# COM4506/6506: Testing and Verification in Safety Critical Systems

Dr Ramsay Taylor

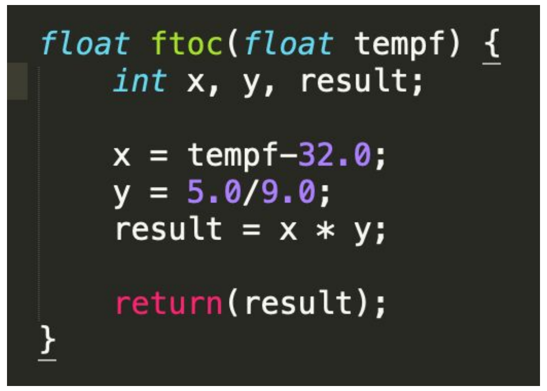
## Contents

- What happens when we compile?
- What can we Statically Analyse if we don't know what will happen?
- Coding Standards - not just making things pretty!

## Type Theory

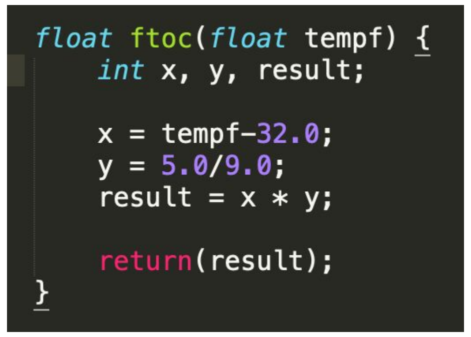
$$\begin{array}{|l}
\textit{ConvertFtoC} \\
tempF? : FLOAT32 \\
tempC! : FLOAT32 \\
\hline
tempC! = (tempF? - 32) \times \frac{5}{9}
\end{array}$$

=?=



```
float ftoc(float tempf) {
    int x, y, result;

    x = tempf-32.0;
    y = 5.0/9.0;
    result = x * y;

    return(result);
}
```

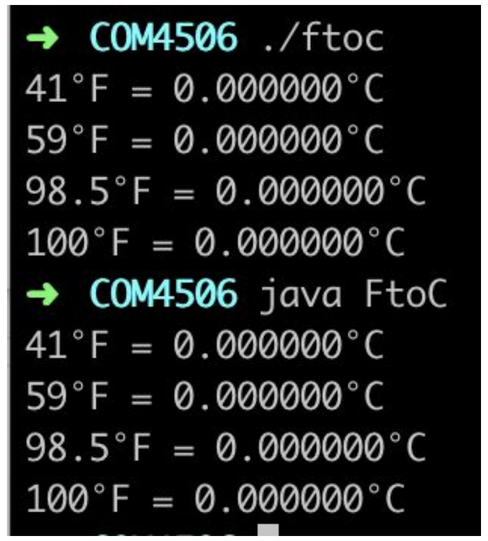
## Type Theory



```
float ftoc(float tempf) {
    int x, y, result;

    x = tempf-32.0;
    y = 5.0/9.0;
    result = x * y;

    return(result);
}
```

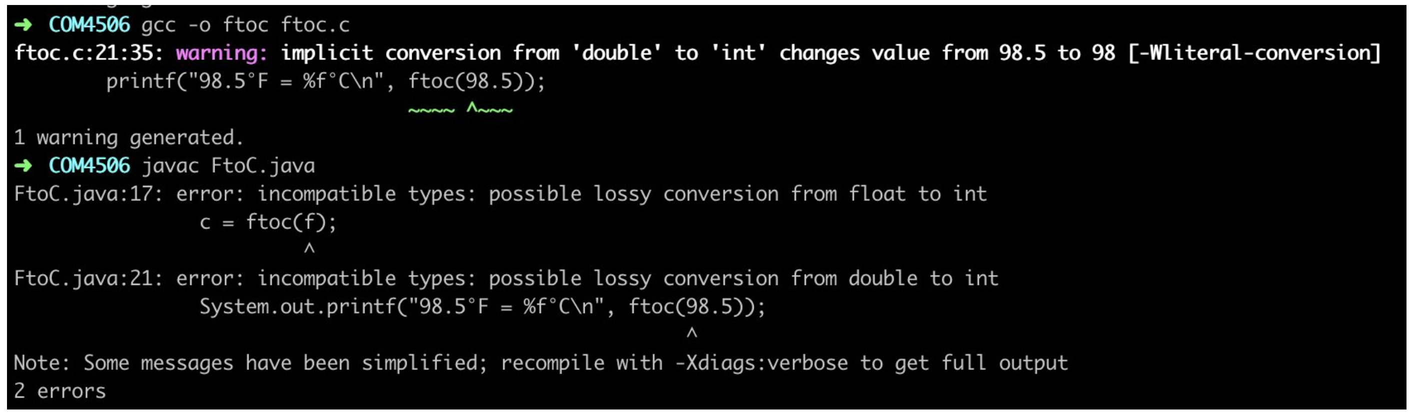
$$\begin{array}{lll}
\{tempF = TF\} & x = tempF - 32; & \{tempF = TF;\ x = TF - 32\} \\
 & y = 5.0/9.0; & \{tempF = TF;\ x = TF - 32;\ y = \frac{5}{9}\} \\
 & tempC = x * y; & \{tempF = TF;\ x = TF - 32;\ y = \frac{5}{9};\ tempC = (TF - 32) \times \frac{5}{9}\}
\end{array}$$

