Code Quality

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

pollev.com/tqm

- How are you? Any feedback?

Administrivia

- HW02 Due Yesterday
  - I *hope* it was less work than HW01
- HW03 posted after lecture
  - Algorithmically more complex than HW02
  - But you have the C++ standard library
  - I *think* it is less work than HW02
- Check-in posted tomorrow
  - “Finish” what we didn’t get to in lecture + 1 or 2 new short questions

Pre-Semester Survey Results & Discussion

- Concerns
  - Too much work
  - Course may explode due to it being the first offering

  - How has it been so far?

- Anonymous feedback form: https://forms.gle/2YzMuyC8kxCq1F2n8

Pre-Semester Survey Results & Discussion

- Wants:
  - Learn C++
  - Prepare people for upper-level courses
  - Take a course w/ me

  - Can't comment on second topic, but hopefully first one is going ok so far? Third one is hopefully meeting expectations for those that wanted that

- Anonymous feedback form: https://forms.gle/2YzMuyC8kxCq1F2n8

Lecture Outline

- **C++ Aside: namespaces**
- Idioms
- Reasoning About Code
- Clang-tidy and Cognitive complexity

C++ Namespaces: Motivation

- Let's say we wanted to define our own implementation for string
- How do we differentiate between which string we want to use?
  - My string or C++ string
- (other than changing the name of our string)

```
class string {
 public:
  // ctor
  string();

  // ...

 private:
  // SECRETE PATENT PENDING MATERIAL
};
```

```
int main() {
  string x; // is this my string or the C++ string?
  // ...
}
```

C++ Namespaces: definition example

- Let's say we wanted to define our own implementation for string
- We can put our string definition in a namespace.
- Namespaces:
  - A way to avoid name conflicts
  - Usually a way to group various items from the same module / library under one "name"
  - Now this string is under the "travis_lib" name

```
namespace travis_lib {

class string {
 public:
  // ctor
  string();

  // ...

 private:
  // SECRETE PATENT PENDING MATERIAL
};

};
```

C++ Namespaces: definition example

- Let's say we wanted to define our own implementation for string
- Can specify which string I want by explicitly stating the namespace

```
int main() {
  std::string x;
  travis_lib::string y;
  // ...
}
```

```
namespace travis_lib {

class string {
 public:
  // ctor
  string();

  // ...

 private:
  // SECRETE PATENT PENDING MATERIAL
};

};
```

C++ Namespaces: using namespace

- Let's say we wanted to define our own implementation for string

- Using namespace <namespace> makes the specified namespace the "default" assumed unless specified

```
using namespace std;

int main() {
  std::string x;
  travis_lib::string y;
  string z;
  // ...
}
```

Assumes std::string

```
namespace travis_lib {

class string {
 public:
  // ctor
  string();

  // ...

 private:
  // SECRETE PATENT PENDING MATERIAL
};

};
```

C++ Namespaces: using namespace

- Using statements related to namespaces are never used in .hpp files (because it then applies to every file that #includes it)
- Using a whole namespace in .cpp file (like we have been doing with using namespace std) is generally frowned on.

- Instead, you either
  - always specify namespace
    e.g. always type `std::string` instead of `string`
  - specify at the top of the cpp file, the things
    you are using from each any namespaces.

```
using std::string;
using std::vector;

int main() {
  string x;
  travis_lib::string y;
  string z;
  // ...
}
```

Lecture Outline

- C++ Aside: namespaces
- **Idioms**
- Reasoning About Code
- Clang-tidy and Cognitive complexity

Prelude:

- A lot of what I am stating is subjective, but I tried to keep it broadly acceptable
- A lot of this style stuff also depends on who you are coding with (e.g. the company/lab you are working with)
  - Different workplaces have different style guides
- You may not have access to more modern language features

What are programming idioms?

- A code idiom is just a piece of code / code pattern that shows up frequently across code.
- Some idioms can even exist across multiple programming languages
- Very important!

