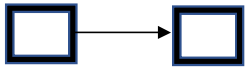
Reducing Risk

Klaus Havelund
NASA Jet Propulsion Laboratory
October 17, 2023

Jet Propulsion Laboratory
California Institute of Technology

Our weapons

Modular design



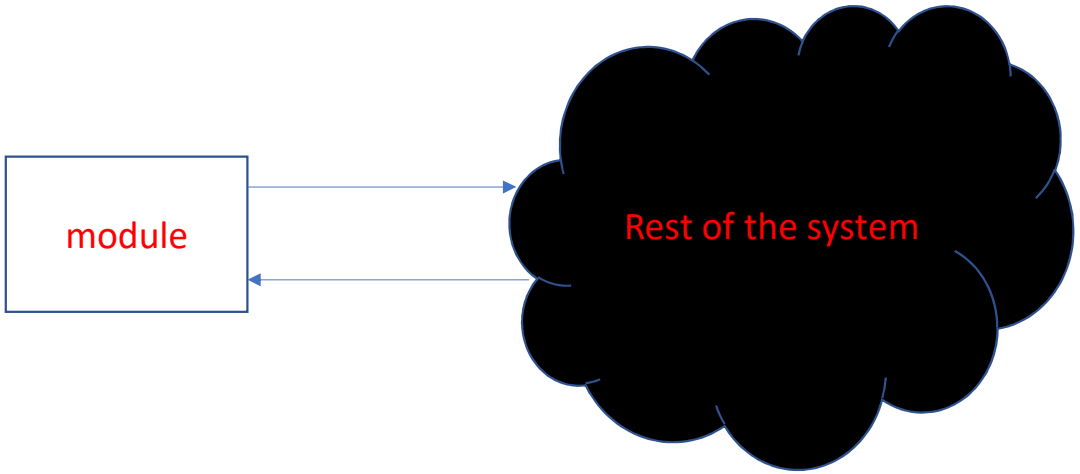
Be aware of bounds

Apply defensive coding

Follow coding standard

Modular design

- A key mechanism for building reliable code is modularity
- A module is a logical unit with a well defined interface
- An interface is a contract between the module and the rest



Design interface **carefully**

Assume the worst
Guarantee the best

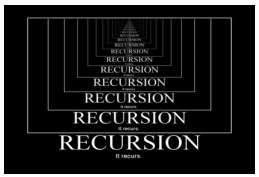
Awareness of bounds

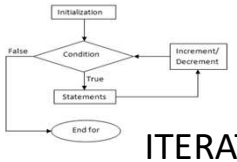
Be aware of bounds when dealing with:

- Numbers (precision, wrap-around, …)

$$\mathbb{Z} \to \texttt{int}$$
$$\mathbb{R} \to \texttt{float}$$

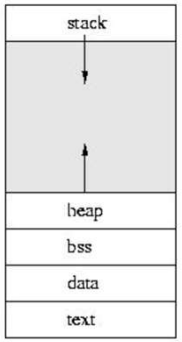
- Loops (termination, complexity, …)





ITERATION

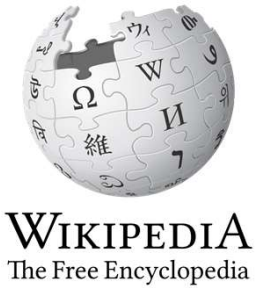
- Memory (aliasing, null pointers, dangling pointers, finiteness, …)



Defensive programming

Defensive programming is an approach to improve software:

- Reducing the number of **software bugs**
- Making the software behave in a **predictable** manner despite unexpected inputs or user actions
- Making the source code **comprehensible**



Assertions

"In order that the man who checks may not have too difficult a task the programmer should make a number of definite *assertions* which can be checked individually, and from which the correctness of the whole program easily follows."

Alan Turing, *Checking a large routine*, in Conf. on High Speed Automatic Calculating Machines, Cambridge, UK, 1949, pp. 67-69.

source: L.A. Clarke, D.S. Rosenblum, *A historical perspective on runtime assertion checking in software development*, ACM SIGSOFT Software Eng. Notes, Vol. 31, Issue 3, 2006

```
...
assert(y != 0);
r = x / y;
...
```

Check or assert that

- parameters are not null and have the right values
- calculations do not under or overflow
- that loops terminate
- called functions do not return error values

Assertions should, however, only be used in "cannot happen" cases.

Assertion density vs. defect density

Assessing the Relationship between Software Assertions and Faults: An Empirical Investigation

Gunnar Kudrjavets [1], Nachiappan Nagappan [2], Thomas Ball [2]
[1] *Microsoft Corporation, Redmond, WA 98052*
[2] *Microsoft Research, Redmond, WA 98052*
{gunnarku, nachin, tball} @microsoft.com

Abstract

The use of assertions in software development is thought to help produce quality software. Unfortunately, there is scant empirical evidence in commercial software systems for this argument to date. This paper presents an empirical case study of two commercial software components at Microsoft Corporation. The developers of these components systematically employed assertions, which allowed us to investigate the relationship between software assertions and code quality. We also compare the efficacy of assertions against that of popular bug finding techniques like source code static analysis tools. We observe from our case study that with an increase in the assertion density in a file there is a statistically significant decrease in fault density. Further, the usage of software assertions in these components found a large percentage of the faults in the bug database.

code and add assertions. This causes the analysis to be skewed as size plays a huge factor in these results.

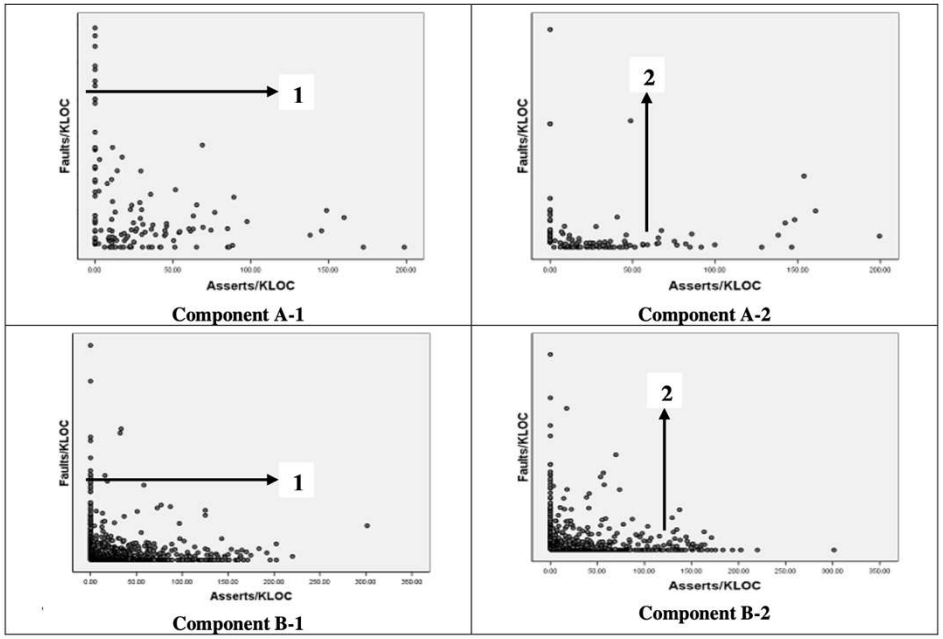
Assertions have been studied for many years [5, 11, 13]. In 1998 there was an investigation [14] performed for the NIST (National Institute for Standards and Technology) in order to evaluate software assertions. This report raised the following generalized observation regarding assertions:

> ... *"Interest in using assertions continues to grow, but many practitioners are not sure how to begin using assertions or how to convince their management that assertions should become a standard part of their testing process."*

In this paper, we focus on addressing the above underlined statement. There have been several research papers on the formal analysis of assertions in code, their utility in contracts, investigations on the placement of assertions etc. but none on the practical implications of using assertions in commercial code bases. This paper attempts to address this question

17th International Symposium on Software Reliability Engineering (ISSRE'06)
0-7695-2684-5/06 $20.00 © 2006 IEEE

We observe from our case study that with an increase in the assertion density in a file there is a statistically significant decrease in fault density.



