overload 156
APRIL 2020 £4.50

Deconstructing Inheritance

Inheritance can be overused. We consider how it can be abused, and what alternatives exist.

Pass the Parcel

A look at the advanced features of Python's module and package system

Quick Modular Calculations

We conclude this series with a new algorithm that works for 64-bit operands

It's About Time

How easy is it to make code wait for a time period before doing something?

Profiting from the Folly of Others

We learn about private access in C++

Using Compile Time Maps for Sorting

How to achieve compile time sorting of data with a map

A magazine of ACCU
ISSN: 1354-3172

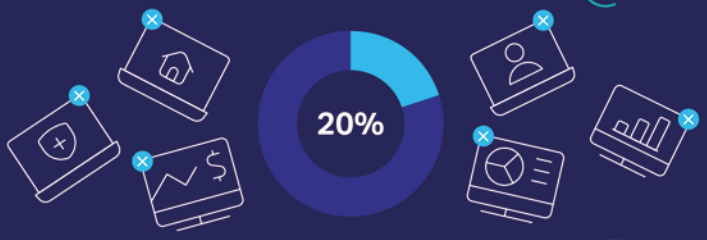
Remote access software:

opentext™

Ensuring productivity by eliminating single points of failure

Unresponsive programs

Knowledge workers on average see productivity drop by as much as **20%** because of unresponsive applications.[1]



Increase productivity and eliminate downtime

A remote access platform maximizes end user productivity with:

- Extremely fast responsiveness resulting from highly efficient data compression protocol.
- Access to Windows®, UNIX® and Linux® applications and desktops.
- Auto-resume for network interruptions.
- Remote collaboration and built-in screen sharing.
- Stable sessions.
- Highly available architecture out of the box.

How it works

Example:
Ensure a highly available, load balanced remote desktop infrastructure for users.

1. Establish load balancing rules based on the number of current sessions and CPU or memory utilization of available hosts.
2. Assign users to the least loaded servers in their host pool with the load balancer.
3. Install connection nodes to offload CPU-intensive tasks and provide faster performance of application servers.
4. Ensure a user's work is never lost with auto-reconnect, which resumes a session when the device is back online.
5. Configure servers in a high availability (HA) cluster to distribute website load and eliminate single points of failure.

The result

Provide users with reliable and fast access to work desktops and server applications from any platform and location, avoiding costly downtime, interruptions and loss of work.

OpenText™ Exceed™ TurboX empowers a global workforce with a high-performance remote access solution that ensures accessibility to graphically demanding applications and desktops through a web browser.

Key partners include:

Agisoft · ASPOSE · ATLASSIAN · axure · BrowserStack · DevExpress · embarcadero · Hex-Rays · intel Software · Microsoft Silver Partner · qbs PUBLISHING · Sketch · SMARTBEAR · SPARX SYSTEMS · Telerik Develop experiences · think-cell · Visual Paradigm

For your latest software needs, contact our team on:

020 8733 7101 **sales@qbs.co.uk** **www@qbs.co.uk**

QBS PUBLISHING

OVERLOAD 156

April 2020

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Jon Wakely
accu@kayari.org

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 157 should be submitted by 1st May 2020 and those for Overload 158 by 1st July 2020.

The ACCU

The ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Overload is a publication of the ACCU
For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

R.E.S.P.E.C.T.

Respect can mean many different things. Frances Buontempo muses on its myriad meanings.

Respect to our writers. We've had so many submissions for this issue, I have spent all my time reading through them instead of planning an editorial. Good work people! I also had to stay in the office late for two nights running in order to help with a release, which obviously used even more of my time. I bet several of you manage to release your code during office hours. Perhaps you should write in and tell us how. Anyway, this adds to my excuse for not writing an editorial. On the plus note, one of the business people said thank you on a slack channel when we were finally done. Never under estimate the power of saying thanks. Years ago, in another company, one of the senior devs always said thank you at the end of the day, which seemed a bit odd on the face of it. I'd turned up to work, and mostly done what I was told. Why a word of thanks after that? It turns out, it made me feel appreciated and keener to work harder. Or talk to him when I had ideas of how to make things better, quicker, leaner, meaner. You know. Saying thank you is a small act of kindness that can make a big difference. Respect Keith. If you're reading this.

Who do you respect? Do you have a favourite writer or speaker? Or band or composer? I'll bet there's someone. What makes you respect them? Consistent quality performance? Ability to adapt and react to an audience or situation? Who would you choose to partner with you on a late night release? Why? We recently got a new team member. He's a rubber duck, called Quackson. He's the best listener ever. I don't know how he manages to sit still and say nothing while others rant and rail at him. I find it very hard to do that. I tend to say, "Hang on, that can't be right." Or, "I don't understand." That sort of thing. I have much to learn from the duck. The colleague who brought in Quackson had never heard of rubber duck debugging [RubberDuck] before, but spontaneously started telling the duck what he was up to. It's a bit like an imaginary friend you tell your woes to, explaining what you are up to and why, and what you need help with. Then,

> At some point you will tell the duck what you are doing next and then realise that that is not, in fact, what you are actually doing. The duck will sit there serenely, happy in the knowledge that it has helped you on your way.

Sometimes you give people grudging respect. My editor of choice is Vim. At the expense of fuelling an editor war, a colleague some years ago used Emacs. Watching him find entries in logs and reformat text was a wonder to behold. I know how to exit Emacs, which is good enough for me; however, seeing a man who could drive his tool of choice so well commanded immediate respect. Good work, Moshe.

Respect usually revolves around interactions between people. Respect can take the form of being mindful of other's needs. Don't stand in front of the white board and talk to it, if you expect a room of people to hear you. If someone in the meeting is deaf or partially sighted, you need to be even more thoughtful about the layout of the physical space and the format taken. Bigger font sizes. Making sure the person speaking can be seen, if someone needs to lip read. If people are dialled-in to a meeting, make sure they get a chance to speak as well. Don't speak over people. I'm sure you can draw up your own list. Alternatively, avoid the problem completely and never have a meeting.

Many organisations have a hierarchy, no matter how flat they claim it to be. This often carries an implicit assumption that more important people will automatically be obeyed no matter what. Respect the badge. Respect your elders and betters. This sometimes means blind obedience. In some situations, there isn't time to argue and no harm will come from doing something now and dealing with any fallout later. In other situations, much harm can happen. A classic, often quoted, example is the so-called 'Charge of the Light Brigade'. During the Crimean War, in 1854, Lord Cardigan led the light cavalry, armed with swords, against Russian forces, armed with guns. Due to a miscommunication, they were sent straight up against the artillery and most ended up dead or injured. [Wikipedia-1].

Respecting those in authority, your elders, or even your parents is not the same as doing exactly what they ask. On another late night release, many years ago, a senior manager said he thought you shouldn't call fabs directly as it could be slow. A team mate thereby halted proceedings and tried to make us go through the code and swap out the calls for a hand-crafted piece of code. I eye-balled the disassembler with another co-worker and could see it was one floating point instruction. Now, I believe the absolute floating point function may have been slower than hand crafted versions once upon a time, but things change. While respecting what the manager said, we showed him the compilation to a single instruction. He was horrified that we'd even considered holding up the release to hack around the code at midnight. Not the charge of the light brigade, but… Try talking to senior people once in a while, and checking what they say. You might get home earlier. Respect is not the same as mindless obedience.

Respect and obedience, though often conflated, are not the same thing. The root of 'respect' is 're' for 'back', and 'specere' for 'to look at', giving a similar word 'regard', re+gard, or 'back' plus 'guard' or 'watch'. You should have watched Moshe driving Emacs, though! Perhaps respect has something to do with looking and seeing. Not just glancing and vaguely guessing what's going on, but actually looking and paying attention. James 1:23-24 says:

> Anyone who listens to the word but does not do what it says is like someone who looks at his face in a mirror and, after looking at himself, goes away and immediately forgets what he looks like.

Do you remember what you look like? Perhaps. Look, carefully; watch, in detail; act, respecting the people and situation you observe.

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

Respect isn't always about people. Given coding guidelines, you can either respect them, perhaps automatically enforcing them, or you can subvert or blatantly disobey from time to time. If you code in C++ and request a function to be inlined, does the compile respect your wishes? Inline is a request, and therefore might not be respected.

> No matter how you designate a function as inline, it is a request that the compiler is allowed to ignore: the compiler might inline-expand some, all, or none of the places where you call a function designated as inline. (Don't get discouraged if that seems hopelessly vague. The flexibility of the above is actually a huge advantage: it lets the compiler treat large functions differently from small ones, plus it lets the compiler generate code that is easy to debug if you select the right compiler options.) [CPP]

I wonder what the 'right' compiler options are? Have an experiment and report back. It's not just inline. Introduced in C++11, **`_Alignas`** is also a keyword in C11, where alignas is a preprocessor macro. [CPPref]. Aside from this potential clash, the request for alignment may not be respected. Over-alignment (asking for a bigger number) may not be respected [Stackoverflow]. Some may say alignas is always respected for a reasonable use, i.e. no more than **`max_align_t`**. If surprising things happen in your code, it might be down to your misunderstanding. Optimisations can uncover data races and similar undefined behaviour (UB). Nasal demons may result [Maudel13].

> One of the responsibilities is to learn and understand the contract between yourself and the compiler. If you break the contract, then anything can, and will, happen.

I've never witnessed demons flying from someone's face, but have been confused by UB once in a while. You should respect the tools you are using, and take time to learn them. There will always be more to learn, but that's what makes programming fun. If you get stuck, ask for help. The accu-general mailing list is a good place to turn to.

Enough of your compiler not respecting your requests, or you not respecting stated boundaries or requirements. Sometimes, respect is meant in a sense of perspective, or looking in a certain direction. This fits well with the sense of looking back. We talk of velocity changing with respect to time in physics, of the derivative of y with respect to x in calculus. It crops up so often, you tend to see WTR as short hand. It is important to clarify when ambiguity is present. If a function has several variables, talking of its derivative might be unclear. In computer science, the complexity of a function can be with respect to execution time or memory usage. This respect aims to give precision and clarity. These are fundamentally important for clear communication. I suspect most blazing rows in meetings happen when communication has broken down. Finding ways to reword ideas and suggestions can bring about revelatory changes in perspective and attitude.

I mentioned hierarchies earlier. Sometimes you may find yourself in a minion-type role. If you have to pass a code review before being allowed to commit code, this may feel like a subordinate position. However, code reviews can be an enabler. I recently had a great code review. We refactored my changes together and ended up with much clearer, and neater code. The reviewer encouraged me to fly in the face of some anti-patterns that were starting to emerge in the code base. We made everything feel better, just for a small corner of the code. We had fun. Not all code reviews work like this. Sometimes someone demands a specific code style or approach, and seems to be nit-picking for the sake of it. Nits should be picked, but a mixture of kindness and encouragement goes a long way. Pair programming can be contentious too. Todd Sedano gave a workshop at *Agile 2019*, entitled 'Considerate Pair Programming' [Sedano19]. An important point he raised was "adjusting for any power imbalances". There will always be expertise imbalances, but real experts are usually good at working with other people, apart from the occasional diva moment. We all have our off-days. I encouraged the student on our team to review my code a while ago. He was surprised, since he saw himself as too inexperienced to comment on my code. I reassured him that he could at least tell me if he could follow the code. Power imbalance levelled. If you do code reviews or pair programming, are they conducted with respect? Find out what it means, to you at least.

Lean software development talks of empowering the team [Wikipedia-2]. Wikipedia emphasises managers empowering rather than telling workers how to do their own job. We are told, "Respecting people and acknowledging their work is one way to empower the team." The idea is an empowered team gets things done. The article also mentions trust; "trust them to get the work done". In some senses, trust and respect are similar. You might claim respect is earned. If someone manages to do something amazing or useful, or consistently manages to simplify a difficult problem, or steer a meeting to clear action points, they will earn respect for their particular talent. If a new manager is parachuted in, they may well be treated with suspicion until they prove themselves. You could say until they earned your respect, or trust. This is orthogonal to being thoughtful, considerate and kind, which could also be described as respect. If things are going wrong, don't forget Hanlon's razor – "Never attribute to malice that which is adequately explained by stupidity." But my corollary is, "Don't call people stupid." Some actions are daft, and everyone has their off-days. It's OK to refuse to obey orders, and question things. But strive to be kind. Let's help each other. And keep writing articles.

References

[CPP] https://isocpp.org/wiki/faq/inline-functions

[CPPref] https://en.cppreference.com/w/cpp/language/alignas

[Maudel13] Olve Maudel 'Demons may fly out of your nose', *Overload* 115, https://accu.org/index.php/journals/1857

[RubberDuck] https://rubberduckdebugging.com/

[Sedano19] https://agile2019.sched.com/event/OD03/considerate-pair-programming-an-interactive-workshop-todd-sedano

[Stackoverflow] https://stackoverflow.com/questions/35365624/alignas-keyword-not-respected

[Wikipedia-1] https://en.wikipedia.org/wiki/Charge_of_the_Light_Brigade

[Wikipedia-2] https://en.wikipedia.org/wiki/Lean_software_development#Empower_the_team

Pass the Parcel

Python's module and package system has many features. Steve Love explores some more advanced ones.

This is the second instalment, following on from the introduction to Python modules [Love20]. In that article, we looked at how to create your own modules, a little on how to split your program into modules to make sharing of the code easier, and how to structure packages to make testing them easier. In this article, we will take a more detailed look at making the packages you create easier to import and use. We will explore more ways to share your packages with others, and some ways of ensuring you can always have a dependable environment in which your code runs.

A little more on the import statement

In the previous article, we described a simple package with code to take input in one structured format, e.g. JSON or CSV, and turn it into another format, perhaps performing simple transformations on the way.

Listing 1 shows the basic usage of the code in our own package **`textfilters`**. For the sake of keeping the package contents tidy, we created some sub-packages so that the code to perform transformations was separate from the main package, and the tests for the package were all in one place, also separate. The package structure we ended up with is shown below.

```
<project root>/
  |__ main.py
  |__ textfilters/
      |__ __init__.py
      |__ csv.py
      |__ json.py
      |__ transformers/
          |__ __init__.py
          |__ change.py
          |__ choose.py
      |__ tests/
          |__ __init__.py
          |__ test_filters.py
          |__ test_change.py
          |__ test_choose.py
```

This structure explains the two import statements in Listing 1: the first such import brings in the main filters for taking (in this case) CSV input and turning it into JSON output. The second import is pulling a single function – **`change_keys`** – from a module called **`change`**. This module is in a package named **`transformers`**, which is a sub-package of the **`textfilters`** package.

As we mentioned in the previous article [Love20], there are a few ways we could arrange the import statements, with alterations to the usage. The portion of the import line after the **`import`** statement effectively defines the namespace, so that first import line could be:

```
import textfilters.csv
```

Steve Love is an independent developer constantly searching for new ways to be more productive without endangering his inherent laziness. He can be contacted at steve@arventech.com

```
from textfilters import csv, json
from textfilters.transformers.change import
change_keys
import sys
if __name__ == '__main__':
  def key_toupper( k ):
    return k.upper()
  data = csv.input( sys.stdin )
  result = [ change_keys( row, key_toupper )
    for row in data ]
  print( json.output( result, sort_keys=True,
    indent=2 ) )
```

Listing 1

And the corresponding use of the csv object would become:

```
data = textfilters.csv.input( sys.stdin )
```

This demonstrates why namespaces are so important. Python already has a built-in module named **`csv`** (which our package's **`csv`** module uses), and it's not unimaginable that you would want to import *both* of those. Explicitly fully naming the **`textfilters.csv`** module allows Python's **`csv`** module to also be used alongside it.

Python provides a shortcut to import all the names from a module. Consider the following:

```
from textfilters.csv import *
data = input( sys.stdin )
```

The import statement here requests that all the names from the **`textfilters.csv`** module are imported into the current namespace. On the face of it, this seems great – we get to use the **`input`** function unadorned! However, there are pitfalls to this approach. Programming is more than a typing exercise, and names matter.

Whilst that **`import *`** directive did indeed bring the name of the function we wanted into the current scope, it also brought in *every other name* exported by the **`csv`** module (we will return to what 'exported' means later). This may, or may not be what you intended. To see why it's important, create a file called `namespace.py` with the code below (a cut-down version of the `textfilters.csv` contents).

```
import csv
def input( data ):
  return list( csv.DictReader( data ) )
```

Now run a Python interpreter session in the same directory, and try the following:

```
>>> csv = '1,2,3'
>>> csv
'1,2,3'
>>> from namespace import *
>>> csv
<module `csv` from `...`>
```

Here, we're creating a variable called **`csv`**, and assigning it a value. Importing **`*`** from the **`namespace`** module then *over-writes* that value. I'm

while you can be disciplined and always avoid the use of import *, you can't very well impose that on everyone who might use your package

Explicit is better than implicit

When you import code from a module, take advantage of the namespace mechanism to ensure your own names don't get hijacked by imported ones, and to minimize the risk of hiding other names, such as built-in modules. Prefer explicitly qualified names.

sure you can guess why, but to make this completely clear, when the **`namespace`** module invokes **`import csv`**, it's bringing the name **`csv`** into *its* scope as an exported name along with the name **`input`**. When *you* import the **`namespace`** module, any exported names are brought into *your* scope, over-writing your own variable names where they clash.

Of course, while you can be disciplined and always avoid the use of **`import *`**, you can't very well impose that on *everyone* who might use your package. There are ways of helping to prevent your users from shooting themselves in their own feet.

Private names

Not *all* names are imported when you use the **`from module import *`** form. Python has a convention for making names private to a module (or indeed, a class – the mechanism is the same) by prefixing it with an underscore. Consider the code in Listing 2.