The for loop

- Basic for loop

```
#include <vector>

using namespace std;

int product(const vector<int>& items) {
  int result = 1;
  for (int i = 0; i < items.size(); i++) {
    result *= items[i];
  }
  return result;
}
```

- Does this work?
- Any complaints on it?
- Is this C++?

The “raw” for loop

- GOTO loop
  More similar to asm

```
int product_raw(const vector<int>& items) {
  int result = 1;
  int i = 0;
  LOOP:;
    if (i >= items.size()) goto END;
    result *= items[i];
    i = i + 1;
    goto LOOP;
  END:;
    return result;
}
```

- Does this work?
- Is this faster?
- Any complaints on it?

The “modern” for loop

- “modern” for loop

```
int product(const std::vector<int>& items) {
  int result{1};
  for (std::size_t idx{}; idx != items.size(); ++idx) {
    result *= items[idx];
  }
  return result;
}
```

- Does this work?
- How readable is this?
- Is this better?
- Any complaints on it?

```
int x = 5;
int a = ++x;

int y = 5;
int b = y++;
// final values: x,y,a = 6. b = 5

// y++ makes a temporary value
// that is often unecessary
```

The “auto” for loop

- Some people believe “use auto everywhere”

```
int product(const std::vector<int>& items) {
  auto result{1};
  for (auto idx{0uz}; idx != items.size(); ++idx) {
    result *= items[idx];
  }
  return result;
}
```

- Does this work? (resolve to the correct types)
- How readable is this?
- Is this better?
- Any complaints on it?

The “efficient” for loop

- Simplify the for loop to combine update and check into one step.

```
int product(const std::vector<int>& items) {
  int result{1};
  for (auto idx{items.size()}; idx--;) {
    result *= items[idx];
  }
  return result;
}
```

- Does this work?
- Is this faster?
- Any complaints on it?

The range-for loop

- Use a range-for loop

```
int product(const std::vector<int>& items) {
  int result{1};
  for (int item : items) {
    result *= item;
  }
  return result;
}
```

- Does this work?
- Any complaints on it?

Removing the loop

- Use <algorithm> and take a functional approach

```
#include <algorithm>
#include <functional>

using std::reduce;
using std::vector;
using std::multiplies;

int product(const vector<int>& items) {
  return reduce(items.begin(), items.end(), 1, multiplies);
}
```

- Does this work?
- How readable is this?
- Any complaints on it?

Idioms

- Code can be written in many ways that do the same thing
  - Many ways that generate the same assembly, near the same assembly, or have negligible difference in performance

- While writing code, you should make sure that the **code expresses intent to your audience** (co-workers, the compiler, yourself, etc.)
  - What an idiom is will depend on your audience. What practices have they seen before? What coding style do you and your co-workers work with?
  - In other words: “the best code is as self-documenting as possible”. Comments are great, but people have the habit of not fully reading those.

- Following idioms generally makes code easier to reason about
  - Similar for “simplifying” the code as much as possible.
    The less there is, the less there is to think about

Other C++ idioms: const ref

- Parameters for anything that isn't a primitive is a const reference
  - Some exceptions to this (e.g. `string_view`) but mostly true.

- Why?
  - Avoids making a copy
  - Avoids modifying data that we don't want a function to modify

Other C++ idioms: in/out parameters

- Which parameters are inputs? Which are outputs?

```
int main() {
  string x = "hello";
  string y = "bye";
  string z = mystery(x, y);
}
```

- Hard to tell based off of usage :(
  (not very self documenting)

Other C++ idioms: in/out parameters

- Which parameters are inputs? Which are outputs?

- Some places (like google) will use pointers to indicate a parameter is intended to be used for output:

- We aren't enforcing this. This idiom is pretty varied in adoption.

```
string mystery(string* a, const string& b) {
  *a += b;
  return *a;
}

int main() {
  string x = "hello";
  string y = "bye";
  string z = mystery(&x, y);
}
```

Other C++ idioms: naming conventions

- Which variables here are Globals? Constants? Member variables?

```
string Object::function(const string& input) {
  counter += 1;
  return storage[counter * factor];
}
```

Other C++ idioms: naming conventions

- Which variables here are Globals? Constants? Member variables?

```
string Object::function(const string& input) {
  g_counter += 1;
  return m_storage[g_counter * k_factor];
}
```

- Some C++ programmers prefix variable names to self-document
  - g_ -> global variable
  - k_ -> constant
  - m_ -> non-public member variable

- Again, this idiom varies in usage.