## Type ~~Theory~~ Practice



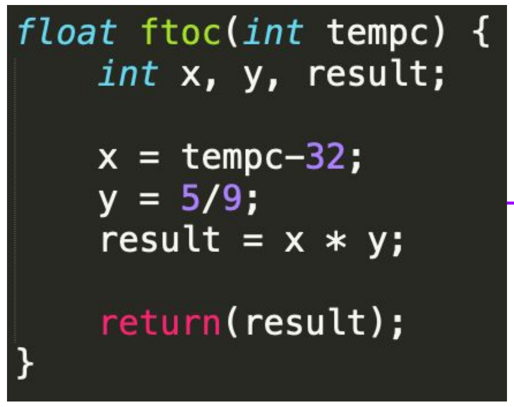
```
➜ COM4506 ./ftoc
41°F = 0.000000°C
59°F = 0.000000°C
98.5°F = 0.000000°C
100°F = 0.000000°C
➜ COM4506 java FtoC
41°F = 0.000000°C
59°F = 0.000000°C
98.5°F = 0.000000°C
100°F = 0.000000°C
```

## Type ~~Theory~~ Practice

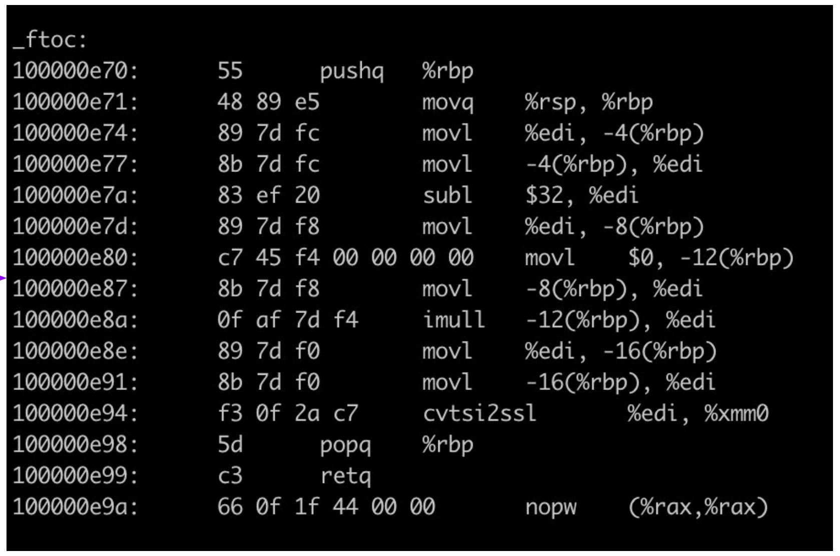


```
➜ COM4506 gcc -o ftoc ftoc.c
ftoc.c:21:35: warning: implicit conversion from 'double' to 'int' changes value from 98.5 to 98 [-Wliteral-conversion]
        printf("98.5°F = %f°C\n", ftoc(98.5));
                                  ~~~~ ^~~~
1 warning generated.
➜ COM4506 javac FtoC.java
FtoC.java:17: error: incompatible types: possible lossy conversion from float to int
                c = ftoc(f);
                         ^
FtoC.java:21: error: incompatible types: possible lossy conversion from double to int
                System.out.printf("98.5°F = %f°C\n", ftoc(98.5));
                                                          ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
2 errors
```

## Compiler Optimisations...

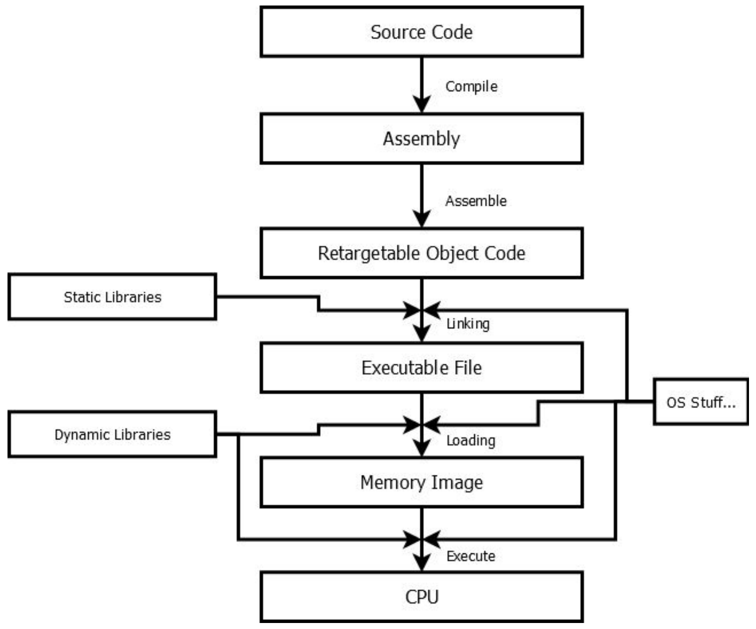


```c
float ftoc(int tempc) {
    int x, y, result;

    x = tempc-32;
    y = 5/9;
    result = x * y;

    return(result);
}
```

