overload 173
FEBRUARY 2023 £4.50

Floating Point Comparison

Paul Floyd shows how you should perform floating-point comparisons (and how not to do it)
.

Determining If A Template Specialization Exists

Lukas Barth determines whether a class or function template can be used with specific arguments

Stack Frame Layout On x86-84

Eli Bendersky describes the x86-64 stack layout in detail

Value-Oriented Programming

Lucian Radu Teodorescu explores this paradigm

Afterwood

Chris Oldwood asks us to meet him halfway when considering the value of meetings

A magazine of ACCU
ISSN: 1354-3172

ACCU 2023

Hosting a wide selection of keynote speakers, training sessions and workshops, this Conference is designed by programmers, for programmers, about programming.

Wednesday 19th to Saturday 22nd April 2023 at Bristol Marriott Hotel City Centre
Pre-conference workshops on Monday 17th & Tuesday 18th April 2023

The content

The ACCU Conference has C and C++ at its core, but also has sessions relating to other languages – such as C#, D, F#, Go, Haskell, Java, Kotlin, Lisp, Python, Ruby, Rust, and Swift.

It's not just about languages, though. There are sessions on tools, techniques and processes (such as TDD, BDD and how to 'do programming right').

The format

This is an in-person conference, with much of the content over the conference also being streamed.

The main event starts on Wednesday 19th April and closes on Saturday 22nd April. Each day has five concurrent streams, focusing on different areas of programming and development.

The pre-conference workshops (not accessible virtually) run on Monday 17th and Tuesday 18th April.

The full programme – including confirmed keynote speakers – is at www.accuconference.org

The attendees

This is a great event to attend whether you are a software developer or programmer working as an employee or as a consultant or contractor.

The event welcomes:

- Talented and innovative programmers and developers
- Industry leaders and game changers
- Internationally renowned speakers
- Industry organisations

The tickets

	Member	Non-member
Day ticket	From £195	From £225
In-person (4 days)	From £635	From £745
Virtual (4 days)	From £425	From £535

Early bird discount until Tuesday 28 February 2023

The Conference is always popular. We recommend you book now to avoid disappointment at:

www.accuconference.org

Check out our social media channels for updates ahead of the event, and keep an eye on our YouTube channel for exclusive content!

- Facebook: www.facebook.com/accuorg
- Twitter: twitter.com/ACCUConf
- YouTube: @ACCUConf

Book today to attend the ACCU Conference 2023

February 2023

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Ben Curry
b.d.curry@gmail.com

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.co.uk

Balog Pal
pasa@lib.hu

Tor Arve Stangeland
tor.arve.stangeland@gmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Lari Bat (from iStock Photo).

Copy deadlines

All articles intended for publication in *Overload* 174 should be submitted by 1st March 2023 and those for *Overload* 175 by 1st May 2023.

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

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.

Copyrights and Trademarks

Under Pressure

Mounting pressure can be problematic. Frances Buontempo takes a step back and wonders if pressure is always a bad thing.

The news in the UK has told me everything is under pressure, and I know the feeling. I have been trying to write two conference talks, along with doing a variety of other tasks and there are just not enough hours in the day. Suffice it to say, I haven't written an editorial, but instead thought long and hard about being under pressure. Much of this is of my own making and I must learn to say, "No" more often. Maybe the start of a new year is a good time to reflect and think about how to make changes.

Mind you, pressure, in and of itself, is not a bad thing. We own a pressure cooker, which is useful for cooking beans and pulses. My hazy memory of physics from school tells me that the cooker is sealed, so keeps the volume fixed, meaning as the pressure increases when heat is supplied, the temperature also increases, cooking food more quickly than in an unsealed pan. The internet suggests this is related to Gay Lussac's law, "the pressure of a gas varies directly with temperature when mass and volume are kept constant", so that

$$\frac{P_1}{V_1} = \frac{P_2}{V_2}$$

for two pressures, P_1, P_2 and two volumes, V_1, V_2 [ChemTalk]. Similar physics means trying to boil a kettle up a mountain is somewhat difficult, or rather the water boils, but at a much lower temperature than some people would want. Since the atmospheric pressure is lower there, water boils below 100º, making a substandard cup of tea. Pressure is sometimes useful!

