Stuff

- Homework 2 due Thurs before class
- Lab 2 due in 2 weeks
- Questions?
Important From Last Time

- Volatile is tricky
- To write correct embedded C and C++, you have to understand what volatile does and does not do
  - What is the guarantee that it provides?
- Don’t make the 8 mistakes shown in lecture
  - What were they?
Today

- MISRA-C
  - Subset of C language for critical systems
- Interesting MISRA rules
- MISRA-aware tools
- MISRA limitations
- Other language subsets
Safety-Critical Systems

◆ System is safety-critical if people might die due to software bugs
◆ Examples:
  ➢ Automobile stability / traction control
  ➢ Medical automation
  ➢ Many military applications
◆ You develop safety-critical software differently from non-critical software
◆ We’ll cover this topic in more detail later
MISRA-C

◆ MISRA – Motor Industry Software Reliability Association

◆ Their bright idea:
  ➢ Can’t avoid C
  ➢ But can force developers to avoid features of C that are known to be problematic
    ➢ Some language flaws
    ➢ Some legitimate features that happen to be bad for embedded software

◆ Most of MISRA-C is just good common sense for any C programmer
Terminology

- Execution error: Something illegal done by a program
  - Out-of-bounds array reference
  - Divide by zero
  - Uninitialized variable usage

- Trapped execution error: Immediately results in exception or program termination

- Untrapped execution error: Program keeps running
  - But may fail in an unexpected way later on
    - E.g., due to corrupted RAM
  - In C, operations with undefined behavior are not trapped
Safety

- A safe language does not allow untrapped execution errors
- A statically safe language catches all execution errors at compile time
- Useful languages can’t be completely statically safe
  - Java is dynamically safe
  - C and C++ are very unsafe
  - MISRA C is not safe either
- However, adherence to MISRA-C can largely be statically checked
  - This eliminates or reduces the likelihood of some kinds of untrapped execution errors
MISRA-C Rule 1.2

- No reliance shall be placed on undefined or unspecified behavior.
  - Lots of things in C have undefined behavior
    - Divide by zero
    - Out-of-bounds memory access
    - Signed integer overflow
  - Lots of things in C have implementation-defined and unspecified behavior
    - `printf ("a") + printf ("b");`

- Both of these hard to detect at compile time, in general

- Implementation-defined behavior is fine in MISRA-C
  - Why?
MISRA-C Rule 5.2

- Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

```c
int total;
int foo (int total) {
    return 3*total;
}
```

- What does this code mean?
- Why is it bad?
More MISRA-C

- Rule 6.3: Typedefs that indicate size and signedness should be used in place of the basic types.
  - For example `uint32_t` or `int8_t`
  - Why?
  - Good idea in general?

- Rule 9.1: All automatic variables shall have been assigned a value before being used.
  - Data segment: Initialized by programmer
  - BSS segment: Initialized to zero
  - Stack variables:Initialized to garbage
More MISRA-C

- Rule 11.1: Conversions shall not be performed between a pointer to a function and any type other than an integral type.
  - Discuss

- Rule 11.5: A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.
  - Discuss
More MISRA-C

◆ Rule 12.1: Limited dependence should be placed on C’s operator precedence rules in expressions.

◆ What does this program print?

```c
int main (void)
{
    int x = 0;
    if (x & 1 == 0) {
        printf ("t\n");
    } else {
        printf ("f\n");
    }
}
```
More MISRA-C

- **Rule 12.2**: The value of an expression shall be the same under any order of evaluation that the standard permits.

- **Rule 12.3**: The `sizeof` operator shall not be used on expressions that contain side effects.
  - E.g. `sizeof(x++)`;
  - What does this code mean?
  - Absurd that this is permissible in the first place
More MISRA-C

- Rule 12.4: The right-hand operand of a logical && or || operator must not contain side effects.
  - && and || are short-circuited in C
    - Evaluation terminates as soon as the truth of falsity of the expression is definite
  - if (x || y++) { ... }
  - Can this be verified at compile time?
  - What is a side effect anyway?
    - Page fault?
    - Cache line replacement?
More MISRA-C

◆ 12.10: The comma operator shall not be used.
  ➢ Some of the most unreadable C makes use of commas

\[(C--=Z=!Z) \quad \mid \mid \quad (\text{printf}("\n"), \quad C = 39, \quad H--)\];

◆ 13.3: Floating-point expressions shall not be tested for equality or inequality.
  ➢ Why?
More MISRA-C

- **14.1:** There shall be no unreachable code.
  - Good idea?

- **14.7:** A function shall have a single point of exit at the end of the function.
  - Good idea?
More MISRA-C

- 16.2: Functions shall not call themselves, either directly or indirectly.
  - Good idea?

- 16.10: If a function returns error information, then that error information shall be tested.
  - Good idea?
  - What does scanf() return? printf()? fclose()?
More MISRA-C

- **17.6**: The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

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

- This is a common (and nasty) C/C++ error
- How is this avoided in Java?
More MISRA-C

- **18.3:** An area of memory shall not be reused for unrelated purposes.
  - No overlays!

- **19.4:** C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
  - Avoids some problems we talked about earlier

- **20.4:** Dynamic heap memory allocation shall not be used.
  - Woah!
MISRA Limitations

- What cannot be accomplished within the MISRA framework?
  - Safety
  - Eliminating the preprocessor
  - Generics
- “A shack built on a swamp”
Tool Support for MISRA

◆ Goals:
  - Compiler should emit warning or error for any MISRA rule violation
  - Should not emit warnings or errors for code not violating the rules

◆ Tools:
  - Compilers from Green Hills, IAR, Keil
  - PC-Lint

◆ Reportedly there is considerable variation between tools
Other Language Subsets

- **SPARK Ada**
  - Subset of Ada95
  - Probably the most serious attempt to date at a safe, statically checkable language for critical software
  - Too bad Ada is so uncool…

- **Embedded C++**
  - No multiple inheritance
  - No RTTI
  - No exceptions
  - No templates
  - No namespaces
  - No new-style type casts
More Subsets

- **J2ME**
  - Not actually a language subset
  - Restricted Java runtime environment that has far smaller memory footprint
  - Popular on cell phones, etc.

- **JavaCard**
  - Very small – targets 8-bit processors

- **Basic ideas:**
  - A good language subset restricts expressiveness a little and restricts potential errors a lot
  - All languages have warts (at least in the context of embedded systems)
  - Simpler compilers may be better
Summary

- C has clear advantages and disadvantages for building safety-critical embedded software
  - MISRA-C mitigates some of the disadvantages
- Language subsetting can be a good idea