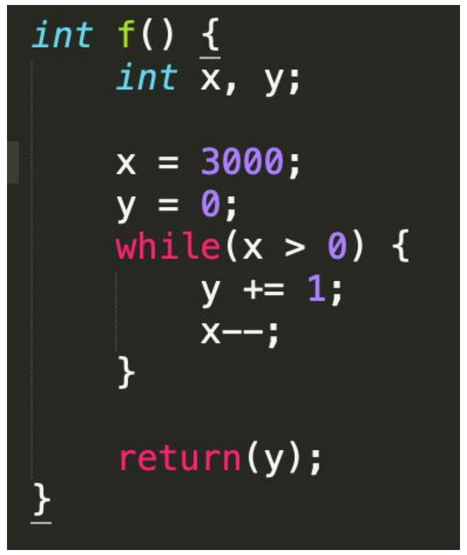


```
_ftoc:
100000e70:      55      pushq   %rbp
100000e71:      48 89 e5        movq    %rsp, %rbp
100000e74:      89 7d fc        movl    %edi, -4(%rbp)
100000e77:      8b 7d fc        movl    -4(%rbp), %edi
100000e7a:      83 ef 20        subl    $32, %edi
100000e7d:      89 7d f8        movl    %edi, -8(%rbp)
100000e80:      c7 45 f4 00 00 00 00    movl    $0, -12(%rbp)
100000e87:      8b 7d f8        movl    -8(%rbp), %edi
100000e8a:      0f af 7d f4     imull   -12(%rbp), %edi
100000e8e:      89 7d f0        movl    %edi, -16(%rbp)
100000e91:      8b 7d f0        movl    -16(%rbp), %edi
100000e94:      f3 0f 2a c7     cvtsi2ssl       %edi, %xmm0
100000e98:      5d      popq    %rbp
100000e99:      c3      retq
100000e9a:      66 0f 1f 44 00 00       nopw    (%rax,%rax)
```

## Compilers

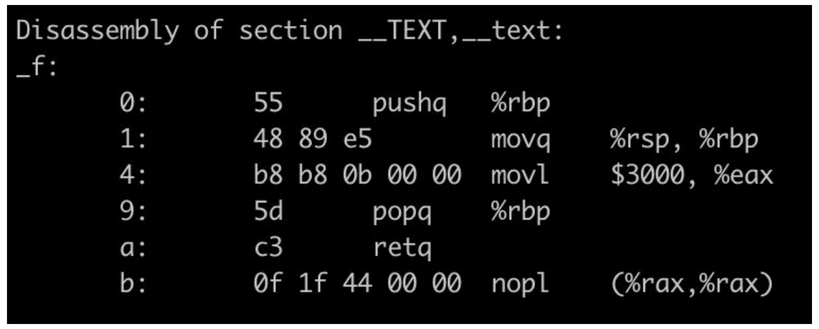


## Compiler Optimisations...



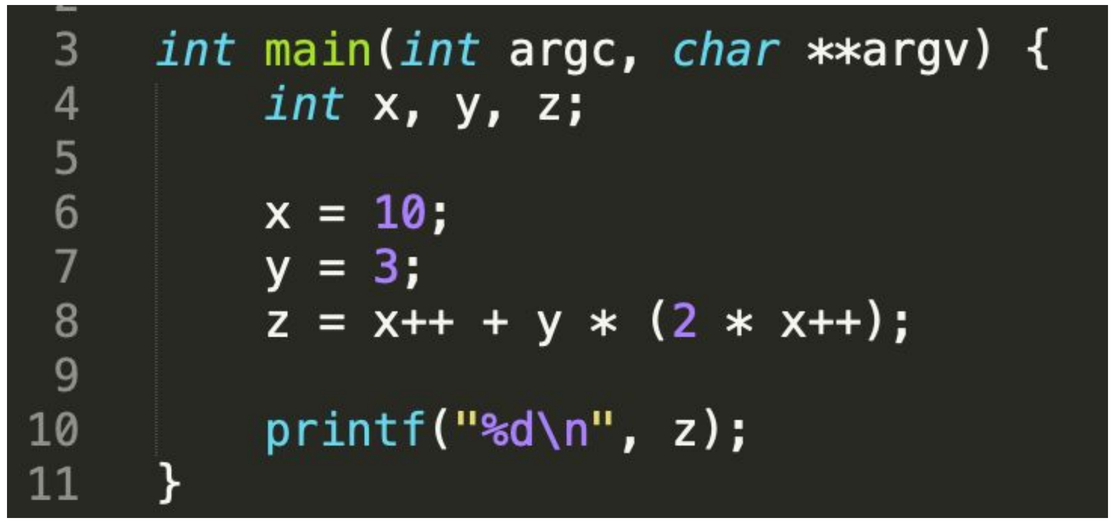
```
int f() {
    int x, y;

    x = 3000;
    y = 0;
    while(x > 0) {
        y += 1;
        x--;
    }

    return(y);
}
```

$\longrightarrow$



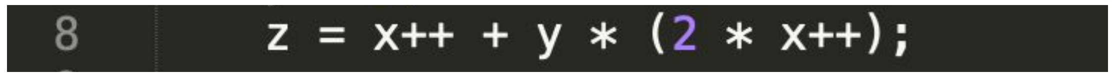
```
Disassembly of section __TEXT,__text:
_f:
       0:       55      pushq   %rbp
       1:       48 89 e5        movq    %rsp, %rbp
       4:       b8 b8 0b 00 00  movl    $3000, %eax
       9:       5d      popq    %rbp
       a:       c3      retq
       b:       0f 1f 44 00 00  nopl    (%rax,%rax)
```