The **`import`** statement allows you to alter the names of things you import, and by renaming **`csv`** as **`_csv`**, we make that name *private* to the module. If a user of this module now invokes **`from textfilters.csv import *`**, those names are not brought into scope. Note how this affects the usage within the module's code. You can still *explicitly* request private names when you import from a module, because in Python, private doesn't mean *really private*, it just means you have to try a little harder to get access to it.

Define a public API

You can also limit the set of names brought into local scope when using **`from module import *`** by defining a module-level list of strings called **`__all__`**. If this value exists when **`from module import *`** is encountered, it is taken to mean 'this is the list of all public names in the module'. It's just a list of the names from the module you wish to be public. In the instance of the code in Listing 2, this would be defined as:

```
__all__ = [ 'input', 'output' ]
```

Adding this line to `textfilters/csv.py` will change the behaviour of **`import *`** for everyone so that only the names you defined will get imported.

```
import csv as _stdcsv
from io import StringIO as _StringIO

def input( data ):
    parser = _stdcsv.DictReader( data )
    return list( parser )
...
```

Listing 2

What have we learned?

- Using **`import *`** imports all the public names from a module.
- You can rename imported things in the import statement.
- Prefixing names with an underscore makes them 'private', so **`import *`** *does not* import them.
- As the author of a module, you can also limit the names that **`*`** imports by defining a value for the special **`__all__`** list.
- As the user of a module, avoid using **`import *`**, as it can bring in unexpected names that may hide names in your code.

Package initialization

In the previous instalment [Love20], we explored how packages are a special kind of Python module which can have sub-modules – some of which may also be packages. Python identifies a package by the existence of a file named `__init__.py`. What we didn't mention was that this file gets 'run' by the Python interpreter when the package is imported, in much the same way that the top-level code of a simple module is run when imported.

This file can contain any Python code you like, but it's useful for bringing sub-module names into a narrower scope. Consider again the directory layout of our package:

```
|__ textfilters/
  |__ __init__.py
  |__ csv.py
  |__ json.py
  |__ transformers/
    |__ __init__.py
    |__ change.py
    |__ choose.py
```

Functions inside the `change.py` sub-module of the sub-package `transformers` need a full-qualification when they're imported:

```
from textfilters.transformers.change import
  change_keys
```

This is a bit unwieldy, but arises from the *physical* separation of the **`change`** module from the **`choose`** module. That physical separation helps us as the package author to structure the code for ease of maintenance, but imposes some unnecessary complexity on the users of our package. Listing 3 (overleaf) shows how I'd prefer to present the API to users.

I've already mentioned there is more to programming than typing, but there is more to this than reducing key-presses. Your public API needn't be constrained by the physical structure of the code, and how you choose to lay out your package needn't be limited by how you wish your users to use it. We can take advantage of the fact that Python, by default, exports all public names from a module – including the modules *it* imports.

In order to achieve my desired result, a couple of changes are required. The first is to the `transformers/__init__.py` file:

```
from .change import change_keys
```

A common mistake is to presume that importing a package causes Python to go and find all of its sub-modules and import the published names from them all

```
from textfilters import csv, json
from textfilters import reshape
import sys
if __name__ == '__main__':
  def key_toupper( k ):
    return k.upper()
  data = csv.input( sys.stdin )
  result = [ reshape.change_keys( row,
    key_toupper ) for row in data ]
  print( json.output( result, sort_keys=True,
    indent=2 ) )
```

Listing 3

This brings the name **change_keys** into the scope of the **transformers** namespace, and removes the need for users to explicitly name the intermediate **change** module name.

The second alteration is to the top-level package `__init__.py`.

```
from . import transformers as reshape
```

This *renames* the namespace of **transformers** to be **reshape**. Naturally, you could just rename the **transformers** folder, but one reason you might not want to do that could be if you already have a version 'in the wild', but you'd like new users to have a new API, while still supporting existing users on the 'old' API.

We can streamline the API even further. A common pattern when using complex modules is to import the whole package and have access to its contents, as in Listing 4.

As things stand, however, this will not work. You'll get an error:

```
AttributeError: module 'textfilters' has no
attribute 'csv'.
```

A common mistake is to presume that importing a package causes Python to go and find all of its sub-modules and import the published names from them all. Such behaviour could be quite expensive! This is why the `__init__.py` file is so important – it is how a package defines all of its published names. In order to achieve what we want in Listing 4, we just need to bring the names **csv** and **json** into the package scope, using the top-level package's `__init__.py`:

```
import textfilters as tf
import sys
if __name__ == '__main__':
  def key_toupper( k ):
    return k.upper()
  data = tf.csv.input( sys.stdin )
  result = [ tf.reshape.change_keys( row,
    key_toupper ) for row in data ]
  print( tf.json.output( result, sort_keys=True,
    indent=2 ) )
```

Listing 4

```
from . import transformers as reshape
from . import csv, json
```

A similar mistake is to presume that **from textfilters import *** would cause Python to automatically load all the sub-modules. For the same reason as above, it does not. Not even the top-level modules (**csv** and **json**). The documented behaviour is that this imports the **textfilters** package, but in our case, **textfilters** is 'just' a directory. It does, however, run the `textfilters/__init__.py`. and import any published names that result from that.

As with simple modules, packages also recognise the special **__all__** value as a list of strings naming the sub-modules to import. It's crucial to note, however, that using **__all__** isn't transitive. Suppose you have the following:

- In `textfilters/__init__.py`:
  __all__ = ['transformers']
- In `textfilters/transformers/__init__.py`:
  __all__ = ['change', 'choose']

If you invoke **from textfilters import ***, it will import the **transformers** sub-package, but the sub-packages defined by the **__all__** value in `transformers/__init__.py` will not be loaded. You would also need to invoke **from textfilters.transformers import *** to also bring those names.

You can't use the top-level **__all__** value to import sub-packages, either. For example, the following will not work:

- textfilters/__init__.py
  __all__ = ['transformers',
  'transformers.change']

The consequence of this is that defining the public API for a package is best done by importing or defining the names you want in `__init__.py`. It's not necessary to also specify **__all__**, since importing * from a package won't bring any unexpected names into scope, as it might with a simple module.

What have we learned?

- A package's `__init__.py` file gets run when it's imported, and this file can contain Python code.
- You can use the `__init__.py` to alter the public API of your package.
- Importing * from a package does not automatically bring in any of the public names, only what is defined in the `__init__.py`.

Creating an installable package

Sharing a package directly by copying the package directory, or even better, including it in a shared version control system, is sufficient in most cases. There can be benefits to having a cleaner separation between application and library code, however. One example might be that a package is used across multiple applications. In such a case, it is wasteful and error-prone to have the package sources duplicated in different

```
from setuptools import setup, find_packages

setup(
  name = 'TextFilters',
  version = '0.0.0.dev1',
  packages = find_packages(),
)
```

Listing 5

repositories. It makes more sense to have the shared code separately version-controlled in its own shared repository.

Most modern version control systems have the facility to build a working copy from multiple repositories, so this shouldn't present a problem. However, you can avoid the need for that by creating your own installable package. If you've used Python for anything more sophisticated than simple scripts, you'll almost certainly have come across **pip**: the standard Python package installer[1]. In this section we'll explore how to create a package that can be installed using **pip**.

The very simplest installable package just needs a file named `setup.py`, located in the parent directory of the package itself (i.e. in the same directory as `main.py` in the example). Listing 5 shows the bare minimum contents.

The name and version properties are used to create the file name of the package. The version number here follows the recommended practice that is based on Semantic Versioning (see [PEP440] and [SemVer]). The pre-release specifier (**.dev1** in this case) departs from the Semantic Version spec, and is the format understood by **pip**, which – when installing from a shared package repository like **PyPI** – *ignores* pre-releases unless they're explicitly requested.

The last line uses a tool which automatically detects and includes any sub-packages (directories containing `__init__.py`). The packages property is merely a list of package and module names to be included, so you could explicitly name them:

```
packages = [ 'textfilters',
  'textfilters.transformers' ]
```

This invocation would exclude the **tests** sub-package, which might be what you intend. Note that sub-packages have to be explicitly named. If you have a large package with several sub-packages, the **find_packages()** utility is much more convenient. Note also that the file `main.py` will *not* be included. In our case, that's intentional, because it's not inside a package.

There are many more parameters accepted by the **setup()** function; we'll examine a few of the common ones here, but a complete description, along with recommendations on version numbering schemes, and restrictions on things like the **name** property, can be found in the *Python Packaging Guide* [PPG]. Many of those properties are used by the Python Package Index, PyPI.

For now, we have the bare essentials needed to create an installable package. To build it, run this command within the directory containing `setup.py`:

```
python setup.py bdist_wheel
```

This invocation creates a 'binary distribution', also known in Python circles as a wheel (see [PEP427] for all the gory details). If all went well[2], you will see a couple of new directories: `build` and `dist`, and the `dist` folder should have your installable package in it, named `TextFilters-0.0.0.dev1-py3-none-any.whl`. You can create 'source distributions', too, if the package is pure Python code, but it doesn't have any real benefit over a wheel format package.

The components of the file name are partly taken from the name and version parameters given to the **setup()** function in `setup.py` (refer back to Listing 5). The last 3 parts identify the targeted Python language version (`py3`), the ABI (`none`, in this case) and the required platform (which we didn't specify, and so is `any`). You can control these with other parameters to the **setup()** function, but for our purposes, the code in the package is indeed intended for Python 3, and is pure Python code, with no ABI or platform requirements, so the defaults are appropriate.

The file itself is just a normal Zip file with a `.whl` extension, so you can examine the contents for yourself (I find 7-zip especially useful).

Before we install our shiny new package, however, we should talk about segregation.

Partitioning and separation

Python comes with a rich standard library of tools, some of which our example package is using – **csv** and **json**. You can also install 3rd party modules, and our package is using **pytest**. In [Love20], we looked at how Python locates modules when they're imported. As a reminder, here is the basic Python algorithm for finding modules:

1. The directory containing the script being invoked, *or* an empty string to indicate the current working directory in the case where Python is invoked with no script – i.e. interactively.
2. The contents of the environment variable **PYTHONPATH**. You can alter this to change how modules are located when they're imported.
3. System defined search paths for built-in modules.
4. The root of the **site** module.

It's number 4 we're interested in now – the **site** module.

When you install a 3rd party package (such as pytest), it is installed into a directory named `site-packages`, which is a well-known location for the Python interpreter (the location may differ, depending on your platform). Whilst it is obviously convenient to have all the packages you want in one place, easily available for use in your Python programs, it can easily become cluttered. In particular, you might not want (or be able) to install the packages you create to the global site location, especially when they're in early development.

One way to handle this might be to have multiple installations of Python, but this is wasteful unless you genuinely need multiple versions of Python available. A more light-weight way of handling it is to take advantage of Python's virtual environments. These are a fully-featured Python environment, but cut back to the bare minimum needed. They don't contain the 3rd party modules installed in the global Python install location (but you can choose to give a virtual environment *access* to those libraries) except for a few necessities – including the **pip** installer module. The important thing is that a virtual environment is entirely independent of all other virtual environments, with its own `site-packages` location.

The implication of this is that you can create Python virtual environments with *different* libraries for different needs. This is useful now as a way of quarantining our custom package so that it doesn't interfere with either the installed Python instance, or anyone else's virtual environments. You should consider creating your environment somewhere outside of your code folders, maybe by putting the code beneath a new directory (named something like `src`, for example), and using the parent to hold the new environment.

```
python -m venv localpy
```

On some platforms you may be prompted to install a package for **venv** to work, for example on my Ubuntu-based Mint distribution, I had to install **python3-venv**.

This creates a new Python environment in a directory named `localpy` as a child of the current directory. You can choose wherever you like for it. If all's gone to plan, you should now have a directory structure like this:

```
<project root>/
  |__src/
    |__main.py
    |__setup.py
    |__textfilters/
      |__ ...
  |__localpy/
    |__ ...
```

1. **pip** comes as part of the Python install for versions later than 3.4
2. You may need to install the **wheel** package from PyPI.

The structure of the environment will differ, depending on your platform, but will contain Python itself (on Windows, in `localpy/Scripts`, on *nix it's in `localpy/bin`), along with **`pip`** to install more libraries, and a script named **`activate`**.

The **`activate`** script ensures that the virtual environment's Python and **`pip`** are at the front of the current session's path. It's not necessary to always activate a virtual environment, however: you can invoke the Python interpreter by fully-qualifying the directory name, and it will 'just work'. This extends to using **`pip`** to install packages.

- Windows

```
.\localpy\Scripts\pip.exe install [package
name]
```

- Mint (Ubuntu)

```
./localpy/bin/pip install [package name]
```

Python internally keeps track of where to find the platform-independent and platform-dependent files it needs in order to run, and where to find installed libraries. These are:

```
sys.prefix
sys.exec_prefix
```

When a virtual environment is in use (either by activation, or by running the Python program), these values will point to the respective locations *within* the virtual environment. When no virtual environment is in use, these values point to the locations of the respective Python *installation* locations. Furthermore, when a virtual environment is in use, two more values can be used to find the location of the Python install from which the virtual environment was created:

```
sys.base_prefix
sys.base_exec_prefix
```

These values enable the virtual environment to operate independently of the main Python installation(s), as well as any other virtual environments. You can find much more detailed information on how these things work in [venv] and [site], but for our purposes, all that remains is to install our local package into the independent environment. It's as simple as (on Windows):

```
.\localpy\Scripts\pip install
 src\dist\TextFilters-0.0.0.dev1-py3-none-any.whl
```

If you now run a Python session using the virtual environment's Python, you can import the **`textfilters`** package, and see from where it was imported:

```
>>> import textfilters
>>> textfilters
<module 'textfilters' from
'\\path\\to\\localpy\\lib\\site-packages\\
textfilters\\__init__.py'>
```

(This will look slightly different on non-Windows platforms, but the idea is the same).

What have we learned?

- You can create your own installable package to make sharing code even easier.
- Python wheels are zip-files.
- The site module is where Python looks for installed packages for use in code.
- Python virtual environments are a powerful way of segregating requirements with its own, independent site module.

It depends

Sometimes, a package you create will require other packages to be installed. In the case of our package, it can be used without anything other than Python's standard libraries, but it does have some tests. Whilst they don't depend exclusively on **`pytest`**, which is the testing package we used in [Love20] (other frameworks are available, such as Nose2 [Nose2], which would also work just fine), we can use it to explore another feature of package creation.

```
from setuptools import setup, find_packages

setup(
  name = 'TextFilters',
  version = '0.0.0.dev1',
  packages = find_packages(),
  install_requires = [ 'pytest' ],
)
```

Listing 6

In the `setup.py` file we created for our package, we can indicate that our package requires other libraries. In this case, we can tell the setup tools that the package **`pytest`** should also be installed when our package is installed.

Listing 6 shows a change to `setup.py` with the addition of a parameter to the **`setup()`** function named **`install_requires`**. This is a list of packages, which in this case has only one item, but you can specify as many as you need here.

Now re-create the package, and re-install it with an upgrade:

```
localpy\scripts\python src\setup.py bdist_wheel
localpy\scripts\pip install --upgrade
  dist\TextFilters-0.0.0.dev1-py3-none-any.whl
```

You will see that **`pytest`**, along with *its* requirements, is also automatically installed.

Sometimes you need a particular version of a dependent package, or perhaps you've tested on a particular stable release, and wish to constrain the versions of your dependencies. This is also specified in `setup.py`[3]:

```
install_requires = [ 'pytest>=5.0' ],
```

You can also depend on specific versions of *Python* itself in the `setup.py` parameters. In the case of our package, we may well want to ensure our users are on Python v3 or above. There are many reasons to do this, but chief among them is that the code in a package depends on some feature that was introduced in a specific Python release.

```
python_requires = '>=3',
```

There is much more you can specify, and describe, about your package in the `setup.py` file, but you can find a wealth of documentation on that in the Python packaging guide ([dist]). We do need to revisit one aspect we've already looked at briefly – the version number.

As we've already seen, the version number specified in `setup.py` gets used to generate the file name of the resulting package wheel. In our example, we marked the version with a trailing **`.dev1`**, which marks the package as a pre-release – specifically, still in development – which is used by **`pip`** when performing upgrades.

Given a package with a version number indicating it's stable (e.g. **`0.0.1`**), and a *later* version that's marked as a pre-release (e.g. **`0.1.0a1`**), when performing an upgrade, **`pip`** will by default give you the latest applicable stable release, which in this case is **`0.0.1`**. You can explicitly request that pre-releases are considered by passing the **`--pre`** argument to **`pip`** on the command line, or by specifically requesting a pre-release version.

Whilst we're in development mode, and installing specific locally-created wheels, this isn't an issue for our package, of course, but it *does* make a difference for the dependent packages in the **`install_requires`** list.

It also makes a difference in a file that's normally named `requirements.txt` (but needn't be, necessarily), which is a file you

Know your dependencies

Knowing the full set of dependent packages, right down to individual versions, makes sharing an *application* code base easier. Allowing different versions of libraries within a team can lead to very difficult-to-track errors.

3. Setting an upper limit on the version is possible too, but be careful of that. If you tie down your requirements too tightly, it might make your package unusable.

can use alongside a virtual environment to have **pip** install a whole collection of packages. This is a useful technique for specifying the library contents of a virtual environment, with needed packages at specific versions. It's common to want this to ensure, for example, that different developers on a team have *identical* environments; if one person is developing against version 1 of some package, and someone else is using version 2, chaos is bound to ensue! The `requirements` file provides a way of creating a coherent environment that the whole team can use.

