from one type to another is still available in C++ but its use should be discouraged. The traditional C cast is an instruction to the compiler to override any type information, which it may hold about an expression.

Note that this can lead to portability problems because a cast, which may be valid on one system, may not be valid on another but the compiler assumes that the programmer knows best.

The C++ standard defines four new cast-operators that are more specific in their operation and therefore the intention of the cast is clearer. Three of them are of interest here.

The first one `static_cast<type>(expression)` is equivalent to a traditional C cast but is more obvious in the code.

The second one `const_cast<type>(expression)` allows the programmer to remove the const or volatile nature of the expression.

The third one `reinterpret_cast<type>(expression)` is intended for particularly complicated and non-portable casting such as that one concerning pointers to functions.



## 4.2. Order of Evaluation

**The programmer shall make sure that the order of evaluation of the expression is defined by typing in the appropriate syntax (estec 93).**

The order of evaluation of individual expressions within a statement is undefined, except when using a limited number of operators ("(", "&&", "||", "?").

All expressions, which appear as function arguments, are evaluated before the function call actually takes place. Note that the order of evaluation of the arguments is not specified.

In addition, a "full expression" is the enclosing expression that is not a subexpression.

The compiler will evaluate each full expression before going further.

The operators which are explained above are known as "sequence points" and can be used to determine the order of evaluation of the expressions in some statements. However the order of evaluation of expressions between each sequence point is undetermined.

This means that different compilers may produce different answers for particular statements. This is most notable when using expressions that contain side-effects. For example the outcome of the following statement is undefined.

```
firstArray[i] = secondArray[i++]; /* UNDEFINED!! */
```

**Use of Parentheses: The programmer must use parentheses to make intentions clear (estec 95).**

As well as considering the order of evaluation of a statement, the programmer must also consider the precedence of the operators used in the statement because these affect the way the compiler breaks down the statement into expressions. For example, the relative precedence of the addition and multiplication operators mean that the expression

`a + b * c`

is really treated as

`a + ( b * c )`

rather than

`( a + b ) * c`

The programmer is advised to make explicit use of parentheses to reduce any possible confusion about what may have been intended and what the compiler will assume by applying its precedence rules.

**The programmer must always use parentheses around bitwise operators (estec 96).**



The use of the bitwise operators is not always intuitive because they appear to offer two different types of behaviour. On the one hand they behave like arithmetic operators while on the other hand they behave like logical operators. The precedence rules may mean that a bitwise operator behaves as expected when used in one context, but not when used in another context. The safest course is to use parentheses with the bitwise operators so that the code is clear.

```
if (statusWord & PARTICULAR_STATUS_BIT)
```

lacks an explicit comparison (see Rule 92) so the programmer modifies it to what would initially appear to be an equivalent form:

```
if (statusWord & PARTICULAR_STATUS_BIT != 0)
```

Unfortunately the programmer has just introduced an error into the code! The precedence rules mean that the bitwise-AND operator has a lower precedence than the != comparison operator and as a result PARTICULAR\_STATUS\_BIT is tested against zero and the outcome is then combined with the statusWord using the bitwise-AND. The programmer must use parentheses to restore the code to working order:

```
if ((statusWord & PARTICULAR_STATUS_BIT) != 0)
```

**Do not use spaces around the "." and "->" operators or between a unary operators and their operands (estec 97).**

These operators are tightly coupled with their operands and the precedence rules mean that they are evaluated before the other operators. Adding spaces between the operator and the operand makes them appear to be more loosely coupled than they really are and this can lead to confusion.

**Other operators should be surrounded by white space (estec 98).**

In contrast with the previous rule, other operators should be surrounded by white space in order to give additional visual separation of the operands in order to show that they are more loosely coupled. Note that parentheses may be used to group operands for evaluation purposes and white space may be omitted for "tightly-coupled" operands.

```
y = (a * x * x) + (b * x) + c; // or y = a*x*x + b*x + c;
```

However, when in doubt about operator precedence, use parentheses!