Running performance tests on a system can be illuminating. Stress or load tests simulate high traffic to a website, or service or similar, to see what happens. There is a subtle difference between them. A load test shows what happens under an expected load, meaning you know how long you expect a response or similar to take. In contrast, a stress test finds the upper limits of a system's capacity, meaning you can prepare for a DDoS attack, or at least be aware of what might happen to your system [LoadNinja]. You can even run such tests as part of your CI pipeline, to keep an eye on timings and so on. Putting your system under pressure in a dev environment prepares you for production. In theory.

Knowing upper limits can be important. Big-oh notation is a fundamental part of many computer science courses. For once, that was not a typo, the technical term is big-o, but, like me, you might say uh-oh internally if someone asks what the space or time complexity of a specific algorithm is, when you haven't thought about this for a long time. Nothing like the pressure of an interview or similar to make your brain seize up. Of course, big-o is short for order of approximation [Wikipedia]. It tells us how many calculations, or how much space, an algorithm might take in the worst case, giving a sense of how an algorithms scales with the item count. Roger Orr wrote an article a long while ago, looking at what can actually happen in real life [Orr14]. He reminded us that big-O "may ignore other factors, such as memory access costs that have become increasingly important in recent years." Given the article was written in 2014, I wonder how different the results might look today. Feel free to re-read the article and run the test cases to see what happens now.

The runtime behaviour of a system can be hard to predict, and different architectures will have different runtime profiles. If a system appears to be under strain, you can try throwing more hardware at it, as the phrase goes. Not literally, of course. Now, we've all read, or are at least aware of, *The Mythical Man Month* by Fred Brooks, originally published in 1975. He shows that adding more people to a software project that is behind schedule is likely to delay it even longer. I wonder if adding more hardware to a system can have a similar effect. Changing the hardware can improve some performance measures. Certainly an SSD is likely to be quicker than a spinning disk, and probably use less energy and generate less heat. The SSD may not improve the performance of your linked list, though. And adding a second computer might make it take even longer to traverse the nodes of the list. Changing the data structure is more likely to improve performance. Adding more hardware can speed up calculations, provided you get the parallelism right. Some problems are embarrassingly parallel, in the sense that they do not need to communicate, which is often the bottleneck, both in the *Mythical Man Month* and many parallel algorithms. Others are not, which is why we often see multithreaded code run slower than equivalent single threaded code. The point about swapping to an SSD shows that changing a setup can make more difference than adding more of the same. In order to cope with pressure, finding a way to do things differently often helps. If a deadline is looming, the best approach might be to limit what gets delivered, so a minimum viable product (MVP). Many people have written about this, and an MVP is about more than doing the bare minimum. The Agile Alliance attributes the term to Eric Ries in his *Lean Startup* book [Agile Alliance], and emphasizes the MVP as the core piece of a strategy of experimentation. Finding a "version of a new product which allows a team to collect the maximum amount of validated learning about customers with the least effort," means you get feedback quickly. This might mean teams "dramatically change a product that they deliver to their customers or abandon the product together based on feedback they receive from their customers." Furthermore the MVP is supposed to reduce stress or pressure. As with many agile ideas, small baby steps FTW.