Pre and post conditions in C as assertions

```c
int32_t mid(int32_t left, int32_t right)
{
    /* Precondition: */
    assert (0 <= left && left < right);

    /* Calculation: */

    int32_t m = left + ((right - left) / 2);

    /* Postcondition: */
    assert (left <= m && m <= right);

    return m;
}
```

Avoid loosely defined C constructs

18 pages of loosely defined C constructs in Appendix A of JPL std.

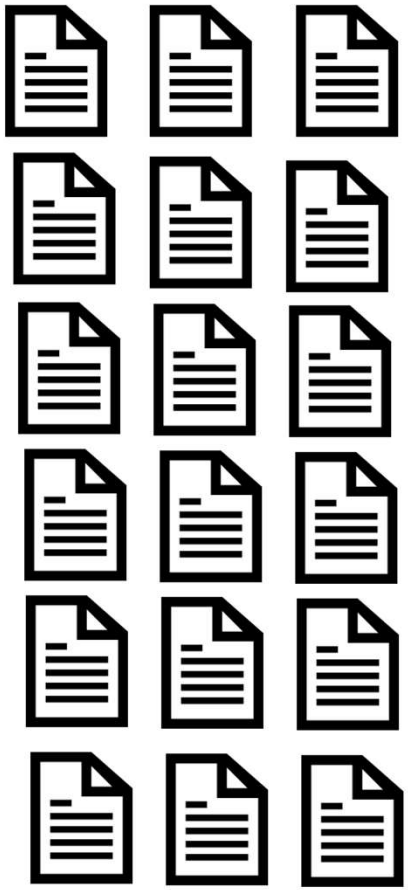
defensive coding: secure language compliance

avoid unspecified, undefined, or implementation defined code

- **Unspecified**
  The compiler has to make a choice from a finite set of alternatives, but that choice is not in general predictable by the programmer.
  Example: *the order in which sub-expressions are evaluated in a C expression.*

- **Implementation defined**
  The compiler has to make a choice, and the choice required to be documented and available to the programmer,
  Example*: the range of C variables of type short, int, and long*.

- **Undefined**
  The language definition gives no indication of what behavior can be expected from a program – it may be some form of catastrophic failure (a 'crash') or continued execution with arbitrary data.
  Example: *dereferencing a null pointer in C*.



JPL Institutional Coding Standard for the C Programming Language

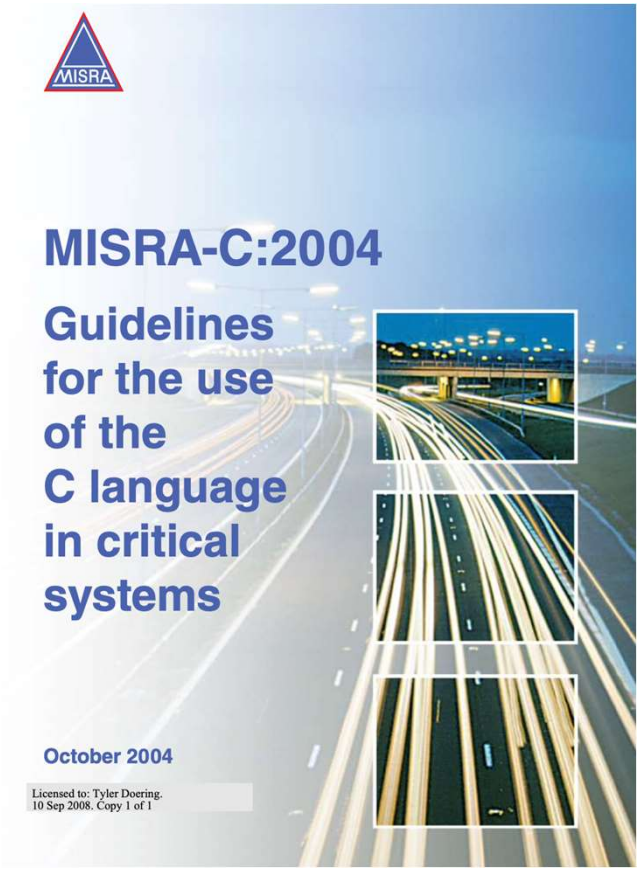
Version: 2.0

Date: May 22, 2023

1 Language Compliance	
1	Do not stray outside the language definition.
2	Compile with all warnings enabled; use static source code analyzers.
2 Predictable Execution	
3	Use verifiable loop bounds for all loops meant to be terminating.
4	Do not use direct or indirect recursion.
5	Do not use dynamic memory allocation after task initialization.
*6	Use IPC messages for task communication.
7	Do not use task delays for task synchronization.
*8	Explicitly transfer write-permission (ownership) for shared data objects.
9	Place restrictions on the use of semaphores and locks.
10	Use memory protection, safety margins, barrier patterns
SC1	Use atomic read and write operations from stdatomic.h for shared data
11	Do not use goto, setjmp or longjmp.
12	Do not use selective value assignments to elements of an enum list.
3 Defensive Coding	
13	Declare data objects at smallest possible level of scope.
14	Check the return value of non-void functions, or explicitly cast to (void).
15	Check the validity of values passed to functions.
16	Use static and dynamic assertions as sanity checks.
*17	Use U32, I16, etc instead of predefined C data types such as int, short, etc.
18	Make the order of evaluation in compound expressions explicit.
19	Do not use expressions with side-effects.
SC2	Use only safe string functions (avoid strcpy etc.)
SC3	Do not use variable-length arrays as parameters to functions
4 Code Clarity	
20	Make only very limited use of the C pre-processor.
21	Do not define macros within a function or a block.
22	Do not undefine or redefine macros.
23	Place #else, #elif, and #endif in the same file as the matching #if or #ifdef.
*24	Place no more than one statement or declaration per line of text.
*25	Use short functions with a limited number of parameters.
*26	Use no more than two levels of indirection per declaration.
*27	Use no more than two levels of dereferencing per object reference.
*28	Do not hide dereference operations inside macros or pointers in typedefs.
*29	Do not use non-constant function pointers.
30	Do not cast function pointers into other types.
31	Do not place code or declarations before an #include directive.
5&6 Appendix A:	
73+16*	All remaining MISRA *shall* (73x) & *should* rules (16x)

*) All rules are *shall* rules, except those marked with an asterix.