## Compiler Ambiguity

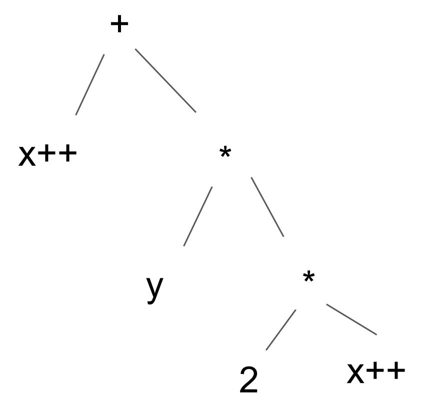


```
int main(int argc, char **argv) {
    int x, y, z;

    x = 10;
    y = 3;
    z = x++ + y * (2 * x++);

    printf("%d\n", z);
}
```

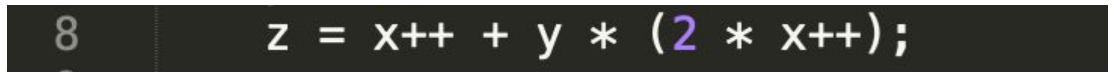
## Compiler Ambiguity



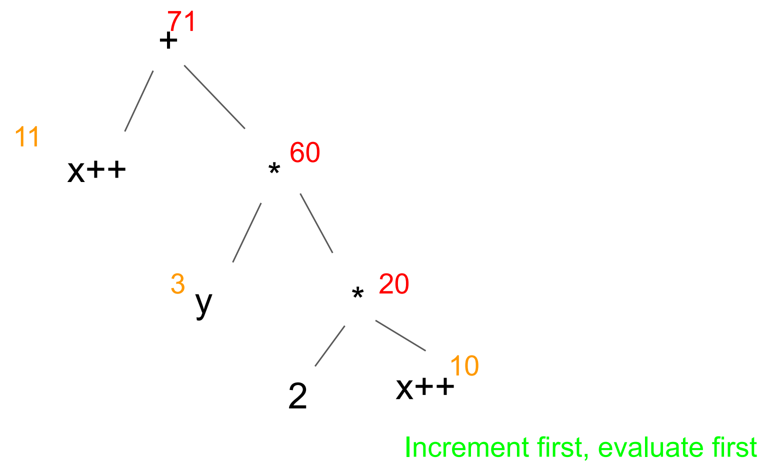
```
8        z = x++ + y * (2 * x++);
```



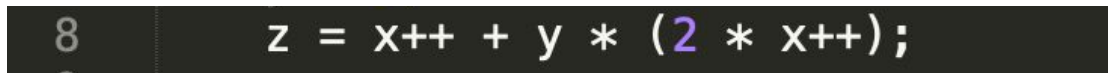
## Compiler Ambiguity



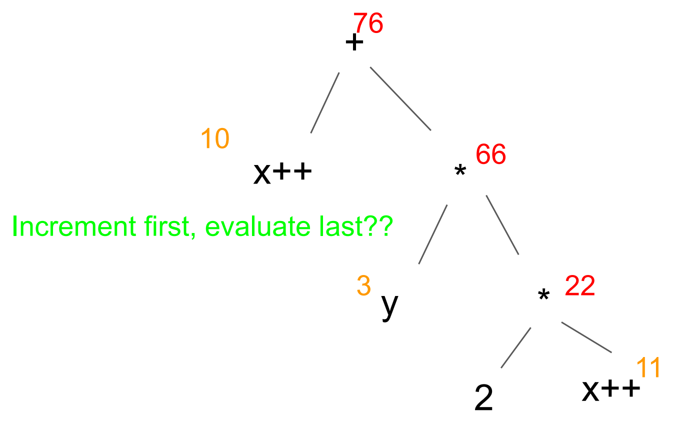
```
8        z = x++ + y * (2 * x++);
```



