

Simplicity Not Just for Beginners

Kate Gregory
kate@gregcons.com
@gregcons

When We Teach, We Start Simple

- Omit error checking
- Assume we're given a positive or otherwise reasonable number
- Assume all input is well intentioned
- Show how to move things up but not down, or forward but not back



Why?

- So we can show what we're trying to teach
- So the learner can concentrate on one thing at a time
- So it fits on a page with a largish font
- To reduce the cognitive burden on those who read it
- Because the sample is artificial and lacks context

So What Happens?

- Real life is complicated
- You can't omit all that error checking and input sanitizing and handling both directions
- Code grows
- It gets more complicated

What Happens to Developers?

- We reject simple
 - After all, we're not beginners
 - And real life is complicated
- Maybe we even show off a little
 - If it was hard to write, it should be hard to read
 - If it was easy, anyone could do it



Constexpr_{Edge} References_{Resources} Toolchain Impossible
Practices_{MSVC} Learn Nope Code_{Modules} Compiler
Hard to Understand New Features_{Colleagues}
Standard_{Amount} Language_{Tools} Difficulty_{Evolves}
New Stuff_{Dependencies} Older_{Past} Books Difficult to Understand

What is simple code?

- Expressive
- Readable
- Understandable
- Unsurprising
- Transparent
- Self explanatory
- Reassuring
- Pleasant

Is Simpler Better?

- Better means?
 - Faster to write the first time
 - More correct
 - Runs faster or in less memory or less of some other resource
 - Easier to read and understand the next hundred+ times
 - Easier to modify when the world changes
 - More fun to create and have created

Is it faster to write simple code?

- Definitely not
- Much-misattributed quote about no time for a shorter letter
- New habits required
- New ways of looking
- Reviewing, revisiting, refactoring



Is simpler code more correct?

- Usually, yes
- RAI is less to write, and also less to forget
- Take away opportunities to be inconsistent
 - One function with default parameters instead of two similar functions
 - One function that is called with params instead of blocks of copy-and-paste-and-mostly-edit
 - One template instead of two (or ten) similar functions
- Code that moves complexity to abstractions often has less bugs
 - When you move complexity, can it disappear?
- Library code is already tested and has thought of edge cases

Does simpler code run faster?

- Usually, no
 - `for (auto p : people)`
 - `for (auto& p : people)`
- To get faster code you typically have to know and remember something about the language
- Try not to choose simplicity over performance **if** a real choice exists
- But
 - Compilers and optimizers are often much better than you
 - They're guaranteed to be better than someone who's not measuring
- Library code may be faster than what you would write yourself

What's in it for you?

- Simpler code is more readable and debuggable
 - Often more correct too
- Unsurprising code is more maintainable code
- Expressive code is fun to work with
- Other people's code is beautiful



What I Have Learned

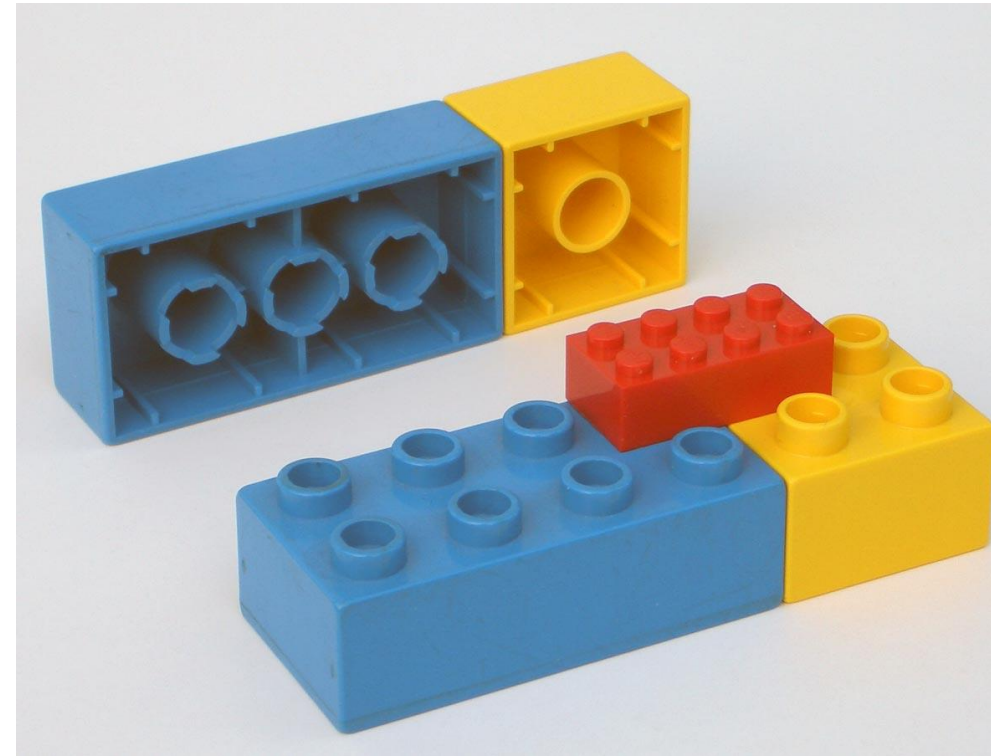
- True simplicity is very hard
- You have to know your tools
 - The language
 - The libraries
 - Our idioms
- Simplicity that is complete is utterly different from “I left that out for simplicity”