Influenced by the Misra C standard



The power of 10 – rules to remember

defensive coding: coding standards

follow a machine-checkable, risk-based, standard

1. *Restrict to simple control flow constructs*
2. *Do not use recursion and give all loops a fixed upper-bound*
3. *Do not use dynamic memory allocation after initialization*
4. *Limit functions to no more than ~60 lines of text*
5. *Target an average assertion density of 2% per module*
6. *Declare data objects at the smallest possible level of scope*
7. *Check the return value of non-void functions; check the validity of parameters*
8. *Limit the use of the preprocessor to file inclusion and simple macros*
9. *Limit the use of pointers. Use no more than 2 level of dereferencing*
10. *Compile with all warnings enabled, and use source code analyzers*

Gerard Holzmann


The Power of 10: Rules for Developing Safety-Critical Code

IEEE Computer 39(6) 95-97 (2006)

http://spinroot.com/p10/

JPL Institutional Coding Standard for the C++ Programming Language

JPL DOCID X-XXXXX
Version 0.1

September 13, 2023

Part 1: Basic Rules	
1	All code **shall** conform to ISO/IEC 14882:2014 and **shall** not use deprecated features [A1-1-1].
2	The JPL C coding standard **shall** be followed.
3	A function definition **shall** be placed in a class definition only if (1) it is intended to be inlined or (2) it is a member function template or (3) it is a member function of a class template [A3-1-5].
4	The only null pointer value **shall** be `nullptr` [A4-10-11].
5	A lambda expression object **shall** not outlive any of its reference-captured objects [A5-1-4].
6	No pointer used in a pointer arithmetic expression or accessed through an array index **shall** have a base class as its element type [A5-0-4].
*7	Casts from a base class to a derived class **should** not be performed on polymorphic types [M5-2-3].
8	(a) C-style casts (*type*) *expression* **shall** not be used, except when *type* is `void`. (b) Function-style casts *type* (*expression*) **shall** not be used [M5-2-4].
9	Neither `reinterpret_cast` nor `const_cast` **shall** be used [A5-2-3, A5-2-4].
10	None of the following operators **shall** be overloaded: the comma operator, the `&&` operator, the `||` operator, and the unary `&` operator [M5-2-11, M5-3-3].
11	Pointers to incomplete class types **shall** not be deleted [A5-3-3].
*12	Pointers to members **should** not be used. If pointers to members are used, then a pointer to a member virtual function **shall** only be tested for equality with the null-pointer constant [A5-10-1].
13	The semantic equivalence between a binary operator and its assignment operator form **shall** be preserved [M5-17-1].
*14	(a) Using directives `using namespace` *name* **should** not be used [M7-3-4]. (b) No header file **shall** contain (1) an unnamed namespace `namespace { ... }`; or (2) a using declaration `using` *name* that appears outside any class or function scope [M7-3-3, M7-3-6].
*15	No two namespaces on the same path in the namespace tree **should** have the same unqualified name.
16	The global namespace **shall** contain only the `main` function, namespace declarations, and `extern "C"` declarations [M7-3-1].
17	References **shall** be used instead of pointers whenever possible [A8-4-10].
18	In the initialization list of a constructor, the order of initialization **shall** be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition [A8-5-1].
19	Member functions **shall** not return non-`const` pointers or references to private or protected data owned by the class [A9-3-1].
20	If a member function can be made `constexpr` then it **shall** be made `constexpr`; otherwise if it can be made `static` then it **shall** be made `static`; otherwise if it can be made `const` then it **shall** be made `const` [M9-3-3].
21	No class **shall** be derived from more than one base class that is not an interface class [A10-1-1].
22	No member function **shall** be redefined in a derived class, unless it is a virtual function [A10-2-1].
23	Each overriding virtual function **shall** be declared with `override` or `final` [A10-3-1].
24	Virtual functions **shall** not be introduced in a final class [A10-3-3].
25	No user-defined operator **shall** be virtual.
26	Friend declarations **shall** not be used [A11-3-1].
27	(a) If a class declares a copy constructor, a copy assignment operator, or a destructor, either via `=default`, `=delete`, or via a user-provided declaration, then all others of these three special member functions **shall** be declared as well. (b) In non-flight code that uses rvalue references and move semantics, if a class declares any of these three special member functions, or a move constructor, or a move assignment operator, then all others of these five special member functions **shall** be declared as well [A12-0-1].
28	Each constructor **shall** explicitly initialize all virtual base classes, all direct non-virtual base classes, and all non-static data members [A12-1-1].
29	All constructors that are callable with a single argument of fundamental type **shall** be declared `explicit` [A12-1-4].
30	A destructor of a base class **shall** be public virtual, public `override`, or protected non-virtual [A12-4-1].
*31	If a public destructor of a class is non-virtual, then the class **should** be declared `final` [A12-4-2].
32	In each constructor, all non-static class data members **shall** be initialized using member initializers [A12-6-1].
33	Each copy assignment operator and move assignment operator **shall** handle self-assignment [A12-8-5].
34	In any base class, copy and move constructors and copy assignment and move assignment operators **shall** be declared protected or defined `=delete` [A12-8-6].
35	The arguments and return values of a user-defined operator **shall** follow the pattern established by the corresponding built-in operator [A13-2-1, A13-2-2, A13-2-3].
36	Reserved identifiers, macros, and functions in the C++ standard library **shall** not be defined, redefined, or undefined [A17-0-1].
*37	C-style arrays **should** not be used [A18-1-1]. If C-style arrays are used, then they **shall** be encapsulated inside class objects that store the array size and provide bounds checking of indices.
*38	Dynamic memory allocation **should** not be used after initialization. If dynamic memory allocation is used after initialization, then operators `new` and `delete` **shall** not be called explicitly [A18-5-2].
39	C library symbols **shall** be accessed via C++ library headers [A18-0-1].
40	When converting a pointer `p` to a type `T*`, `p` **shall** point to a valid memory region that is aligned for type `T` and is large enough to hold an object of type `T` [A18-5-10].
41	(a) If dynamic memory is used after initialization, then the following rules apply: • `std::unique_ptr` **shall** be used to represent exclusive ownership [A20-8-2]. • `std::shared_ptr` **shall** be used to represent shared ownership [A20-8-3]. • `std::weak_ptr` **shall** be used to represent temporary shared ownership [A20-8-7]. (b) If dynamic memory is not used after initialization, then the `std` smart pointers **shall** not be used.
*42	All uses of C-style strings, except for string literals, and all calls to C string library functions **should** be encapsulated inside class objects.

Part 2: Flight Software and Mission-Critical Code	
43	The standard library's I/O streams **shall** not be used.
44	Multiple and virtual inheritance **shall** not be used.
45	Run-time type information (RTTI) **shall** not be used.
46	Exceptions **shall** not be used.
47	Only simple templates **shall** be used.
48	Dynamic memory allocation **shall** not be used after initialization.
49	The `std` container classes **shall** not be used. Services from `std` **shall** be used only if they do not allocate memory.
50	`std::string` **shall** not be used
51	rvalue references and move semantics **shall** not be used.

An example rule

Rule 6 (Pointers)

No pointer used in a pointer arithmetic expression or accessed through an array index **shall** have a base class as its element type [A5-0-4].

A base class is a class that is the base of at least one derived class. A pointer to a base class type may provide incorrect size information for pointer arithmetic, including array indexing. For example, this code is incorrect:

```
class A { ... };
class B : public A { ... };
B b_array[10];
// a_ptr points to element zero of b_array
A* a_ptr = &b_array[0];
// Undefined behavior if A and B have different sizes
aptr_[1] = B();
```

Based on the Misra & Autosar C++ standards

AUTOSAR

Guidelines for the use of the C++14 language in critical and safety-related systems
AUTOSAR AP Release 18-03

Document Title	Guidelines for the use of the C++14 language in critical and safety-related systems
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	839

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	18-03

Document Change History

Date	Release	Changed by	Description
2018-03-29	18-03	AUTOSAR Release Management	• New rules resulting from the analysis of JSF, HIC, CERT, C++ Core Guideline • Improvements of already existing rules, more details in the Changelog (D.2) • Covered smart pointers usage • Reworked checked/unchecked exception definitions and rules
2017-10-27	17-10	AUTOSAR Release Management	• Updated traceability for HIC, CERT, C++ Core Guideline • Partially included MISRA review of the 2017-03 release • Changes and fixes for existing rules, more details in the Changelog (D.1)
2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Other standards …

Carnegie Mellon University
Software Engineering Institute

Confluence Spaces Search Log in

SEI CERT C Coding Standard

Pages
Boards

SPACE SHORTCUTS
Dashboard
Secure Coding Home
Android
C
C++
Java
Perl

PAGE TREE
1 Front Matter
2 Rules
3 Recommendations
4 Back Matter
5 Admin
Wiki Contents
CERT manifest files

Pages

SEI CERT C Coding Standard

Created by Admin, last modified by David Svoboda on Dec 05, 2018

Account disable announcement

As of Friday, September 8, 2023, the SEI Secure Coding Wiki no longer provides the ability to sign up for new user accounts. In addition, all accounts th
page content will be disabled (accounts that have had recent activity will not be disabled).