The simplest way to create the requirements file is to have **pip** itself create one:

```
localpy\scripts\pip freeze > requirements.txt
```

The requirements file should contain something similar to this (truncated here for brevity):

```
...
pytest==5.4.1
six==1.14.0
TextFilters==0.0.0.dev1
...
```

Here, the file requires a *specific* version of each installed package. You can modify the version numbers if you need versions after a particular one, or within a range of versions, for example. Note that our own package, **TextFilters**, is explicitly naming the pre-release version. Suppose we had been working on the package for a while, and had a few releases available in our `dist` directory:

```
TextFilters-0.0.0.dev1-py3-none-any.whl
TextFilters-0.0.1-py3-none-any.whl
TextFilters-0.0.2.dev1-py3-none-any.whl
TextFilters-0.0.2a1-py3-none-any.whl
TextFilters-0.0.3a1-py3-none-any.whl
```

We have stable **0.0.1** and **0.0.2** versions, but only a pre-release for **0.0.3**. Our `requirements.txt` file might have this line:

```
TextFilters>=0.0.1
```

We might create our virtual environment from scratch as follows:

```
python -m venv localpy
localpy\scripts\pip install -r requirements.txt
-f src\dist
```

Here, the **-r** parameter to **pip** instructs it to read the list of packages to install from the indicated file. By default, pip looks on PyPI [PyPI] for packages, but we haven't published our package there yet, so the **-f** parameter tells **pip** to *find* packages in the specified location (which might, for example, be a file share available to the team), and look in PyPI for packages not found there.

This would result in our new environment having version **0.0.2** of our TextFilters package, because it's the latest stable version available. If we had also added the parameter **--pre** to the pip command line, the latest *pre-release* version – **0.0.3a1** – would have been installed.

What have we learned?

- An installable package can explicitly define other packages upon which it depends.
- The **pip** installer makes sophisticated use of the version numbers exposed by a package to determine how to install requirements.
- You can easily create a canned fully-working virtual environment by using a library requirements file.

A wider audience

In this article we've explored in more detail the idea of Python 'namespaces', and how you can take advantage of package initialization to make using your package easier for your users. We've looked at some of the pitfalls of wild-card imports, and highlighted the benefits of creating a public API for your modules that might not match its physical structure. We also explored virtual environments, and how to create and install your own package 'wheels', and looked at why this segregation is important. Finally we looked at package dependencies, and how to manage them in concert with virtual environments and the **pip** installer.

Taken all together, these things will help you structure your packages so they can be shared easily, and your users will find your packages easier to install and use as a result.

There is more you can do with your own packages. For example, in the previous article we looked at the **pytest** unit-testing framework, and in this article we've looked at Python's **venv**. Both of these are installable modules that can be *run*, e.g.:

```
python -m venv
```

This is achieved by adding another special file to the package: `__main__.py`, which is executed when the package is run in this way[4].

The ultimate sharing of packages with the wider community means publishing it to the Python Package Index ([PyPI]). There is excellent documentation on this in the Python packaging guide ([PPG]). Taking this extra step involves some extra responsibility, of course, in maintaining and documenting your package.

These things – and more! – I leave for you to discover. ■

References

[Love20] Steve Love (2020) 'The path of least resistance' in *Overload* 155, February 2020, https://accu.org/index.php/journals/2749

[Nose2] Nose2: https://docs.nose2.io/en/latest/

[PEP440] 'Python Version Identification and Dependency Specification', https://www.python.org/dev/peps/pep-0440/

[PEP427] 'The Wheel Binary Package Format' (PEP 427), https://www.python.org/dev/peps/pep-0427/

[PPG] The Python packaging guide, 'Packaging and distributing projects' at https://packaging.python.org/guides/distributing-packages-using-setuptools/

[PyPI] The Python Package Index, https://pypi.org/

[SemVer] 'Semantic Versioning Scheme Specification', https://semver.org/

[site] Python Documentation – Site specific configuration hook, https://docs.python.org/3/library/site.html

[venv] Python Documentation – Creation of virtual environements, https://docs.python.org/3/library/venv.html

Other resources

'Packaging a Python library', https://blog.ionelmc.ro/2014/05/25/python-packaging/#the-structure

4. I wanted to explore this a bit more in the example package, but was defeated by the fact I'd (deliberately) used names that clashed with built-in Python modules. Another example of why not to do that!

Quick Modular Calculations (Part 3)

This article concludes the 3-part series. Cassio Neri presents a new algorithm that also works for 64-bit operands.

The first two instalments of this series [Neri19] and [Neri20] showed three algorithms, *minverse*, *mshift* and *mcomp*, for evaluating expressions of the form `n % d ⋚ r`, where `d` is known by the compiler and ⋚ denotes any of `==`, `!=`, `<`, `<=`, `>` or `>=`. While *minverse* is restricted to expressions where ⋚ is either `==` or `!=`, *mshift* and *mcomp* are not. However, the last two must perform intermediate calculations in domains larger than their input. Specifically, for 32-bit data, computations are done in 64-bit registers. What if the input is already large? Will we still need it when it is 64? Yes, but these algorithms will not send you a Valentine, birthday greetings or bottle of wine. This article presents another algorithm that overcomes this limitation. I shall refer to it as *new_algo* since, to the best of my knowledge, it is original.[1]

All algorithms, including *new_algo*, are implemented in [qmodular]. Recall once again that the intention is not to 'beat' the compiler but, on the contrary, to help it. The hope is that compiler writers will consider incorporating these algorithms into their products for the benefit of all programmers. As I reported in [Neri19], *minverse* has been implemented by GCC since version 9.1 but the implementation falls back to a less efficient algorithm for certain values of `r`. Clang 9.0 also uses *minverse* (only when `r == 0` but its trunk version extends the usage for all other remainders). (See [Godbolt].) Other major compilers do not implement *minverse* and none implements any of the other algorithms presented in this series.

Recall and warm up

Figure 1[2] graphs the time taken by different algorithms to check whether each element of an array of 65,536 uniformly distributed unsigned 64-bit dividends in the interval $[0, 10^6]$ leaves a remainder less than 5 when divided by 7. As usual, *built_in* corresponds to `n % 7 < 5` as emitted by the compiler. (As a motivational note, when `n` is the number of days since a certain Monday, `n % 7 < 5` is a check for weekdays.) Bars labelled *mshift* and *mcomp* correspond to algorithms covered in [Neri20]. Finally, *new_algo* is the subject of this article.

Observe that some previously seen algorithms are absent from Figure 1. As explained, *minverse* cannot be used with `<` and efficient implementations of *mshift_promoted* and *mcomp_promoted* for 64-bit inputs require full hardware support for 128-bit calculations, which is not provided by x86_64 CPUs.

Recall that *mshift* and *mcomp* have preconditions and yield wrong results when `n` is above a certain threshold. Therefore, although very fast, the lack of generality forces the compiler to discard them.

The time taken to scan the array of dividends is used as unit of time. All measurements encompass this time. They are[3] 3.70 for *built_in*, 1.78 for *mshift*, 1.66 for *mcomp* and 2.46 for *new_algo*. Subtracting the scanning time and taking results relatively to *built_in*'s yields $0.78 / 2.70 \approx 0.29$ for mshift, $0.66 / 2.70 \approx 0.24$ for *mcomp* and $1.46 / 2.70 \approx 0.54$ for *new_algo*.

Cassio Neri has a PhD in Applied Mathematics from Université de Paris Dauphine. He worked as a lecturer in Mathematics before moving to the financial industry. He can be contacted at cassio.neri@gmail.com.

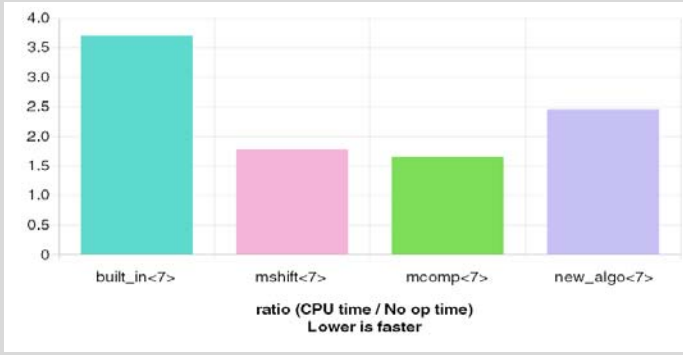


Figure 1

These numbers, however, depend on the divisor. Listing 1 contrasts the code generated by GCC 8.2.1 with `-O3` for *built_in* and *new_algo*.

Finally, recall that we are interested in modular expressions where **the divisor is a compile time constant and the dividend is a runtime variable. The value compared to the remainder can be either. They all have the same unsigned integer type which implements modulus 2^w arithmetic**. (Typically, $w = 32$ or $w = 64$.) We focus on GCC 8.2.1 for the x86_64 target but some ideas might also apply to other platforms.

The new_algo

The fractional part of n / d corresponds to the remainder of the division. Indeed, Euclidean division states that any integer n can be uniquely written as $n = q \cdot d + r$, where q and r are integers with $0 \leq r < d$. Dividing this equality by d gives that q and r / d are, respectively, the integer and fractional parts of n / d. Hence, knowing the fractional part of n / d, or an approximation of it, is enough to identify r. Since d is known by the compiler, an approximation M of $1 / d$ is precomputed at compile time and only the cheaper multiplication $n \cdot M$ is performed at runtime. The multiplication has the effect of increasing the error and when n is large enough the result is unreliable to allow the identification of r.

The last paragraph's arguments supported the works of *mshift* and *mcomp* [Neri20] and they equally support *new_algo*. The novelty is that this algorithm, rather than accepting the approximation error until the result becomes unreliable, takes steps to reduce the error. As a consequence, the algorithm's applicability is extended. As usual in this series, we shall present *new_algo*'s main ideas by means of examples. (Deeper

1. I would be grateful if a well-informed reader could point me towards a previous work on the same algorithm.
2. Powered by quick-bench.com. For readers who are C++ programmers and do not know this site, I strongly recommend checking it out. In addition, I politely ask all readers to consider contributing to the site to keep it running. (Disclaimer: apart from being a regular user and donor, I have no other affiliation with this site.)
3. YMMV, reported numbers were obtained by a single run in quick-bench.com using GCC 8.2 with -O3 and -std=c++17 [QuickBench]. I do not know details about the platform it runs on, especially, the processor.

The hope is that compiler writers will consider incorporating these algorithms into their products for the benefit of all programmers

```
built_in
   0: movabs $0x2492492492492493,%rdx
   a: mov    %rdi,%rax
   d: mul    %rdx
  10: mov    %rdi,%rax
  13: sub    %rdx,%rax
  16: shr    %rax
  19: add    %rax,%rdx
  1c: shr    $0x2,%rdx
  20: lea    0x0(,%rdx,8),%rax
  28: sub    %rdx,%rax
  2b: sub    %rax,%rdi
  2e: cmp    $0x4,%rdi
  32: setbe  %al
  35: retq

new_algo
   0: movabs $0x2492492492492492,%rcx
   a: mov    %rdi,%rax
   d: mul    %rcx
  10: add    %rcx,%rax
  13: lea    (%rax,%rdx,2),%rdx
  17: movabs $0xb6db6db6db6db6da,%rax
  21: cmp    %rax,%rdx
  24: setbe  %al
  27: retq
```

Listing 1

mathematical proofs of correctness can be seen in [qmodular] and references therein.)

Although a rigorous proof is out of scope, the fundamental idea behind *new_algo*'s error reduction has elementary school level[4]: the periodicity of decimal expansions of rational numbers. For example, $1 / 3 = 0.333\ldots$ and the sequence of 3s goes on indefinitely. Also, $1 / 7 = 0.142857\ldots$ and 142857 repeats over and over. Some readers might object and point to terminating expansions like $1 / 2 = 0.5$ or, even more obvious, $1 / 1 = 1$. Nevertheless, a terminating expansion can be identified with a periodic one by appending an infinity of trailing 0s. For instance, $0.5 = 0.5000\ldots$ and $1 = 1.000\ldots$. Furthermore, a terminating expansion is also identified with yet another periodic representation ending in 9s. Indeed, recall (or try to convince yourself) that $0.5 = 0.4999\ldots$ and $1 = 0.999\ldots$. More generally, periodicity occurs for any base and, in particular, in binary expansions. For instance, $1 / 7 = (0.001001...)_2$ with repeating 001 built_in.

Reality kicks in again to remind us that CPUs have finite precision. In practice `n`, `d` and `r` are 32 or 64 bits long but, for ease of exposition, we assume the number of bits is $w = 10$. Hence, truncation at the 10^{th} bit after the binary point yields $1 / 7 \approx (0.0010010010)_2$. Keeping the example of the previous section in mind, we set $d = 7$ and the approximation $M = (0.0010010010)_2$ of $1 / 7$.

4. I cannot help myself from highlighting the beauty of this simplicity.

n	$(n / 7)_2$	$(n \cdot M)_2$
0	0. 000 000 000 000 ...	0. 000 000 000 0 ...
1	0. 001 001 001 001 ...	0. 001 001 001 0 ...
2	0. 010 010 010 010 ...	0. 010 010 010 0 ...
3	0. 011 011 011 011 ...	0. 011 011 011 0 ...
4	0. 100 100 100 100 ...	0. 100 100 100 0 ...
5	0. 101 101 101 101 ...	0. 101 101 101 0 ...
6	0. 110 110 110 110 ...	0. 110 110 110 0 ...
7	0. 111 111 111 111 ...	0. 111 111 111 0 ...
8	1. 001 001 **001** 001 ...	1. 001 001 **000** 0 ...

Table 1

Table 1 contrasts, for all $n \in \{0, \ldots, 8\}$, the binary expansions of $n / 7$ and $n \cdot M$. Bits are grouped in triples to highlight the period. Observe that for $n \leq 7$, multiplication by $1 / 7$ and by M can be done separately on each triple, since the result of one group does not spill to its left. (Take notice that the 2^{nd} column shows $n / 7 = (0.111111\ldots)_2$ which is the binary analogon of $1 = 0.999\ldots$. This exemplifies the relevance for *new_algo* of the periodic representation of terminating expansions.)

The row for $n = 8$ is the first where the fractional parts of $n / 7$ and $n \cdot M$, up to the 9^{th} bit, differ. (The relevant triple of bits is emphasised.) To understand the origin of this difference, we observe that this row can be obtained from the one for $n = 1$ by multiplication by 8 or, equivalently, by left shift by 3. The 2^{nd} column illustrates infinite precision and the periodicity ensures that any triple of bits after the binary point has a replica on its right which is also left-shifted. This contrasts to the 3^{rd} column, where the bits feeding the left shift at the rightmost position are 0s. Having realised that there is an error coming from the right, we shall see how *new_algo* reduces it.

The previous paragraph pointed out a discrepancy between the fractional parts of $n / 7$ and $n \cdot M$ for $n = 8$. Observe now the disparity between the integral parts. It turns out the two divergences compensate each other and by uniting the two parts we can correct the error of division.

Figure 2 illustrates the steps of the process (grey 0-bits are included for clarity) as applied to $n = 8$: right shift the integer part of $n \cdot M$ by 9 bits and add the result to $n \cdot M$. Comparing the outcome with $8 / 7$ (shown in Table 1) we realise it is much closer than the original value $8 \cdot M$ is.

The procedure is very effective in making the fractional parts of the result for $n = 8$ identical to that for $n = 1$, that is, $(0.001\ 001\ 001\ 0)_2$.

The fact that multiplication by $n = 8$ is equivalent to left shift by 3 bits makes clear that the quantity on the left of the binary point is the exact amount required to correct the error on the right. For other values of n, this property might be more difficult to see but it still holds. For instance, consider $n = 15$, which is the next dividend with remainder 1. Then $n \cdot M$

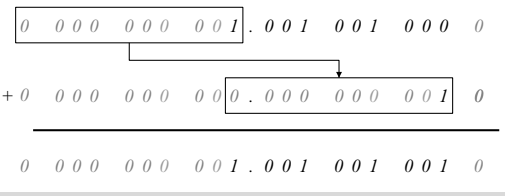


Figure 2

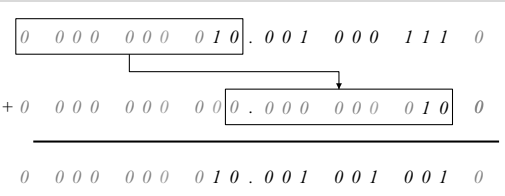


Figure 3

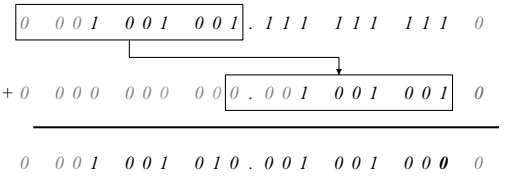


Figure 4

= $(10.\ 001\ 000\ 111\ 0)_2$ and the correction process in shown in Figure 3. Again, the result's fractional part matches the one obtained for $n = 8$. Therefore, the outcomes of the correction for $n = 1$, $n = 8$ and $n = 15$ have all the same fractional part.

Unfortunately, the process is not so good for all n. Indeed, for $n = 519$ we have $n \cdot M = (1\ 001\ 001.\ 111\ 111\ 111\ 0)_2$ and Figure 4 shows that the fractional part of the outcome is off by deficiency when compared to the one obtained for $n = 1$. The disparity appears at the 9th bit (emphasised) and thus, the error is still small. By proximity, the result suggests that the remainder is 1, which is correct.

It is worth noticing that $n = 519$ is *not* the smallest value for which the correction attempt does not zero the error out. Indeed, the row for $n = 7$ of Table 1 shows that the integer part of $n \cdot M$ is 0 and thus, the correction attempt does not provoke any change to the fractional part of $n \cdot M = (0.\ 111\ 111\ 111\ 0)_2$. Moreover, the outcome is quite far from the one for $n = 0$. This sounds like a showstopper, given that proximity is the key to recognise remainders and it has failed to hold here. Fortunately, this is more an annoyance than a real issue.

Important points to retain follow. As n takes increasing values with the same remainder $r > 0$, the fractional part of the outcome $f(n)$ starts, for $n = r$, at $f(r) = r \cdot M$ and, at each stage, it either stays the same or decreases by a tiny amount. As long as $f(n)$ does not fall enough to reach $f(r - 1)$, we are sure the remainder is r. Furthermore, when r is large enough, $f(n)$ does not change at all, that is, $f(n) = r \cdot M$ for all n with remainder r in the range of interest.

