



Naming is Hard: Let's Do Better

Kate Gregory

kate@gregcons.com

www.gregcons.com/kateblog

[@gregcons](#)

Naming Things

- It matters
- This is how we explain ourselves
 - To each other
 - To our customers, users, managers, ...
- C++ people are famously bad at it
- It's a learned skill
 - That means we can improve

Names Carry Meaning

- Inactive date, end date or expiry date?
 - Can you use it on that date?
 - Is it actually a date and time?
 - What do the users call it?
- Does `empty()` empty a collection, or tell you whether or not it's empty?
 - What does `clear()` do?

It Matters Everywhere



Nicole Archambault

@lavie_encode

Follow



it's really hard at times for me to follow examples where they use one-letter variables.

if you're trying to teach coding, maybe try to make it a bit more descriptive or creative than a/b/c? Newbies can shorten them once they feel more comfortable.

9:43 PM - 11 Jul 2019

4 Retweets 23 Likes



Names tell a story

- If you name things well, you need less comments
- Bad names confuse

```
void setStatus(ApplicationStatus s)
{
    status = s;
}
```

```
void setStatus(ApplicationStatus s)
{
    status = s;
    lastUpdated = now();
}
```

Names tell a story

- If you name things well, you need less comments
- Bad names confuse

```
void setStatus(ApplicationStatus s)
{
    status = s;
    lastUpdated = now();
    if (status == ApplicationStatus::Approved)
    {
        // . . .
    }
    if (status == ApplicationStatus::Denied)
    {
        // . . .
    }
}
```

Names tell a story

- If you name things well, you need less comments
- Bad names confuse

```
void Approve()
{
    status = ApplicationStatus::Approved;
    lastUpdated = now();
    // . . .
}
void Deny()
{
    status = ApplicationStatus::Denied;
    lastUpdated = now();
    // . . .
}
```


Naming is hard

- Giving something the correct name may happen long after it's first written
- When refactoring, one technique is to give functions literally nonsense names
 - Dfhtjd
- Or extremely verbose ones
 - SetShippingCostsAdjustTotalAndMarkAsShipped
- Eventually you give things their “true names”

The background is a blue gradient with decorative white circuit-like lines in the corners. The text is centered and reads:

Naming
requires
empathy

An <algorithm> story

- sort

1	5	4	2	9	7
---	---	---	---	---	---

- partial_sort

1	2	4	5	7	9
---	---	---	---	---	---

1	2	4	9	5	7
---	---	---	---	---	---

- partial_sort_copy

1	2	4
---	---	---

- top_n

Consistency

User Nomenclature

- Names exist outside your code
 - Headings on reports
 - Emails
 - Prompts
 - Human conversations about the system
- Use the same words/names in all contexts
 - Everyone should call things by their proper names, everywhere
- Don't use the same words for different things
 - Be arbitrary: a certification expires but a coupon becomes invalid
 - Then stick to it
- Don't accept similar English words in conversation; stay precise
 - Expired/inactive/invalid
 - Coupon/voucher/discount

Don't Invent Business Words

- Naming pieces of a function is hard
- Avoid pre/post and other “dependent” names
 - Unless the business uses them
- Prefer single English words like Save or Location to implementation-focused words like UpdateConfigFile or StorageCoOrdinates
 - Eventually

```
PreLoad(user, section);  
Load(begin, end, filter);  
PostLoad(user, category);
```

Don't Mismatch Natural Pairs

- Begin goes with end, not last (last goes with first)
- Create goes with destroy, not cleanup
- Open goes with close, not release
- Next goes with previous, not rewind
- Put goes with get, not retrieve
- Source goes with destination, not target

Some Heuristics for Functions

Verbs Help Functions Make Sense

- Ideally a service of the system or object: `Update`, `CalculateTax`, `DeductFees`, `MarkAsRead`
- You may want to use helper verbs
 - `IsEmpty()` is less ambiguous than `Empty()`
 - `[[nodiscard]]` is a signal that people misunderstand the name you're using now
 - `HasX()` and `CanX()` are also useful
 - `IsShippable()` vs `CanShip()` vs `getShipStatus()`

Order Matters

- If you are going to have a noun and a verb in each function name, should they be VerbNoun or NounVerb?
 - We never say TaxDetermine() or FeesCharge() but we do say EmployeeUpdate() and InventoryCheck() – why?
- Make a deliberate choice, think about it

Tools Matter

- If similar functions all start the same
 - They are listed together in IDEs that show alphabetical lists of functions
 - They may be sorted together by tidiers that do so
 - You may have to type more of them before you can autocomplete
- Do HasEntries and HasRisks belong together, away from GetRisks and GetEntries, and AddEntry and AddRisk?