If you have a specific need to keep your account active, or you feel your account has been disabled in error, please submit a message.

Access to view pages will remain for the public users without an account.

Active Account Requirements

- Actively contributing content to the wiki (e.g. page edits, comments)
- Demonstrated on-going need to access information contained within the external wiki

The C rules and recommendations in this wiki are a work in progress and reflect the current thinking of the secure coding community. Because this is a
As rules and recommendations mature, they are published in report or book form as official releases. These releases are issued as dictated by the need

Create a sign-in account if you want to comment on existing content. If you wish to be more involved and directly edit content on the site, you still need

Front Matter

Introduction

Rules

Rule 01. Preprocessor (PRE)
Rule 02. Declarations and Initialization (DCL)
Rule 03. Expressions (EXP)
Rule 04. Integers (INT)
Rule 05. Floating Point (FLP)
Rule 06. Arrays (ARR)
Rule 07. Characters and Strings (STR)
Rule 08. Memory Management (MEM)
Rule 09. Input Output (FIO)
Rule 10. Environment (ENV)
Rule 11. Signals (SIG)
Rule 12. Error Handling (ERR)
Rule 13. Application Programming Interfaces (API)
Rule 14. Concurrency (CON)
Rule 48. Miscellaneous (MSC)
Rule 50. POSIX (POS)
Rule 51. Microsoft Windows (WIN)

Recommendations

Rec. 01. Preprocessor (PRE)
Rec. 02. Declarations and Initialization (DCL)
Rec. 03. Expressions (EXP)
Rec. 04. Integers (INT)
Rec. 05. Floating Point (FLP)
Rec. 06. Arrays (ARR)
Rec. 07. Characters and Strings (STR)
Rec. 08. Memory Management (MEM)
Rec. 09. Input Output (FIO)
Rec. 10. Environment (ENV)
Rec. 11. Signals (SIG)
Rec. 12. Error Handling (ERR)
Rec. 13. Application Programming Interfaces (API)
Rec. 14. Concurrency (CON)
Rec. 48. Miscellaneous (MSC)
Rec. 50. POSIX (POS)
Rec. 51. Microsoft Windows (WIN)

Back Matter

AA. Bibliography
BB. Definitions
CC. Undefined Behavior
DD. Unspecified Behavior
EE. Analyzers
FF. Related Guidelines
GG. Risk Assessments

want to request privileges to participate in standards development

SEI CERT C++ Coding Standard

Rules for Developing Safe, Reliable, and Secure Systems in C++

2016 Edition

Aaron Ballman

CERT | Software Engineering Institute Carnegie Mellon University

JOINT STRIKE FIGHTER

AIR VEHICLE

C++ CODING STANDARDS

FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM

Document Number 2RDU00001 Rev C

December 2005

PRQA Programming Research

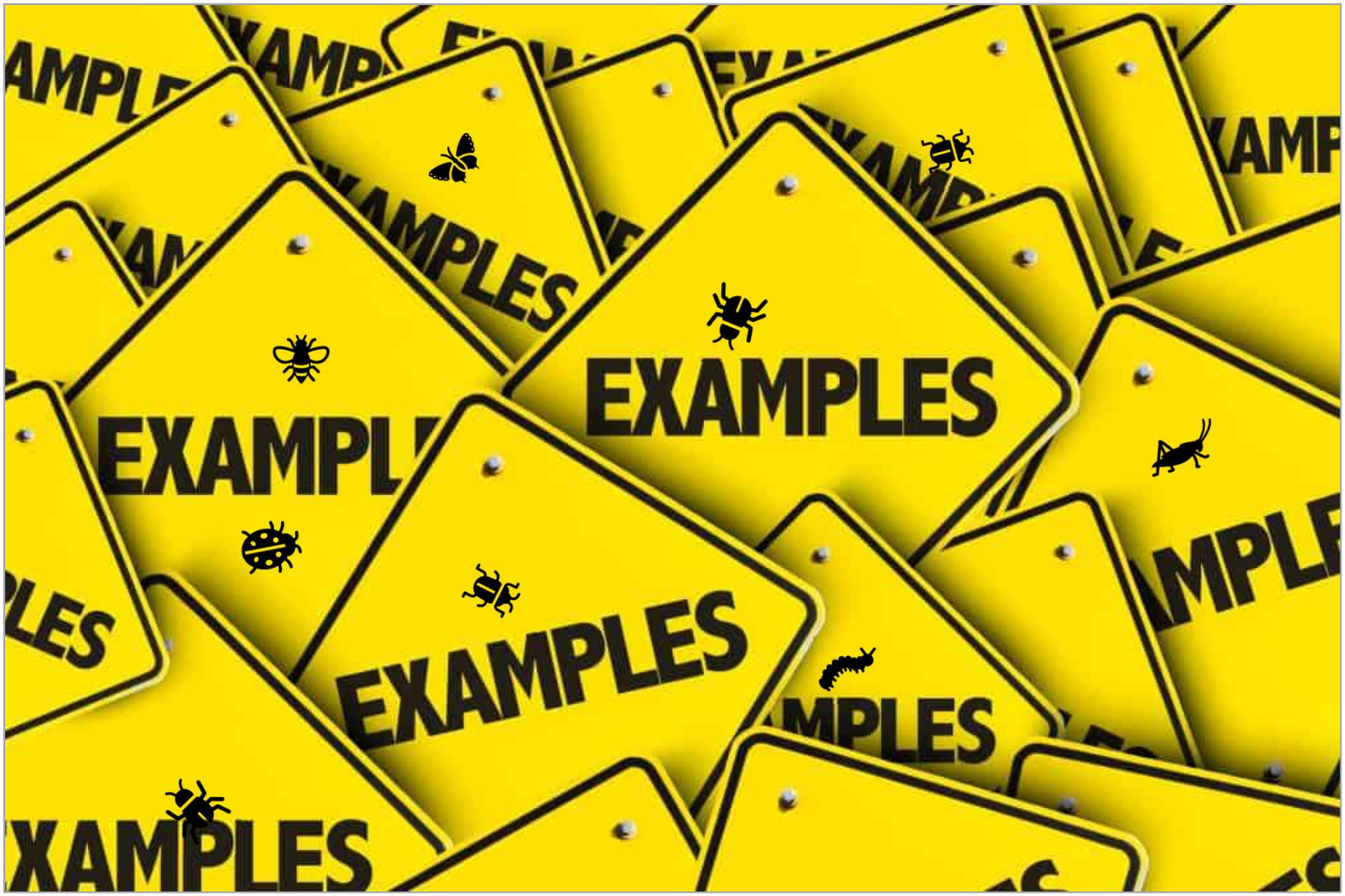
High Integrity C++

Coding Standard
Version 4.0

www.codingstandard.com

3 October 2013

Programming Research Ltd
Mark House - 9/11 Queens Road - Hersham - Surrey KT12 5LU - United Kingdom
Tel: +44 (0) 1932 888 080 - Fax: +44 (0) 1932 888 081
info@programmingresearch.com - www.programmingresearch.com
Registered in England No. 2844401