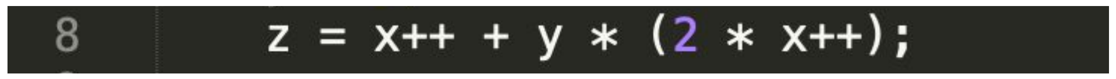
## Compiler Ambiguity



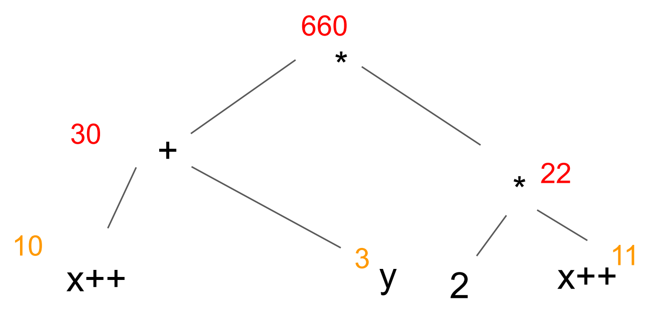
```
z = x++ + y * (2 * x++);
```



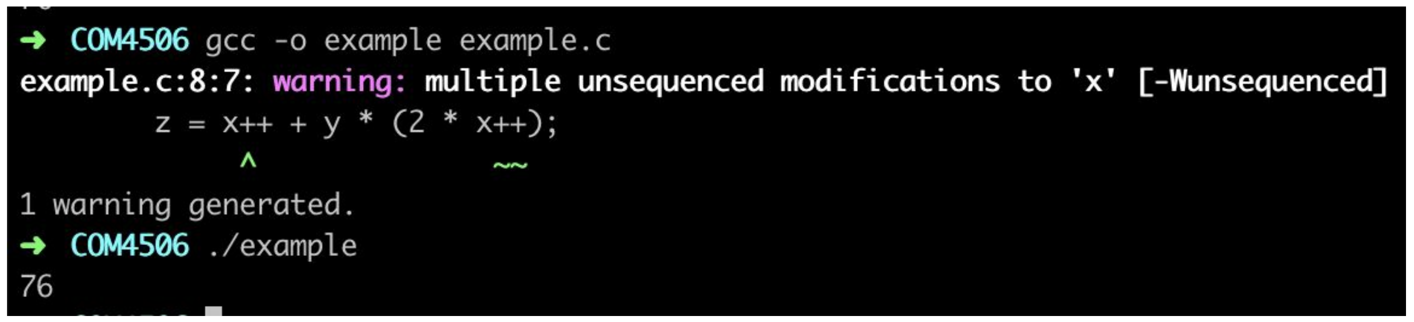
## Compiler Ambiguity



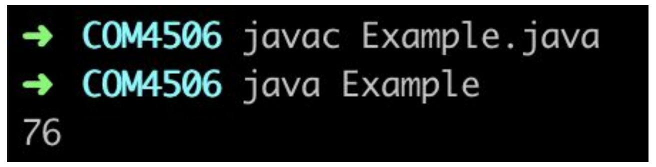
```
z = x++ + y * (2 * x++);
```



## Compiler Ambiguity

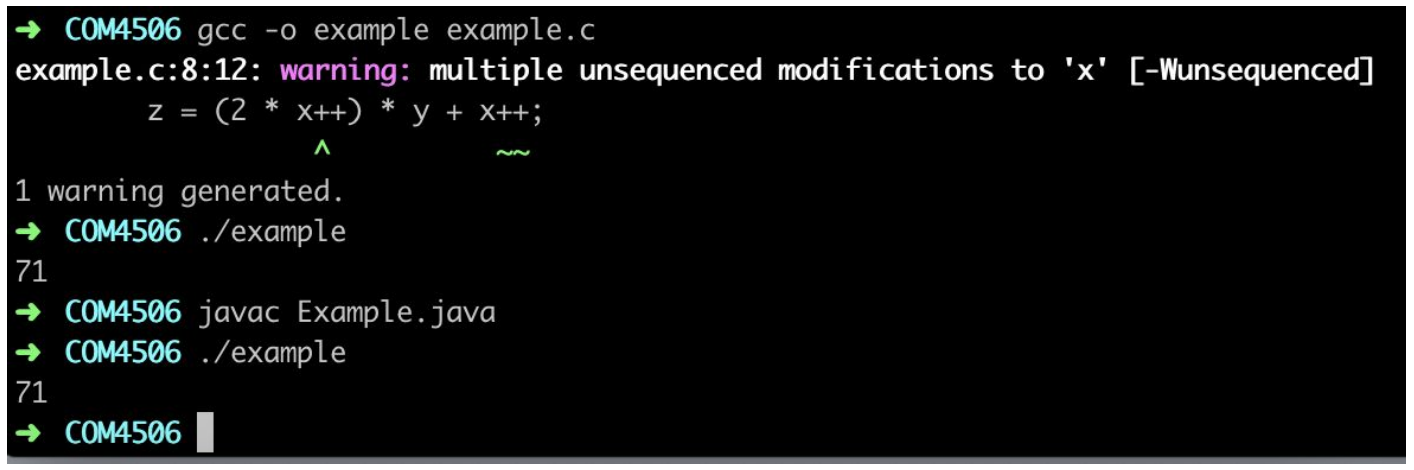


```
➜ COM4506 gcc -o example example.c
example.c:8:7: warning: multiple unsequenced modifications to 'x' [-Wunsequenced]
        z = x++ + y * (2 * x++);
             ^              ~~
1 warning generated.
➜ COM4506 ./example
76
```



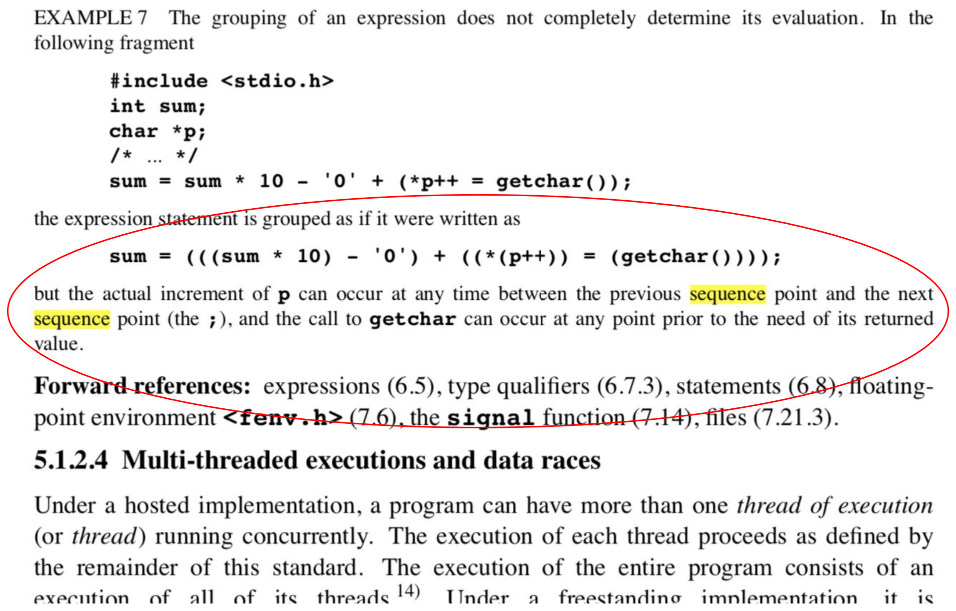
```
➜ COM4506 javac Example.java
➜ COM4506 java Example
76
```

