Reducing Risk

Klaus Havelund
NASA Jet Propulsion Laboratory
October 18, 2022

Jet Propulsion Laboratory
California Institute of Technology

Evolved from

Gerard Holzmann's 2016 and 2019 FSW courses

Rajeev Joshi's 2016 FSW course

Contents

- General introduction to minimizing risk
- Code examples
- Tool support

“Programs are fascinating objects”

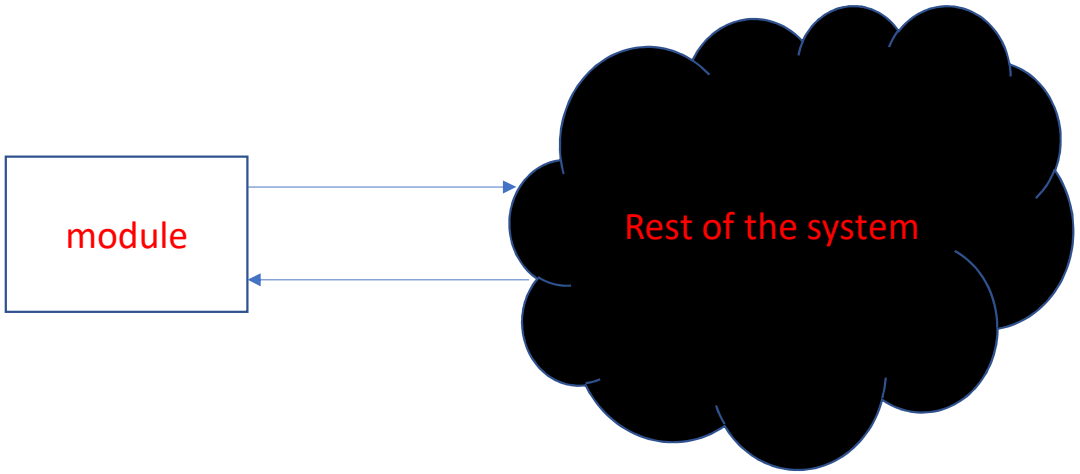
Professor Neil Jones

Because they embody an infinite number of behaviors and are challenging to analyze.

But that also means that a lot of things can go wrong.

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

unchecked use of parameters

```
void
dont_do_this (struct A *ptr)
{
   int y = ptr->cnt;        // don't assume the ptr is non-null
   int x = 1024/y;          // don't assume the value is non-zero
   int sum = x * x;         // don't assume this wont overflow
                            // do use sensible variable names

   itoa((int) n, (char *)str, /*base */ 10);
                            // do you know what you're calling?
                            // will it work for negative value of n?
                            // is str non-null and large enough?
                            // can itoa return an error code?
   ...

}
```

The readability of programs is immeasurably more important than their writeability. Hints on Programming Language Design, C.A.R. Hoare 1973

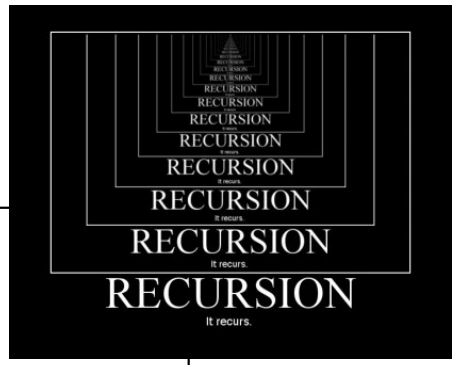
From math to code

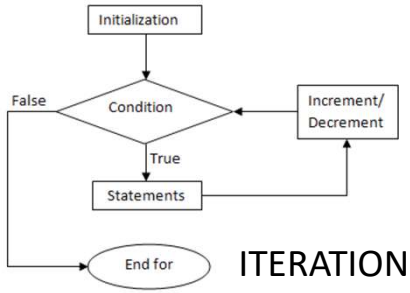
- We would like to believe that we can use **math** to reason about our **code**, and that it is a simple matter to do the right conversions.

However: converting from math to code requires consideration of:

- Resource limits
- Resource sharing
- Complexity

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





ITERATION

$$\mathbb{Z} \rightarrow \texttt{int}$$

$$\mathbb{R} \rightarrow \texttt{float}$$

$$\{f(x) \mid x \in S \cdot p(x)\} \rightarrow \texttt{for(nxt = start;...)\{...\};}$$

Awareness of bounds

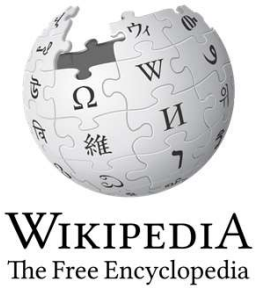
Be aware of bounds when dealing with:

- Numbers (precision, wrap-around, …)
- Loops (termination, complexity, …)
- 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

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.

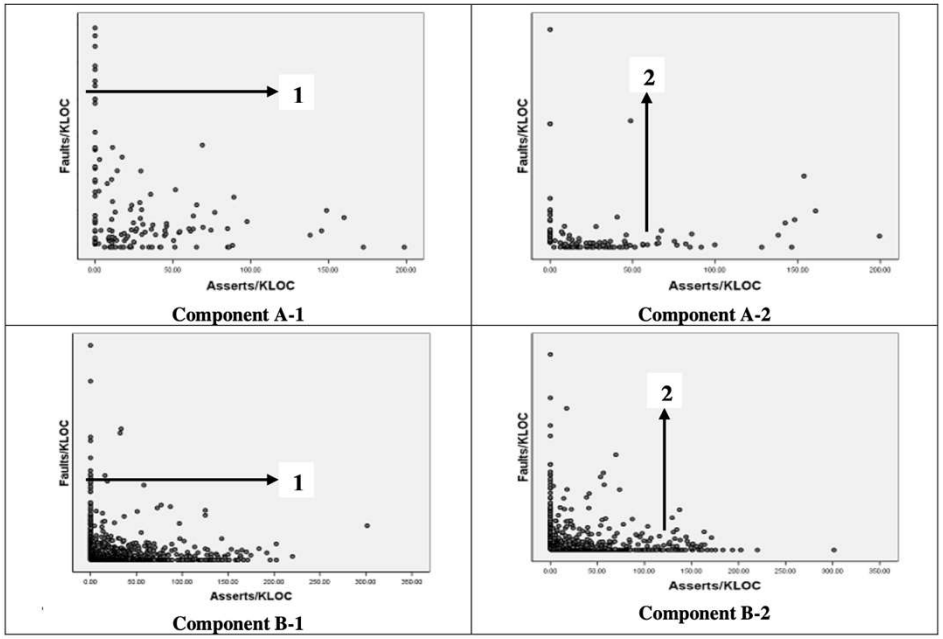
Assertions have been studied for many years [5, 11, 13]. In 1998 there was an investigation [14] performed for the NIST (National Institute for Standards and Technology) in order to evaluate software assertions. This report raised the following generalized observation regarding assertions:

> ... *"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

Some languages have them built in, including e.g. Dafny: https://github.com/dafny-lang/dafny

```
method max(arr: array<int>) returns (max: int)
  requires arr.Length > 0
  ensures forall j : int ::   j >= 0 && j < arr.Length  ==>  max >= arr[j]
  ensures exists j : int ::   j>=0 && j < arr.Length && max == arr[j]
{
  ...
}
```

Pre and post conditions in C as assertions

```c
int sum_of_squares (int *a, int count)
{
    /* Precondition: */
    assert (a != NULL);
    assert (count >= 0);

    /* Calculation: */

    int i, result = 0;
    for (i = 0; i < count; ++i) {result = result + (a[i] * a[i]);}

    /* Postcondition: */
    assert (result >= 0);

    return result;
}
```

https://riptutorial.com/c/example/1810/precondition-and-postcondition

Avoid loosely defined C constructs

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*.

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 C coding standard

https://rules.jpl.nasa.gov/cgi/doc-gw.pl?DocID=78115

Standard:	**Institutional Coding Standard for the C Programming Language, Rev. 0**
Process:	**Develop Software Products**
Document Owner:	**Gerard Holzmann**
Effective Date:	**April 17, 2009**

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.
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.
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 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 – MISRA *shall* compliance	
73 rules	All MISRA *shall* rules not already covered at Levels 1-4.
6 – MISRA *should* compliance	
*16 rules	All MISRA *should* rules not already covered at Levels 1-4.

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

LOC-1: Language Compliance

Rule 1 (language)
All C code *shall* conform to the ISO/IEC 9899-1999(E) standard for the C programming language, with no reliance on undefined or unspecified behavior. [MISRA-C:2004 Rule 1.1, 1.2]

The purpose of this rule is to make sure that all mission critical code can be compiled with any language compliant compiler, can be analyzed by a broad range of tools, and can be understood, debugged, tested, and maintained by any competent C programmer. It ensures that there is no hidden reliance on compiler or platform specific behavior that may jeopardize portability or code reuse. The rule prohibits straying outside the language definition, and forbids reliance of undefined or unspecified behavior. This rule also prohibits the use of `#pragma` directives, which are by definition implementation defined and outside the language proper. The `#error` directive is part of the language, and its use is supported. The closely related `#warning` directive is not defined in the language standard, but its use is allowed if supported by the compiler (but note Rule 2).

The C language standard explicitly recognizes the existence of undefined and unspecified behavior. A list of formally unspecified, undefined and implementation dependent behavior in C, as contained in the ISO/IEC standard definition, is given in Appendix A.

Rule 2 (routine checking)
All code *shall* always be compiled with all compiler warnings enabled at the highest warning level available, with no errors or warnings resulting. All code *shall* further be verified with a JPL approved state-of-the-art static source code analyzer, with no errors or warnings resulting. [MISRA-C:2004 Rule 21.1]

This rule should be considered routine practice, even for *non*-critical code development. Given compliance with Rule 1, this means that the code should compile without errors or warnings issued with the standard gcc compiler, using a command line with minimally the following option flags:

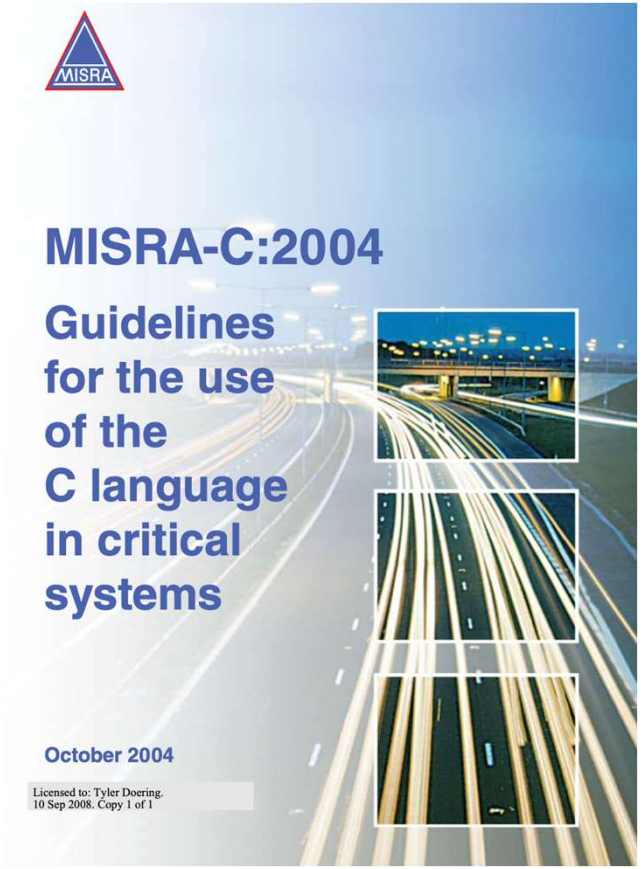
```
gcc -Wall -pedantic -std=iso9899:1999 source.c
```

A suggested broader set of gcc compiler flags includes also:

```
-Wtraditional
-Wshadow
-Wpointer-arith
-Wcast-qual
-Wcast-align
-Wstrict-prototypes
-Wmissing-prototypes
-Wconversion
```

The Misra coding standard

https://www.misra.org.uk



6. Rules 20
6.1 Environment 20
6.2 Language extensions 21
6.3 Documentation 22
6.4 Character sets 24
6.5 Identifiers 25
6.6 Types 28
6.7 Constants 29
6.8 Declarations and definitions 29
6.9 Initialisation 32
6.10 Arithmetic type conversions 33
6.11 Pointer type conversions 44
6.12 Expressions 46
6.13 Control statement expressions 53
6.14 Control flow 56
6.15 Switch statements 59
6.16 Functions 61
6.17 Pointers and arrays 64
6.18 Structures and unions 66
6.19 Preprocessing directives 70
6.20 Standard libraries 76
6.21 Run-time failures 79

Rule 14.7 (required): **A function shall have a single point of exit at the end of the function.**

[IEC 61508 Part 3 Table B.9]

This is required by IEC 61508, under good programming style.

Rule 14.8 (required): **The statement forming the body of a *switch*, *while*, *do* ... *while* or *for* statement shall be a compound statement.**

The statement that forms the body of a *switch* statement or a *while*, *do* ... *while* or *for* loop, shall be a compound statement (enclosed within braces), even if that compound statement contains a single statement.

For example:

```
for (i = 0; i < N_ELEMENTS; ++i)
{
   buffer[i] = 0;         /* Even a single statement must be in braces */
}

while ( new_data_available )
   process_data ();      /* Incorrectly not enclosed in braces         */
   service_watchdog (); /* Added later but, despite the appearance
                           (from the indent) it is actually not part of
                           the body of the while statement, and is
                           executed only after the loop has terminated */
```

Note that the layout for compound statements and their enclosing braces should be determined from the style guidelines. The above is just an example.

CERT C coding standard

https://wiki.sei.cmu.edu/confluence/display/c

Rules

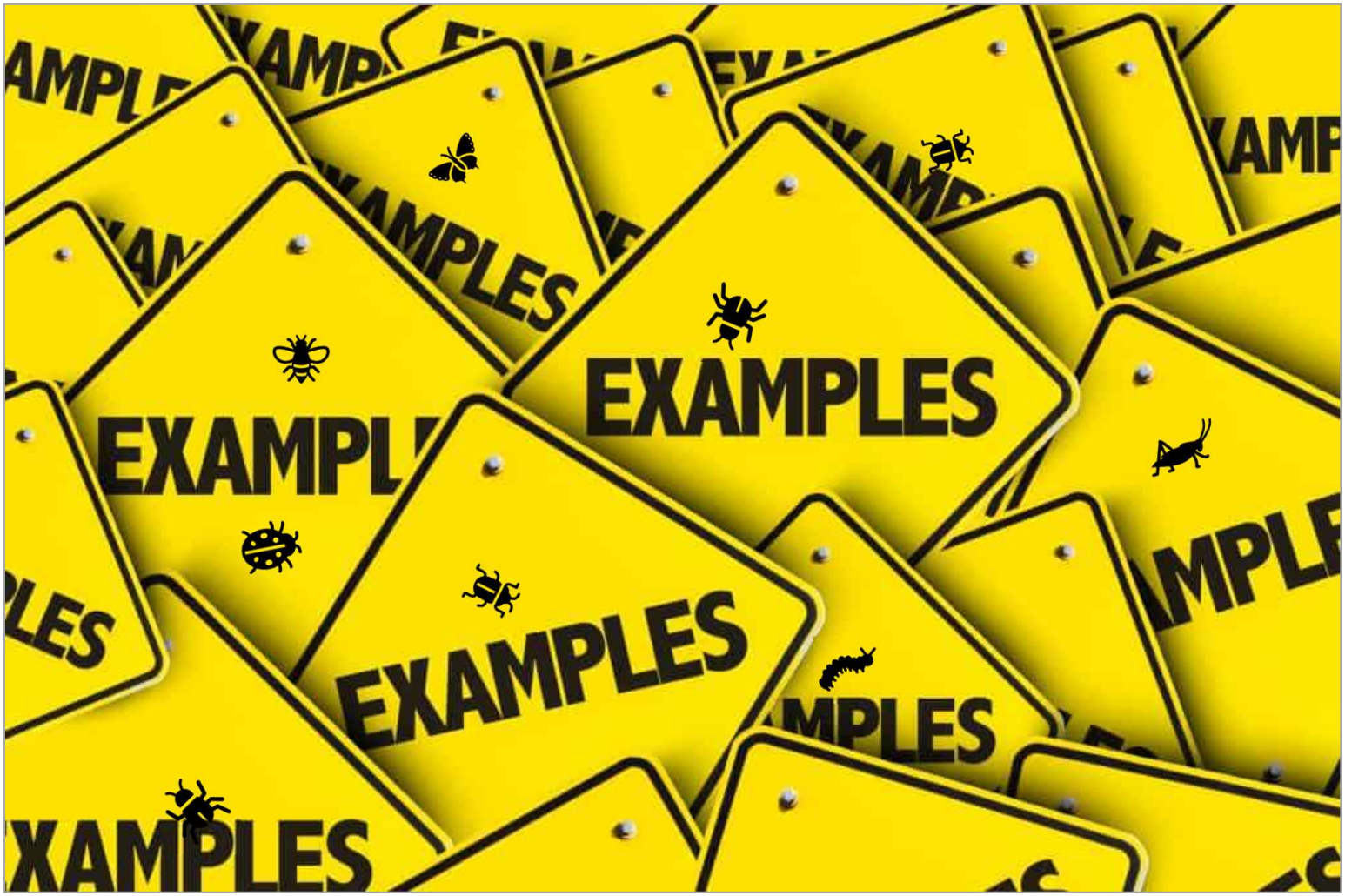
- Rule 01. Preprocessor (PRE)
- Rule 02. Declarations and Initialization (DCL)
- Rule 03. Expressions (EXP)
- Rule 04. Integers (INT)
- Rule 05. Floating Point (FLP)
- Rule 06. Arrays (ARR)
- Rule 07. Characters and Strings (STR)
- Rule 08. Memory Management (MEM)
- Rule 09. Input Output (FIO)
- Rule 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)

Not an official JPL C coding standard
but worth being aware of as well.

Several of the following code examples are from the CERT standard!


EXAMPLES
EXAMPLES
EXAMPLES

Macros

Avoid side effects in arguments to unsafe macros

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

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

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

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

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)

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

Numbers

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
```

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
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```
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 goes wrong

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(unsigned int ui_a, unsigned int ui_b) {
    unsigned int usum = ui_a + ui_b;
    printf("%u + %u = %u\n", ui_a, ui_b, usum);
}
```

From:

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

we derive that:

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

The negation is:

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

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

```
void func(unsigned int ui_a, unsigned int ui_b) {
    unsigned int 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);
    }
}
```

4294967295 + 1 = 0

overflow!

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

```c
int main() {
    unsigned long int u_a = ULONG_MAX;
    signed char sc;
    sc = (signed char)u_a;
    printf("%lu -> %c\n", u_a, sc);
    return 0;
}
```

18446744073709551615 -> ?

```c
int main() {
    unsigned long int u_a = ULONG_MAX;
    signed char sc;
    if (u_a <= SCHAR_MAX) {
        sc = (signed char)u_a;
        printf("%lu -> %c\n", u_a, sc);
    } else {
        printf("to big: %lu!\n", u_a);
    }
    return 0;
}
```

to big: 18446744073709551615!

Ensure that operations on signed integers do not result in overflow

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

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

2147483647 + 1 = -2147483648

overflow!

```c
void func(signed int si_a, signed int si_b) {
    signed int sum;
    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);
    }
}
```

Prevent or detect domain and range errors in math functions

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

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

The average of two numbers

```c
int main() {
    short 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
```

Loops

Bounds on loops

bounds on loops: the microsoft zune 30 GB

stopped working midnight december 31, 2008

source:
public Microsoft FAQ webpage
for the Zune 30, Dec. 2008

Bounds on loops

bounds on loops: the zune code

input : days elapsed since Jan 1, 1980
output: year + day of year

```
year = 1980;
while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    } else
    {
        days -= 365;
        year += 1;
    }
}
```

December 31, 2008 was the 366th day of the year.
(2008 was a leap year: a multiple of 4, but not of 100 or 400).
-> the zune got stuck in an infinite loop

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 {
    int value;
};

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

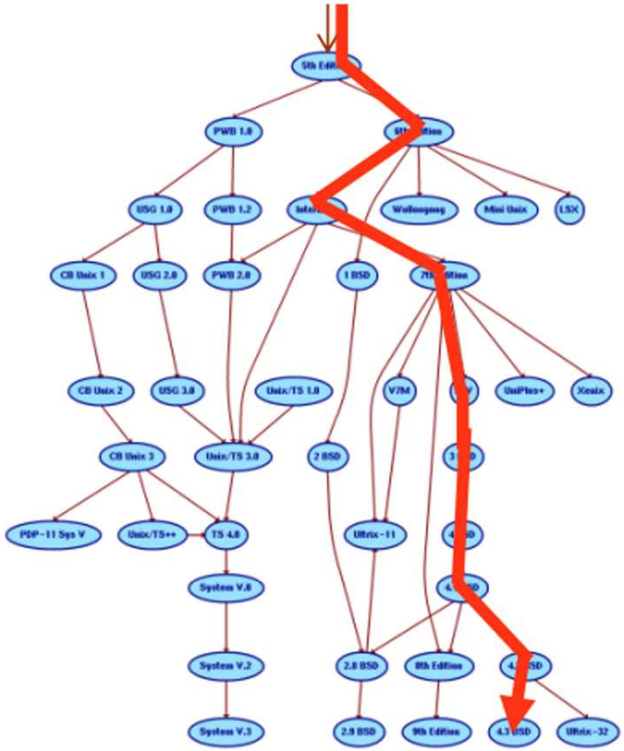
What if the list is circular?

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;
    int 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");
        }
    }
}
```

No recursion & bounded loops



Acyclic function call graph:
every path through the call graph
has a finite length.

Memory bounds

If we label each node (function) with its stack requirements, we can compute a bound on the maximal stack size needed.

Time bounds

We can compute the maximal time spent In each function, and we can derive a bound on the maximum execution time for each call.

Memory

Defensive coding example

```
void
vulnerable (char *input)
{   char str[1024];

    strcpy(str, input);
    ...
}
```

```
int
defensive( const char *input )
{   char str[1024];

    if (input == NULL)
    {    return ERROR;
    }
    strncpy(str, input, sizeof(str));
    str[sizeof(str)-1] = '\0'; // null terminate
    ...
    return SUCCESS;
}
```

let the compiler help you catch glitches

if this is an expected case
if not: use "assert(input != NULL)"

never use strcpy()
always strncpy()

make fewer assumptions
in this case: about the validity of parameters
(now, *and in future revisions of the code)*

use assertions
to indicate assumed "cannot happen" cases.
because when they do happen, you want to know

Do not read uninitialized memory

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

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

```c
void set_flag(int number, int *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;
    }
}

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

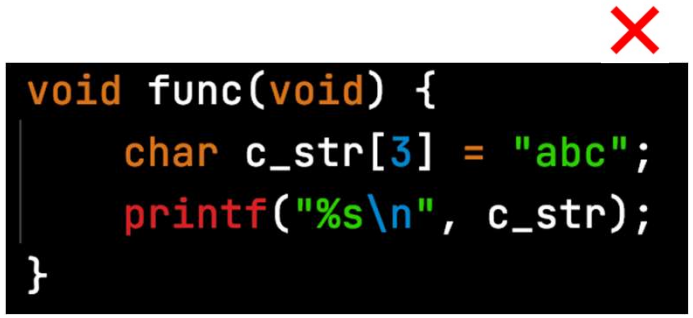
Do not form or use out-of-bounds pointers or array subscripts

```
enum { TABLESIZE = 100 };

static int table[TABLESIZE];
```

```
int *f(int index) {
    if (index < TABLESIZE) {
        return table + index;
    }
    return NULL;
}
```

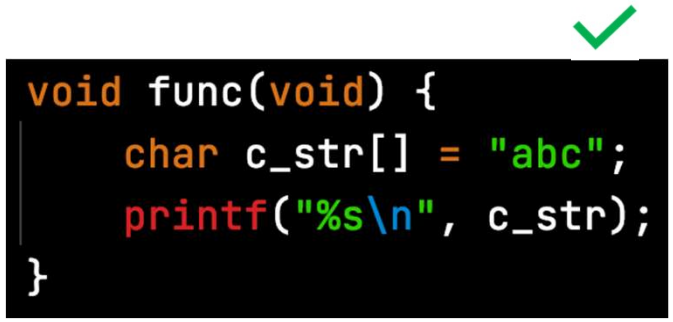
```
int *f(int index) {
    if (index >= 0 && index < TABLESIZE) {
        return table + index;
    }
    return NULL;
}
```

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);
}
```

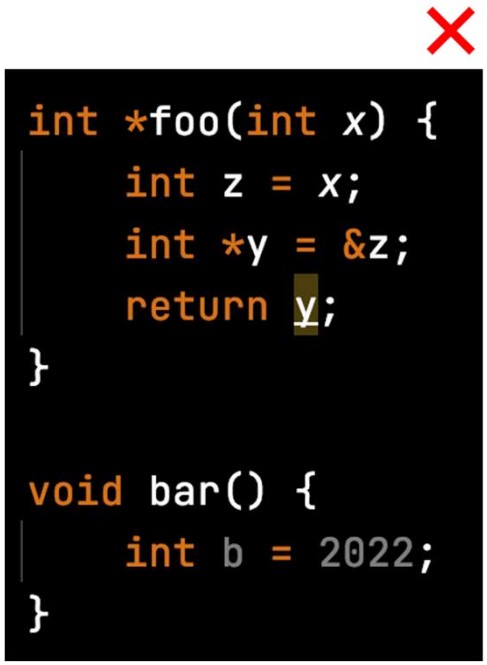
abc??=??



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

abc

Out of lifetime access



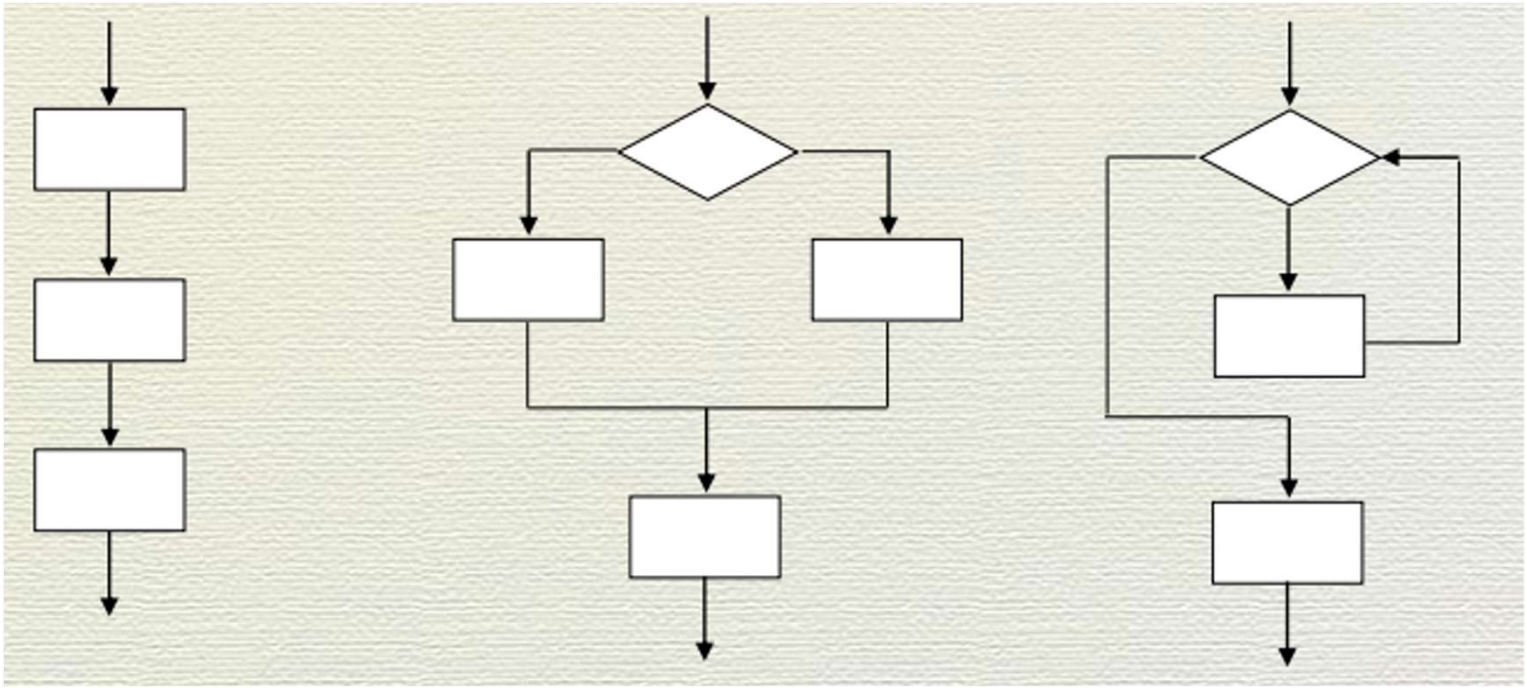
```c
int *foo(int x) {
    int z = x;
    int *y = &z;
    return y;
}

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

```c
int main() {
    int *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

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

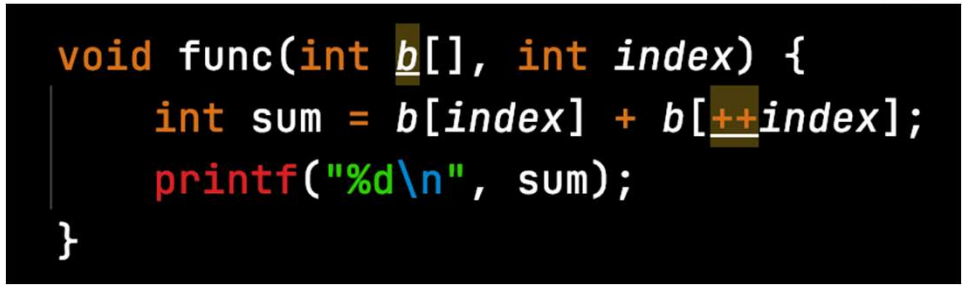
high

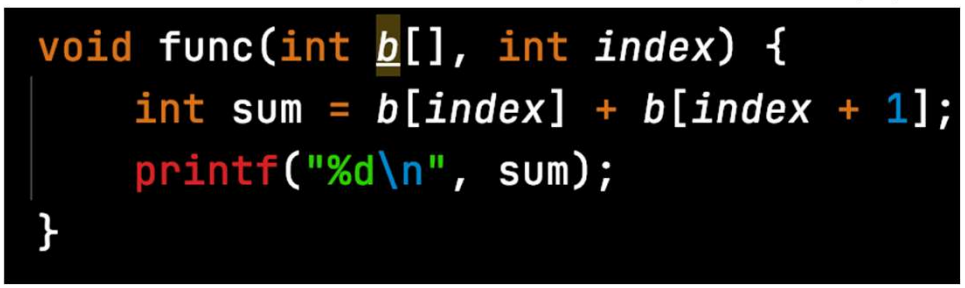
Operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

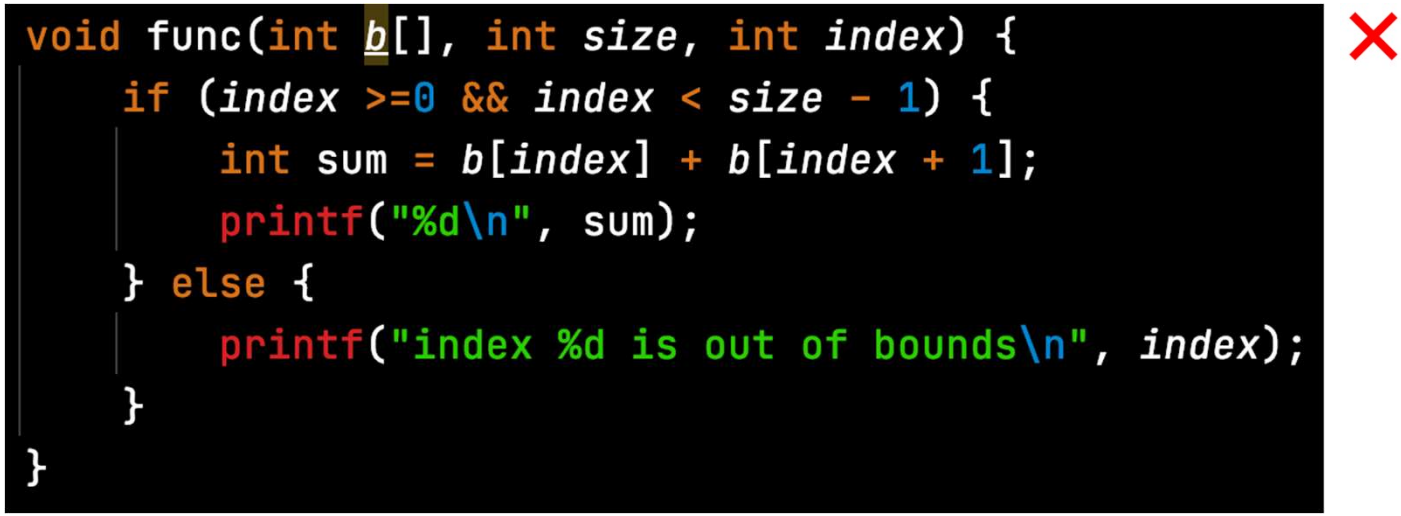
low

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

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







Etc, etc, etc, …

```c
void func(int b[], int size, int index) {
    if (index >=0 && index < size - 1) {
        int value1 = b[index];
        int 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.

Ensure that control never reaches the end of a non-void function

```
bool checkpass(const char *password) {
    // reject "password" as password:
    if (strcmp(password, "password") == 0) {
        return false;
    }
}
```

```
bool checkpass(const char *password) {
    // reject "password" as password:
    if (strcmp(password, "password") == 0) {
        return false;
    } else {
        return true;
    }
}
```

```
int main() {
    char password[] = "bii283@1#17";
    if (checkpass(password)) {
        printf("Success\n");
    } else {
        printf("Failure\n");
    }
}
```

Failure

Success

Concurrency

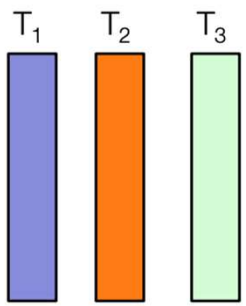
Concurrent programming

Concurrent Programming with Threads

We consider a programming model in which there are multiple threads that execute asynchronously with each other.

Each thread executes a (traditional) sequential program.

Threads may interact with each other by accessing shared memory.



Threads when run in isolation

Threads running concurrently on a single CPU

Note: sometimes threads are called tasks (e.g., in VxWorks)

Data races

A data race occurs when two threads access the same shared data, with no mutual coordination, and at least one of the threads is performing writes.

shared memory

```
int = 0;
```

T_1

```
x = x + 1;
```

L_1

T_2

```
x = x + 2;
```

L_2

Q: what are the possible outcomes ?

A: 3 ... and 1 ... and 2!

Q: Why 1 and 2?

task	action	L1	L2	x
T_1	L_1 = x	0		0
T_2	L_2 = x	0	0	0
T_1	x = L_1 + 1	0	0	1
T_2	x = L_2 + 1	0	0	1

x becomes 1

Race conditions

Races can occur at a higher level than just variable updates.

T_1

```
while (true) {
   …
   if (!new_events()) {
      wait();
   };
   …
}
```

T_2

```
new event!;
```

```
(defun daemon ()
 (loop
  (if (check-locks)
      (do-automatic-recovery))
  (unless
   (changed?
    (+ (event-count *database-event*)
       (event-count *lock-event*)))
   (wait-for-events
    (list *database-event* *lock-event*)))))
```

The solution

Mutual Exclusion

A protocol for ensuring that at most one thread can execute a piece of code (which is called a *critical section* of code)

T_1

```
enter() ;
... critical section ...
exit() ;
```

T_2

```
enter() ;
... critical section ...
exit() ;
```

The `enter()` code allows at most one thread to gain entry to the critical section. The `exit()` code does cleanup after the thread is done so another thread can perform its critical section.

Any other threads that try to enter the critical section are **blocked** until the current thread is done.

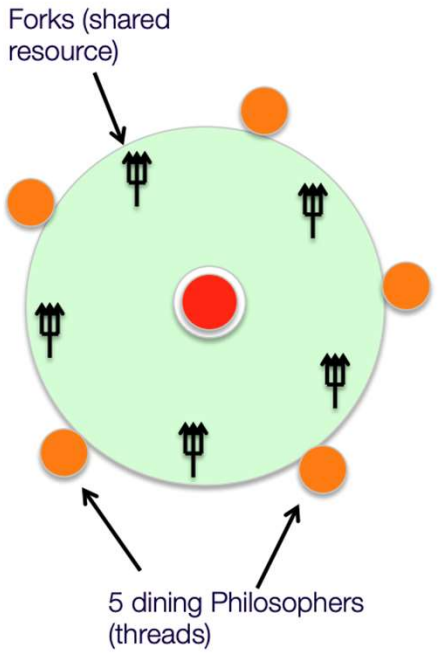
Deadlocks

The Dining Philosophers Problem

A famous problem invented by Dijkstra

DINING RULES

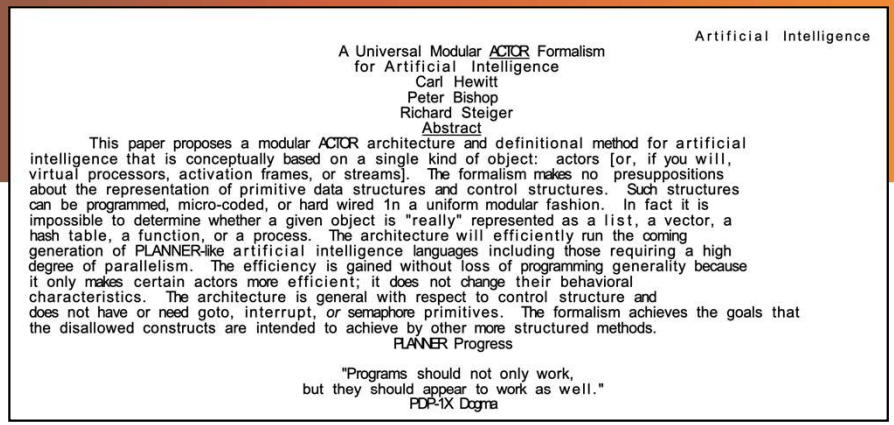
1. PHILOSPHERS CANNOT TALK TO ONE ANOTHER.
2. EACH PHILOSOPHER REQUIRES TWO FORKS TO EAT.
3. A PHILOSOPHER CAN GRAB ONLY THOSE FORKS ADJACENT TO HIM, ONE FORK AT A TIME.
4. MAXIMUM ONE PHILOSOPHER PER FORK.
5. EACH FORK IS RETURNED WHEN DONE EATING.



What if each philosopher starts by picking up their left fork?

The system will deadlock!

The Actor model



Artificial Intelligence

A Universal Modular ACTOR Formalism
for Artificial Intelligence

Carl Hewitt
Peter Bishop
Richard Steiger

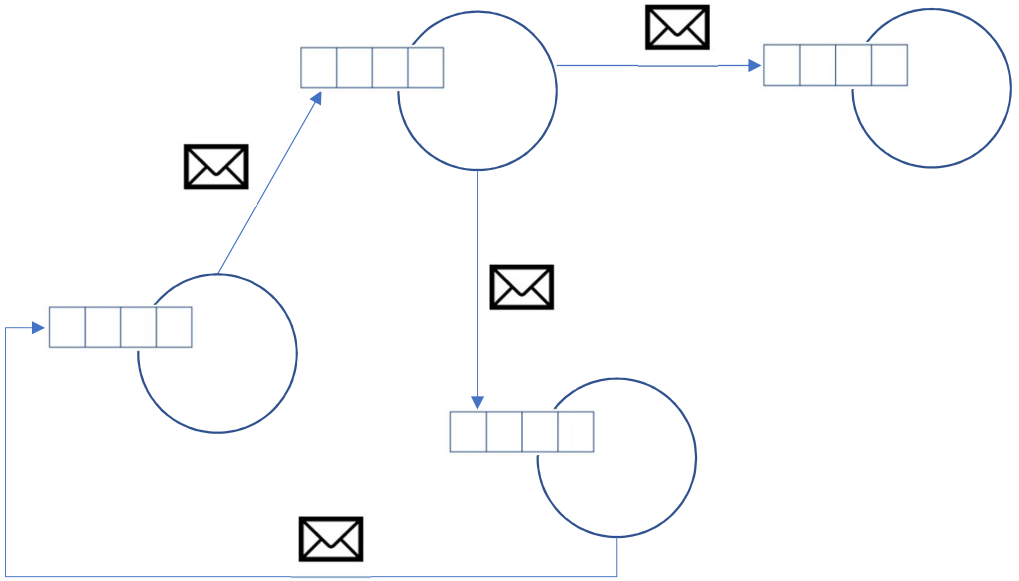
Abstract

This paper proposes a modular ACTOR architecture and definitional method for artificial intelligence that is conceptually based on a single kind of object: actors [or, if you will, virtual processors, activation frames, or streams]. The formalism makes no presuppositions about the representation of primitive data structures and control structures. Such structures can be programmed, micro-coded, or hard wired 1n a uniform modular fashion. In fact it is impossible to determine whether a given object is "really" represented as a list, a vector, a hash table, a function, or a process. The architecture will efficiently run the coming generation of PLANNER-like artificial intelligence languages including those requiring a high degree of parallelism. The efficiency is gained without loss of programming generality because it only makes certain actors more efficient; it does not change their behavioral characteristics. The architecture is general with respect to control structure and does not have or need goto, interrupt, *or* semaphore primitives. The formalism achieves the goals that the disallowed constructs are intended to achieve by other more structured methods.

PLANNER Progress

"Programs should not only work,
but they should appear to work as well."
PDP-1X Dogma

- Proposed by Hewitt, Bishop, and Steiger in 1973.
- System consists of a collection of concurrently executing actors.
- Actors communicate only via non-blocking message passing.
- At JPL the actor model has formed the basis for flight software in various forms.



Properties of concurrent programs

Safety vs Liveness

There are two kinds of properties we may want to assert about a program:

nothing bad ever happens ← *safety property*
can be falsified with a finite observation

no two processes are ever in their critical section at the same time
this linked list will never become circular

something good happens eventually ← *liveness property*
cannot be falsified with a finite observation

once the email is sent, it will eventually be delivered
a thread waiting on a shared resource will eventually get to use it

Safety and liveness properties can be expressed in a logical notation called *temporal logic* and there are automated tools (e.g., the SPIN model checker) that can check if a given program satisfies a set of safety and liveness properties

Tools and Techniques

Testing

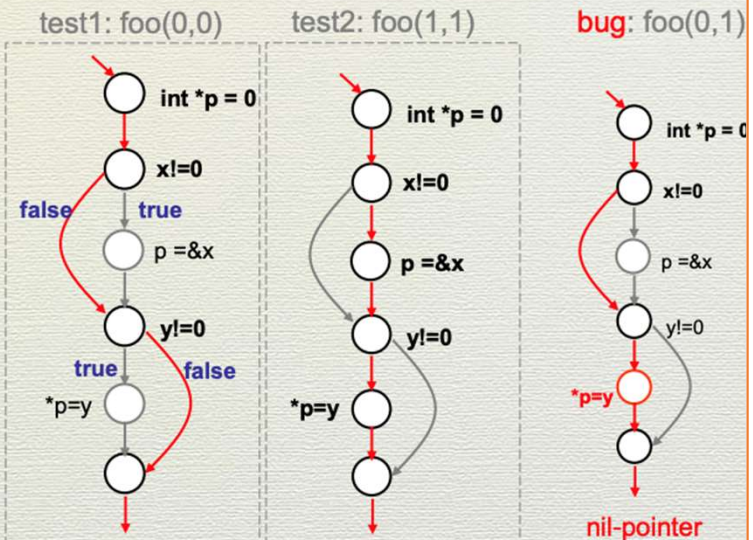
- Test driven development: https://en.wikipedia.org/wiki/Test-driven_development
- Fuzz testing: https://www.fuzzingbook.org (Andreas Zeller)
- Valgrind: https://valgrind.org
- Mutation testing: https://github.com/mull-project/mull

Testing

is 100% statement and branch coverage sufficient to detect all software defects?

small example:

```
void foo(int x, int y)
{
    int *p = NULL;

    if (x != 0) {
        p = &x;
    }

    if (y != 0) {
        *p = y;
    }
}
```



nil-pointer dereference missed

100% statement and branch coverage

not sufficient to detect all defects

test randomization (fuzz testing)

- for all input parameters:
  - define the range of valid values
- randomly pick
  - a value within the valid range
  - a value near or at the upperbound of the valid range
  - a value near or at the lower-bound of the valid range
  - a value outside the valid range
- run the test, check the result, repeat

Static analysis

- Overview of static analyzers: https://spinroot.com/static
- Power of 10 rules: https://spinroot.com/p10
- JPL's C coding standard: https://rules.jpl.nasa.gov/cgi/doc-gw.pl?DocID=78115
- The MISRA C coding standard: https://www.misra.org.uk

- 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)

Model checking

- SPIN: https://spinroot.com
- nuSMV: https://nusmv.fbk.eu
- FDR4: https://cocotec.io/fdr
- UPPAAL: https://uppaal.org (for real-time modeling)
- P: https://github.com/p-org/P
- TLA+: http://lamport.azurewebsites.net/tla/tla.html
- CBMC: https://www.cprover.org/cbmc (of C code)

Theorem proving

- PVS: https://pvs.csl.sri.com
- Dafny: https://dafny.org (of code – in the Dafny programming lang.)

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

END

Code safely!