EXAMPLES
EXAMPLES
EXAMPLES

Macros

Macro traps

defensive coding: testable code

example: make very limited use of the C preprocessor

Q1: will this code trigger
a compilation error (and if so, where)?

```
#define A

int
main(void)
{
#ifndef A
        printf("hello world\n");
#else1
        goto *main();
#endif
        return 0;
}
```

```
$ gcc –Wall –pedantic gobble.c
$ ./a.out
$
```

Q2: how many different ways can this
code be compiled (i.e., how many
ways would it need to be tested)?

```
#if (a>0)
….
        #ifdef X
        ….
                #ifndef Y
                …
                        #if b
                        …
                        #endif
                …
                #endif
        …
        #endif
…
#endif
```

A: 16 different ways (2^4)

fewer tests would suffice,
but that relies on the
programmer being able
to calculate all the options

Avoid side effects in arguments to unsafe macros

```c
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int32_t n) {
    int32_t m = ABS( x: ++n);
    printf("m = %d\n", m);
}
```

```c
#define ABS(x) (((x) < 0) ? -(x) : (x))

void func(int32_t n) {
    ++n;
    int32_t m = ABS(n);
    printf("m = %d\n", m);
}
```

Numbers

Use size specific macros for basic types

```
typedef          char   char_t;
typedef signed   char   int8_t;
typedef signed   short  int16_t;
typedef signed   int    int32_t;
typedef signed   long   int64_t;
typedef unsigned char   uint8_t;
typedef unsigned short  uint16_t;
typedef unsigned int    uint32_t;
typedef unsigned long   uint64_t;
typedef          float  float32_t;
typedef          double float64_t;
typedef long     double float128_t;
```

For a 32-bit integer machine
Misra rule 6.3

Factorial

Let us start a well known concept from mathematics - the factorial function

$$\begin{aligned} n! &= n \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 3 \times 2 \times 1 \\ &= n \times (n-1)! \end{aligned}$$

Reducing software risk

We would like to use math to reason about the function

In the **PVS** theorem prover (https://pvs.csl.sri.com):

```
factorial(n): RECURSIVE posnat =
  IF n = 0 THEN 1
  ELSE n*factorial(n-1)
ENDIDF
MEASURE n
```

Problem: Use induction to prove that the factorial of any number strictly greater than 1 is even. Lemma `factorial_even` specifies this statement in PVS. The predicate `even?` is defined in the PVS prelude library as follows.

```
even?(i): bool = EXISTS j: i = j * 2
```

```
factorial_even : LEMMA
  FORALL (n:above(1)) :
    even?(factorial(n))
```

No stack overflow

No integer overflow

Just pure math

Factorial in Python

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

Factorial in Python of 997

```
print(fac(998))
```

```
Traceback (most recent call last):
  File "…/factorial.py", line 12, in <module>
    print(fac(998))
  File "…/ factorial.py", line 6, in fac
    return n * fac(n-1)
  File "…/ factorial.py", line 6, in fac
    return n * fac(n-1)
  File "…/ factorial.py", line 6, in fac
    return n * fac(n-1)
  [Previous line repeated 995 more times]
  File "…/ factorial.py", line 3, in fac
    if n == 0:
RecursionError: maximum recursion depth exceeded in comparison
```

```
print(fact(997))
```

4035972446164539023429265276529074021109033524613938914303079737351966319010684237263758833853587102137006631984661977193944113181265519616824478
0819892496805164333079279054597565865236698495341010299472919339792786270786066331221142813965733919921028883982924596589308458677218884794980135
4437616450752245066665598898009417557796737695167521343249479413631414534202184726421479392615730781173164526393982880263279118925406206180689438
6833086446963341339551842355400772424511659038110182771983218003159582798999413815661514909173799810545498524832232927524389811980802708882543991
9757453646057047343037159587240316948675716615429494125804531124138293083686200505239196747842903536298319905066323058686625761240280494240344233
1663944341683350732204123565349869446216232111598995678724462182568501131746383857706790400107507266739002631612931124112227909672935742104968533
2780747960003358559304320605174471952264361873012311950910589161415000050344865688475996490049406776931852322183786594448546457039088249340151445
5003570460531797737862031185509535676948889221713020001125049115164153109012008376515922196975531443788020928170857449369384012533872207051402936
2985801732618715060934298236579096167095859504053310608725711198457200226544350445941157734863407428532431126485686678788466148681975019174010453
2976390040068195207044638407735286912244552652299854897643569096753838002459692766798724077579242119184881795985303822666475479072261654798029765
4789693965688881325682653906791569527887851625739692098351138902956310111232537239546473978314336136287987257855014757116813608339135424273514280
3988735616917749898060073075542403509536490539404444972668319521415425667918323473675966566332390993259591959049424070380861864682206986463729281
5573387474665466278592062875719964916067979790641428194695892008126790265612881240871363598309598670345134414348502128648186015045295201958285280
4560086942064644286372048541496836531269052383502650854577265971210516113769359526291937135884001947338380202834453118167941771656301350124247729
1139042422814166369601152223293596957527530934652046662174154235850073391729650007182794396630407081318880947107940245036774649857429379220776637
3568902115965400093490922559880479094175947783757057238419181676630262770090339396547856717150451221853157302493936160447379021701169807360000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

```
print(fact(997) / fact(996))
```

997.0

It seems as if Python is doing the right thing

*recall that: fac(997) = 997 * fac(996)*

Factorial in C of 13 wraps around

bounds on data: computing factorials

a simple recurrence relation

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n\,(n-1)! & \text{if } n > 0 \end{cases} \quad \forall n \in \mathbb{N}.$$

```
#include <stdio.h>
#include <assert.h>

int fact(int n) { return (n == 0) ? 1 : n*fact(n-1); }

int
main(int argc, char *argv[])
{       int n;

        if (argc != 2)
        {  printf("usage: fact N\n");
           exit(1);
        }

        n = atoi(argv[1]);
        assert(n >= 0);

        printf("%d! = %d\n", n, fact(n));
        exit(0);
}
```

```
$ for i in `seq 13`
  do  ./fact $i
  done
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 1932053504
```

```
1932953504 / 479001600 =
    4.03538005718561
13*12*11*10*9*8*7*6*5*4*3*2*1 =
    6227020800
6227020800 - 2^32
    1932053504
```

Ensure that unsigned integer operations do not wrap

```
void func(uint32_t ui_a, uint32_t ui_b) {
    uint32_t usum = ui_a + ui_b;
    printf("%u + %u = %u\n", ui_a, ui_b, usum);
}
```

```
int32_t main() {
    func( ui_a: UINT_MAX, ui_b: 1);
    return 0;
}
```

4294967295 + 1 = 0

From:

$$UINT\_MAX \geq ui\_a + ui\_b$$

we derive that:

$$UINT\_MAX - ui\_a \geq ui\_b$$

The negation is:

$$UINT\_MAX - ui\_a < ui\_b$$

```
void func(uint32_t ui_a, uint32_t ui_b) {
    uint32_t usum;
    if (UINT_MAX - ui_a < ui_b) {
        printf("overflow!\n");
    } else {
        usum = ui_a + ui_b;
        printf("%u + %u = %u\n", ui_a, ui_b, usum);
    }
}
```

overflow!