## Compiler Ambiguity



```
➜ COM4506 gcc -o example example.c
example.c:8:12: warning: multiple unsequenced modifications to 'x' [-Wunsequenced]
        z = (2 * x++) * y + x++;
                  ^          ~~
1 warning generated.
➜ COM4506 ./example
71
➜ COM4506 javac Example.java
➜ COM4506 ./example
71
➜ COM4506
```

## Compiler Ambiguity



EXAMPLE 7 The grouping of an expression does not completely determine its evaluation. In the following fragment

```
#include <stdio.h>
int sum;
char *p;
/* ... */
sum = sum * 10 - '0' + (*p++ = getchar());
```

the expression statement is grouped as if it were written as

```
sum = (((sum * 10) - '0') + ((*(p++)) = (getchar())));
```

but the actual increment of **p** can occur at any time between the previous sequence point and the next sequence point (the **;**), and the call to **getchar** can occur at any point prior to the need of its returned value.

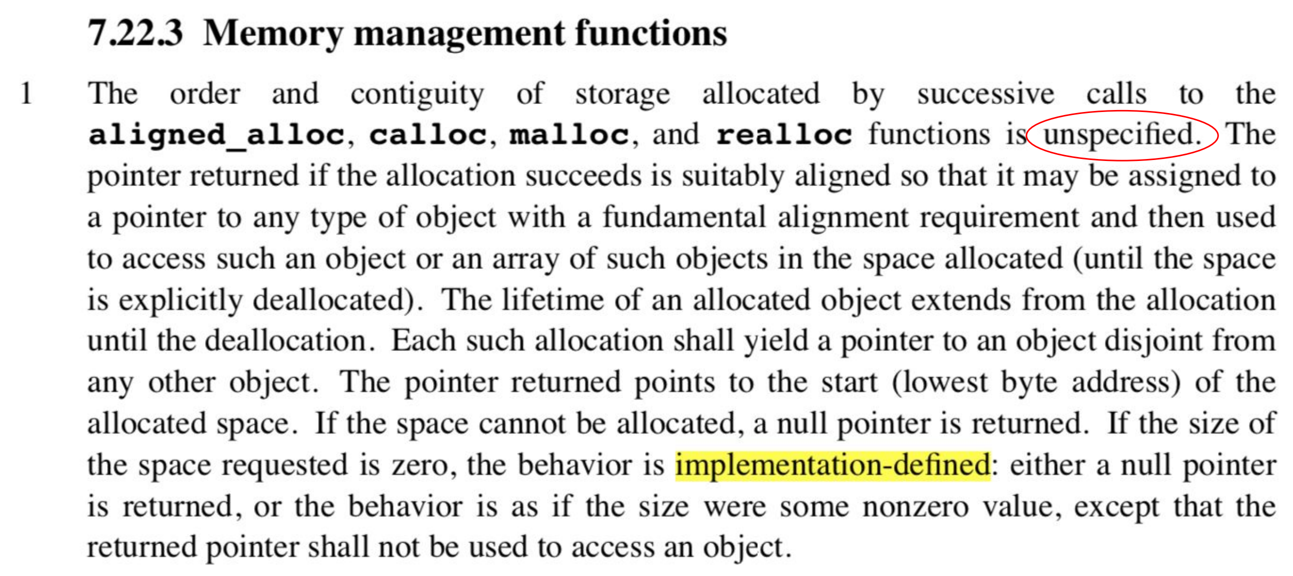
**Forward references:** expressions (6.5), type qualifiers (6.7.3), statements (6.8), floating-point environment **<fenv.h>** (7.6), the **signal** function (7.14), files (7.21.3).

**5.1.2.4 Multi-threaded executions and data races**

Under a hosted implementation, a program can have more than one *thread of execution* (or *thread*) running concurrently. The execution of each thread proceeds as defined by the remainder of this standard. The execution of the entire program consists of an execution of all of its threads.14) Under a freestanding implementation, it is

**ISO/IEC 9899:2011 (E)**

## Compiler Ambiguity



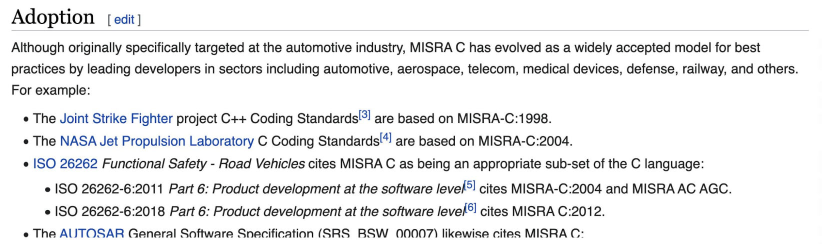
**7.22.3 Memory management functions**

1 The order and contiguity of storage allocated by successive calls to the **aligned_alloc**, **calloc**, **malloc**, and **realloc** functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated). The lifetime of an allocated object extends from the allocation until the deallocation. Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.