Lecture Outline

- C++ Aside: namespaces
- Idioms
- **Reasoning About Code**
- Clang-tidy and Cognitive complexity

Reasoning about code

- Our computers are automata

- It follows a sequence of operations to transition from one state to another.

- We can reason about our code like:
  - Program is in some state
  - Some assembly/line of code is executed
  - We are in a different state

Reasoning about code

- Consider this example. We star by assuming there is some int w that has a value greater than 0.

```
// Assume w is > 0

// x == 5 && w > 0
int x = 5;

// x == 5 && y == 32 && w > 0
int y = 32

// x == 5 && y == 32 && w > 0 && z >37
int z = x + y + w;
```

- By reasoning through our code, we can guarantee z is > 37 (assuming our initial assumption holds)

Reasoning about code (backwards)

- We can also go backwards to see “what must be true for what I want to be true”

```
int y = 2;


int z = 32;


int a = x * y + z;

// want a == 4562
```

Reasoning about code (backwards)

- We can also go backwards to see “what must be true for what I want to be true”

```
int y = 2;


int z = 32;

// x * y + z = 4562
int a = x * y + z;

// want a == 4562
```

Reasoning about code (backwards)

- We can also go backwards to see “what must be true for what I want to be true”

```
int y = 2;

// x * y = 4530
int z = 32;

// x * y + z = 4562
int a = x * y + z;

// want a == 4562
```

Reasoning about code (backwards)

- We can also go backwards to see “what must be true for what I want to be true”

```
// x = 2265
int y = 2;

// x * y = 4530
int z = 32;

// x * y + z = 4562
int a = x * y + z;

// want a == 4562
```

Reasoning about If

- We need to reason through all possible conditions to see what assumptions we can make before/after
- State is a logical OR of all possible branches

```
// assume x >= 0

int z = 0;

// z == 0 && x >= 0
if (x != 0) {
  // z == 0 && x > 0
  z = x;
  // z == x && x > 0 && z > 0
} else {
  // z == 0 && x == 0
  z = x + 1;
  // x == 0 && z == 1
}

// (x > 0 && z == x)
// OR
// (x == 0 && z == 1)

// another way to look at it:
// x >= 0 && z > 0
```

Reasoning about loops

- We can apply this logic to loops.
- Loops should maintain some invariant (something that is true before each iteration)

```
int product(const vector<int>& items) {
  int result = 1;

  for (int i = 0; i < items.size(); i++) {
    // i < items.size()
    // result = 1 * product(items[0], ... items[i - 1])
    result *= items[i];
  }

  // result = 1 * product(items[0], items[items.size() - 1])

  return result;
}
```

Reasoning about objects

- Classes are designed usually to implement some abstraction.
  To implement that abstraction, the class's member variables must maintain some property between each other.
- These properties are the class's invariant
- This invariant should be held before and after every function called on an object until it is deleted or moved

```
// data points to `capacity` number of integers on the heap
// length <= capacity
// data[0 .. length - 1] are valid integers for a user to access
// data[length ... capacity - 1] should not be accessed
class Vec {
  int* data;
  size_t capacity;
  size_t length;
};
```

Reasoning about objects

- Generally if there is no (or minimal) invariant to maintain, then you make a struct.
  - (inner helper-structs can have invariants since those are already "private")

```
// no invariant
struct StringAndInt {
  string str;
  int integer;
};
```

Debugging

- Backwards reasoning model is how I debug.
  - Find where the error is occurring
    - (e.g. I am looking up a key that doesn't exist in a map)
  - What assumptions does my code expect to be true
    - (e.g. state of the key and map)
  - Work backwards from there
    - what needs to happen for my assumption to hold true. Do those assumptions still hold true?