Therefore, for n in a certain range, the remainder of n divided by d is r if, and only if, $(r - 1) \cdot M < f(n) \leq r \cdot M$ or, equivalently, $r \cdot M < f(n) + M \leq (r + 1) \cdot M$. The analysis of the case $r = 0$ is a bit trickier but the same result holds. It also follows that the remainder of n divided by d is less than r if, and only if, $0 < f(n) + M \leq r \cdot M$.

To finish this section, a very important limitation of *new_algo* must be mentioned: it is not available for all divisors. Indeed, it is easy to see that, for the correction to work, at least one full period must fit in 10 bits but, as it turns out, the period of 1 / 13 in binary has length 12. Therefore, *new_algo* cannot be used for $d = 13$ in our idealised CPU. In a real 64-bit machine the smallest divisor with this issue is $d = 67$. (The period of 1 / 67 has length 66.)

Towards an implementation

The presentation so far has evolved around the idea of splitting numbers into their integer and fractional parts. We shall see now how to turn this idea into a working implementation based on unsigned integers values only. Again, for ease of exposition, we assume that these numbers and CPU registers are 10-bits long.

The algorithm's first step is calculating $n \cdot M$ where n is an integer and M has 10 bits after the binary point. To bring the product to the realm of integers, the multiplicand M is substituted by $M \cdot 210$. To keep the notation simple, the latter quantity is still denoted M. Hence, in our example we set $M = (0010010010)_2$.

Another practical issue remains. Now n and M are 10 bits long and thus, the product $n \cdot M$ has up to 20 bits. How can a 10-bit CPU calculate such number? In the real world, the question is how can a x86_64 CPU compute the 128-bit product of two 64-bit operands? The `mul` instruction (see Listing 1) does exactly that, by splitting the 128-bit product into its 64-bit higher and lower parts and storing them in registers `rdx` and `rax`, respectively. Coming back to our exposition, we assume that our imaginary 10-bit CPU provides a similar `mul` instruction.

Notwithstanding the change in the definition of M, figures 2, 3 and 4, still illustrate the correction with little differences. Previously, the small dot symbolised the binary point but now it separates the higher and lower parts. To correctly align the bits of the higher part to those of the lower one, the former should be left shifted by $k = 1$. Finally, we were originally interested in the fractional part of the outcome but now it is the lower part that we care about. In particular, the addition does not need to be carried over to the higher part, it can be performed in modulus 2^{10} arithmetic.

Putting all pieces together, a C++ implementation of *new_algo* to evaluate `n % d < r` looks like this:

```
bool has_remainder_less(uint_t n, uint_t r) {
  auto [high, low] = mul(M, n);
  uint_t f = low + (high << k);
  return f + M <= r * M;
}
```

where `mul(M, n)` returns a pair of `uint_t` with the higher and lower parts of `M * n`. The last line is the condition $0 < f(n) + M \leq r \cdot M$ in simplified form since it can be shown that $0 < f(n)$ always holds.

For readers accustomed to x86_64 assembly, it should not be difficult to recognise the C++ code above in Listing 1. (With compile time constants `M = 0x2492492492492492`, `k = 1` and `r * M = 5 * M = 0xb6db6db6db6db6da`).

A naïve implementation of *new_algo* for `n % d == r` follows:

```
bool has_remainder(uint_t n, uint_t r) {
  auto [high, low] = mul(M, n);
  uint_t f = low + (high << k);
  uint_t fpM = f + M;
  return r * M < fpM && fpM <= (r + 1) * M;
}
```

The last line comes from $r \cdot M < f(n) + M \leq (r + 1) \cdot M$. This code contains many inefficiencies (e.g., the branch implied by `&&`) and is shown for exposition only. A faster implementation is provided in [qmodular]. Depending on a number of factors, many optimisations are possible. For instance, for small values of k, the addition and left shift in the second line can be combined in a single `lea` instruction. (See Listing 1.) Also, as we have seen, for larger values of r the only condition to be tested is $f(n) = r \cdot M$. The important point here is that *new_algo*'s final form depends on several aspects that have a visible impact on the performance, as we shall see in the next section.

Performance analysis

As in the warm up, all measurements shown in this section concern the evaluation of modular expressions for 65,536 uniformly distributed unsigned 64-bit dividends in the interval $[0, 10^6]$. Charts show divisors on the x-axis and time measurements, in nanoseconds, on the y-axis. Timings are adjusted to account for the time of array scanning.

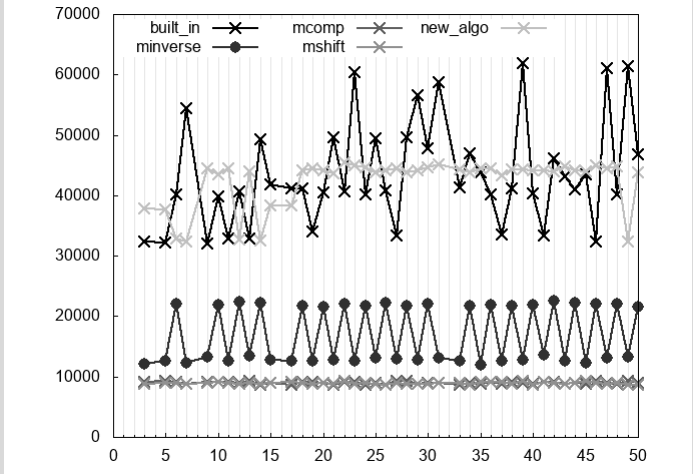


Figure 5

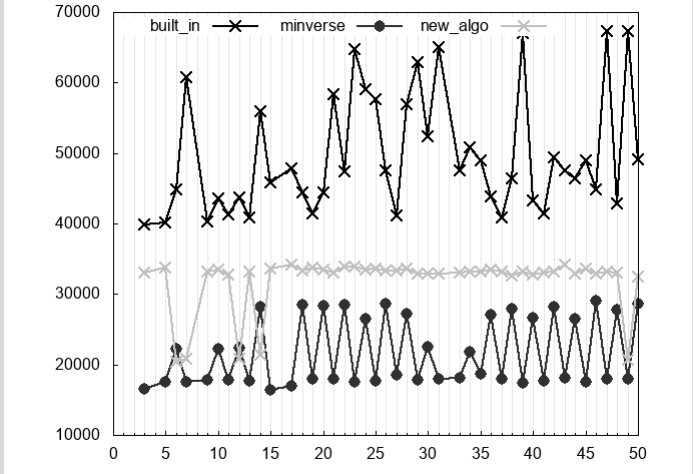


Figure 6

For clarity, we restrict divisors to [1, 50] which suffices to spot trends. (Results for divisors up to 1,000 are available in [qmodular].) In addition, we filter out divisors that are powers of two since the bitwise trick (see [Warren13]) is undoubtedly the best algorithm for them. The timings were obtained with the help of Google Benchmark [Google] running on an AMD Ryzen 7 1800X Eight-Core Processor @ 3600Mhz; caches: L1 Data 32K (x8), L1 Instruction 64K (x8), L2 Unified 512K (x8), L3 Unified 8192K (x2).

Figure 5 concerns the evaluation of `n % d == 0`. Readers might already be familiar with *minverse*'s zigzag and its great performance. Although *mcomp* and *mshift* are even faster and have a pretty regular performance across divisors (a good feature on its own), recall they are not available for all values of *n*. They are shown here for the sake of completeness but in practice a compiler cannot use them. Looking at *new_algo*, we observe that its performance changes considerably across divisors depending on the availability of different micro-optimisations. Actually, *new_algo* is not very performant here and given the limitations of *mcomp* and *mshift*, we conclude that *minverse* is the best option.

Figure 6 shows the evaluation of `n % d == 1`. Due to *mshift*'s and *mcomp*'s limitation, they have now been excluded from this picture. The situation changed considerably with respect to the previous case. Indeed, *new_algo* beats the *built_in* algorithm for all divisors shown and for a handful of them (e.g., $d = 14$) it even beats *minverse*.

Finally, Figure 7 considers the expression `n % d > 1`. Recall that *minverse* cannot evaluate this expression. It is fair to say that *new_algo* beats the *built_in* algorithm for most of the divisors shown in the picture.

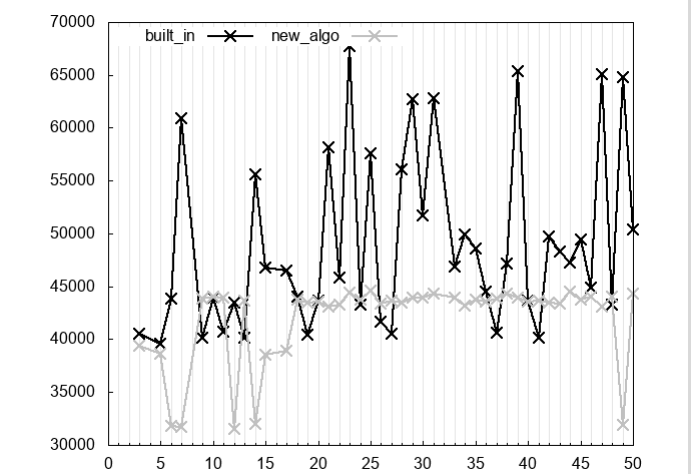


Figure 7

Conclusion

We presented a new algorithm, designated here as *new_algo*, for the evaluation of certain modular expressions. It overcomes limitations of other algorithms previously seen in this series [Neri19] and [Neri20]. Specifically, *minverse* cannot be used for expressions like `n % d < r` and *mshift* and *mcomp* cannot be efficiently implemented in 64-bit CPUs. Alas, the *new_algo* has its own limitation: it is not available for all divisors.

Like *mshift* and *mcomp*, *new_algo* operates on an approximation of n / d. which contains an error that increases with the numerator. Contrarily to the others, *new_algo* performs steps to delay the error growth by using the periodicity of binary expansions of rational numbers. In essence, errors on the right side of the truncated expansion can be corrected using bits appearing on the left.

Performance analysis shows that, in some cases, *new_algo* can be faster than others. However, it is worth mentioning that no algorithm seen in this series beats all others in all circumstances. Therefore, a compiler aiming to emit the most efficient code for modular expressions needs to implement all these algorithms and carefully pick the one that is best for the particular case in hand. Amongst other aspects, this decision must consider the value of the divisor, the type of the expression (e.g., `n % d == r` as opposed to `n % d > r`), the size of operands (32 versus 64 bits). A particularly interesting point about *new_algo* is that to emit efficient code just for this one algorithm, the compiler (writer) has already to deal with many choices of micro-optimisations.

This article brings this series to an end but more research is needed. To compiler writers: "I don't know why you say goodbye, I say hello." ■

Acknowledgements

I am deeply thankful to Fabio Fernandes for the incredible advice he provided during the research phase of this project. I am equally grateful to Lorenz Schneider and the *Overload* team for helping improve the manuscript.

References

[Godbolt] https://godbolt.org/z/xsMLeP

[Google] https://github.com/google/benchmark

[Neri19] Cassio Neri, 'Quick Modular Calculations (Part 1)', *Overload* 154, pages 11–15, December 2019.

[Neri20] Cassio Neri, 'Quick Modular Calculations (Part 2)', *Overload* 155, pages 14–17, January 2020.

[QuickBench] http://quick-bench.com/oF3Bm1mHz3_pbSuLHV4NdqY1edw

[qmodular] https://github.com/cassioneri/qmodular

[Warren13] Henry S. Warren, Jr., *Hacker's Delight*, Second Edition, Addison Wesley, 2013.

Deconstructing Inheritance

Inheritance can be overused. Lucian Radu Teodorescu considers how it can go wrong and the alternatives.

After glancing at the title, the reader might accuse me of trying to destroy inheritance; probably by arguing that it should be replaced by some other mechanism. But that is not the case; that is not my intent. According to Merriam-Webster [MW], *deconstruction* is defined as:

> : a philosophical or critical method which asserts that meanings, metaphysical constructs, and hierarchical oppositions (as between key terms in a philosophical or literary work) are always rendered unstable by their dependence on ultimately arbitrary signifiers
>
> : the analytic examination of something (such as a theory) often in order to reveal its inadequacy

My intent here is to reveal inheritance's actual meanings versus the meanings that most Object-Oriented programmers will infuse it with; to show hidden oppositions in its structure, to show that some signifiers are somehow arbitrary, and finally to reveal inner inadequacies. The main point is to test the limits of inheritance, and how far we can go until our beliefs about inheritance break.

One of the main topics of the article will be the relation between inheritance and the *is-a* relationship, and how this connects to the principle of correspondence (the common design belief that modelling OOP software should maintain a correspondence to the real-world that the software somehow models). Another important topic that is frequently referred to in this article is the Liskov Substitution Principle (LSP) [Liskov94] [Liskov88].

These two topics are a crucial point in analysing inheritance. They both define what inheritance is, but also subversively work against it, creating this amorphous concept that encompasses both good and bad.

Some simple problems are hard

Let's look at a very simple OOP modelling problem: we want to model the *Rectangle* and the *Square* concepts in software. For our problem, we are only interested in dimensions. As the two concepts are closely related in the real-world, we want to relate them with an inheritance in our software. There are 2 main options:

- make *Rectangle* inherit from *Square*
- make *Square* inherit from *Rectangle*

Let us analyse both options.

Rectangle is-a Square

First thing: this is mathematically incorrect. In the real-world, the *is-a* relationship is revered. But, let's ignore this for a moment. Let's look at the code in Listing 1, which is modelling *Rectangle is-a Square*.

Lucian Radu Teodorescu has a PhD in programming languages and is a Software Architect at Garmin. As hobbies, he is working on his own programming language and he is improving his Chuck Norris debugging skills: staring at the code until all the bugs flee in horror. You can contact him at lucteo@lucteo.ro

```
class Square {
  int size;
public:
  virtual int getSize() const { return size; }
  virtual void setSize(int x) { size = x; }
  virtual int getArea() const {
    return size*size; }
};
class Rectangle: public Square {
  int width;
public:
  virtual int getWidth() const { return width; }
  virtual int getHeight() const {
    return Square::getSize(); }
  virtual void setSize(int x) {
    Square::setSize(x); width = x; }
  virtual void setWidth(int x) { width = x; }
  virtual void setHeight(int x) {
    Square::setSize(x); }
  virtual int getArea() const {
    return width*getSize(); }
};
```

Listing 1

Not only is the mathematical relation broken, but the interface of the `Rectangle` class is polluted by concerns that it doesn't have (size is confusing for *Rectangle*). Moreover, we can easily find an example (see Listing 2) in which this breaks the LSP test (if you change the type, does the code still functions well?).

Passing a `Rectangle` object to the `increaseArea` function will make the code break. This variant is definitely not right. Let's try the other one.

Square is-a Rectangle

Let's look at the code in Listing 3 (overleaf).

Mathematically, this seems to be correct. And the interface of `Square` is not necessarily polluted with the unneeded stuff (the inherited methods can be hidden). Let's now try to see if it passes the LSP test (see Listing 4).

Similarly to the previous test, if we assume that `r` is a veritable rectangle, doubling the width will double the area. But, if `r` is a square, then the area will increase by 4 times.

```
void increaseArea(Square& square) {
  auto oldArea = square.getArea();
  square.setSize(square.getSize() * 2);
  auto newArea = square.getArea();
  assert(newArea == 4 * oldArea);
}
```

Listing 2

Even though we have a good insight into what the real-word concepts mean when we place them into code, the metaphor breaks

```
class Rectangle {
  int width, height;
public:
  virtual int getWidth() const { return width; }
  virtual int getHeight() const { return height; }
  virtual void setWidth(int x) { width = x; }
  virtual void setHeight(int x) { height = x; }
  virtual int getArea() const {
    return width*height; }
};
class Square: public Rectangle {
public:
  virtual int getSize() const {
    return Rectangle::getWidth(); }
  virtual void setSize(int x) {
    Rectangle::setWidth(x);
    Rectangle::setHeight(x); }
};
```

Listing 3

Another problem is with the existence of the **setWidth** and **setHeight** functions of the base class. No matter how we override them in the derived class, the existence of these setters will make possible clients of **Rectangle** break. If we ignore to override them, it's easy to see that a call to any of these will break the invariants of **Square**. If we throw exceptions, we may break **Rectangle** clients that used to work ok. If we change both the width and the height when one setter is called, then we can find examples similar to Listing 4. There is no reasonable override to these methods that cannot be proven to be wrong with the help of LSP.

More discussion

The previous examples showed us that, if we want to force the mathematical relationship between area and the sizes of the square/rectangle, no matter how we do the inheritance, we cannot do it right.

One good observation that will allow us to fix things is to remove the setters; make objects of those two classes immutable. Something similar to the code in Listing 5.

This code doesn't break LSP as, once created, the objects cannot be made to break their invariants. However, then the main question that arises is what are actually gaining from the inheritance anyway? We occupy more memory for the **Square** objects, and we make a few functions virtual, that most probably are not used. The only thing that is reused is the **area()**

```
void increaseAreaNew(Rectangle& r) {
    auto oldArea = r.getArea();
    r.setWidth(r.getWidth() * 2);
    auto newArea = r.getArea();
    assert(newArea == 2 * oldArea);
}
```

Listing 4

```
class Rectangle {
protected:
  const int width, height;
public:
  Rectangle(int w, int h) : width(w), height(h) {}
  virtual int getWidth() const { return width; }
  virtual int getHeight() const { return height; }
  virtual int getArea() const {
    return width*height; }
};
class Square: public Rectangle {
public:
  Square(int s) : Rectangle(s, s) {}
  virtual int getSize() const {
    return Rectangle::getWidth(); }
};
```

Listing 5

method, which, mostly by coincidence, did not need rewritten. Adding inheritance here does not help us.