Ensure that signed integer operations do not wrap

```c
void func(int32_t si_a, int32_t si_b) {
    int32_t sum = si_a + si_b;
    printf("%d + %d = %d\n", si_a, si_b, sum);
}
```

```c
int32_t main() {
    func(INT_MAX, si_b: 1);
    return 0;
}
```

2147483647 + 1 = -2147483648

overflow!

```c
void func(int32_t si_a, int32_t si_b) {
    int32_t sum = 0;
    if (((si_b > 0) && (si_a > (INT_MAX - si_b))) ||
        ((si_b < 0) && (si_a < (INT_MIN - si_b)))) {
        printf("overflow!\n");
    } else {
        sum = si_a + si_b;
        printf("%d + %d = %d\n", si_a, si_b, sum);
    }
}
```

The average of two numbers

```c
int32_t main() {
    int16_t a, b, sum, avg1, avg2;
    a = b = SHRT_MAX;
    sum = a + b;
    avg1 = (a + b) / 2;
    avg2 = sum / 2;
    printf("a = %d, b = %d, sum = %d\n", a, b, sum);
    printf("avg1    = %6d\n", avg1);
    printf("avg2    = %6d\n", avg2);
    printf("sum/2   = %6d\n", sum/2);
    printf("(a+b)/2 = %6d\n", (a+b)/2);
    return 0;
}
```

```
a = 32767, b = 32767, sum = -2
avg1    =  32767
avg2    =     -1   }
sum/2   =     -1   }   sum/2
(a+b)/2 =  32767
```

1 isn't always 1

```
#include <stdio.h>

int
main(void)
{   float f, of, cnt, i;

    f = 1;

    for (cnt = 1; cnt < 47; cnt++)
    {
        f /= 10;

        for (of = f, i = 1; i <= cnt; i++)
        {    of *= 10;
        }

        printf("%9.3g    %9.8f\n", f, of);
    }
}
```

```
      0.1       1.00000000
     0.01       0.99999994
    0.001       0.99999994
   0.0001       0.99999994
    1e-05       0.99999994
    1e-06       0.99999994
    ...
    1e-38       0.99999994
    1e-39       1.00000024
    1e-40       0.99999452
    1e-41       0.99996656
    1e-42       1.00052714
 9.95e-44       0.99492157
 9.81e-45       0.98090899
  1.4e-45       1.40129852
        0       0.00000000
```

Prevent or detect domain and range errors in math functions

```c
void func(double x) {
    double result = sqrt(x);;
    printf("sqrt(%lf) = %lf\n", x, result);
}
```

```c
void func(double x) {
    double result = 0;
    if (x < 0.0) {
        printf("%lf is negative", x);
    } else {
        result = sqrt(x);
        printf("sqrt(%lf) = %lf\n", x, result);
    }
}
```

Ensure that integer conversions do not result in lost or misinterpreted data

```c
int32_t main() {
    uint64_t u_a = ULONG_MAX;
    char sc;
    sc = (char)u_a;
    printf("%llu -> %c\n", u_a, sc);
    return 0;
}
```

18446744073709551615 -> ?

```c
int32_t main() {
    uint64_t u_a = ULONG_MAX;
    char sc;
    if (u_a <= CHAR_MAX) {
        sc = (char)u_a;
        printf("%llu -> %c\n", u_a, sc);
    } else {
        printf("to big: %llu!\n", u_a);
    }
    return 0;
}
```

to big: 18446744073709551615!

Loops

Defensive coding: bounded loops

- In mission critical code, loops must be *provably bounded*
  - In such a way that a static analysis tool can *verify* this
  - Never modify a loop index inside the body of a loop
  - If there is no clear bound, create one (e.g. INT_MAX)

```
int cnt = 0;
for (ptr = head; ptr != NULL; ptr = ptr->nxt, cnt++)
{   assert(cnt < INT_MAX);
    …
}
```

Potentially unbounded loops

```c
struct Data {
    int32_t value;
};

struct Node {
    struct Data* data;
    struct Node* next;
};
```

What if the list is circular?

What if some list elements have non-null data fields?

```c
void printList(struct Node* n) {
    struct Node* ptr;
    for (ptr = n; ptr; ptr = ptr->next) {
        printf("%d\n", ptr->data->value);
    }
}
```

```c
void printList(struct Node* n) {
    struct Node* ptr;
    int32_t count = 0;
    for (ptr = n; ptr; ptr = ptr->next, count++) {
        assert(count < 1000);
        if (ptr->data) {
            printf("%d\n", ptr->data->value);
        } else {
            printf("NO DATA\n");
        }
    }
}
```

Memory

Do not form or use out-of-bounds pointers or array subscripts

```c
enum { TABLE_SIZE = 100 };

static int32_t table[TABLE_SIZE];
```

```c
int32_t *f(int32_t index) {
    if (index < TABLE_SIZE) {
        return table + index;
    }
    return NULL;
}
```

```c
int32_t *f(int32_t index) {
    if (index >= 0 && index < TABLE_SIZE) {
        return table + index;
    }
    return NULL;
}
```

Don’t pass non-null-terminated char seq. to a library function that expects a string

```
void func(void) {
    char c_str[3] = "abc";
    printf("%s\n", c_str);
}
```

abc??=??

```
void func(void) {
    char c_str[] = "abc";
    printf("%s\n", c_str);
}
```

abc

Out of lifetime access

```
int32_t *foo(int32_t x) {
    int32_t z = x;
    int32_t *y = &z;
    return y;
}

void bar() {
    int32_t b = 2022;
}
```

```
int32_t main() {
    int32_t *x = foo( x: 42);
    printf("%i\n",*x);
    foo( x: 255);
    printf("%i\n",*x);
    bar();
    printf("%i\n",*x);
    return 0;
}
```

42

255

1846018143

Out of lifetime access

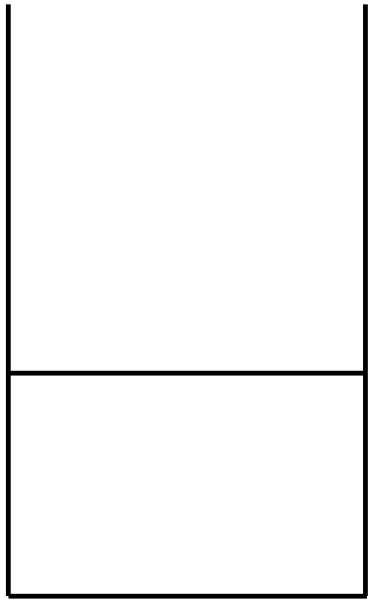
```
int32_t *foo(int32_t x) {
    int32_t z = x;
    int32_t *y = &z;
    return y;
}

void bar() {
    int32_t b = 2022;
}
```

```
int32_t main() {
    int32_t *x = foo( x: 42);
    printf("%i\n",*x);
    foo( x: 255);
    printf("%i\n",*x);
    bar();
    printf("%i\n",*x);
    return 0;
}
```

42

255

1846018143



Out of lifetime access

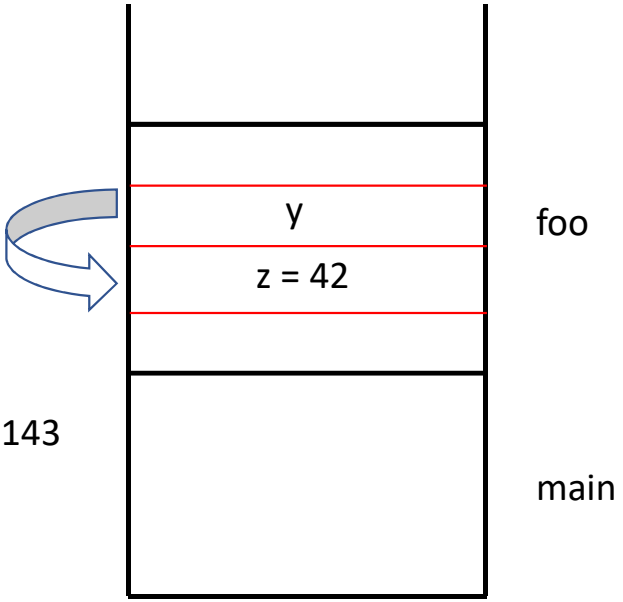
```
int32_t *foo(int32_t x) {
    int32_t z = x;
    int32_t *y = &z;
    return y;
}

void bar() {
    int32_t b = 2022;
}
```

```
int32_t main() {
    int32_t *x = foo( x: 42);
    printf("%i\n",*x);
    foo( x: 255);
    printf("%i\n",*x);
    bar();
    printf("%i\n",*x);
    return 0;
}
```

