

CODING AND SOLUTIONING

(Code Layout ,Reliability ,Reusability)

Date	13 November 2022
Team ID	PNT2022TMID26105
Project Name	Real-Time River Water Quality Monitoring and Control System
Maximum Mark	2 Marks

KEY IDEA 1 - Code should be easy to understand.

KEY IDEA 2 - Code should be written to maximize the usage for as many applications as possible

Naming Convention:

KEY IDEA 3 -

- Choosing specific words
- Avoiding generic names (or knowing when to use them)
- Using concrete names instead of abstract names
- Attaching extra information to a name, by using a suffix or prefix
- Deciding how long a name should be
- Using name formatting to pack extra information

Choose Specific Words

You have to choose the words that are very specific and avoiding 'empty' words. For example, the word **get** is very unspecific, as in this example: `def GetPage(url):`

The word `GetPage()` doesn't really say much. Does this method get a page from a local cache, from a database, or from the Internet? If it's from the Internet, a more specific name might be `FetchPage()` or `DownloadPage()`

The name `Size()` doesn't convey much information. A more specific name would be `Height()`, `NumNodes()`, `MemoryBytes()`, etc.

The name `Stop()` is okay, but depending on what exactly it does, there might be a more specific name: `Kill()` if it's a heavyweight operation that can't be undone. `Pause()` if there is a way to `Resume()` it.

Finding more colourful words

Don't be afraid to use a thesaurus or ask a friend for better name suggestions. English is a rich language, and there are a lot of words to choose from.

send ~ deliver, dispatch, announce, distribute, route find ~ search, extract, locate, recover start ~ launch, create, begin, open make ~ create, set up, build, generate, compose, add, new

KEY IDEA 4 - It's better to be clear and precise than to be cute.

Avoid Generic Names like `tmp` and `retval`

Instead of using an empty name like this, **pick a name that describes the entity's value or purpose**

Using a generic name sometimes will help you to detect a bug **`tmp` or**

`tmp`

```
if (right < left) { tmp
= right; right = left;
left = tmp; }
```

In cases like these, the name `tmp` is perfectly fine. The variable's sole purpose is temporary storage, with a lifetime of only a few lines.

But here's a case where `tmp` is just used out of laziness:

```
String tmp = user.name(); tmp += " "
+ user.phone_number(); tmp += " "
+ user.email(); ...
template.set("user_info", tmp);
```

Even though this variable has a short lifespan, being temporary storage isn't the most important thing about this variable. Instead, a name like `user_info` would be more descriptive.

In the following case, `tmp` should be in the name, but just as a part of it:

```
tmp_file = tempfile.NamedTemporaryFile() ...
SaveData(tmp_file, ...)
```

Notice that we named the variable `tmp_file` and not just `tmp`, because it is a file object. Imagine if we just called it `tmp`:

```
SaveData(tmp, ...)
```

Looking at just this one line of code, it isn't clear if `tmp` is a file, a filename, or maybe even the data being written.

Loop Iterators

`i`, `j`, `iter`, it can be used as indices and loop iterators (In fact, if you used one of these names for some other purpose, it would be confusing - So, Don't do that). But sometimes there are better iterator names than `i`, `j` and `k`

```
for (int i = 0; i < clubs.size(); i++)
for (int j = 0; j < clubs[i].members.size(); j++)
for (int k = 0; k < users.size(); k++)
if (clubs[i].members[k] == users[j])
cout << "user[" << j << "] is in club[" << i << "]" << endl;
```

In the if statement, `members[]` and `users[]` are using the wrong index. Bugs like these are hard to spot because that line of code seems fine in isolation:

```
if (clubs[i].members[k] == users[j])
```

In this case, using more precise names may have helped. You can name them as `club_i`, `member_i`, `user_i` or more succinctly (`ci`, `mi`, `ui`). This approach would help the bug stand out more: `if (clubs[ci].members[ui] == users[mi]) # Bug! First letters don't match up.`

As you've seen, there are some situations where generic names are useful. A lot of the time, they're overused out of pure laziness. This is understandable—when nothing better comes to mind, it's easier to just use a meaningless name like `foo` and move on. But if you get in the habit of taking an extra few seconds to come up with a good name, you'll find your **naming muscle** builds quickly.

Prefer Concrete Names over Abstract Names

For example, suppose you have an internal method named `ServerCanStart()`, which tests whether the server can listen on a given TCP/IP port. The name `ServerCanStart()` is somewhat abstract, though. A more concrete name would be `CanListenOnPort()`. This name directly describes what the method will do.

Please don't try to smash two orthogonal ideas into one. (Follow Single Responsibility Rule can help you easier to naming a method)

Attaching Extra Information to a Name

Values with Units

```
var start = (new Date()).getTime(); // top of the page ...
var elapsed = (new Date()).getTime() - start; // bottom of the page
document.writeln("Load time was: " + elapsed + " seconds"); More explicit:

var start_ms = (new Date()).getTime(); // top of the page ...
var elapsed_ms = (new Date()).getTime() - start_ms; // bottom of the page
document.writeln("Load time was: " + elapsed_ms / 1000 + " seconds");
Start(int delay) - delay -> delay_secs CreateCache(int size) - size -
> size_mb ThrottleDownload(float limit) - limit -> max_kbps Rotate(float angle) - angle - >
degrees_cw
```

Encoding Other Important Attributes

Many security exploits come from not realizing that some data your program receives is not yet in a safe state. For this, you might want to use variable names like `untrustedUrl` or `unsafeMessageBody`. After calling functions that cleanse the unsafe input, the resulting variables might be `trustedUrl` or `safeMessageBody`.

A password is in “plaintext” and should be encrypted before further processing - password - better name: `plaintext_password`

A user-provided comment that needs escaping before being displayed - comment - better name: `unescaped_comment`

Bytes of html have been converted to UTF-8 - html - better name: `html_utf8`

Incoming data has been “url encoded” - data - `data_urlenc`

You shouldn't use attributes like `unescaped_` or `_utf8` for every variable in your program. They're most important in places where a bug can easily sneak in if someone mistakes what the variable is, especially if the consequences are dire, as with a security bug. Essentially, if it's a critical thing to understand, put it in the name.

How long should a Name be

How do you decide between naming a variable `d`, `days` The answer depends

on exactly how the variable is being used.

Shorter Names Are Okay for Shorter Scope

Identifiers that have a small scope (how many other lines of code can "see" this name) don't need to carry as much information.

```
if (debug) { map<string,int> m;  
    LookUpNamesNumbers(&  
        m);  
    Print(m);  
}
```

you can get away with shorter names because all that information (what type the variable is, its initial value, how it's destroyed) is easy to see.

Even though `m` doesn't pack any information, it's not a problem, because the reader already has all the information she needs to understand this code.

```
class Program  
{  
    0 references  
    static async Task Main(string[] args)  
    {  
        WriteLine("Please type the username for the desired user:");  
        var username = ReadLine();  
  
        var github = new GitHubClient(new ProductHeaderValue("MyAmazingApp"));  
  
        try  
        {  
            var user = await github.User.Get(username);  
            WriteLine($"The user {user.Name} was successfully retrieved!");  
            WriteLine($"{user.Name} has {user.PublicRepos} public repositories. Do you want to see the list? (y/n)");  
            var response = ReadLine();  
  
            if (string.Equals(  
                "y",  
                response,  
                StringComparison.InvariantCultureIgnoreCase))  
            {  
                var repos = await github.Repository.GetAllForUser(username);  
                foreach (var repo in repos.OrderBy(x => x.CreatedAt))  
                {  
                    WriteLine($"{repo.CreatedAt:yyyy-MM-dd} | {repo.Name}");  
                }  
            }  
        }  
    }  
}
```