Let's look at what others are saying about this problem:

> The truth is that Squares and Rectangles, even immutable Squares and Rectangles, ought not be associated by inheritance – Robert C. Martin
>
> The class Square is not a square, it is a program that represents a square. The class Rectangle is not a rectangle, it is a program that represents a rectangle. […] The fact that a square is a rectangle does not mean that their representatives share the ISA relationship. – Robert C. Martin
>
> ISA is useful when trying to model real world relations to make class hierarchies intuitive, but classes are metaphors, and metaphors, if extended too far will break – Bjørn Konestabo

One cannot use inheritance to model a very simple mathematical problem. Even though we have a good insight into what the real-word concepts mean when we place them into code, the metaphor breaks. Inheritance doesn't work the way the *is-a* relationship works in mathematics.

Concepts, is-a and inheritance

The previous sections showed us that inheritance cannot always properly model the *is-a* relationships from the real-world. Let's look at some more cases in which the analogy with the real-world breaks.

Let's think of modelling an elevator system. Besides the elevator car, motor or doors, we have a lot of buttons. We have buttons on each floor (up/down), we have buttons inside the elevator, both for floors and for cancelling or alerting. We can model the system as shown in Figure 1 (overleaf). But we can also model it with Figure 2.

Or, we can simply model everything with just one **Button** class (Figure 3).

To be honest, I would probably go for the last option, but that doesn't matter too much for this discussion.

each time we look at inheritance, instead of thinking about is-a relationships, we should think of behaves-like-a relationships

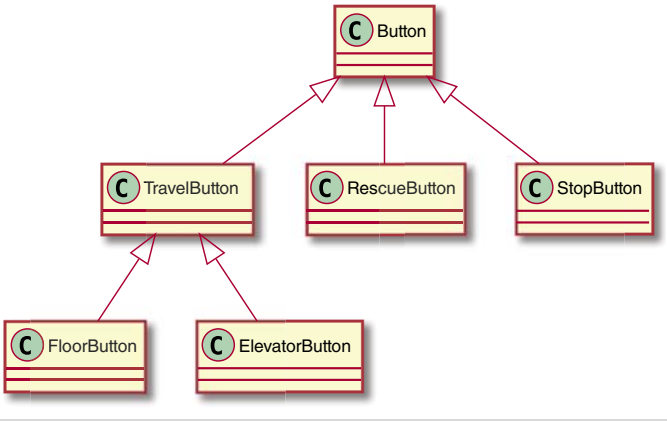


Figure 1

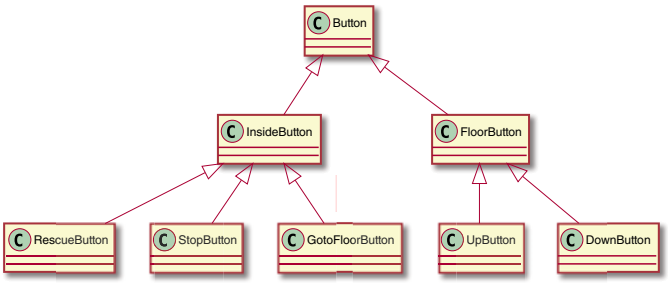


Figure2

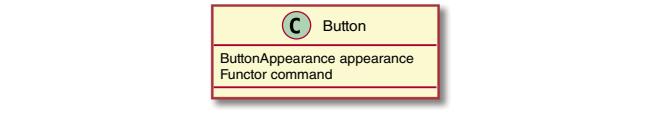


Figure 3

If we carefully look at the various methods (and others can easily be found), we realise that the *Button* concept is probably the only real-world concept. Things like *TravelButton*, *InsideButton*, and *FloorButton* are concepts invented in software modelling, and then somehow look real. Nobody thinks of an *inside-button* concept in the real world. Yes, we sometimes distinguish between buttons that are inside the lift car and the ones that are fixed to the floors, but that's a property of the objects themselves; *inside-button* and *outside-button* are not strong enough to be concepts by themselves.

Without dwelling too much on the semantics of the *concept* concept, we observe a discrepancy between concepts inspired between real-life (concepts as building blocks of thoughts) and concepts that are generated through our design process (concepts as sets, that can always be divided into smaller sets). If we want to stick to the classes that should be inspired by real-world concepts, we should probably abandon the second type of concepts.

And now we've reached the fun part: what does 'is-a' mean? What does it take for a concept to *be* another concept? Too bad for us that metaphysics has not been able to figure out the answer to this issue in the last 2000+ years. While we wait for the philosophers to figure this out, we can safely assume at least that we cannot say *A is-a B* if *A* and *B* belong to different species. And, in our case, we just argued that *InsideButton* and *Button* belong to different species: one in an artificially constructed concept and one is a real-world inspired concept. That means that is improper to say that *InsideButton is-a Button* (at least, not while considering *Button* as a real-world concept).

A far more useful relation would be the *behaves-like-a* relationship. We can safely say that *InsideButton behaves-like-a Button*, even if the two concepts come from different worlds (e.g., a dolphin can behave like a fish even if it's not a fish). Moreover, from a software perspective, we are only interested in the *behaves-like-a* relationship, and we can leave the *is-a* to metaphysics. When I say *D behaves-like-a B*, what I mean is that *D* inherits all the properties of *B*, that **I** can use *D* in all the places that I would use *B*. But this is exactly what the Liskov Substitution Principle says.

So, in other words, each time we look at inheritance, instead of thinking about *is-a* relationships, we should think of *behaves-like-a* relationships, and then immediately think of LSP.

If my digression into semiotics and metaphysics left the users too confused, maybe a quote would do better [Sutter04]:

> The "is-a" description of public inheritance is misunderstood when people use it to draw irrelevant real-world analogies: A square "is-a" rectangle (mathematically) but a Square is not a Rectangle (behaviorally). Consequently, instead of "is-a," we prefer to say "works-like-a" (or, if you prefer, "usable-as-a") to make the description less prone to misunderstanding.

To prove my point, this quote was taken from the chapter named 'Public inheritance is substitutability. Inherit, not to reuse, but to be reused'. Public inheritance is substitutability. q.e.d.

On the fine details of LSP

Formally, LSP states the following [Liskov94]:

> Subtype requirement: Let ?(x) be a property provable about objects x of type T. Then ?(y) should be true for objects y of type S where S is a subtype of T.

The informal principle goes the following way [Liskov88] (see also [ObjectMentor03]):

> If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

It basically defines what a subtyping relation should be, and by extension, it describes what successful inheritance should be.

Inheritance was supposed to be an abstraction feature, one that reduces the complexity of the software

Now, if we assume a strict interpretation of the formal principle, we can argue that polymorphism cannot happen under subtyping, rendering inheritance useless.

Let us quickly sketch a proof. Let's say that we have classes *D* and *B*, *D* being a subtype of *B*. Then we have a method *m*, implemented as *m1* in *B* and as *m2* in *D*, using a different implementation. In this case we can define the provable property $\phi(x)$ as being $\phi(x)$ = *x.m() has-exactly-the-same-behaviour-as m1*. It is clear that this property holds for any object of type *B*, but it does not hold for objects of type *D*. The only way to make LSP work for such properties is to make the methods have identical behaviour, so, therefore, to eliminate polymorphism. q.e.d.

The reader might accuse us of exaggerating the matter by artificially constructing counterexamples; something that cannot happen in practice. But if we consider our statement in the context of the Open/Closed Principle, then we can easily imagine how this is not exaggerating. After all, with the types under the incidence of inheritance can be a closed system, and we should be able to extend this system with functionality that contains our counterexample. And this is not just a theoretical problem. Myself, I've encountered violations of LSP built on the same pattern as our proof above multiple times (of course, unintentional).

At this point, some readers might still argue that we should probably not be applying the LSP principle so strictly. We should only consider the properties relevant to the program in question. That is, if we want to make *D* derive from *B*, we should consider all the 'practical' properties associated with *D* and *B*. This idea is similar to the one that Sutter and Alexandrescu argue in their 'Public inheritance is substitutability. Inherit, not to reuse, but to be reused' recommendation; they argue that creating inheritance should be the focus on the external code that may be able to use this inheritance relationship.

But there is a great danger if we go in this direction. Inheritance was supposed to be an abstraction feature, one that reduces the complexity of the software – instead of looking at a large number of classes, we should look at fewer classes. But instead, LSP forces us to consider all the visible properties of the code, for all the users of the classes. It's an anti-abstraction feature.

Let's take an example. In the previous section, the method to compute the area of a rectangle or a square is a relatively good abstraction. It decouples the implementation details (in this case very simple) from the code that utilises it. We can easily document it and explain it to other engineers. But, suddenly, if we add inheritance to any of these classes, we need to also consider how this method can be used by the callers, coupling it to a possibly large number of implementations. If we are not careful and capture the usage patterns, we might end up with examples that break LSP, and thus render the program invalid.

So, theoretically, LSP cannot be formally applied, and if we are using our practical sense to apply it, we may be creating a larger problem for us to solve. To paraphrase a programmer's joke: we have one problem to solve; let's try to throw in inheritance to solve it – now we have two problems to solve, and one of them is hard to solve.

LSP and invariants

Let's now analyse how LSP applies from a different point of view: that of maintaining invariants. The base class has some invariants. The main question would be for a derived class on how it can change the behaviour of the objects while maintaining the same invariants – after all, invariants are visible properties, and LSP dictates that they should not change.

One can think of invariants as predicates that can be applied to objects. Whenever the predicate returns false for an object, the invariant doesn't hold. Moreover, it can be chained as a conjunctive form giving a series of conditions $C_1, C_2, \ldots, C_n$. The predicate holds if all the conditions are true. Can we add or remove conditions in the derived classes? Let's look at both cases. Let's assume that the invariant of the base class is $I_B = C_1 \wedge C_2 \wedge C_3$.

First, let's consider that case in which the derived class removes a constraint, let's say C_3. The invariant for the derived class will be $I_D = C_1 \wedge C_2$. Now, all the *D* objects that properly satisfy I_D, but do not satisfy C_3, will not satisfy I_B. LSP will not apply to those objects. Thus, removing constraints in derived classes will break LSP.

In the other case, when we add constraints, things appear to work well. Mathematically we can easily prove that the invariants of the base class are met for the derived objects: $I_D \Rightarrow I_B$.

But this works well only when all the invariants are known upfront. And, most of the time, as software is in its essence just complexity [*Brooks95*], not all invariants are known upfront – there are a lot of implicit assumptions. Let's say that $I_B = C_1 \wedge C_2 \wedge C_3$ and $I_D = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, and that C_4 never applies to any object of *B*. In such cases, a user can accidentally assume that C_4 never happens. This becomes an accidental assumption in *B*'s invariant: $I_B = C_1 \wedge C_2 \wedge C_3 \wedge \neg C_4$. If this happens, then again LSP is broken.

So, to make LSP work, we have to survey all user code to check for hidden assumptions, before we can derive from *B*; both when trying to keep the same or when adding new constraints to the invariant of the derived class.

Inheritance and composition

We can start with Robert C. Martin's quote to set the basis for the discussion in this section:

> A pox on the ISA relationship. It's been misleading and damaging for decades. Inheritance is not ISA. Inheritance is the redeclaration of functions and variables in a sub-scope. No more. No less.

Well, the content misses one big point: inheritance also adds subtyping (allowing us to implicitly convert derived-class objects into base-class objects); which in turn can be used to implement polymorphism. But the rest is true.

If inheritance is just a redeclaration of functions and variables, then we can easily transform it into composition. Instead of making *D* derive from *B*, we can make *D* contain a *B*.

Therefore, if subtyping is not needed, it's better to just use composition instead of inheritance as we don't have to deal with the complications introduced by subtyping.

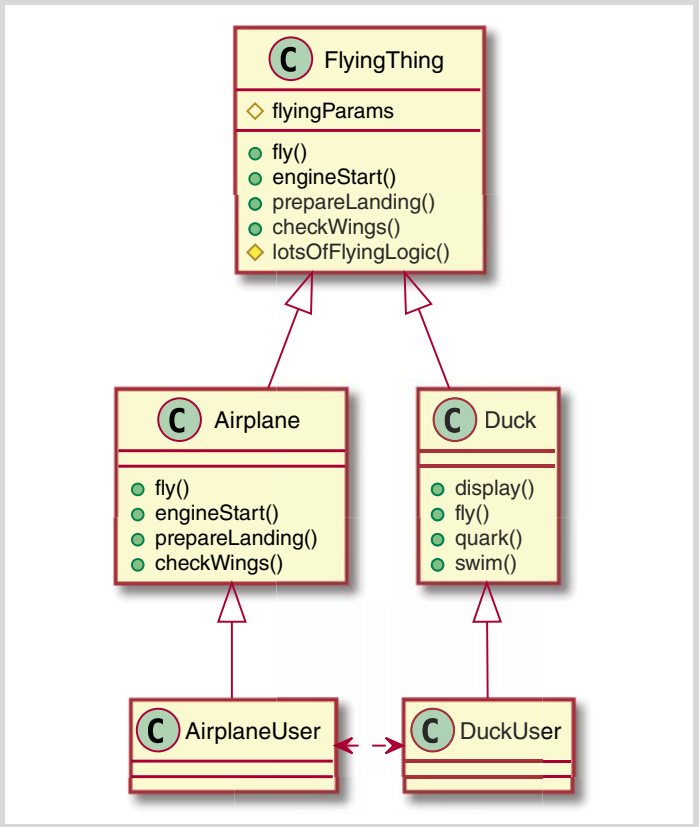


Figure 4

The same conclusion reaches Sutter and Alexandrescu in their *C++ coding standards: 101 rules, guidelines, and best practices* book [Sutter]; see the section called 'Prefer composition to inheritance'. I won't repeat the arguments, as the idea is relatively straightforward.

Inheritance is stronger than friendship

Sutter and Alexandrescu argue that inheritance is almost as strong as friendship (same section of [Sutter]). My opinion is that we should move forward and argue that inheritance is even stronger than friendship.

Making an analogy with real-life, one's children are closer than one's friends. The children can be friends, but in general, are more than that. Yes, children, especially young ones, may not have access to all the information their parent shares with friends but may dramatically alter the life of the parent.

The analogy works with classes. Yes, derived classes may not access all the fields of the base class, but they can seriously affect the space of the invariants. Future development in the derived classes may involve changing the invariants in the base class, and therefore affecting all the other derived classes, and all their users.

Friendship can affect private data, the internals of the class. But, if encapsulation is done right, this will not change the public interface of the class, and therefore the behaviour of the clients. Like any abstraction, class-level encapsulation restrains the impact of a change. In contrast, an inheritance that changes the invariants (directly or indirectly) is a public change, and it affects all the clients of the class.

To exemplify the impact of inheritance, let's look at Figure 4. Let's assume that the `AirplaneUser` needs a change to the flying behaviour. This affects the `Airplane` class, which changes the invariants of `FlyingThing`, which, in turn, affects `Duck` and finally `DuckUser`. In effect, the `AirplaneUser` and `DuckUser` classes are indirectly coupled.

This is stronger than friendship. Friendship may change your internal state, but if it's not done particularly badly, it tends not to break your invariants, and your users are isolated from the change.

Looking only at the difference between protected and private access is missing the larger impact of inheritance. However, if we look at the bigger picture, if we consider LSP and the example above, we can conclude that inheritance implies more coupling than friendship.

Conclusions

In this article, we have tried to cast a critical perspective on inheritance, as one of the most used (or abused) features of OOP. The intent of this critical perspective was not to prove that inheritance should not be used at all, but rather to test its limits. What becomes apparent is that this is not a feature that should be abused, and great care needs to be taken when adding new inheritance to a software system, not to break existing code. In other words, we don't have local guarantees when adding inheritance.

We started with a classic problem of *Rectangle* and *Square,* and have shown that inheritance doesn't quite make sense in code, even though a square is a rectangle in mathematics. We explored the meaning of the *is-a* relationship and its relation to the real-world; after all, a common strategy in OOP modelling is to use real-world analogies. We concluded that this analogy works only to a point. Inheritance is not *is-a*. Furthermore, the term *is-a* can be confusing (unless we solve a large part of metaphysics). A slightly more appropriate way to think of inheritance is to think of it as a *behaves-like-a* relationship; i.e., what LSP preaches.

Moving forward, we showed that LSP is hard to apply. If we want to be strictly formal, we cannot apply it. In practice, we can, however, apply it, but not as easily as we may think. We cannot have local reasoning (looking just at the base class and the derived code). We need to look at all the user code and all the implicit assumptions that this code makes. Depending on the software, there may be semantic leaks towards all parts of the code; yes, that may be a badly structured code, but there is no clear algorithm that indicates whether we have such semantic leaks. As much as we would like to put boundaries to the implications of inheritance and LSP, it seems that we can't.

We also briefly argued that whenever possible composition should be preferred to inheritance. And, to add one more negative aspect to inheritance, we've argued that inheritance is a stronger relationship than friendship, relation widely considered harmful.

And, as we are enumerating some negative aspects of inheritance, we should mention the presentation called *Inheritance Is The Base Class of Evil*, by Sean Parent [Parent13] – the name is too good not to be mentioned.