OK, Give me the Simple
Rules to Write Simple Code

The Easiest Step

- Know what simple looks like
- Try to write code simply from the beginning
- As it grows, expands, and twists, recognize when it is too complex
 - Do something to make it simpler
- Prevent opportunities to be inconsistent



Names really help

- Often hiding in comments

```
//total of the numbers in the vector
int i = 0;
for (auto n : v)
{
    i += n;
}
```

- Becomes

```
int total =
    accumulate(begin(v), end(v), 0);
```

Using names

- Functions
 - Especially from <algorithm> et al
- Enums
- Constants
- Variables (avoid a, x, i, d1, d2, d3, ...)

Remembering a Variable Name

```
double d3; //total revenue
// . . .
d3 = getGrossReceipts(...);
// . . .
if (something)
    d3 *= 0.95;
// . . .
if (d3 > d7)
{
    // . . .
}
```

Better Name, Simpler Life

```
double totalRevenue;  
// . . .  
totalRevenue =  
    getGrossReceipts(...);  
// . . .  
if (something)  
    totalRevenue *= 0.95;  
// . . .  
if (totalRevenue > oldRevenue)  
{  
    // . . .  
}
```

Short Functions

- Not for readability or to print on a single page
- But so they can be named
- If a function does two things, perhaps it's two functions?
- Consider also “emotionally short” functions such as those in `<algorithm>`
 - Code you didn't write feels very short indeed
 - Code everybody “knows” is also short – no learning and absorbing needed

```
char *ptr;

ptr = strchr(lpCmdLine, ' ');
if ( !ptr )
    return FALSE;
*ptr = 0;
strncpy_s(szDriverName, lpCmdLine,32);
*ptr = ' ';
while( *ptr && isspace(*ptr) )
    ptr++;
if ( !*ptr )
    return FALSE;
sprintf_s(lpszPipename, 256, "\\\\.\\pipe\\%s", ptr);

return TRUE;
```

Use Other People's Code

```
stringstream ss(CmdLine);  
ss >> DriverName >> PipeName;  
  
if (DriverName == "" || PipeName == "")  
    return false;  
Pipename = R"(\\.\pipe\)" + Pipename;  
return true;
```


Avoid really long lists of parameters

- Abstraction is your friend
 - Don't pass 7 bools, pass an Options
 - Don't pass 4 ints, pass a Rectangle or two Points
 - Don't pass 3 strings and a float, pass an Order or Employee
- Maybe this function needs 10 pieces of information because it's really 3 functions, that could be called with smaller parameter lists?
- Maybe this should be a member function of something that knows most of this already?

Don't nest deeply – return early

```
bool Order::Calculate(double x, double y)
{
    if (x < limit)
    {
        if (y >= 0)
        {
            if (shipping)
            {
                //... actual calculation setting some member variable
                return true;
            }
            else
            {
                error = Errors::NotShipping;
                return false;
            }
        }
        else
        {
            error = Errors::YNegative;
            return false;
        }
    }
    else
    {
        error = Errors::XTooLarge;
        return false;
    }
}
```

Don't nest deeply – return early

```
bool Order::Calculate(double x, double y)
{
    if (x >= limit)
    {
        error = Errors::XTooLarge;
        return false;
    }
    if (y < 0)
    {
        error = Errors::YNegative;
        return false;
    }
    if (!shipping)
    {
        error = Errors::NotShipping;
        return false;
    }
    //... actual calculation setting some member variable
    return true;
}
```

Const all the things

- Beyond just “const correctness”
- Mark everything const that you possibly can
- To lower the cognitive burden of future readers
 - Yes, there are 10 local variables here, but only 2 of them vary
- Also a reason to avoid out params and in/out params in functions
 - Return a struct or `std::optional` or even a `std::tuple`
 - Perhaps this should be a member function of the in/out thing
 - Abstraction again

Keep up with the standard

- The *mutable* keyword is 25 years old yet people don't know it
 - Lets you stay more const correct than you otherwise would be
 - Yes, yes, thread-safe, but...
- Use ranged-for loops if you must use loops
- Instead of making certain constructors private to prevent others creating or copying objects, make them *deleted*
- Use non static member initializers
- Use the library

Programming is a social activity in which communication is a vital skill. The code you leave behind speaks.

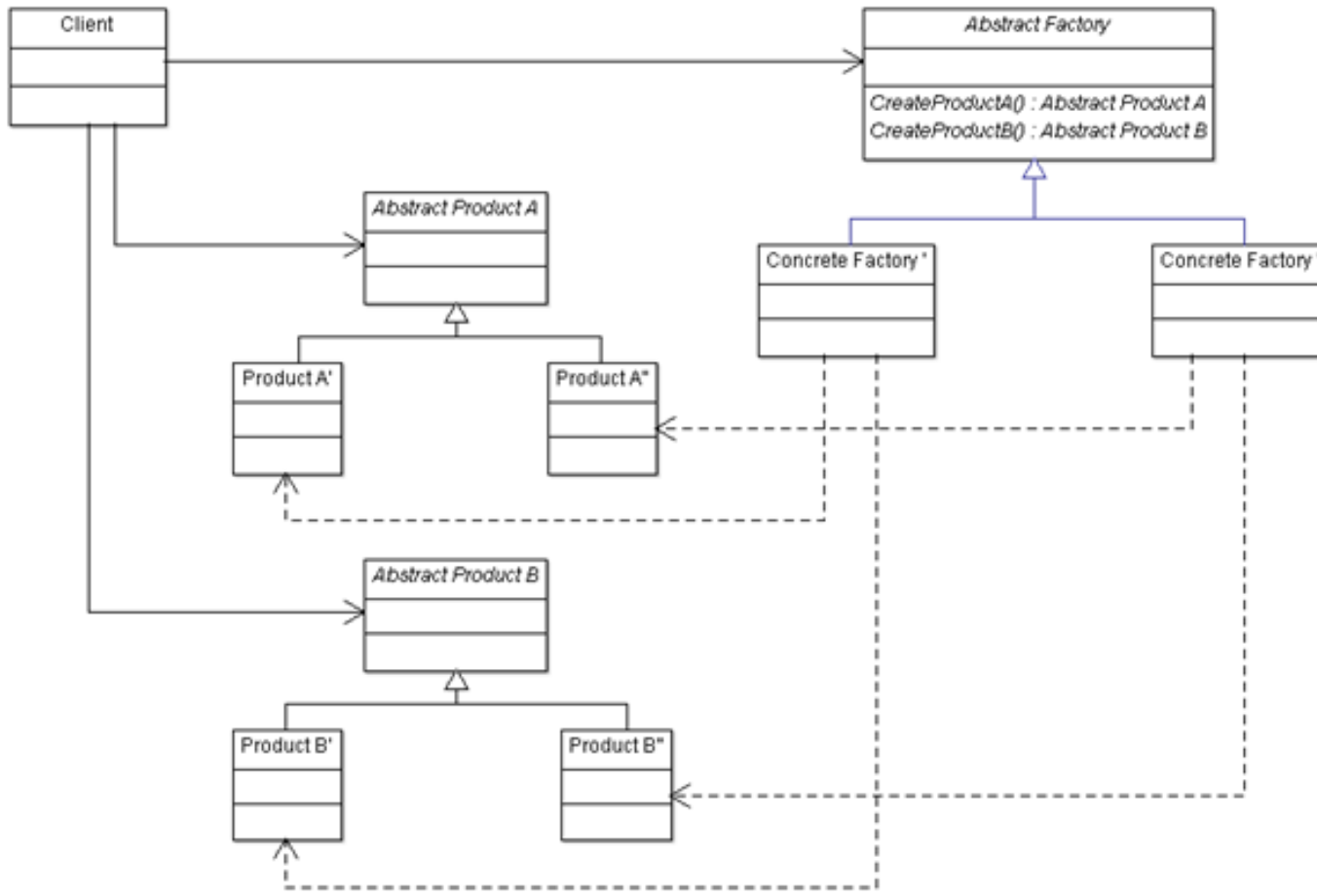
The pit of success

- We can control a lot of the defaults we leave for the next developer
 - Or show to the developer next to us
- Opportunities to be inconsistent are rotten things to leave behind
 - Two versions of a function? They will have to remember to change both
 - One version? No chance to be inconsistent
 - Initialization to defaults with nonstatic member init – ctors can't get inconsistent
- All cleanup in the destructor?
 - They don't have to remember to clean up
 - No need for changes when exceptions are added
- Const correct?
 - They don't need to play chase-the-const later
 - Might also make concurrency less terrifying later
- Good names for everything? Short functions?
 - They will keep the pattern going





Don't be an architecture astronaut



```
AbstractFactory* factory =  
    FactoryMakerSingleton::getInstance()->getFactory();  
shared_ptr<Subject> subject = factory->createSubject();  
subject->attach(factory->createObserver());  
shared_ptr<Command> command =  
    factory->createCommand(subject);  
command->execute();
```

Simplicity Paradox

- The things you do to make code simple can make it more complex
- It is NOT POSSIBLE to write simple rules for how to write simple code
 - Unless you write vague rules
 - “good”, “short”, “a lot”, “not many”
 - “usually”, “without a good reason”
- This is a law of the universe
 - What speed should you drive at? What lane should you be in? When do you change lanes?
 - The baby is crying. What should you do?

Not all questions have simple answers

- Should you use exceptions?
- How long should a function be?
- What is a good variable name?
- Are default parameters confusing?
- Are overloads confusing?
- Should we really never use raw loops or raw pointers?

Moving to harder steps

- Simple practices like naming and keeping things short are easy enough
 - They require some judgment
- Ideally you write your code like this from the beginning
 - You can refactor to be simpler when it snarls up
 - You might be brought in to fix snarled code
- But that is not the whole story
- For big gains
 - In performance
 - In understandability, reusability
 - In maintenance pain
- You must change the norms of your team
 - Current and future

Kinds of Complexity

```
for(uint8_t i=0; i < GetSize(); i++)  
{  
    //...  
}
```

- Guess what the return type of GetSize() is?
 - uint16_t
 - And it needs to be – won't fit in 8 bits
 - So that means?
- C++ is so complicated with all those darn different types

Failure to Encapsulate

- `i` and `GetSize()` have no relationship
- There isn't a real collection here
- Fix that

```
for(x& : coll)
{
    //...
}
```

Idioms, Library Abstractions, Commonality

- These are old friends you can learn to recognize
- This loop touches every element in the collection; I should use a ranged for instead of a traditional for loop
 - Or something from `<algorithm>`
- “This is obviously a rotate”
- There is already a stack in the Standard Library
- I bet someone already wrote a pretty good json parser, logger, http-getter, etc
- If I move the initialization of this object to a function or immediately-invoked lambda, I can make it const
 - And with a lambda I can just [&] instead of working out what the parameters to the function should be
- Using raw string literals here will help readability

The Harder Step

- Know what we all should know
 - Is surprising people simple?
 - It is not enough that you know something. The reader must know it
- Replace your complicated things with
 - **Familiar** idioms and language constructs that express your intent
 - Well known library classes and functions **that others will recognize**
 - Appropriate abstraction that becomes **a thing to learn** in your code
 - Moving complexity inside your abstraction
- Without
 - Omitting needed capabilities
 - Hiding core information behind abstractions and indirections
 - Factories, interfaces, InjectorFactoryAdapter
 - Preventing future changes
 - Global mutable state, singletons, hardcoding things because “it’s simpler”

The Future

- As simple as possible, but no simpler!
- Simplicity in the larger context
 - Using a magic number is simpler now than setting up a const variable (or an enum for several of them) but will it be simpler to understand later?
 - Adding a global is simpler now than adding a parameter to a long chain of function calls, but later when people don't understand what controls behaviour, was it simpler?
- Remember simpler code isn't always faster or easier to write
 - Take the time to write the shorter letter

The Hardest Steps

- Knowing that border between “skipping stuff to make it easy” and genuinely elegant simplicity
- Being brave enough to present simple code
 - “Is that all you did?”
 - “I thought you were creative/innovative/an architect?”



The Bravery

- Which side of that border are you on?
 - Is this simple-didn't-think-it-through or simple-brilliant?
- If you're relying on knowing your language and library, do others?
- Now your code is expressive and transparent, can you be replaced?
- Does your code reflect you and your abilities?
 - What are you leaving behind? How does it speak?
- How far are you from being a beginner?

Call to Action

- Learn
- Read
- Care
- Test
- Communicate