42

255

1846018143



Out of lifetime access

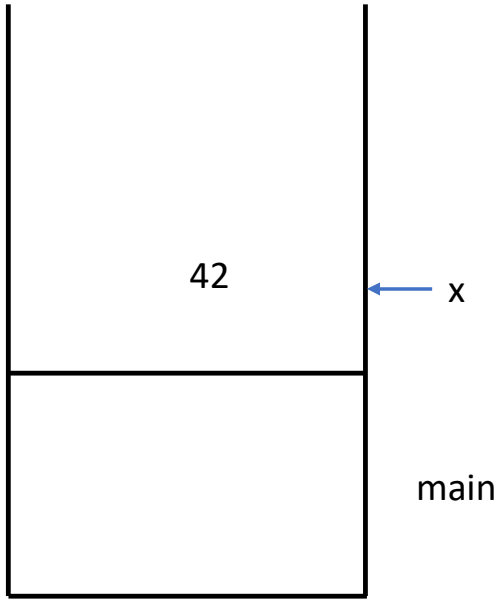
```c
int32_t *foo(int32_t x) {
    int32_t z = x;
    int32_t *y = &z;
    return y;
}

void bar() {
    int32_t b = 2022;
}
```

```c
int32_t main() {
    int32_t *x = foo( x: 42);
    printf("%i\n",*x);
    foo( x: 255);
    printf("%i\n",*x);
    bar();
    printf("%i\n",*x);
    return 0;
}
```

42

255

1846018143



Out of lifetime access

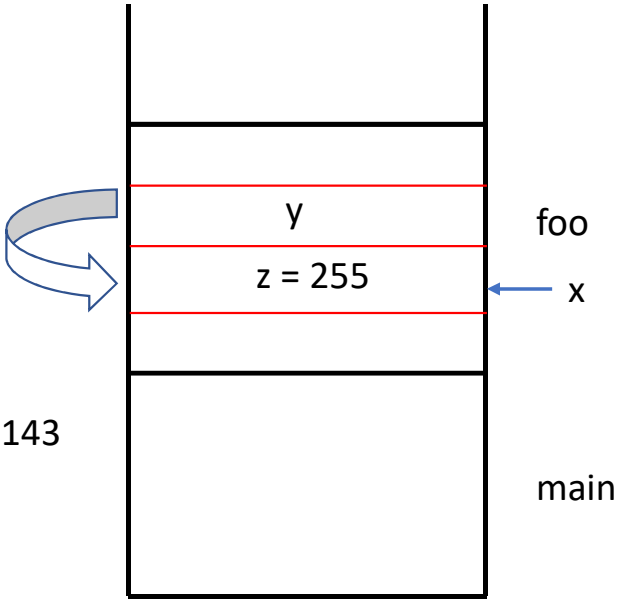
```c
int32_t *foo(int32_t x) {
    int32_t z = x;
    int32_t *y = &z;
    return y;
}

void bar() {
    int32_t b = 2022;
}
```

```c
int32_t main() {
    int32_t *x = foo( x: 42);
    printf("%i\n",*x);
    foo( x: 255);
    printf("%i\n",*x);
    bar();
    printf("%i\n",*x);
    return 0;
}
```

42

255

1846018143



Out of lifetime access

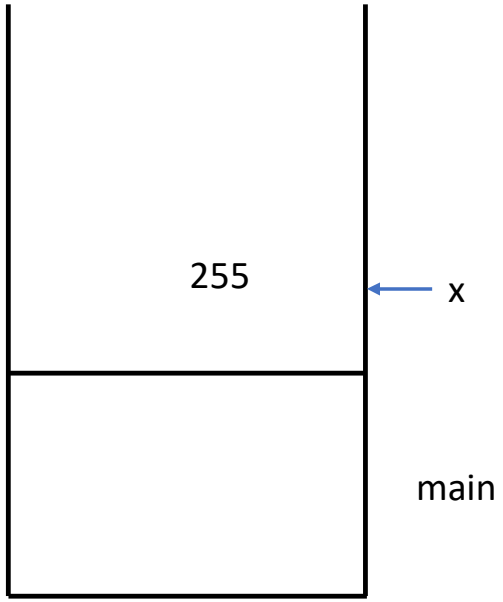
```c
int32_t *foo(int32_t x) {
    int32_t z = x;
    int32_t *y = &z;
    return y;
}

void bar() {
    int32_t b = 2022;
}
```

```c
int32_t main() {
    int32_t *x = foo( x: 42);
    printf("%i\n",*x);
    foo( x: 255);
    printf("%i\n",*x);
    bar();
    printf("%i\n",*x);
    return 0;
}
```

42

255

1846018143



Out of lifetime access

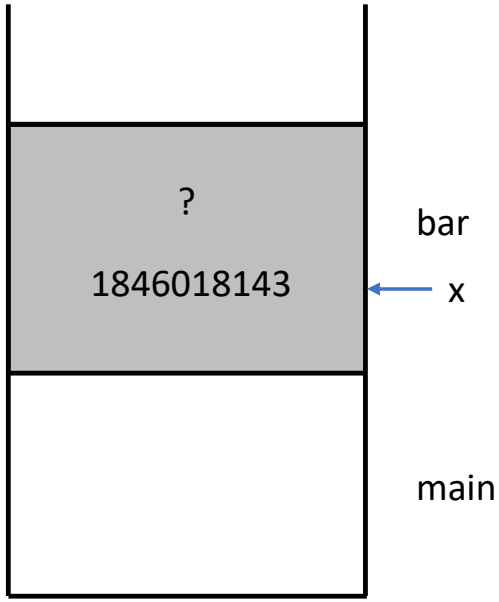
```c
int32_t *foo(int32_t x) {
    int32_t z = x;
    int32_t *y = &z;
    return y;
}

void bar() {
    int32_t b = 2022;
}
```

```c
int32_t main() {
    int32_t *x = foo( x: 42);
    printf("%i\n",*x);
    foo( x: 255);
    printf("%i\n",*x);
    bar();
    printf("%i\n",*x);
    return 0;
}
```

42

255

1846018143



Out of lifetime access

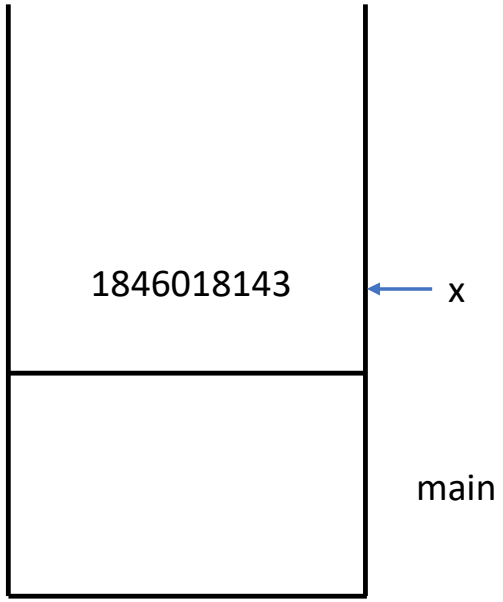
```c
int32_t *foo(int32_t x) {
    int32_t z = x;
    int32_t *y = &z;
    return y;
}

void bar() {
    int32_t b = 2022;
}
```

```c
int32_t main() {
    int32_t *x = foo( x: 42);
    printf("%i\n",*x);
    foo( x: 255);
    printf("%i\n",*x);
    bar();
    printf("%i\n",*x);
    return 0;
}
```