## Coding Standards

The **Motor Industry Software Reliability Association** stepped in and made some rules!

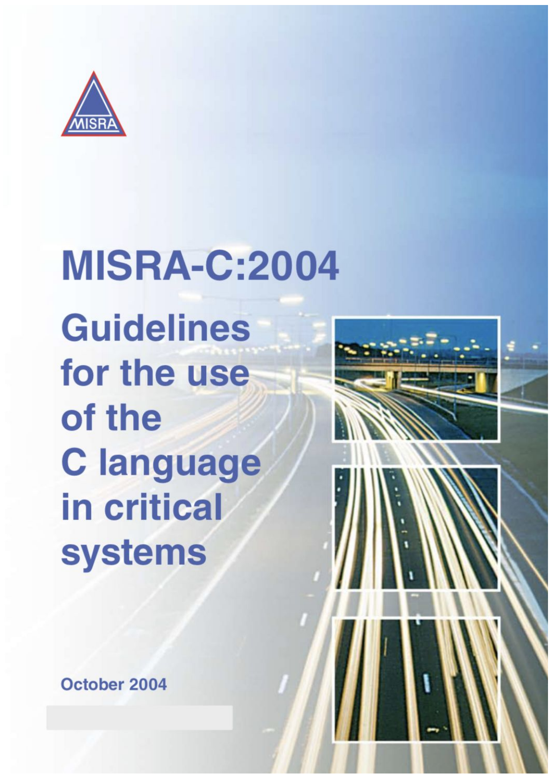
Lots of other people then accepted MISRA's rules:



Adoption [ edit ]

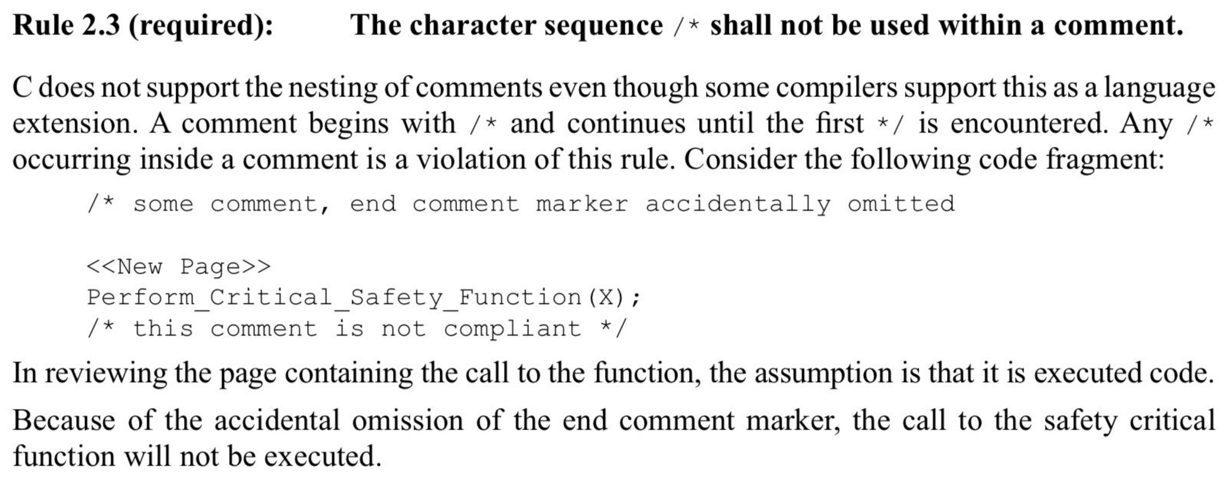
Although originally specifically targeted at the automotive industry, MISRA C has evolved as a widely accepted model for best practices by leading developers in sectors including automotive, aerospace, telecom, medical devices, defense, railway, and others. For example:

- The Joint Strike Fighter project C++ Coding Standards[3] are based on MISRA-C:1998.
- The NASA Jet Propulsion Laboratory C Coding Standards[4] are based on MISRA-C:2004.
- ISO 26262 *Functional Safety - Road Vehicles* cites MISRA C as being an appropriate sub-set of the C language:
  - ISO 26262-6:2011 *Part 6: Product development at the software level*[5] cites MISRA-C:2004 and MISRA AC AGC.
  - ISO 26262-6:2018 *Part 6: Product development at the software level*[6] cites MISRA C:2012.
- The AUTOSAR General Software Specification (SRS_BSW_00007) likewise cites MISRA C:


MISRA
MISRA-C:2004
Guidelines for the use of the C language in critical systems
October 2004

## Coding Standards

*"Soft" rules are also good!*



**Rule 2.3 (required): The character sequence /* shall not be used within a comment.**

C does not support the nesting of comments even though some compilers support this as a language extension. A comment begins with /* and continues until the first */ is encountered. Any /* occurring inside a comment is a violation of this rule. Consider the following code fragment:

```
/* some comment, end comment marker accidentally omitted

<<New Page>>
Perform_Critical_Safety_Function(X);
/* this comment is not compliant */
```

In reviewing the page containing the call to the function, the assumption is that it is executed code.