You cannot add more people to solve certain problems, such as taking an exam. You have a limited amount of time to prepare beforehand and a maximum amount of time in the examination itself. Though you can find

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD using AI and data mining. She's written a book about machine learning: *Genetic Algortithms and Machine Learning for Programmers*. 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.

friends to help you revise, you cannot parallelise that task, with three or four people learning different parts and taking a section of the exam each. Well, you could try, but the invigilator will almost certainly spot what you are doing. However, if a group of friends split up to learn different aspects of a subject, and reconvene to share what they have learnt, this can work. Each person will know some of the subject in depth and by explaining it to others may understand even better. Those they explain to might pick up some of the subset explained to them. Collaborating can take some of the pressure off, and give an arena to vent frustration in. Without a group of friends to help, you can choose to focus on a smaller subset yourself, and at least ensure you can answer some of the exam questions.

If you are building a software system with several moving parts, you might split the work up between teams. However you make the split, you need to make sure the whole system works when the pieces are glued together. I've seen this succeed when pieces from another team are mocked out; for example, a service returning hard coded results, until the database is in place, and so on. When this happens, the teams can program to an interface and be sure the parts will slot together. Without an initial discussion on the interfaces to use, trying to make the parts work together at the last minute often leads to trouble, later night and fraught discussions. Some pressure is avoidable.

Pressure can lead to innovation, but so can the offer of a reward. King Oscar II of Sweden and Norway offered a prize in 1885 for a solution to the problem of determining the stability of the solar system. The question was, "Will the planets of the solar system continue forever in much the same arrangement as they do at present? Or could something dramatic happen, such as a planet being flung out of the solar system entirely or colliding with the Sun?" [Britannica].

At the time, modelling the motion of two bodies was possible, but three or more, let alone all of our solar system's planets and moons. Poincaré won the prize, though he couldn't fully solve the problem. His ideas lead to differential equations. Differential equations can be used to model all kinds of dynamical systems, and every now and then people find relatively simple looking equations that lead to very complex solutions. "Even when there is no hint of randomness in the equations, there can be genuine elements of randomness in the solutions." [Britannica]. Now, whether the solutions exhibit genuine randomness is a big question. Certainly you might see one small change in initial conditions leading to a large change in outcomes. The recurrence relationship governing the familiar Mandelbrot set gives a clear example of this. The simple looking equation used is

$$z_{n+1} = z_n^2 + c$$

where c is a complex number and z starts at 0. Any numbers which leave z bounded belong to the set. The boundary of the set is very complicated and if you zoom in, you start to see patterns repeating. This is often cited as an example of chaos. I am sure this is something very different to randomness. Surprising complex behaviour emerging from a simple model is deterministic, and I suspect randomness is often uses synonymously with non-determinism. Radioactive decay is often regarded as random, in the sense that we are not able to predict when a specific atom will decay, even if we can be very precise about the half-life of a radioactive material. Einstein said "God does not play dice with the universe" in response to the probabilistic laws used in quantum mechanics. He did not like the idea of randomness as a fundamental feature of any theory. [Natarajan08]. He had other objections too, but that would require a digression into linear models and more maths and physics. The salient point is that sequence regarded as random do often have predictable properties. Whether anything is truly random is another matter. I shall attempt to broach this subject in my ACCU conference talk this year, if I ever finish my slides.

We've looked briefly at pressure and how it can have positive aspects. If people or systems are under too much pressure, they usually crack or fail in some way. Having knowledge on the upper bound that a setup can handle is useful, but if the environment strays towards that upper bound, trouble is on the way. That Poincaré won a prize for developing a new area of mathematics is marvelous. Sometimes curiosity and a carrot, rather than a stick, is a great motivator. If you are under pressure to complete something by a deadline, work extra hours, or do something else you can't manage, learn to say "No". I haven't yet, but might give it a go this year.