42

255

1846018143



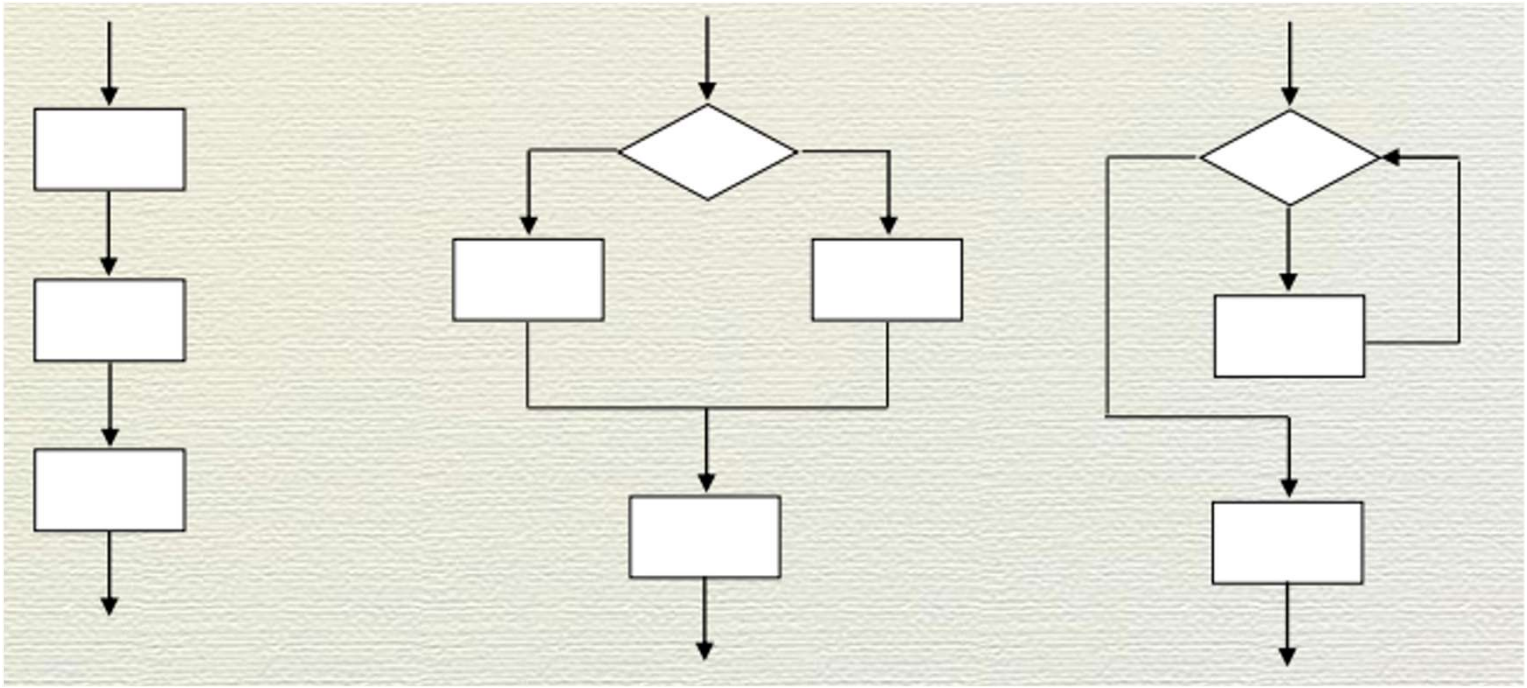
Do not read uninitialized memory

```c
void set_flag(int32_t number, int32_t *sign_flag) {
    if (NULL == sign_flag) {
        return;
    }
    if (number > 0) {
        *sign_flag = 1;
    } else if (number < 0) {
        *sign_flag = -1;
    }
}

int32_t main() {
    int32_t num = 0;
    int32_t sign;
    set_flag(num, &sign);
    printf("sign of %d = %d", num, sign);
    return 0;
}
```

```c
void set_flag(int32_t number, int32_t *sign_flag) {
    if (NULL == sign_flag) {
        return;
    }
    if (number >= 0) { // account for number being 0
        *sign_flag = 1;
    } else if (number < 0) {
        *sign_flag = -1;
    }
}

int32_t main() {
    int32_t num = 0;
    int32_t sign = 0; // initialize for being safe
    set_flag(num, &sign);
    printf("sign of %d = %d", num, sign);
    return 0;
}
```

Misc

Restrict to simple control flow constructs



A frequent source of mistakes is operator confusion

```
x + y * z << a

x + (y * z) | a

a & b == c

a = b(x) != SUCCESS
```

```
(x + (y * z)) << a

(x + (y * z)) | a

a & (b == c)

a = (b(x) != SUCCESS)
```

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

high

Operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

low

https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm

Do not depend on the order of evaluation for side effects … etc …

```
void func(int32_t b[], int32_t index) {
    int32_t sum = b[index] + b[++index];
    printf("%d\n", sum);
}
```

```
void func(int32_t b[], int32_t index) {
    int32_t sum = b[index] + b[index + 1];
    printf("%d\n", sum);
}
```

```
void func(int32_t b[], int32_t size, int32_t index) {
    if (index >=0 && index < size - 1) {
        int32_t sum = b[index] + b[index + 1];
        printf("%d\n", sum);
    } else {
        printf("index %d is out of bounds\n", index);
    }
}
```

Etc, etc, etc, …

```
void func(int32_t b[], int32_t size, int32_t index) {
    if (index >=0 && index < size - 1) {
        int32_t value1 = b[index];
        int32_t value2 = b[index + 1];
        if (INT_MAX - value1 >= value2) {
            int sum = value1 + value2;
            printf("%d\n", sum);
        } else {
            printf("overflow! of %d + %d", value1, value2);
        }
    } else {
        printf("index %d is out of bounds\n", index);
    }
}
```

We here must trust that the value passed in `size` is correct.

Some websites

Power of 10 rules for C: https://spinroot.com/p10
The MISRA coding standards: https://www.misra.org.uk
The AUTOSAR coding standards: https://www.autosar.org

Static analysis

- Overview of static analyzers: https://spinroot.com/static
- CodeSonar: https://www.grammatech.com/codesonar-cc
- Coverity: https://www.synopsys.com/software-integrity.html
- KlockWork: https://www.perforce.com/products/klocwork
- Semmle: https://github.blog/2019-09-18-github-welcomes-semmle
- Cobra: https://spinroot.com/cobra (free)
- Frama-C: https://frama-c.com (free)
- uno: https://spinroot.com/uno (free)

Dynamic analysis

- Test driven development: https://en.wikipedia.org/wiki/Test-driven_development
- Fuzz testing: https://www.fuzzingbook.org (Andreas Zeller)
- Valgrind: https://valgrind.org
- Mutation testing: https://github.com/mull-project/mull

Summary

- **Be aware of bounds**
  - data (numbers), time (loops), resources (memory)
  - Math and code are very different
- **Apply defensive coding**
  - Don’t assume: assert
  - Assertions flag the “cannot happen” cases.
  - Assertions are not for “can happen” error cases.
- **Follow coding standard**
  - Always compile with all warnings enabled
  - Use strong static analyzers on every build