But again, the purpose of our deconstruction is not to show that inheritance should not be used. The main idea is to better know its limits, to find its weak points, and to find its internal inadequacies; when it can be applied, and where it can generate problems. There are cases in which inheritance is, at least useful, if not more. But this can be the topic of another article. ■

References

[Brooks95] Frederick P. Brooks Jr. (1995), *The Mythical Man-Month: Essays on Software Engineering*, 20th Anniversary Edition, Addison-Wesley

[Liskov88] Barbara H. Liskov (1988), *Data Abstraction and Hierarchy*

[Liskov94] Barbara H. Liskov, Jeannette M. Wing (1994), *A behavioral notion of subtyping, ACM Transactions on Programming Languages and Systems*

[MW] Merriam-Webster (2020), Definition of 'deconstruction', https://www.merriam-webster.com/dictionary/deconstruction

[ObjectMentor03] Object Mentor (2003), 'The Liskov Substitution Principle', https://web.archive.org/web/20030403055009/http://www.objectmentor.com:80/resources/articles/lsp.pdf

[Parent13] Searn Parent (2013), 'Inheritance Is The Base Class of Evil', *GoingNative 2013*, https://channel9.msdn.com/Events/GoingNative/2013/Inheritance-Is-The-Base-Class-of-Evil

[Sutter04] Herb Sutter, Andrei Alexandrescu (2004), *C++ coding standards: 101 rules, guidelines, and best practices*, Addison-Wesley Professional

Using Compile Time Maps for Sorting

Compile time sorting can be interesting. Norman Wilson shows how to sort a map.

In the previous article I described a way of doing compile time sorting. One of the questions that came out of this was why would anyone want to do that? The first answer is just for fun, it's just pretty for its own sake, and I think we as programmers ought to be able to appreciate that even if there is no practical use. To put it another way, quoting Albert Einstein, what use is a newborn baby? Secondly, the sorting algorithm illustrates some techniques that can be applied more generally to solve other problems – really it's just playing with parameter pack expansion. Last but not least, there are real reasons why you'd want to sort at compile time and in this article I'm going to show you one.

Firstly though I'm going to show another little trick, apply it to create yet another compile time sorting algorithm and then use that to solve a practical problem.

A simple compile time map

Look at the following bit of code.

```
template<typename... Members>
struct Map: Members... {};
```

If we instantiate **Map** with a pack of types, (assuming we can derive from all of the types) we end up with a type that is a composition of these. Furthermore we also know we can implicitly cast from the composed type to any of its bases. So given:

```
struct A {}; struct B {}; struct C {};
auto abc = Map<A, B, C>{};
```

Then I can select the **A** base of **abc** like this:

```
inline decltype(auto) getA(const A& a)
{ return a; }
const A& a = getA(abc);
```

And similarly for the other bases. Nothing particularly radical about that, but what if our bases look like this?

```
#include <cstddef>

template<std::size_t key, typename Value>
struct Pair { using type = Value; };
```

Now we can write a generalised **get**.

```
template<std::size_t key, typename Value>
inline constexpr decltype(auto)
  get(const Pair<key, Value>& result)
{ return result; }
```

The crucial thing is that type deduction now comes into play. So given:

```
using ABCMap = Map<Pair<0, A>, Pair<1, B>,
  Pair<2, C> >;
ABCMap abcMap;
```

I can write:

```
#include <utility>
#include <type_traits>

const auto& x = get<1>(abcMap);
```

We have constrained key and, since our keys are unique, when the compiler tries to deduce **Value** there is only one possible solution, **abcMap** must be cast to **const Pair<1, B>&**. The expression is unambiguous and **x** ends up referencing the appropriate base of **abcMap**.

```
static_assert(std::is_same_v<std::decay_t
  <decltype(x)>, Pair<1, B> >,
  "get() pulls out the corresponding base.");
```

We can translate this back to the world of types with this slightly ugly incantation:

```
template<typename M, std::size_t key>
using Get = typename std::decay_t<
  decltype(get<key>(std::declval<M>()))>::type;
static_assert(std::is_same_v<Get<ABCMap, 2>, C>,
 "Get works too.");
```

So we have a very simple way of mapping from integers to types. The **key** doesn't have to be a non-type and it doesn't have to be an **int**, but it does have to be a compile time construct. There are other ways of making compile time maps but I quite like this one as it's simple, and uses the basic rules of C++ that we all (should) understand. This **map** type is obviously very similar to **std::tuple** – but we'll come back to that later. The order we declare the pairs in the mapping is not important. We could have said:

```
using ABCMap2 = Map<Pair<2, C>, Pair<1, B>,
  Pair<0, A> >;
```

Type deduction will still produce the same result – another MacGuffin.

Another way to sort

If we can rank each element of a set, then sorting is just forming a mapping from rank to element. In other words a sorted sequence allows us to access the elements by rank. Let's express that in code.

Let's start with a simple type:

```
template<typename... Ts> struct TypeList {};
```

Primary definition. We require an **index_sequence**, some traits to generalize how we do the sort and some types to sort.

```
template<typename Index, typename Traits,
  typename... Ts> struct MapSortImpl;
```

Partially specialize to break out the **index_sequence**. The index pack gives us 0..**sizeof..(Ts) - 1**. This corresponds to the position of each **T** in **Ts**. We'll need this throughout the following code and a shorthand to use traits.

Norman Wilson has been coding since he was a spotty teenager in the early 80s and learned C++ while at university. Since then he's spent most of his career in finance. When not staring at template error messages, he rock climbs, makes music and helps bring up three daughters. You can contact him at norman.wilson+accu@gmail.com

To get a space-efficient layout, we should sort our members by size, biggest to smallest. Efficient layouts make better use of cache and that makes code go faster.

```
template<std::size_t... index, typename Traits,
typename... Ts>
struct MapSortImpl<std::index_sequence<index...>,
  Traits, Ts...>: Traits
{
```

Note we've derived from `Traits` to make this easier.

```
  template<typename T>
  static constexpr auto value()
  { return Traits::template value<T>(); }
  using Traits::compare;
```

How do we find the rank of an element? Firstly we count up how many other elements rank lower than it using the traits `compare` function.

We then consider equal ranking elements. We want a stable sort, so we add on the count of the equal elements preceding this element in the list. This keeps equal ranking elements in their existing order and ensures every element has a well defined place.

In order to find the rank of element `T` which was at position `pos` in `Ts`, we need the following function:

```
  template<typename T, std::size_t pos>
  static constexpr auto rankOf()
  {
    auto countOfTsWithLesserValue =
      (compare(value<Ts>(), value<T>()) +...);
    auto countOfEqualTsPrecedingInList =
      ((value<T>() == value<Ts>() && index < pos)
      +...);
    return countOfTsWithLesserValue +
      countOfEqualTsPrecedingInList;
  }
```

Now we can define the ranking as a mapping from rank to `T` for each `T` in `Ts`.

```
  using Ranking = Map<Pair<rankOf<Ts, index>(),
    Ts>...>;
```

Can you dig it? What I'm saying here is `Ranking` is a map whose keys are the ranks (defined by `rankOf()`) of the corresponding `Ts`. In other words, it's a map from sorted position to type.

We can produce the sorted result by `Get`ting from the ranking map in sequence.

```
  static constexpr auto sort()
  { return TypeList<Get<Ranking, index>...>{}; }
};
```

Finally some syntactic sugar.

```
template<typename Traits, typename... Ts>
inline constexpr auto mapSort(TypeList<Ts...>)
{
  return
    MapSortImpl<std::index_sequence_for<Ts...>,
    Traits, Ts...>::sort();
}
```

Does it work? See Listing 1.

```
#include <functional>

template<int i>
using Int = std::integral_constant<int, i>;

struct Traits {
  template<typename T>
  static constexpr auto value()
  { return T::value; }
  static constexpr std::less<> compare;
};
using In = TypeList<Int<42>, Int<7>, Int<13>,
  Int<7> >;
using Out = decltype(mapSort<Traits>(In{}));
static_assert(
  std::is_same_v< Out, TypeList<Int<7>, Int<7>,
  Int<13>, Int<42> > >, "mapSort works!");
```

Listing 1

What's the point?

Consider:

```
using T1 = std::tuple<bool, std::int16_t, char,
  std::int32_t, std::int64_t>;
static_assert(sizeof(T1) == 24,
  "pathological case std::tuple pads");

using T2 = std::tuple<std::int64_t, std::int32_t,
  std::int16_t, bool, char>;
static_assert(sizeof(T2) == 16,
  "efficient layout sorts by size");
```

We all know that on most architectures types have an alignment. And we ought to know that to get a space-efficient layout, we should sort our members by size, biggest to smallest. Efficient layouts make better use of cache and that makes code go faster.

In many cases where we define a `std::tuple` we can rearrange the members by hand for efficiency, but sometime we might not be able to – maybe the type is generated by TMP, or maybe we want future proof our code against changes which affect the alignments. So can we come up with something like a `std::tuple` but which automatically lays itself out efficiently? Let's try.

Firstly we'll revisit the compile time map we started with, but with a little tweak. Field now holds a value.

```
template<std::size_t key, typename T>
struct Field { T value; };
```

`get()` returns that value.

```
template<std::size_t key, typename T>
decltype(auto) get(const Field<key, T>& f)
{ return (f.value); }
```

```
template<std::size_t key, typename T,
  typename Tuple>
constexpr auto checkType(Tuple&& t)
{
  return
    std::is_same_v<std::decay_t
    <decltype(get<key>(t))>, T>;
}
auto boolShort = Tuple<Field<0, bool>, Field<1,
  std::int16_t> >{};
static_assert(checkType<0, bool>(boolShort),
  "pulls out bool");
static_assert(checkType<1,
  std::int16_t>(boolShort), "pulls out short");
```

Listing 2

We can actually reuse the `Map` template but the name no longer makes sense so I'll rename it.

```
template<typename... Ts>
using Tuple = Map<Ts...>;
```

To prove this works I define the contents of Listing 2.

So we've defined the very barest bones of a tuple template. I leave it to the reader to flesh out the missing parts. The important difference from `std::tuple` is that we explicitly associate a key with an element. The keys could be given out of order and could be non-contiguous, but the association between key and type is still maintained. So we can define a tuple where the elements can be reordered to form a space efficient layout? All we need to do now is sort the types by size. Listing 3 shows we do that.

Footnote: is it any good?

The sort I presented in the previous article – no. It's actually astonishingly bad. Never in all my time as a programmer have I burned so many clock cycles on so simple a problem. So lets add another reason to why you'd want to do a compile time sort – you can go and make a coffee while your code is compiling.

What about `mapSort`? That's actually much better but is still at least n^2. I've pulled down the code for `skew_sort` that comes from the *Stack Overflow* thread [Stackoverflow] that started all this off to compare. The graphs show this is radically faster than both, but blows the compiler's maximum template recursion depth for data sets larger that 1024. Clearly the story is not over. Can we write a non-recursive sort in `n log(n)`? Will it be any good? Find out next time. ■

Reference

[Stackoverflow] 'C++ calculate and sort vector at compile time', available from https://stackoverflow.com/questions/32660523/c-calculate-and-sort-vector-at-compile-time

```
// First we define an appropriate traits class.
struct EfficientLayoutTraits
{
  // We want to sort by size.
  template<typename T>
  static constexpr auto value()
  { return sizeof(T::value); }

  // Biggest first.
  static constexpr std::greater<> compare;
};

// A little helper that transforms a TypeList
// to a Tuple.
template<typename... Ts>
auto typeListToTuple(TypeList<Ts...>)
{ return Tuple<Ts...>{}; }

// Shorthand to expand index_sequence. This little
// function deserves an article in itself.
template<std::size_t... index, typename F>
auto applySequence(std::index_sequence<index...>,
  F&& f)
{
  return f(std::integral_constant<std::size_t,
    index>{}...);
}

// Given some Ts create a Tuple with efficient
// layout.
template<typename... Ts>
auto makeEfficientLayout()
{
  // Form a list of Fields containing Ts,
  // indexed by declaration order.
  auto unsortedFields = applySequence(
    std::index_sequence_for<Ts...>{},
    [](auto... index)
    { return TypeList<Field<index, Ts>...>{}; });
  // Sort it using our traits.
  auto sortedFields =
    mapSort<EfficientLayoutTraits>
    (unsortedFields);
  // Turn it into a Tuple.
  return typeListToTuple(sortedFields);
}
// Our previous pathological case.
auto efficientTuple =
  makeEfficientLayout<bool, std::int16_t,
  char, std::int32_t, std::int64_t>();

// Tuple elements have been reordered to use
// minimum space.
static_assert(sizeof(efficientTuple) ==
  16, "QED");
// But are accessed by original declaration order.
static_assert(checkType<0,
  bool>(efficientTuple));
static_assert(checkType<1,
  std::int16_t >(efficientTuple));
static_assert(checkType<2,
  char>(efficientTuple));
static_assert(checkType<3,
  std::int32_t >(efficientTuple));
static_assert(checkType<4,
  std::int64_t >(efficientTuple));

int main(){}
```

Listing 3

Profiting from the Folly of Others

Code referred to as a hack can raise an eyebrow. Alastair Harrison learns about accessing private members of C++ classes by investigating a header called UninitializedMemoryHacks.h

I always enjoy browsing through the source code of libraries written by other people. With so many dark corners in C++ I often come across new and interesting ideas. I'd like to share one such example from the 'Folly' library. Not because I think it illustrates best practice (it doesn't!), but because I learned something about C++ in the process of deciphering it.

Folly [GitHub] is a C++ library developed at Facebook and released under the open-source Apache 2.0 licence. It contains useful algorithms, vocabulary types and utility functions. Hidden amongst the main-stream functionality are some utilities tailored towards the more unusual situations that a C++ developer may find themselves in.

The code I'd like to focus on lives in a header with the ominous title of `UninitializedMemoryHacks.h`. Its subtle use of loopholes and language features is fascinating, despite its obviously questionable nature.

The file contains a collection of helper functions that do reprehensible things in the name of performance. In particular, it provides a set of overloaded functions in the **`folly::`** namespace, named **`resizeWithoutInitialization`** and whose purpose is to 'resize **`std::string`** or **`std::vector`** without constructing or initializing new elements'.

It does *what?*

Normally when we call **`resize`** to increase the size of a **`std::vector`**, the container first checks to see if the existing capacity is sufficient to hold the requested number of elements. Even when the existing capacity is sufficient, the implementation needs to do something with the newly added elements. They each need to be constructed or initialized to ensure that they are in a valid state. For trivial types such as **`int`** it's actually OK to leave the values uninitialized, as long as we don't try to read them before we've first written something to them. But **`std::vector`** always forces us to pay the cost of initialization, even if we were intending to overwrite all of the newly initialized elements straight after calling **`resize`**.

In contrast, when we call **`folly::resizeWithoutInitialization`** on a **`std::vector`** with sufficient capacity, it simply *reaches in to the private implementation* and moves the pointer representing the end of the sequence. The memory for the new elements is left uninitialized, leaving the caller responsible for that task.

The first time I looked at the implementation of this function, I was amazed and alarmed to see it somehow bypassing the normal C++ access restrictions to modify a private member variable of a standard library component. I say 'somehow' because the precise mechanism was so thoroughly obfuscated behind layers of macros, template trickery and arcane member function pointer syntax that it might as well have been magic. The baffling part was that it claimed to pull off this magic trick without invoking any undefined behaviour. I had to know how this worked!

I won't dwell further on the specifics of how Folly meddles with the internals of the standard containers. The interesting part is how it bypasses the access control mechanisms of C++. Herb Sutter has a *Guru of the Week* article [GotW] discussing three nefarious techniques for accessing private members of a class, though none of them quite matches the applicability of the method in the Folly library. The first two are illegal and the third involves writing a sneaky member function specialization, which makes it relevant only to classes that contain member function templates.

What's interesting about the technique used in Folly is that it's able to freely access private members of any class, without any particular structural requirements. It does this with a clever combination of infrequently-used language features and a small loophole allowed by the C++ standard.

The effect

Let's take a simple class with a private member function:

```
class Widget {
 private:
  void forbidden();
};
```

Our aim is to write a free function called **`hijack`** which takes a reference to a **`Widget`** as input and calls the **`Widget::forbidden()`** member function on it. Assume that the **`Widget`** class is closed for modification by us, so we can't just change the **`private`** to **`public`**, or make **`hijack`** a friend of **`Widget`**.

Obviously we can't call the private member function directly:

```
void hijack(Widget& w) {
  w.forbidden();  // ERROR!
}
```

because the compiler will stop us:

```
In function 'void hijack(Widget&)':
error: 'void Widget::forbidden()' is private
within this context
      |     w.forbidden();
      |                 ^
```

Using techniques from the Folly library, we'll build up a solution piece-by-piece. This article covers the specific case of calling private member functions, but the approach is equally applicable to accessing and mutating private member variables in a class. The underlying techniques all work in C++98, but some more modern features will be used to ease exposition.

A syntax refresher for pointers to member functions

We'll be using pointers to member functions (PMFs) extensively, so it's worth revisiting their syntax before we dive in further. PMFs enable a

Alastair Harrison Alastair started out as a robotics researcher but accidentally became a C++ build engineer because nobody else wanted to do it. These days he appreciates the fact that his customers are all in the same building as him and that they are apparently unfazed by his eagerness to delete old code. He can be contacted at aharrison24@gmail.com

There's a curious loophole in the C++ standard around the use of explicit template instantiation which allows us to refer to private class members

```
class Calculator {
  float current_val = 0.f;
 public:
   void clear_value() { current_val = 0.f; };
   float value() const {
     return current_val;
   };

   void add(float x) { current_val += x; };
   void multiply(float x) { current_val *= x; };
};
```

Listing 1

primitive form of polymorphism over methods in a class. For the sake of exposition, let's start with a hypothetical calculator class (Listing 1).