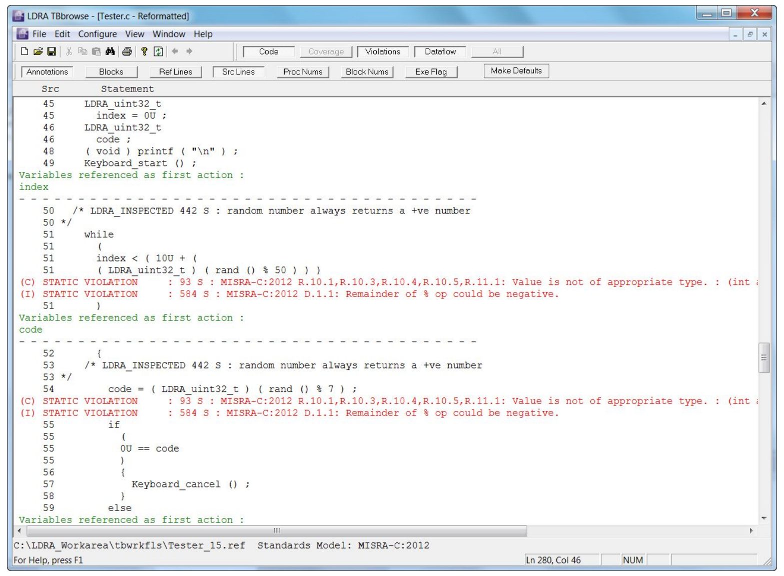
Because of the accidental omission of the end comment marker, the call to the safety critical function will not be executed.

## Coding Standards

Well written standards can be automatically checked!

Standard tools exist for the standard MISRA rules.

The tools can often take additional (well defined) rules too.

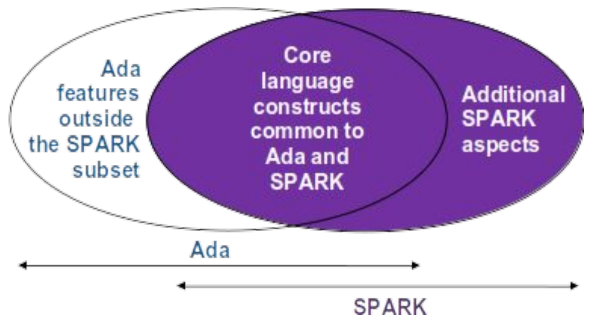

LDRA TBbrowse - [Tester.c - Reformatted]
File Edit Configure View Window Help
Code Coverage Violations Dataflow All
Annotations Blocks Ref Lines Src Lines Proc Nums Block Nums Exe Flag Make Defaults

```
    Src        Statement
    45     LDRA_uint32_t
    45       index = 0U ;
    46     LDRA_uint32_t
    46       code ;
    48     ( void ) printf ( "\n" ) ;
    49     Keyboard_start () ;
Variables referenced as first action :
index
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    50   /* LDRA_INSPECTED 442 S : random number always returns a +ve number
    50 */
    51     while
    51       (
    51       index < ( 10U + (
    51       ( LDRA_uint32_t ) ( rand () % 50 ) ) )
(C) STATIC VIOLATION     : 93 S : MISRA-C:2012 R.10.1,R.10.3,R.10.4,R.10.5,R.11.1: Value is not of appropriate type. : (int
(I) STATIC VIOLATION     : 584 S : MISRA-C:2012 D.1.1: Remainder of % op could be negative.
    51       )
Variables referenced as first action :
code
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    52       {
    53     /* LDRA_INSPECTED 442 S : random number always returns a +ve number
    53 */
    54         code = ( LDRA_uint32_t ) ( rand () % 7 ) ;
(C) STATIC VIOLATION     : 93 S : MISRA-C:2012 R.10.1,R.10.3,R.10.4,R.10.5,R.11.1: Value is not of appropriate type. : (int
(I) STATIC VIOLATION     : 584 S : MISRA-C:2012 D.1.1: Remainder of % op could be negative.
    55         if
    55           (
    55           0U == code
    55           )
    56           {
    57             Keyboard_cancel () ;
    58           }
    59         else
Variables referenced as first action :
C:\LDRA_Workarea\tbwrkfls\Tester_15.ref  Standards Model: MISRA-C:2012
For Help, press F1                                   Ln 280, Col 46    NUM
```

## Coding Standards

SPARK Ada includes a set of rules and limitations of the use of Ada syntax and constructions.

It does this for all the same rules as MISRA C, but also because the various Static Analysis couldn't work if the code was ambigious!


Ada features outside the SPARK subset
Core language constructs common to Ada and SPARK
Additional SPARK aspects
Ada
SPARK

## Summary

- Source code isn't the final product!
- The various steps in compiling and assembling a program can *interpret* the source code in various different ways.
- This makes Static Analysis, Formal Methods, and even just *reading the code* irrelevant.
- Coding Standards exist to reduce the ambiguity in Safety Critical environments (and to make *reading the code* more effective!)