Ed Discussion

- Go there ☺

Lecture Outline

- C++ Aside: namespaces
- Idioms
- Reasoning About Code
- **Clang-tidy and Cognitive complexity**

Clang-tidy

- Static analyzer
- Does something like how we reasoned about code to try and notice bugs
- Also checks to enforce some amount of idioms

Cognitive Complexity

- Most errors are straight forward enough just from reading what the error says.
  - e.g.

```
error: parameter name 'i' is too short, expected at least 3 characters
[readability-identifier-length,-warnings-as-errors]
                    size_t i,
```

- There is one that is not clear and shows up enough to be worth going over now: “Cognitive complexity”
  - The tool calculates “cognitive complexity” of your code and will complain about anything that is too complex. This means you should think about how to break your code into helpers, because if you don’t, clang-tidy will complain and you will face a deduction.

Cognitive Complexity

- This function has Cognitive Complexity of *3*.

```
int function3(bool var1, bool var2) {
  if(var1) { // +1, nesting level +1
    if(var2)  // +2 (1 + current nesting level of 1), nesting level +1
      return 42;
  }

  return 0;
}
```

- Each if statement, loop, etc adds a +1. How “nested” it is can make it worth more

Cognitive Complexity

- Consider the code on the left
- It has a much higher complexity than the one on the right

```
bool foo(string param) {
  if ( !error1 ) {
    if ( !error2 ) {
      if ( !error3 ){
        // do some computation
        return true;
      }
    }
  }
  return false;
}
```

```
bool foo(string param) {
  if ( error1 ) {
    return false;
  }
  if ( error2 ) {
    return false;
  }
  if ( error3 ) {
    return false;
  }
  // do some computation
  return true;
}
```

Cognitive Complexity & Reasoning

- Code that is more cognitively complex is said to be harder to reason about

- Many organizations and standards will mandate that code has a low complexity score.

Cognitive Complexity Example:

- What does this code do? Does it do it correctly?

```
void bar(const matrix<N, N>& m1, const matrix<N, N>& m2, matrix<N, N>& out) {
  for (size_t i = 0; i < m1.rows; i += r) {
    for (size_t j = 0; j < m2.cols; j += r) {
      for (size_t k = 0; k < N; k += r) {
        for (size_t ib = i; ib < i + r; ++ib) {
          for (size_t jb = j; jb < j + r; ++jb) {
            for (size_t kb = k; kb < k + r; ++kb) {
              out[ib, jb] += m1[ib, kb] * m2[kb, jb];
            }
          }
        }
      }
    }
  }
}
```

Cognitive Complexity Example:

- What does this code do? Does it do it correctly?

```
void foo(const matrix<N, N>& m1, const matrix<N, N>& m2, matrix<N, N>& out) {
  for (size_t i = 0; i < m1.rows; ++i) {

    for (size_t j = 0; j < m2.cols; ++j) {

      for (size_t k = 0; k < m2.rows; ++k) {
        out[i, j] += m1[i, k] * m2[k, j];
      }

    }

  }
}
```

Cognitive Complexity Solution

- Break your code out into reasonable functions
- The less you have to reason about at once, the easier it is to reason (in my experience)
- Instead of the first matrix multiply, we can refactor it:

```
void bar(const matrix<N, N>& m1, const matrix<N, N>& m2, matrix<N, N>& out) {
  for (size_t i = 0; i < m1.rows; i += r) {
    for (size_t j = 0; j < m2.cols; j += r) {
      for (size_t k = 0; k < N; k += r) {
        multiply_tile(m1, m2, out, i, j, k);
      }
    }
  }
}
```

- (It would also make sense to change the names so that they aren't "r", "k", etc.)

Credits

- Big inspiration from:
  - CSE 331 @ UW by Hal Perkins
  - Comparing 'Classic C++' and 'Modern C++' Ways to Solve Programming Tasks - Roger Orr - ACCU 2023
  - Probably other things I am forgetting, Will add here if I find them :)

That's all for now!

- Next time: talking about the OS :)

- Hopefully you are doing well ☺