Arguably the easiest way to work with pointers to member functions is through type aliases. The type alias is specific to a given class, but the pointer can be bound to any member function in the class that matches the signature. In the case of **`Calculator`**, both **`multiply`** and **`add`** take a single **`float`** argument and return **`void`**, so we can use the same type alias for both. It looks like this:

```
using Operation = void (Calculator::*)(float);
```

We can then store the address of either **`multiply`** or **`add`**. But **`value`** doesn't match the signature, so its address cannot be assigned to an **`Operation`** pointer.

```
// OK
Operation op1 = &Calculator::add;
Operation op2 = &Calculator::multiply;

// ERROR! Signature mismatch
Operation op3 = &Calculator::value;
```

We'll need to make a new alias to match the signature of **`value`**:

```
using Getter = float (Calculator::*)() const;

// OK - signature now matches
Getter get = &Calculator::value;
```

A pointer to a member function isn't very useful unless we know which object instance we want to call it on. Here's the syntax for calling members of **`Calculator`** through their pointers:[1]

```
Calculator calc{};
(calc.*op1)(123.0f); // Calls add
(calc.*op2)(10.0f);  // Calls multiply

// Prints 1230.0
std::cout << (calc.*get)() << '\n';
```

1. C++17 introduced the **`std::invoke`** template, which gives a unified syntax for working with callables.

```
class Widget {
 public:
  static auto forbidden_fun() {
    return &Widget::forbidden;
  }
 private:
  void forbidden();
};

void hijack(Widget& w) {
  using ForbiddenFun = void (Widget::*)();
  ForbiddenFun const forbidden_fun =
    Widget::forbidden_fun();

  // Calls a private member function on the Widget
  // instance passed in to the function.
  (w.*forbidden_fun)();
}
```

Listing 2

One of the interesting things about pointers to member functions is that they can be bound to private member functions. That's the first piece of the Folly puzzle.

Puzzle piece 1: Pointers to private member functions can be called from any scope

Suppose the author of the **`Widget`** class had helpfully provided a means to obtain a pointer to the **`Widget::forbidden()`** member function. Once we have that pointer, we are able to call it from *any* scope where we have a **`Widget`** available (Listing 2).

That's useful to know, but most classes don't offer to hand out pointers to their private member functions. We need to find a sneakier way to get hold of one from outside of the class scope.

There's a curious loophole in the C++ standard around the use of explicit template instantiation which allows us to refer to private class members. That gives us the second piece of the Folly puzzle.

Puzzle piece 2: The explicit template instantiation loophole

The C++17 standard contains the following paragraph (with the parts of interest to us marked in bold):

> **17.7.2 (item 12)**
>
> **The usual access checking rules do not apply to names used to specify explicit instantiations.** [Note: In particular, the template arguments and names used in the function declarator (including parameter types, return types and exception specifications) **may be private types** or objects which would normally not be accessible and the template may be a member template or member function which would not normally be accessible.]

```
class OnSiteEmployeePolicy {
  // ... contains daring and unfettered use of
  // ... hairy template meta-programming tricks.
};
class Company {
 private:
  template <typename EmployeePolicy>
  void update_employee(int employee_id) {
    // ...
  }
};
// Prevents implicit instantiation of a specific
// specialization.
extern template
Company::update_employee<OnSiteEmployeePolicy>;
```

Listing 3

```
#include "company.h"

// Explicit instantiation of the template only
// needs to happen in a single translation unit.
template
Company::update_employee<OnSiteEmployeePolicy>;
```

Listing 4

To understand the reason behind this curiosity, we need to discuss the explicit template instantiation mechanism for a moment.

Suppose we've got a **`Company`** class with an internal private member function template, **`update_employee`**. Perhaps there is one particular template argument, **`OnSiteEmployeePolicy`** which is expensive to compile, but used regularly. We'd like to avoid the cost of instantiating that version of the template in lots of translation units. We can achieve this by explicitly instantiating the member template in just one translation unit and marking it as **`extern`** everywhere else. See Listing 3 (`company.h`) and Listing 4 (`company.cpp`).

Brushing aside the question of how someone ever snuck such an awkward API design through code review, notice how the template instantiation mechanism needs to allow a reference to a *private* member of **`Company`** – **`Company::update_employee`** – in a context where we would not normally be able to (i.e. outside the scope of the **`Company`** class). That's the reason for the exception in the C++ standard that allows for private types to appear in explicit template instantiations.

It's also the crucial loophole that Folly takes advantage of. We can't relax just yet, though. There's still some work to be done.

Puzzle piece 3: Passing a member-function pointer as a non-type template parameter

In C++, template arguments are usually types, but there is some support for non-type template parameters if they are of integral or pointer type.[2] Conveniently enough, it's perfectly legal to pass a pointer-to-member-function as a template argument.[3] Listing 5 is an example of what that looks like.

The intermediate **`SpaceShipFun`** alias hampers the genericity of the **`SpaceStation`** template, so we can move the type of the pointer-to-member-function into the template arguments too (Listing 6).

We can take that a step further and have the compiler deduce the type of the member function for us:

```
SpaceStation<
  decltype(&SpaceShip::dock),
  &SpaceShip::dock
> space_station{};
```

```
class SpaceShip {
 public:
  void dock();
  // ...
};

// Member function alias that matches the
// signature of SpaceShip::dock()
using SpaceShipFun = void (SpaceShip::*)();

// spaceship_fun is a pointer-to-member-function
// value which is baked-in to the type of the
// SpaceStation template at compile time.
template <SpaceShipFun spaceship_fun>
class SpaceStation {
  // ...
};

// Instantiate a SpaceStation and pass in a
// pointer to member function statically as a
// template argument.
SpaceStation<&SpaceShip::dock> space_station{};
```

Listing 5

```
template <
  typename SpaceShipFun,
  SpaceShipFun spaceship_fun
>
class SpaceStation {
  // ...
};

// Now we must also pass the type of the pointer to
// member function when we instantiate the
// SpaceStation template.
SpaceStation<
  void (SpaceShip::*)(),
  &SpaceShip::dock
> space_station{};
```

Listing 6

That relieves us of some of the burden of having to pass the member function signature to the template. We'll stick with this approach for the rest of article as it's what's used in the Folly library, but it's worth mentioning that C++17's **`template <auto>`** feature removes the need for the first template parameter entirely.[4]

Passing a private pointer-to-member-function as a template parameter

Let's combine the explicit template instantiation loophole with the ability to pass member function pointers as template parameters. The **`HijackImpl`** struct receives a pointer to **`Widget::forbidden()`** as a template parameter (see Listing 7).

Brilliant! We've instantiated a template that is able to reach in and call the **`forbidden()`** member function on any **`Widget`** that we care to pass in. So we just have to write the free function, **`hijack`** and we can go back to watching cat videos on YouTube, right?

```
void hijack(Widget& w) {
  HijackImpl<
    decltype(&Widget::forbidden),
    &Widget::forbidden
  >::apply(w);
}
```

2. C++20 will significantly relax the restrictions on non-type template parameters.
3. I imagine it's staggeringly useful to *someone*.
4. Readers lacking both a C++17 compiler and a certain amount of moral fibre have probably already worked out how to use a macro to remove the duplication in the template arguments.

The only problem is that it doesn't work. The compiler sees through our ruse and raps us smartly on the knuckles:

```
error: 'forbidden' is a private member of 'Widget'
  HijackImpl<decltype(&Widget::forbidden),
    &Widget::forbidden>::hijack(w);
```

The use of the **`HijackImpl`** template inside the **`hijack`** function is *not* an explicit template instantiation. It's just a 'normal' implicit instantiation. So the loophole doesn't apply. It's time to phone a friend for help with solving the next piece of the puzzle.

Puzzle piece 4: A friend comes to our aid

Because we're not allowed to refer to **`Widget::forbidden`** inside our **`hijack`** function, we must solve the conundrum of accessing the value of the **`ForbiddenFun`** template parameter *without* making any direct reference to the **`HijackImpl<...>`** template. This apparently unreasonable requirement is easily solved with a shrewd application of the **`friend`** keyword.

Let's take another step back from the task in hand and look at some of the different effects one can achieve when marking a free function as a **`friend`** of a class. The behaviour depends on whether the class contains only a declaration of the function (i.e. function signature only), or whether the complete definition (including the function body) appears inside the class.

'friend' function declarations

Most C++ developers will be familiar with the pattern of placing a free function declaration inside of a class definition and marking it as a **`friend`**. The definition of the free-function still lives outside of the class, but is now allowed to access private members of the class. (See Listing 8.)

If we could make **`hijack`** be a **`friend`** of **`Widget`** then the compiler would allow us to refer to **`Widget::forbidden`** inside the **`hijack`** function. Alas, this option is unavailable because the rules of our game don't allow us to modify **`Widget`**. Let's try something else.

Inline 'friend' function definitions

It's also possible to *define* a **`friend`** function inside a class (as opposed to just declaring it there). This isn't something seen as often in C++ code. Probably because when we try to call the free function, the compiler is unable to find it! (See Listing 9.) Here's the compile error:

```
error: 'frobnicate' was not declared in this scope
       |   frobnicate();
       |   ^
```

```
// The first template parameter is the type
// signature of the pointer-to-member-function.
// The second template parameter is the pointer
// itself.
template <
  typename ForbiddenFun,
  ForbiddenFun forbidden_fun
>
struct HijackImpl {
  static void apply(Widget& w) {
    // Calls a private method of Widget
    (w.*forbidden_fun)();
  }
};

// Explicit instantiation is allowed to refer to
// `Widget::forbidden` in a scope where it's not
// normally permissible.
template struct HijackImpl<
  decltype(&Widget::forbidden),
  &Widget::forbidden
>;
```

Listing 7

```
class Gadget {
  // Friend declaration gives `frobnicate` access
  // to Gadget's private members.
  friend void frobnicate();

 private:
  void internal() {
    // ...
  }
};

// Definition as a normal free function
void frobnicate() {
  Gadget g;
  // OK because `frobnicate()` is a friend of
  // `Gadget`.
  g.internal();
}
```

Listing 8

As before, **`frobnicate()`** is still a free function that lives in the global namespace, but it behaves quite differently under name lookup now that it is defined inside the **`Gadget`** class. A **`friend`** function defined inside a class is sometimes known as a 'hidden friend' [JSS19] [Saks18]. Hidden friends can *only* be found through Argument Dependent Lookup (ADL) and ADL only works if one of the arguments to the function is of the enclosing class type. In the above example **`frobnicate()`** takes no arguments, so argument dependent lookup won't happen. The result is that **`frobnicate()`** can't be called from anywhere. Not even from within **`frobnicate()`** itself!

If we add a parameter of the enclosing class type to **`frobnicate()`** then we're able to call it via ADL (Listing 10, overleaf).

Making hidden friends visible

The hidden-friend ADL trick can be very useful; it's an ideal tool when writing operator overloads for user-defined types. But we'll use a slightly bigger hammer for our **`hijack`** function. There's another way of allowing the compiler to find hidden friends, and that is to put a declaration of the function in the enclosing namespace (Listing 11).

This is exactly the opposite of the usual pattern of defining a free function and then placing a **`friend`** declaration for it inside of a class. The new behaviour is almost identical except for one critical difference: when the enclosing class is a template, the free function has access to the template parameters!

```
class Gadget {
  // Free function declared as a friend of Gadget
  friend void frobnicate() {
    Gadget g;
    g.internal(); // Still OK
  }

 private:
   void internal();
};

void do_something() {
  // NOT OK: Compiler can't find frobnicate()
  // during name lookup
  frobnicate();
}
```

Listing 9

```
class Gadget {
  friend void frobnicate(Gadget& gadget) {
    gadget.internal();
  }

 private:
   void internal();
};

void do_something(Gadget& gadget) {
  // OK: Compiler is now able to find the
  // definition of `frobnicate` inside Gadget
  // because ADL adds it to the candidate set for
  // name lookup.
  frobnicate(gadget);
}
```

Listing 10

```
class Gadget {
  // Definition stays inside the Gadget class
  friend void frobnicate() {
    Gadget g;
    g.internal();
  }

 private:
   void internal();
};

// An additional namespace-scope declaration makes
// the function available for normal name lookup.
void frobnicate();

void do_something() {
  // The compiler can now find the function
  frobnicate();
}
```

Listing 11

Using friends to pilfer template parameters

I trust you will be at least a little unsettled to discover that the program in Listing 12 is valid.

What's happening is that the `observe()` function is not defined until the point at which the `SpookyAction` template is instantiated (by its use in the `main` function). There is a single definition of the `observe()` function, because there is a single instantiation of the `SpookyAction` template. The really useful part is that the `observe()` function gains access to the template parameter of the `SpookyAction<42>` instantiation that caused it to be defined.

```
#include <iostream>

template <int N>
class SpookyAction {
  friend int observe() {
    return N;
  }
};

int observe();

int main() {
  SpookyAction<42>{};
  std::cout << observe() << '\n';  // Prints 42
}
```

Listing 12

Of course things go wrong very quickly if we try to instantiate any more versions of the `SpookyAction` template, as each one results in a redefinition of the `observe()` function and an angry compiler.

Provided we use it carefully, we now have the last piece of our puzzle – a way to access the template parameters of a class from a scope external to that class.

Putting the puzzle pieces together

Let's go back to our original `Widget` example, now that we've got all of the pieces that we need to be able to reach in and call its private member function, `Widget::forbidden()`. In summary:

1. We use the loophole in the explicit template instantiation rules to allow us to refer to `Widget::forbidden()` from outside of the `Widget` class.
2. We inject the address of `Widget::forbidden()` into our `HijackImpl` class as a template parameter.
3. We define the `hijack()` function directly inside of `HijackImpl` so that it can access the template parameter containing the address of `Widget::forbidden()`.
4. We mark `hijack` as a `friend` so that it becomes a free function and we provide a declaration of `hijack` at namespace scope so that it participates in name-lookup.
5. We can now invoke `Widget::forbidden()` on any `Widget` instance through the member-function address that is exposed to the `hijack` function.

The key parts of the mechanism are shown in Listing 13.

Dealing with multiple definitions of the friend function

There's still one more issue to overcome.[5] To avoid violating the One Definition Rule, there must be one – and only one – explicit instantiation of a template (with given template arguments) across all translation units.

```
// HijackImpl is the mechanism for injecting the
// private member function pointer into the
// hijack function.
template <
  typename ForbiddenFun,
  ForbiddenFun forbidden_fun
>
class HijackImpl {
  // Definition of free function inside the class
  // template to give it access to the
  // forbidden_fun template argument.
  // Marking hijack as a friend prevents it from
  // becoming a member function.
  friend void hijack(Widget& w) {
    (w.*forbidden_fun)();
  }
};
// Declaration in the enclosing namespace to make
// hijack available for name lookup.
void hijack(Widget& w);

// Explicit instantiation of HijackImpl template
// bypasses access controls in the Widget class.
template class
HijackImpl<
  decltype(&Widget::forbidden),
  &Widget::forbidden
>;
```

Listing 13

5. If you choose to ignore the pitchfork-bearing members of the C++ standards committee currently approaching your front door with a polite request that you stop doing this sort of thing at all.

```
#pragma once
#include <iostream>

class Widget {
 private:
  void forbidden() {
    std::cout << "Whoops...\n";
  }
};
```

Listing 14

Consider what happens when our `HijackImpl` class is put in a header and is used in multiple translation units. The explicit instantiation of the class template must live outside of that header, otherwise it will appear in every translation unit that includes the header. We need to ensure that there is just one explicit template instantiation in the whole program. What's more, the linker is not actually required to report duplicate instantiations across multiple translation units, so it won't even help us to avoid the problem. That's a recipe for a big maintenance headache.

The approach employed by the Folly library is to add an extra template parameter to the `HijackImpl` class and use it to accept an empty 'tag' struct which is defined in an anonymous namespace.

The anonymous namespace ensures that the tag parameter is of a different type in every translation unit. Every translation unit will therefore get its own unique explicit instantiation of the `HijackImpl` class.

The final solution is short, but packs in a surprising amount of nuance. See `widget.h` (Listing 14), `widget_hijack.h` (Listing 15) and `main.cc` (Listing 16.)

Conclusion

Should you use this access-violation hack in production code? Almost certainly not. Well, not unless you enjoy the excitement and explosive unpredictability of maintaining extremely brittle code.

The C++ class member access rules are there to help authors of types to enforce invariants. If you fool the compiler into mutating private class members then you're likely to be violating the class invariants, which risks leaving the program in an invalid state. You're also relying on intimate knowledge of internal implementation details, for which the library author is under no stability obligations.

Should you experiment with this access-violation technique outside of production code? Absolutely! Learning how to subvert a system in a safe environment is not only fun, but it helps to foster a deeper understanding of that system. Untangling the machiavellian mechanisms in the Folly library tested my knowledge of C++, requiring me to improve my understanding of the language features I encountered along the way. It's almost as if someone had reached in to my brain's internal implementation and fiddled with its contents... ■

A note on the origins of the technique

The idea of using explicit template instantiation to bypass class access rules pre-dates the Folly library by a few years. The first mention I can find is from a 2010 blog post by Johannes Schaub [Schaub10], which describes a method using initialization of static class members. At the time, there was a discussion on the Boost mailing list about how the technique might prove to be a useful addition to the Boost serialization library.

A year later, Schaub offered the dubiously tempting promise of 'Safer Nastiness' in a follow-up blog post [Schaub11] in which he presented an improved version of the code. This removed the need for static class members and is much closer to what's used today by the Folly library.

```
#pragma once
#include "widget.h"

namespace {
// This is a *different* type in every translation
// unit because of the anonymous namespace.
struct TranslationUnitTag {};
}

void hijack(Widget& w);

template <
  typename Tag,
  typename ForbiddenFun,
  ForbiddenFun forbidden_fun
>
class HijackImpl {
  friend void hijack(Widget& w) {
    (w.*forbidden_fun)();
  }
};

// Every translation unit gets its own unique
// explicit instantiation because of the
// guaranteed-unique tag parameter.
template class HijackImpl<
  TranslationUnitTag,
  decltype(&Widget::forbidden),
  &Widget::forbidden
>;
```

Listing 15

```
#include "widget.h"
#include "widget_hijack.h"

int main() {
  Widget w;
  hijack(w); // Prints "Whoops..."
}
```

Listing 16

Acknowledgements

The author would like to thank Geoff Hester, Anthony Kirby and Kirsty McNaught for advice on early drafts of this article and Balog Pal for very patiently explaining some finer points of the one-definition rule.