Parameters

- Serve two purposes
 - They are local variables in the function scope, so you name them with that in mind
 - Never shadow member variables, but please also don't argx
- They are cues to the function caller
 - Never omit them in headers

```
}  
  
Employee newHire("Kate Gregory"s, 1);
```

▲ 3 of 3 ▼ Employee(std::string FullName, int YearlySalary)

Some Heuristics for Classes

Classes Are Nouns

- Anything ending in er (et al) is suspect without a noun
- Don't overdecorate
 - Suffixes like proxy, factory, adapter, interface – are they really needed?
 - Monad? Singleton? AbstractFactory? Base? Impl?
- Don't list the contents
 - NameAndAddress? NameAddressAndPhone? NameAddressPhoneAndEmail?
 - ContactInfo
- Remember the purpose of this class in the larger system

Members

- The class name is implicitly included; don't repeat it
 - `Employee::EmployeeName` should just be `Employee::Name`
 - `Employee::PrintEmployeeRecord()` should be just `Employee::PrintRecord()`
- Adjectives are your friend
 - `FullName` not `Name`
 - `AnnualSalary` not `Salary`
- Avoid encoding type
 - Possible exception for dates (`HireDate` not `Hired`; `ShipDate` not `Shipped`)

Traditional Member Function Names

- If you put real work in a constructor or a destructor, others will know when it happens
 - Eg open/close a file
- We recognize get/set for better or worse
 - Try to reserve get for {return thingy;} and use fetch/read/load/retrieve otherwise
 - People expect getThingy() to be const
- `void Temperature(int t)` and `int Temperature() const` are also well known
 - Some people really hate them; be careful

Enums

- Prefer scoped enums
 - So you don't have to encode enum name into values eg NT_OK, SB_OK etc
- As with members, don't repeat the enum name
- Think about whether or not to “leak” the enum values outside of the class they help

```
if (nextApplication.getStatus() == ApplicationStatus::Approved)
{
    // . . .
}
```

```
if (nextApplication.isApproved())
{
    // . . .
}
```

Some Heuristics for Local Variables

Rarely, Shorter is Better

- *i, j, k* in loops
 - If you must have them: consider a ranged `for` or an algorithm
- *x, y, t, v* etc in scientific calculations
 - Use the same notation as the formula
- When it has a tiny scope
 - almost like a pronoun

```
string r = getNextResponse();
while (!r.empty())
{
    responses.take(r);
}
```

Most of the Time, Longer is Better

- Nouns
- Add Adjectives liberally
 - Next, current, remaining, active, . . .
- Avoid Encoding Only Type
 - And other forms of overdecorating
- Focus on the purpose of the variable, not what it holds
 - Why are you building a collection of Policies? What is that collection for?
- Consider the greppers

```
Employee e;  
Vector<Policy> policies;  
double d;
```

Abbr

- Abbreviations are generally bad
 - Special dispensation to id
- First syllable only may be obvious to you, not others
- First letters is generally write-only
 - ar
 - ti
 - rd
- Vowels are free

Some Heuristics for Templates

All of the Above

- If you write a templated function, it's a function, so use those rules
- If you write a templated class, it's a class, so use those rules
- Then there's the matter of the typenames
 - `template<class T>` or `template<typename T>`?
 - Compiler doesn't care
 - Some humans say `typename` if `int` etc are ok, `class` if they are not
 - Very weak signal

Typename Names

- Only one? T
- Two? Please be meaningful



```
template <class _Elem, class _Traits = char_traits<_Elem>,  
         class _Alloc = allocator<_Elem>>  
class basic_string { // null-terminated transparent array of elements  
  
    // . . .  
  
template <class _Ty, class _Alloc = allocator<_Ty>>  
class vector { // varying size array of values
```


The background is a blue gradient with decorative white circuit-like lines in the corners. These lines consist of straight segments and small circles, resembling a stylized electronic circuit or data paths.

When does naming happen?

When You Know What It Is

- Sometimes before you even write the code
- Sometimes after you've been reading or debugging for some time
- Never miss an opportunity to fix a name
 - Have good tools for this
- Sometimes when code changes, names need to change too

Temporary Names

- Especially in refactoring
- But also in greenfield work
- How will you remember to give it a true name later?
- Never choose something rude or embarrassing

Names as Motivation and Review

- `Process()` can be 5000 lines long and nobody understands it
 - When you insist on a better name, this will change
- Things are hard or impossible to name when they are the wrong size
 - Demanding good names makes things the right size
- Good names make the code tell a story
 - Demanding a story will get you good names

Better Naming

- Care about the code you write, and the people who will read it
- Think about the purposes names serve and how they are used
- Don't be paralyzed by being unable to name something at first
 - Fix it later!
- Demand good names from yourself and those around you
- When you learn what something is, fix its name
- When you change what something is, change its name
- Use consistency and story telling to guide your choices