CubeSat Flight Software Workshop

Suggestions for Coding Style

Rob Bocchino

June 4, 2019

Jet Propulsion Laboratory
California Institute of Technology

Introduction

- Coding style is important for
  - Readability and maintainability
  - Correctness: Obscure code both invites and hides bugs
  - Safety: Good style is a defense against C and C++ hazards
- This section will
  - Provide some suggestions for a (not the only) good style
  - Explain the reasons behind the style choices
  - Provide code examples

Outline

- **Modules and Components**
- Functions
- Expressions and Statements

Encapsulate Operations on Data

- In C++, use an object-oriented style
- In C, write functions that accept pointers to structs as arguments
- Avoid inline operations on data structures

```
Value value;
Status status = NOT_FOUND;
for (…) {
  if (…) {
    value = …
    status = FOUND;
    break;
  }
}
```

OK

```
class Map {
  …
  Status find(const Key& key, Value& value) const {
    …
  }
  …
}
…
Value value;
const Status = map.find(key, value);
```

BETTER

Use Helper Classes (C++ Only)

- In C++, use helper classes inside the component implementation
- You can make them inner classes of the component class
  - C++ inner classes can’t refer to members of outer classes directly
  - To work around this, you can
    - Pass a reference to the outer class into the constructor for the inner class
    - Store the reference in a member of the inner class
- Alternatively, you can use non-inner classes
  - But give the class names appropriate prefixes
  - For example: *MyComponentHelper* instead of *MyComponent::Helper*

Outline

- Modules and Components
- **Functions**
- Expressions and Statements

Factor Functions into Layers

- Factor functions into
  - High-level structure (e.g., testing, branching, looping)
  - Detailed work
- Write a high-level function that has just the logical structure
- Have it call functions to do the detailed work

```
if (this->mode == …) {
  … // Lots of code
}
… // Lots more code
```

OK

```
if (this->mode == …) {
  this->emitTelemetry();
}
this->handleCommands();
```

BETTER

Keep Functions Short

- Move large code blocks to separate functions
- Commented code blocks should become new functions
  - The comment should be reflected in the function name
  - Then the function calls announce what is happening

```
// Do foo
[ code for doing foo ]
// Do bar
[ code for doing bar ]
```

OK

```
foo();
bar();
```

BETTER

Avoid Multiple Return Statements

- Avoid multiple return statements within the same function
  - They make it hard to follow the logic
  - They can lead to resource leaks
- Where possible, functions should follow this pattern (especially in C)

```
Claim resources
Do work
Release resources
Return
```

- Error handling can use the state propagation pattern (see next slide)

Propagate State to a Single Return Point

```
Status status = SUCCESS
const bool opStatus1 = operation1();
status = opStatus1 ? SUCCESS : OP_1_FAILED;
if (status == SUCCESS) {
  const bool opStatus2 = operation2();
  status = opStatus2 ? SUCCESS : OP_2_FAILED;
}
...
if (status == SUCCESS) {
  finish();
}
else {
  // Report error based on status
}
```

- Consider using this pattern:
  - It avoids multiple returns
  - It makes state handling explicit

Prefer Explicit Passing of State Through Arguments

- Don’t use shared variables to pass state between helper methods
  - Put mutable data in struct or class members
  - Update a member only when it is part of the object’s persistent state
- If a function updates shared data, its name should announce that fact

```
U32 calculateSize(const Data& d) {
  U32 size = 0;
  for (…) {
    size += …
    globalState += …
  }
  return size;
}
```

MISLEADING

Move the state update out of the function or name the function *calculateSizeAndUpdateState*

BETTER

Avoid Boolean Flag Arguments

CONFUSING

```
void doSomething(bool foobarMode) {
  ...
}
...
// True means foobar mode
doSomething(true);
```

BETTER

```
void doSomething(Mode mode) {
  ...
}
...
doSomething(Mode::FOOBAR);
```

Outline

- Modules and Components
- Functions
- **Expressions and Statements**

Use Fixed-Width Types as Much as Possible

- **Do not** assume that *int* is a 32-bit integer
  - It may not be on some platforms
  - If you need a 32-bit integer, use *int32_t* (defined in *stdint.h*)
  - F Prime renames these standard types, e.g., *I32*
- Use *int* only when required by the environment
  - For example, to hold the return value of a C library function that returns *int*
  - In this case the value
    - Is platform-specific
    - Does not necessarily have a width that is fixed across platforms
- This rule fosters portability, eliminates surprises

Initialize All Variables

- Never write *I32 x;* . Instead, write *I32 x = 0; .*
  - This includes
    - Local/temporary variables
    - Members in zero-argument constructors
  - If there’s no meaningful initial value, just pick one
    - This guards against nondeterministic behavior
    - E.g., on some platforms or runs, variables happen have the right value
- For C-style structs and arrays, either
  - Initialize all members or
  - Use *memset(0)* to zero out the whole thing
- Note that *memset(0)* does not work for C++ classes

Use Pointers and References Wisely

- In C++, prefer references to pointers
  - References cannot be *NULL*
  - Use pointers only when the value is assigned after the variable is created
    - Initialize each pointer variable *x* = *NULL*
    - Then assign a non-null pointer value to *x*
    - Then assert *x != NULL* before each use of *x*
- Use references (in C++) or pointers (in C) for concision
  - For example, don't write *this->a.b.c[i]* over and over
  - Instead
    - Write *C& c = this->a.b.c* once
    - Write *c[i]* several times

Wrap Accesses to Buffer Pointers and Arrays

- Avoid bare array access *A[i]*
  - It’s not safe, because it’s not bounds checked
  - In C++, use classes
    - Make the bare array a private member
    - Provide access through overloaded *[ ]* operators with bounds checking
  - In C, use functions that access arrays with bounds checking
- Avoid accessing bare buffer pointers **p*
  - Read or write buffer objects that store a pointer and a size
  - Make the pointer private
  - When accessing the buffer, provide bounds checking against the size

Use *const* as Much as Possible (1 of 2)

- C/C++ *const* syntax is weird but worth mastering
- C and C++
  - ***const T x*****:** *x* is a *const T*
  - ***const T *x*****:** *x* is a pointer to *const T*
  - ***T *const x*****:** *x* is a *const* pointer to *T*
  - ***const T *const x*****:** *x* is a *const* pointer to *const T*
- C++ only
  - ***void f() const*****:** *f* is a *const* member function
  - ***T& x*****:** Like *T *const x*
  - ***const T& x*****:** Like *const T *const x*

Use *const* as Much as Possible (2 of 2)

- *const* annotations are useful for
  - Keeping track of where state is changing
  - Keeping track of inputs and outputs
- Example: *void f(const Buffer& in, Buffer& out)*
  - *in* is a *const* input
  - *out* is a mutable output

Avoid Direct Use of C Library String Functions

- **Do not use** *strcpy*, etc. They are not bounds-checked.
- *strncpy*, etc. (note the *n*) are better, but still problematic
  - *strncpy* in particular does not guarantee termination
  - This behavior invites memory overruns and obscure bugs
- In C, wrap problematic functions in other functions with better behavior
- In F Prime, avoid direct use of C string functions altogether
  - In F Prime, use a subclass of *StringBase*
  - For example, *operator*= is a safer alternative to *strncpy*

Jet Propulsion Laboratory
California Institute of Technology

jpl.nasa.gov