References

[GitHub]: Facebook Folly on GitHub: https://github.com/facebook/folly

[GotW]: 'Uses and Abuses of Access Rights at http://www.gotw.ca/gotw/076.htm

[JSS19]: 'The Power of Hidden Friends in C++' posted 25 June 2019: https://www.justsoftwaresolutions.co.uk/cplusplus/hidden-friends.html

[Saks18]: Dan Saks 'Making New Friends' recorded at *CPPCon 18*, available at: https://www.youtube.com/watch?v=POa_V15je8Y

[Schaub10]: Johannes Schaub 'Access to private members. That's easy!', posted 3 July 2010: http://bloglitb.blogspot.com/2010/07/access-to-private-members-thats-easy.html

[Schaub11]: Johannes Schaub 'Access to private members: Safer nastiness', posted 30 December 2011: http://bloglitb.blogspot.com/2011/12/access-to-private-members-safer.html

It's About Time

How easy is it to make code wait for a time period before doing something? Mike Crowe looks at ways to avoid problems when a system clock changes.

Most developers know how to implement a timeout so that an operation can be attempted for a certain period of time before stopping or giving up. Something like Listing 1.

Or, perhaps, in C++, as in Listing 2.

This pattern even works when efficiently blocking for something to happen using a condition variable as in Listing 3.

In these examples, I'm using `std::chrono::system_clock` because its is equivalent to `std::chrono::high_resolution_clock` in libstdc++. You may want to check your standard library documentation to determine which would be best for you.

But what happens if someone changes the system clock during the loop? Not every device has a real-time clock to keep time when the device is off. Even if a device does, it may not be particularly accurate. This could mean that the system clock warps (jumps) by a few seconds or even a few decades after a power cycle when the device does eventually find an accurate time source, perhaps via NTP [Wikipedia-1], when it gains an Internet connection. This can lead to strange hard-to-reproduce bug reports from the field and unhappy users. Note that `std::chrono::system_clock` is required to be Coordinated Universal Time (UTC), which does not change due to daylight saving. Time zones and daylight saving are a completely different subject, one that is not well addressed in standard C++ until C++20 [Hinnant18].

How do we avoid this problem? When possible, just use relative timeouts. Use `std::condition_variable::wait_for` rather than `std::condition_variable::wait_until`. An absolute timeout is like setting an alarm clock – if you change the time shown on the clock then you affect how long it will be until the alarm sounds. A relative timeout is like setting an egg timer, and leaving it alone – the time shown on your clock does not affect how long it will be until the alarm sounds.

Unfortunately a relative timeout doesn't work well for the examples above because the timeout may cover multiple waits. It's possible to recalculate a relative timeout but that's easy to get wrong and it risks the timeout being extended unintentionally as small errors accumulate over many loops.

A better solution is to use a monotonic or steady clock that is immune to the warping of the system clock. Such a clock is defined to keep running at an approximately-consistent rate without warping either forwards or backwards. If the machine has access to an accurate clock source, often via NTP, the monotonic clock can be slewed slightly in order to try to keep it running correctly relative to real time. Clock slewing means slowing down or speeding up the clock by small amounts in order to keep time accurately on average over a longer period of time.

Mike Crowe Mike became a C++ and embedded Linux developer by accident twenty-odd years ago and hasn't managed to escape yet. Working for small companies means that he gets to work on a wide range of high and low-level software, as well as release processes and build tools to stop him getting bored. He can be reached at accu@mcrowe.com.

```
bool do_something_for_a_while(void)
{
  const time_t expire = time(NULL) + 5;
  while (time(NULL) < expire) {
    if (try_to_do_something())
      return true;
  }
  return false;
}
```

Listing 1

```
void do_something_for_a_while()
{
  using namespace std::chrono;
  auto const expire =
    system_clock::now() + seconds(5);
  while (system_clock::now() < expire) {
    if (try_to_do_something())
      return;
  }
  throw std::runtime_error("Timed out");
}
```

Listing 2

```
using namespace std;
void do_something_for_a_while(deque<int> &q,
  mutex &protect_q,
  condition_variable &q_changed)
{
  std::unique_lock<mutex> lock(protect_q);
  auto const expire = system_clock::now()
    +  seconds(5);
  while (system_clock::now() < expire) {
    if (q_changed.wait_until(lock, expire,
       [&q] { return !q.empty(); }))
```

Listing 3

On POSIX systems, this clock is known as `CLOCK_MONOTONIC` and the current time can be retrieved using the `clock_gettime` POSIX function. Unfortunately, the lack of 64-bit types back when this function was invented means that the seconds and nanoseconds are stored separately in a structure. Listing 4 (overleaf) uses a function to tell whether a specified timeout has expired.

This gets more complex if the timeout is not a whole number of seconds because extra housekeeping is required to ensure that the nanoseconds part is kept within permitted bounds. If you find yourself needing to do this then gnulib [GNU] provides helpful functions.

Time points measured against a monotonic clock will usually not be comparable between machines

```
bool expired(const timespec *expire)
{
  struct timespec now;
  clock_gettime(CLOCK_MONOTONIC, &now);
  if (now.tv_sec < expire.tv_sec)
    return false;
  if (now.tv_sec > expire.tv_sec)
    return true;
  return now.tv_nsec > expire.tv_nsec;
}

bool do_something_for_a_while()
{
  struct timespec expire;
  clock_gettime(CLOCK_MONOTONIC, &expire);
  expire.tv_sec += 5;
  while (!expired(&expire)) {
    if (try_to_do_something())
      return true;
  }
  return false;
}
```

Listing 4

libstdc++ and libc++ use `CLOCK_MONOTONIC` to implement C++ `std::chrono::steady_clock`, which provides a much easier way to work with absolute timeouts. Using it is just a matter of changing `system_clock` to `steady_clock` in Listing 2 to get Listing 5 and in Listing 3 to get Listing 6.

The compiler's type checking ensures that you can't accidentally compare time points from different clock sources against each other, making this much safer than the C version, which must rely on the clock passed to `clock_gettime` being consistent.

Both of these examples are now immune to system clock changes.

```
void do_something_for_a_while()
{
  using namespace std::chrono;
  auto const expire =
    steady_clock::now()  + seconds(5);
  while (steady::now() < expire) {
    if (try_to_do_something())
      return;
  }
  throw std::runtime_error("Timed out");
}
```

Listing 5

If you're stuck using C++98 or C++03 then Boost [Boost] provides `boost::chrono`. It also provides a precursor named `boost::posix_time`, but that should probably be avoided for new code.

Time points measured against a monotonic clock will usually not be comparable between machines. On Linux, `CLOCK_MONOTONIC` is actually the system uptime. In a distributed system, such as video playback synchronised across multiple screens, you may have NTP or PTP [Wikipedia-2] working hard to keep the system clock synchronised across multiple devices. In that case it makes more sense to use `std::chrono::system_clock` to agree a specific time to start playback and to control the playback speed. I imagine that a similar situation could occur in other distributed systems.

If we follow the advice above to use relative timeouts where we can and `CLOCK_MONOTONIC` or `std::chrono::steady_clock` where we can't, then all will be lovely, right? Well, yes and no. Unfortunately, current versions of GNU libstdc++ and Clang libc++ lack full support for using `std::chrono::steady_clock` timeouts for thread-synchronisation primitives and tend to convert silently back to `std::chrono::system_clock`, which makes the timeouts subject to misbehaviour when the system clock warps again (although in some cases the window of opportunity can be very small due to the actual wait being a relative one again.) They need to do this because POSIX doesn't currently provide suitable equivalents of the thread functions that are capable of accepting `CLOCK_MONOTONIC` timeouts [Crowe18]. There are new functions [AustinGroup] planned to address this. Glibc v2.30 and later contain these new functions and various patches have already been accepted for libstdc++ to use these functions in GCC 10 to fix the methods on `std::condition_variable`, `std::timed_mutex` and `std::shared_timed_mutex` that accept timeouts. Unfortunately some of the patches [Crowe20] to fix `std::future` didn't make it in before the freeze, but they'll hopefully be in GCC 11. I believe that similar changes are making their way into Clang libc++ too. If you are stuck using

```
using namespace std;
void do_something_for_a_while(deque<int> &q,
  mutex &protect_q,
  condition_variable &q_changed)
{
  std::unique_lock<mutex> lock(protect_q);
  auto const expire =
    steady_clock::now() + seconds(5);
  while (steady_clock::now() < expire) {
    if (q_changed.wait_until(lock, expire,
      [&q] { return !q.empty(); }))
      do_something(q);
  }
  throw std::runtime_error("Timed out");
}
```

Listing 6

If you follow the advice ... you will automatically get the fixes when your code is compiled with newer standard library versions

earlier versions then I believe that at least some of these problems are resolved in the Boost equivalents of the standard library functions.

If you follow the advice above, then the situation will be slightly better than if you'd used `std::chrono::system_clock` in your code right now and you will automatically get the fixes when your code is compiled with newer standard library versions. Many of the functions involved are inline so the fixes require more than upgrading the shared library.

Summary

- Use relative timeouts to standard library functions when performing a single operation.
- Use `CLOCK_REALTIME` or `std::chrono::system_clock` when your times and timeout relate to time in the real world and you want to react to someone warping the system clock. For example, a calendar or public transport tracking application.
- Use `CLOCK_MONOTONIC` or `std::chrono::steady_clock` when your times relate to elapsed time that should not change if someone warps the system clock. For example, network timeouts and refresh intervals.
- Use `CLOCK_REALTIME` and `std::chrono::system_clock` when the devices involved are known to have their clocks synchronised and you wish to share timestamps between those devices.
- Keep your toolchain up to date (and apply patches if you can) to ensure that you have the latest fixes. If you can't then look at using Boost instead. ■

Thanks

Thanks to members of the Austin Group, glibc and libstdc++ maintainers for helping me to turn the scratching of one small itch (in `std::condition_variable`) into fixing this class of problems more widely across POSIX and the C++ standard library. Thanks to the ACCU *Overload* reviewers and Jean-Marc Beaufils for providing feedback.

References

[AustinGroup] Mike Crowe in *Austin Group Defect Tracker*: https://www.austingroupbugs.net/view.php?id=1216

[Boost] https://www.boost.org

[Crowe18] Mike Crowe 'The clock used for waiting on a condition variable is a property of the wait, not the condition variable', at: http://randombitsofuselessinformation.blogspot.com/2018/10/the-clock-used-for-waiting-on-condition.html

[Crowe20] Mike Crowe in *GCC Bugzilla*: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=93542

[GNU] 'Gnulib – The GNU Portability Library' at: https://www.gnu.org/software/gnulib/

[Hinnant18] Howard E. Hinnant and Tomasz Kaminski, 'Extending <chrono> to Calendars and Timezones' posted 16 March 2018 at: https://howardhinnant.github.io/date/d0355r7.html

[Wikipedia-1] 'Network Time Protocol' at: https://en.wikipedia.org/wiki/Network_Time_Protocol

[Wikipedia-2] 'Precision Time Protocol': https://en.wikipedia.org/wiki/Precision_Time_Protocol

A Day in the Life of a Full-Stack Developer

Many roles claim to be full stack. Teedy Deigh shares a day in the life of a full stack developer.

06:00

Uh... what... *Snooze*

06:15

Snooze

06:30

Snooze

06:45

Falls out of bed

Why did I set my alarm for this time of night... morning... whatever...?

Oh yes. Full-stack development. Read and watched some stuff on it yesterday. Couldn't find anything of substance. Sure, a whole load of stuff on web development – JavaScript, HTML, CSS and the occasional database – but nothing on the rest of the stack and how to do it properly.

So, thought I'd better explore it for myself. Full stack, full day, full on!

07:03

OK, dragged a comb across my head, found my way downstairs, drank a cup. Now I've looked up, I've noticed I'm late.

I said was going to get my head down and online by 06:30. Time for another cup.

07:54

OK, someone on the internet was wrong, but I've fixed that now. Where shall we start? Oh yeah, that JavaScript framework I was looking at yesterday – Anglia? Nod? Mithrandir? Whatever. If we're gonna code the full stack properly, we're going to need to write our own framework. Can't just go around reusing other people's stacks – they don't call it partial-stack development, amiright?!

09:47

OK, that's the first version of `OverReact.JS` pushed. No tests, no docs, but still enough to show these so-called full-stack developers how it's done.

Let's just try it out on a pet store example. No, that's not very ethical... mmm, OK, something else that's enterprisey... FizzBuzz it is, then!

09:58

Hmm. Something's not working. There's a bug. Not sure what it could be.

10:58

OK, still don't know what the problem is – and that's after five cups of coffee! Like looking for a needle and thread in a call stack. Bloody legacy code.

Time for a rewrite – `OverReact.JS2`! Maybe this time I'll write some tests. Gonna need a testing framework.

12:03

Right, that only took three more coffees, but now the Jitters unit-test framework is good to go.

Hmm... must be time for breakfast, then the **`OverReact`** rewrite.

12:29

Well, I must confess, I'm a little surprised that that person on the internet is still wrong, especially after all the suggestions I offered!

Anyway. Onward to `OverReact.JS2`!

15:11

OK, that took a little longer than expected, but given what I can now see are the glaring deficiencies of the first version, it's important to account for all possible use cases and make everything more configurable. I think I once heard someone say that it was important to try to please all the people all of the time... or something like that.

15:33

I don't believe it! A bug. Again.

Hmm... perhaps I should have used my Jitters framework to write some tests with?

Is it time for lunch? Must be time for lunch.

15:57

The plan was to write some tests, but I've changed my mind as that will just slow me down, and I'm trying to be Agile™. 'Responding to change over following a plan' is the mantra I'm following today.

The `OverReact.JS2` code doesn't look like it has any issues – in fact, it looks quite happy from where I'm sitting: emoji identifiers FTW! I chose an uncompromisingly minimal coding style that shuns the excesses of commenting, indentation, long identifiers and clunky, procedural control flow. So the problem can't be there, right? Must be somewhere else on the stack.

Perhaps the JavaScript engine is at fault?

18:12

Writing Veg.JooS, my new JavaScript engine, seems to be going well so far, although I suspect that I might be looking at an all-nighter. That, however, is not my main concern: my reliance on an existing C compiler,

Teedy Deigh plans to retire somewhere nice. It is said, however, that no plan of battle survives contact with the enemy. In the meantime, you'll find her hoarding zeroes and ones and toilet paper.

The plan was to write some tests, but I've changed my mind as that will just slow me down, and I'm trying to be Agile™

regardless of its long-standing open-source pedigree, may be a weak link in my tool chain.

Rather than stand on the toes of giants, it may be wiser to write my own compiler.

19:49

Occurs to me that it might be time for lunch... or something.

20:01

Well, this is turning into something of an odyssey, but I can't believe I overlooked an obvious step: I need to create a compiler compiler to save time and trouble in future!

OK, back to food... and, can you believe it, that person on the internet is still wrong?

20:15

The patterns I'm going to use to create YakShavR are clear in my head, but there's a couple of things I think I'm missing.

Ah yes, coffee's what I'm missing!

20:59

It's clear that what I've been missing is the right language. No, I don't mean "@#$%&!" – although there's been plenty of that today – I mean for implementing YakShavR. I started writing it in C but, you know, chickens and eggs!

I've also decided that maybe C is not the right way to go for Veg.JooS. I need a better systems programming language, something with less historical baggage, unsullied by popularity and compromise. Think I might call it ToldUSo, and will post a repo link to the person – now people! – on the internet who is wrong.

Anyway, how to break the chicken-and-egg cycle? Lisp! Pure and simple, and no need for tests as it's already functional.

21:49

Two pomodoros later (minus the wasteful five-minute breaks) and I've got the core Lisp eval function written out on my whiteboard. Or evil, looking at the way I've written it – well, scribbled – which does highlight a bit of a problem: I was going to code it up, but I'm not sure I can read it now. Looks like it was written by a caffeinated spider.

Caffeine. Coffee. Yup, that's the problem. Gonna need more of that if I'm to crack this code.

22:41

OK, I think I've almost got it. Must be time for... a meal.

23:45

Perhaps, a little unbelievably, the people on the internet who are wrong might actually be right. They pointed out that I don't seem to be willing to compromise. Damn straight! Uncompromising is my middle name (albeit one I had to get changed by deed poll, as my parents originally chose Davina).

Andrew Koenig observed that "People who brook no compromise in programming languages should program in lambda calculus or machine language." I need to bootstrap the Lisp interpreter and the answer has been staring me in the face. Time to get back to the metal.

01:23

More coffee. Things are going well.

02:11

More coffee. Things are not going well.

Working at this level is high RISC; I need to go high Church. Alonzo Church, that is. Lambda calculus is the purest of the pure. It's so pure that it makes Haskell look like JavaScript. So pure, in fact, that it doesn't even have numbers – there are only lambdas. Using Church numeral encodings, you define your own numbers.

This is perhaps the truest expression of the software craft movement: hand-crafted, artisanal integers. There are no Booleans, so you make your own truth. Mind-blowing. Want a list? You can define a pair, and then nominate roles for each element – first blow your mind out in a car, second cdr be used as the tail – and cons everything up from there.

02:56

Food? Coffee? Am I hungry? Am I thirsty? What am I doing? Who am I?

03:14

Definitely jitters – no, not the testing framework. Hallucinations? Hallucinations.

04:53

Wuh! Must've dozed off. Went into a dream.

Progress so far: I've got zeroes and ones. Lots of them. Frolicking lambdas ready to be crafted into a full stack.

Instead of FizzBuzz, I'm going to work on some more ambitious and algorithmic: a highly configurable, reusable, general-purpose algorithm that counts anything from sheep in a field, a semiring or a group, to holes in Blackburn, Lancashire (even if they're rather small).

Feeling a little strange. And I'm out of coffee.

06:00

Uh... what... *Snooze*

"The magazines"

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

"The conferences"

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

"The community"

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

"The online forums"

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe