Reducing Software Risk

Rob Bocchino
NASA Jet Propulsion Laboratory
October 22, 2024

Jet Propulsion Laboratory
California Institute of Technology

Reducing Software Risk

- Flight code must perform well and be reliable
- C and C++ are widely used
  - High performance
  - Direct interface to hardware
- But they come with risks
  - Code can have undefined or unexpected behavior
  - Compiler can catch some errors
  - But many it cannot (e.g., integer overflow, dangling pointers)
- This section of the course will show how to reduce the risk

Outline

- **Check numeric bounds**
- Use bounded loops
- Use assertions
- Avoid dangerous code
- Run static analysis
- Follow a coding standard

Check Numeric Bounds

- Integer and FP types in C and C++ are not numbers
- They are approximations of numbers with a finite representation
  - Integer types have a limited range
  - FP types have a limited range and a limited precision
- In FSW, you should avoid using platform-specific integer types
  - Instead of *int* and *unsigned*, use *int32_t* and *uint32_t*
  - This way the bounds are known and are the same across platforms
  - Use *int* if and only if it’s required by a library or system interface
    - E.g., the return value of *main* is *int*
    - Many C library calls return *int* as their error codes

Checking Integer Conversions

```
#include <cinttypes>

int32_t i64_to_i32(int64_t x) {
  return static_cast<int32_t>(x);
}
```

Unchecked Conversion

```
#include <cinttypes>
#include <limits>

enum class Status { FAILURE, SUCCESS };

Status i64_to_i32(int64_t x, int32_t& result) {
  Status status = Status::FAILURE;
  if ((x >= std::limits<int32_t>::min()) &&
      (x <= std::limits<int32_t>::max())) {
    result = static_cast<int32_t>(x);
    status = Status::SUCCESS;
  }
  return status;
}
```

Checked Conversion

Checking Integer Overflow

```
int32_t add_i32(
  int32_t a,
  int32_t b
) {
  return a + b;
}
```

Unchecked Addition

```
Status add_i32(
  int32_t a,
  int32_t b,
  int32_t& result
) {
  int64_t c = static_cast<int64_t>(a) +
    static_cast<int64_t>(b);
  return i64_to_i32(c, result);
}
```

Checked Addition

Exercise: Check the addition using only *int32_t* variables
Hint: Consider cases such as *a >= 0* and *b >= 0, a <= 0* and *b <= 0*, etc.

Beware of Floating Precision Loss

```c
#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
```

Outline

- Check numeric bounds
- **Use bounded loops**
- Use assertions
- Avoid dangerous code
- Run static analysis
- Follow a coding standard

Use Bounded Loops

```
while (true) {
  Queue::Item *item = nullptr;
  const Status status = queue.pop(item);
  if (status != Status::SUCCESS) {
    break;
  }
  dispatch(item);
}
```

Unbounded Loop

```
const size_t queueDepth = queue.getDepth();
for (size_t i = 0; i < queueDepth; i++) {
  Queue::Item *item = nullptr;
  const Status status = queue.pop(item);
  if (status != Status::SUCCESS) {
    break;
  }
  dispatch(item);
}
```

Bounded Loop

Unbounded loops can run forever and hang a thread
This behavior is bad in flight code

Avoid Recursion

- This is a form of a potentially unbounded loop
- It can also cause stack overflow

```
int32_t factorial(int32_t x) {
  if (x <= 1) {
    return 1;
  }
  else {
    return x * factorial(x - 1);
  }
}
```

Recursion

```
int32_t factorial(int32_t x) {
  int32_t result = 1;
  for (int32_t i = 2; i <= x; i++) {
    result *= i;
  }
  return result;
}
```

Bounded Loop

This code should also check for integer overflow

Outline

- Check numeric bounds
- Use bounded loops
- **Use assertions**
- Avoid dangerous code
- Run static analysis
- Follow a coding standard

Use Assertions

- An **assertion** is a macro expression *ASSERT(e)*
  - *e* is a Boolean expression that is expected to evaluate to *true*
  - If *e* does evaluate to *true*, the macro does nothing
  - Otherwise an **assertion failure** occurs
- When an assertion failure occurs
  - In unit testing, we usually halt the program
  - In system test and flight
    - At JPL we generally restart, unless restart could cause mission failure
    - Other institutions turn off assertion responses (ignore the failure)

Use of assertions is important for testing, debugging, and maintenance

When To Use Assertions

- Use assertions when failure “cannot happen” (e.g., it indicates a bug)
  - Pointer is null just before dereference
  - Array index is out of bounds just before array access
  - Computed value is outside expected range
- Otherwise check for and handle the error (e.g., bad user input)
- Common pattern:
  - Check for and reject invalid input
  - After passing the check, assert that input is valid before it is used
  - Assertion failure indicates a bug in the code that did the checking

Never assert that ground or user input is valid

Outline

- Check numeric bounds
- Use bounded loops
- Use assertions
- **Avoid dangerous code**
- Run static analysis
- Follow a coding standard

Avoid Dangerous Code

- C and C++ have many traps for the unwary
- It is easy to write and run programs that have **undefined** behavior
  - Relatively straightforward
    - Read of uninitialized memory
    - Object access through dangling pointer
  - More subtle
    - Left shift of negative signed integer (C, C++)
    - Array index starting at pointer to base-class object (C++)
- Some behavior is defined but obscure
  - Tricky operator precedence (use parentheses to disambiguate)
  - Complex C macros (avoid)
  - Forgetting to declare a base-class destructor *virtual* (C++) (don't do it)

C AND C++ HAZARDS

Avoid Complicated C macros

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 may suffice, but correctness may be difficult to prove

Avoid Side Effects in Macro Arguments

```
#define ABS(X) (((X) < 0) ? –(X) : (X))
```

Macro for computing absolute value

```
int32_t a = get_a();
a++;
int32_t b = ABS(a);
```

OK

```
int32_t a = get_a();
int32_t b = ABS(a++);
```

Probably not what is intended

Avoid Side Effects in Index Expressions

```
#include <cinttypes>
#include <array>

int32_t strangeAddition(
  const std::array<int32_t, 10>& a,
  size_t i
) {
  return a[i] + a[i++];
}
```

Evaluation order is not guaranteed

Guard Against Out-of-Bounds Array Access

```
class History {
  …
 public:
  Item& getItemAt(size_t index) {
    ASSERT(index < HISTORY_SIZE);
    return m_items[index];
  }
  …
 private:
  Item m_items[HISTORY_SIZE];
};
```

- Use error checking or assertions to ensure that array accesses stay in bounds
- Otherwise out-of-bounds access can cause strange behavior that is hard to debug
- If performance is extremely critical, these checks can be disabled in system testing and flight

Avoid Dangling Pointers

```
int32_t *stackReturn() {
 int32_t x;
 return &x;
};

int32_t outOfScope() {
 int32_t *p = nullptr;
 {
  int32_t x = 5;
  p = &x;
 }
 return *p;
}
```

***x* is out of scope on return from the function**

***x* is out of scope when the inner block ends**

Avoid Dynamic Memory Allocation

- Allocate all memory you need when the program starts
- Don’t use *malloc/new* or *free/delete* after initialization
- If you need dynamic behavior, use a memory pool
- This rule
  - Avoids heap fragmentation, which leads to nondeterministic performance
  - Avoids undefined behavior due to dangling pointers and memory leaks

Outline

- Check numeric bounds
- Use bounded loops
- Use assertions
- Avoid dangerous code
- **Run static analysis**
- Follow a coding standard

Run Static Analysis

- Compile with --*Wall* --*Werror*
- Run static analysis tools
  - Free tools, e.g., as GitHub actions
  - Commercial tools such as Coverity and CodeSonar
- Run the tools as often as feasible
  - Compiler: On every build
  - GitHub actions: On every pull request to main
  - Other tools: On every code review

Outline

- Check numeric bounds
- Use bounded loops
- Use assertions
- Avoid dangerous code
- Run static analysis
- **Follow a coding standard**

Follow a Coding Standard

- A **coding standard** is an agreed-upon set of rules for developing code
- Enforcement is usually by a combination of
  - Automated tools
  - Code review
- Examples
  - JPL C, C++
  - MISRA C, C++
  - AUTOSAR C++
  - Google C++ Style Guide

It is important to agree upon and adhere to a coding standard

The Power of 10

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*

http://spinroot.com/p10/

Gerard Holzmann

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

Gerard J. Holzmann

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

References

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