## 5. Preprocessor

**The programmer should use #include system header files using <name.h> and user header files using "name.h" (estec 108).**





There are two different reasons for using different `#include` semantics for system header files and user header files. The first is that the programmer can see from the use of angle brackets or double quotes whether the header file is a system header file or a user header file as this might not be obvious from the name alone.

The second reason is that on some systems the different `#include` semantics imply slightly different behaviour when it comes to searching for the include file in the file system.

### **The `#include` line may not contain the full path to the header file (estec 109)**

If the programmer specifies the full path to a header file then the source file will need to be modified when the header file is moved. Different operating systems also use different ways of specifying a full path and this will therefore need to be changed when porting the software from one system to another.

### **Pre-processor macros, which contain parameters or expressions, must be enclosed in parentheses (estec 111).**

The constants and macros provided by the pre-processor are not really part of the language and are simply substituted directly into the source code. Therefore care should be taken when using expressions (even in constants!). The body of the macro should be enclosed in parentheses.

```
#define TWO 1+1 /* should be (1+1) better use const ... */  
six = 3 * TWO; /* six becomes 3*1+1 i.e. 4 */
```

If a macro contains any parameters, each instance of a parameter in both the macro declaration and in the macro body should be enclosed in parentheses. The parentheses around the parameter in the declaration protect the macro in the event that it is called with an expression that uses the comma operator that would cause confusion about the number of parameters. The parentheses around the parameters in the macro body protect the macro from conflicts of precedence if the parameter is an expression.

```
#define RECIPROCAL(_x) (1/_x) /* should be (1/(_x)) */  
half = RECIPROCAL(1+1); /* half becomes 1/1+1 i.e. 2 */
```

### **Macros should not be used with expressions containing side effects (estec 112).**

This is another facet of Rule 93. The macro hides the underlying expression.

The programmer may only see one expression (with possible side effect) when calling the macro, but underneath the macro body may use this expression several times. The order of evaluation of arguments before calling a function does not apply to a macro because it is not a function but merely a pre-processor convenience.

```
#define SQUARE(_x) ((_x)*(_x)) /* nothing unusual */  
z = SQUARE(y++); /* z becomes y++*y++ i.e. what? */
```

Macros shall not use variables which are extern to the macro.

Eg:



```
#define INCREMENT(_Y) { x++; _y++ }  
int x = 0;  
int a = 2;  
INCREMENT(a);
```

### **The pre-processor may not be used to redefine the language (estec 113).**

Programmers who are more comfortable using another programming language may be tempted to provide pre-processor macros, which map C and C++ constructs into some semblance of this other language.

```
#define IF      if  
#define THEN {  
#define ELSE } else {  
#define FI    }
```

```
IF (x == 1)  
THEN  
statement(s);  
ELSE  
statement(s);  
FI
```

This is expressly forbidden because the programmer is supposed to be using C or C++ and not some other language with which a future maintenance programmer may not be familiar. Certain constructs may not map directly from C into the other language and may therefore have restrictions on their use. Various support tools, such as syntax-aware editors, will be unable to work with the macros.

**(Weak recommendation) #If #else statements should not be used in pre-processor directives.**

## **6. Data Representation**

**The programmer should use "problem domain" types rather than implementation types (estec 114).**

The programmer is encouraged to add a level of data abstraction to the code so that variables are expressed in terms that relate directly to the problem domain rather than to the underlying "computer science" or hardware implementation.

The use of "typedef" can also improve maintainability because only one place in the code needs to be changed if the range of values for that type must be changed. For example, the range of values may be changed by converting a variable, or set of variables, from "short" to "unsigned short".

```
typedef unsigned short TAgeInYears;  
typedef float TKmPerHour;
```

Please note the prefix T for typedefs.

**The programmer may not assume knowledge of the representation of data types in memory (estec 117)**

Different systems may make use of different representations of data types in memory. Such differences may include differences in word ordering, byte ordering within a word (byte sex / Endianity) and even the ordering of bits within a byte. The programmer should therefore avoid any specific knowledge of an underlying representation when manipulating the data because what may be valid on one system may not hold true on another system.

**The programmer may not assume that different data types have equivalent representations in memory (estec 118).**

This rule follows from the previous section and the rule above. If the different types specify different ranges of values it is likely that they are represented differently in memory. The programmer may not assume that different types share a common representation in memory.

**The programmer may not assume knowledge of how different data types are aligned in memory (estec 119).**

Different hardware platforms impose different restrictions on the alignment of data types within memory.

**The programmer may not assume that pointers to different data types are equivalent (estec 120).**

Some hardware platforms actually use different pointer representations for different data types. Therefore assigning one pointer value for one type to a pointer variable for another type is not always guaranteed.

The only exception is the equivalence of void\* pointers to pointers of other types.

**The programmer may not mix pointer and integer arithmetic (estec 121).**

Pointers are not integers. The programmer must not treat pointers as integers. The programmer is prohibited from assigning integers (or integer values) to pointers and vice versa. If we assume the following declarations:

```
int      intValue;
Thing    thingArray[10];
Thing    *thingPointer = thingArray; // i.e. &thingArray[0];
```

```
intValue = thingPointer; // DO NOT DO THIS IN REAL CODE!
intValue += 1; // increase integer value by 1
thingPointer += 1; // adjust pointer value to point to
```

At this point it is unlikely that intValue and thingPointer contain the same value because intValue has been incremented by 1 whereas thingPointer has been adjusted by the size of a Thing object (including possible adjustments for the alignment of Thing objects in memory).

**«Within tolerance» shall be used instead of testing for exact equality when testing equality on floating-point numbers (NASA 3.9.2).**



Not Acceptable

```
if (someVar == 0.1) // may never be evaluated as true
```

Acceptable

```
if (abs(someVar - 0.1) < MAX_TOLERANCE) // safe
```

The constant 0.1 cannot be represented exactly by any finite binary mantissa and exponent.

### **The programmer must use a wider type or unsigned values when testing for underflow or overflow (estec 122).**

This addresses the case when the programmer is dealing with two integer variables, with a relatively narrow possible range (such as chars and shorts), and is worried about overflow or underflow. Then it is possible to convert the values into variables with a wider range in order to detect whether overflow or underflow occurs during the calculation.

```
if ( (long)shortValue + (long)shortValue > SHRT_MAX)
// overflow
```

This is not possible if the variables in question are of type long or have the same range as a long.

However it is possible for the programmer to code around potential problems of overflow or underflow when dealing with two unsigned integer values because this can be detected without relying on the underlying hardware implementation.

```
if (a+b < a || a+b < b) // overflow has occurred!
```

When handling one signed and one unsigned integer value the programmer can still detect potential problems by converting the signed value into an unsigned value and using unsigned arithmetic.

However, when dealing with two signed integer values what happens when overflow or underflow occurs is implementation dependent. The program may abort, it may set some error code, which can be tested, or it may simply continue.

If overflow and underflow are likely to be a problem the programmer may need to convert to using unsigned arithmetic to be able to guarantee detection.

### **The programmer must be careful when assigning "long" data values to "short" ones (estec 123).**

The ranges of the different integer types and floating point types can be different. Therefore the programmer must ensure that a value does not lie outside the permitted range of a particular type before assigning the value to a variable of that type.

## **7. More narrow restrictions: Embedded C++**

### **Embedded C++**

**For on-board software, the subset “Embedded C++” must be used.**

“Avoid those features and specifications that do not meet the requirements of embedded system design. The three major requirements of embedded system designs are:

- Avoiding excessive memory consumption
- Taking care not to produce unpredictable responses
- Making code “ROMable”

"Embedded C++" is a pure subset of ISO/IEC14882 C++ standards and the following are the major restrictions put on normal C++:

- no exceptions
- no RTTI (Run Time Type Information)
- only very simple templates (only to simplify code, highly recommended for type-safe)
- no multiple inheritance
- no dynamic memory allocation

This language is appropriate for the production of on-board software, especially if the criticality level is high.

## 8. Simplicity

To avoid errors and keep code clear:

Make it simpler!

Make it even simpler!

Do not try to optimise it, if it isn't necessary. It is better to keep it simple and clearer

Do not try to save some bytes of memory through clever coding. This is strictly not necessary

Do not try to save some microseconds through optimisation, if not necessary

Simply put, if the code is already good then don't try to make it any better!