References

[Agile Alliance] Minimal Viable Product (MVP): at https://www.agilealliance.org/glossary/mvp/

[Britannica] Dynamic systems theory and chaos: https://www.britannica.com/science/analysis-mathematics/Dynamical-systems-theory-and-chaos

[ChemTalk] Gay-Lussac's Law: https://chemistrytalk.org/gay-lussacs-law/

[LoadNinja] https://loadninja.com/articles/load-stress-testing/

[Natarajan08] Vasant Natarajan (2008) 'What Einstein meant when he said "God does not play dice…"', *Resonance*, published July 2008, pp.655–660. Available online at: https://arxiv.org/ftp/arxiv/papers/1301/1301.1656.pdf

[Orr14] Roger Orr (2014) 'Order Notation in Practice', *Overload*, 22(124):14-20, December 2014. https://accu.org/journals/overload/22/124/overload124.pdf#page=15

[Wikipedia] Big *O* notation: https://en.wikipedia.org/wiki/Big_O_notation

Floating-Point Comparison

Comparing floating point values can be difficult. Paul Floyd shows how you should perform floating-point comparisons (and how not to do it).

It is often said that you shouldn't compare two floating point numbers for equality. To quote one erstwhile guru:

> Any programmer claiming competence that uses equality between floating point values clearly requires re-education on this point.

I agree that in general you need to take care with floating point comparisons, but there are times when using `==` and `!=` is the right thing to do. There are two major problems that I see

1. You don't want to use a tolerance on every single floating-point comparison. Not knowing what you are doing and just randomly sprinkling tolerances into your comparisons won't make the problems go away.
2. Often the way that the tolerance is used is just plain wrong.

When you use a tolerance to make a floating-point comparison you are in effect saying, "This value has some inaccuracy, so I'll do an inaccurate comparison to compensate for it". The two questions that need to be answered are a) is that really the case and b) how inaccurate should the comparison be?

A lot depends on your domain of application. First, let's consider mathematical libraries. Here you want to strive to get as much accuracy as possible (and indeed the precision may be mandated by [IEEE754]). I'll take the C standard library '`pow`' function as an example. One implementation can be found on GitHub [GNU].

Consider this check

```
if (y == 0)
  return 1.0;
```

I think that it would be an extremely bad idea to use a tolerance here. I expect `x**0` to be 1. I do not expect `x**1e-5` to also be 1 (unless `x` is 1 or close to it). There are similar tests for `x**1`, `x**2` and `x**-1`. There is not one use of tolerance in this function, and several floating-point comparisons.

Another situation where you shouldn't use a tolerance is if you are doing 3-way testing like

```
If (a < b) {
  // handle less than
} else if (a == b) {
  // handle equality
} else {
  // handle greater than
}
```

This is fine as it is and using a tolerance instead of `a == b` just adds gratuitous inaccuracy.

Paul Floyd has been writing software, mostly in C++ and C, for about 30 years. He lives near Grenoble, on the edge of the French Alps and works for Siemens EDA developing tools for analogue electronic circuit simulation. In his spare time, he maintains Valgrind. He can be contacted at pjfloyd@wanadoo.fr

If your numbers come from physical observations, then your precision is probably way below the precision of double and even float. Double has a precision of about 1 part in 10^{16} (1 in ten quadrillion). That's about a thousand times more precise than the most accurate measurements that we can make [StackExchange].

Even if your data doesn't come from some inexact source, if you do any sort of operation on the data then there will be some rounding error. One common case that often surprises the uninitiated is the conversion from base 10 strings to binary floating-point (via functions like 'atof', `strtod` or `std::stod`). This doesn't give an exact result in the sense of being identical to the original base 10 value. Furthermore, there will be rounding errors on most floating-point operations. The analysis of floating-point errors is too big a subject to cover even superficially here.

If you'd like to delve deeper into the subject, I recommend *Accuracy and Stability of Numerical Algorithms* [Higham02]. For an overview of techniques to mitigate rounding errors, see the series of articles published by the late Richard [Harris11]. Thirdly, there is 'What every computer scientist should know about floating-point arithmetic' [Goldberg91] which gives a thorough explanation of floating-point. All I will say is that if your numerical code is not too badly written then rounding errors will generally accumulate slowly in the manner of a drunkard's walk.

Potential errors

What could happen if you don't use a tolerance in a floating-point comparison? The most immediate thing is that the wrong branch of your code could execute.

```
if (a == b) {
  action1();
} else {
  action2();
}
```

In the above code, there is the risk that `a` and `b` are very close but not equal and `action2()` gets performed rather than `action1()`.

One possibly worse situation that can occur is that your code hangs in a loop.

```
while (estimation != answer) {
  estimation = refinement(x);
}
```

I say possibly worse since at least if this does get stuck in an infinite loop it will be easy to debug and fix. Algorithms like this that use progressive refinement are common in numerical analysis.

How not to perform floating-point comparisons.

For pedagogical purposes, I'll start with some bad code that shows how not to perform floating-point comparison. This will build up to something reasonably functional. All my examples use `double` but apply equally to `float` and `long double`.

My first example is the worst code. Don't do this.

The idea behind a reltol is to have a tolerance that scales with the values being compared which avoids the problems with big and small values

```
bool cmpEq(double a, double b)
{
  return std::fabs(a - b) <
    std::numeric_limits<double>::min();
}
```

I've seen this in production and it almost totally wrong. This applies equally to the macro `DBL_MIN`. The problem is that the 'min' value here is the smallest possible non-denormalized value of a double, typically something like 2e-308. For the expression to be true, either `a` and `b` have to be equal or either/both denormalized. (For those that are not familiar with denormalized numbers, they are a special case of the set of floating-point numbers that extend the minimum possible value to approx. 5e-324 at the expense of losing precision and slower CPU execution.)

To all intents and purposes, this code only gives a false sense of security and behaves like `operator==`.

The intention here was probably to use `std::numeric_limits<double>::epsilon()`. Epsilon is the smallest representable number that can be added to 1.0 to make a new value. This means that it represents the limits of numerical precision around the value 1.0. The precision is determined by the bits of the hardware – 53 binary digits for 8-byte doubles which is roughly 16 decimal digits.

This is not the same thing as the accuracy, which is the measure of error of a sequence of computations.

Since I've mentioned epsilon, how about using that? Quite often I see code like

```
bool cmpEq(double a, double b)
{
  return std::fabs(a - b) <
    std::numeric_limits<double>::epsilon()*10.0;
}
```

(`DBL_EPSILON` could be used instead of `std::numeric_limits<double>::epsilon()`).

I've somewhat arbitrarily multiplied it by a factor of 10 to give a bit of margin. But what if `a` and `b` are far away from 1.0? If both `a` and `b` are much smaller than 1.0 then the above comparison will always be true. Let's say `a` is 1e-18 and `b` is 1e-24. `a` is a million times bigger than `b` yet the above function will say that they are equal. That may not be what you want. Now what if `a` and `b` are much greater than 1.0? Let's say `a` is 1e10 and `b` is 1.0000000000001e10. That's a difference that could arise from rounding error, but the difference is still 1e-3 which is much greater than the 2e-15 tolerance used above. In short, the above function will only work well for values that are not too far from 1.0. That might be a big restriction.

For my third bad comparison function, I'll introduce the notion of relative tolerance (reltol). The idea behind a reltol is to have a tolerance that scales with the values being compared which avoids the problems with big and small values that the previous version suffered from. Unfortunately, there are still weaknesses and several ways to get this wrong.

```
bool cmpEq(double a, double b)
{
  double reltol = fabs(a)*1e-7;
  return std::fabs(a - b) < reltol;
}
```

There is a potential problem with this function not being commutative if the double tolerance is larger. More specifically, if the factor used for the tolerance is larger than `sqrt(DBL_EPSILON)` (which is about 1.5e-8) and the difference between `a` and `b` is a factor between `(1.0 + DBL_EPSILON)` and `(1.0 + DBL_EPSILON + DBL_EPSILON^2)` then the comparison is not commutative.

This can be illustrated by Listing 1, which outputs only "then these should also equal". That can be the source of nasty bugs that are hard to track down (for instance if this function were used as part of a test in code that requires strict weak ordering, for instance as a predicate for `std::sort` or the ordered containers `std::map` and `std::set`).

The second problem is that it doesn't handle infinity nicely. Infinity is sticky, which mean that the reltol will also be infinite. So, if both `a` and `b` are infinite then the expression `std::fabs(a - b) < reltol` becomes `Inf < Inf` which is false, but `Inf == Inf` is true.

There are a few things that we can do to fix these problems. Firstly, we can apply the reltol factor to a combination of a and b. But what combination? I've seen their sum, average, min and max used. The sum and average have the disadvantage of requiring checks that there is no floating-point underflow or overflow, so I don't recommend using them. That leaves min and max. The only difference is that the choice will slightly narrow or widen the tolerance, respectively. Since the reltol factor itself is somewhat arbitrary, that's not a big difference. I've seen the min version referred to

```
#include <iostream>
#include <limits>
#include <cmath>

const double EPSILON = 1e-7;

bool cmpEq(double a, double b)
{
  double reltol = std::fabs(a) * EPSILON;
  return std::fabs(a - b) < reltol;
}
int main()
{
  double testValue = 42.0;
  double otherValue = testValue *
    (1.0 + EPSILON + EPSILON * EPSILON / 2.0);
  if (cmpEq(testValue, otherValue)) {
    std::cout << "if these are equal then \n";
  }
  if (cmpEq(otherValue, testValue)) {
    std::cout <<
    "then these should also equal\n";
  }
}
```

Listing 1

```
bool cmpEq(double a, double b)
{
  const double EPSILON = 1e-7;
  if (a == b) {
    return true;
  }
  double reltol = std::max(std::fabs(a),
    std::fabs(b)) * EPSILON;
  return std::fabs(a - b) < reltol;
}
```

Listing 2

as "essentially equal to" ([Knuth97]), and the Boost documentation refers to the max version as "close enough with tolerance" and the min version as "very close with tolerance".

The problem with infinity can be fixed by adding a quick check for equality of the arguments.

So now we have Listing 2.

This works with infinity (and NaN which should never compare equal) as well as my corner case with values separated by a factor of about 1.0 + EPSILON.

Are we all done? Sadly not. Using just a reltol is not good for very small values that are close to zero. `cmpEq(1e-85, 2e-85)` will return false. In many situations values these small are just numerical noise possibly arising from cancellation and should be considered equal to zero. To eliminate these, we must add back a test along the lines of our second version using an absolute tolerance (abstol).

One final version is in Listing 3.

This is still not completely safe. If `a` and `b` are very large and of opposite signs, then `a - b` could overflow. If `a` and `b` are both very small, then multiplying by EPSILON could underflow – moving the abstol comparison earlier would fix that.

That only leaves one thing and that is how to choose the relatively arbitrary constants used for abstol and reltol EPSILON? For that you need to apply your domain knowledge as unfortunately there is no one-size-fits-all solution. It also depends on your accuracy requirements. I work in the domain of analogue microelectronic circuit simulation. In this domain voltages are typically 1V and currents are typically 1μA. A good rule of thumb is to have an EPSILON for reltol something like 1e-4 and an abstol something like 1e-6 times your typical domain values. So, for circuit simulation that would be a voltage abstol of 1e-6 and a current abstol of 1e-12 [Kundert95].

Existing implementations

Most unit test libraries will have functions for performing floating-point comparisons, for instance the WithinAbsMatcher of [Catch2]. Boost has `floating_point_comparison.hpp` [Boost]. One thing that I don't like about this is that the code normalizes the difference by dividing by the values. Floating-point division is slow, and I'd rather avoid it when possible. The other problem that I see with this is that it suffers from Boost template bloat. The header when pre-processed is about 54k lines of which about 42k lines are code. That's a lot for what is essentially a 5-line function. A functor `close_at_tolerance` is provided which tests with a relative tolerance. Despite my reservations I would still prefer to see boost being used than a wrong home-rolled comparator.

```
bool cmpEq(double a, double b,
  double epsilon = 1e-7, double abstol = 1e-12)
{
  if (a == b) {
    return true;
  }
  double diff = std::fabs(a - b);
  double reltol = std::max(std::fabs(a),
    std::fabs(b)) * epsilon;
  return diff < reltol || diff < abstol;
}
```

Listing 3

Summary

1. Don't blindly use tolerances in all floating-point comparisons.
2. Don't use `std::numeric_limits<double>::min()` or `DOUBLE_MIN` for tolerances.
3. Consider whether you need an absolute tolerance or not.
4. Choose abstol and reltol EPSILON with care depending on the domain of application.
5. Consider using existing and well-tested libraries. ■

References

[Boost] floating_point_comparison.hpp: https://www.boost.org/doc/libs/1_81_0/boost/test/tools/floating_point_comparison.hpp

[Catch2] catch_matchers_floating_point.hpp: https://github.com/catchorg/Catch2/blob/223d8d638297454638459f7f6ef7db60b1adae99/src/catch2/matchers/catch_matchers_floating_point.hpp

[GNU] e_pow.c: https://github.com/lattera/glibc/blob/master/sysdeps/ieee754/dbl-64/e_pow.c

[Goldberg91] David Goldberg 'What every computer scientist shoudl know about floatig-point arithmetic', published in *ACM Computing Surveys*, Volume 23, Issue 1 March 1991 and availalbe at: https://dl.acm.org/doi/10.1145/103162.103163

[Harris11] Richard Harris wrote a series of four articles in *Overload* from 2010 to 2011:

- Why Fixed Point Won't Cure Your Floating Point Blues, *Overload* 18(100):14-21 available at: https://accu.org/journals/overload/18/100/harris_1717/
- Why Rationals Won't Cure Your Floating Point Blues, *Overload* 19(101):8–11 available at: https://accu.org/journals/overload/19/101/harris_1986/
- Why Interval Arithmetic Won't Cure Your Floating Point Blues, *Overload* 19(103):18-23 available at: https://accu.org/journals/overload/19/103/harris_1974/
- Why Computer Algebra Won't Cure Your Floating Point Blues, *Overload* 20(107):14-19 available at: https://accu.org/journals/overload/20/107/harris_1938/

[Higham02] Nicholas J. Higham (2002) *Accuracy and Stability of Numerical Algorithms*, 2nd Ed.,, SIAM, ISBN 978-0-898715-21-7

[IEEE754] 754-2019 – IEEE Standard for Floating-Point Arithmetic, available at: https://ieeexplore.ieee.org/document/8766229 (Non-free PDF download.)

[Knuth97] Donald E. Knuth (1997) *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, 3rd Ed, Addison-Wesley, ISBN 0-201-89684-2

[Kundert95] Ken Kundert (1995) Appendix A 'Simulator Options' from *The Designers Guide to SPICE and Spectre* available at: https://designers-guide.org/analysis/dg-spice/chA.pdf Details of the book are available at: https://designers-guide.org/analysis/dg-spice/index.html

[StackExchange] Physics: What is the most precise physical measurement ever performed? Question and answers available at: https://physics.stackexchange.com/questions/497087/what-is-the-most-precise-physical-measurement-ever-performed

Determining If A Template Specialization Exists

How do you tell if a class or function template can be used with specific arguments? Lukas Barth details his approach.

One C++17 problem I come across every now and then is to determine whether a certain class or function template specialization exists – say, for example, whether `std::swap<SomeType>` or `std::hash<SomeType>` can be used. I like to have solutions for these kind of problems in a template-toolbox, usually just a header file living somewhere in my project. In this article I try to build solutions that are as general as possible to be part of such a toolbox.

Note that it is not entirely clear what 'a specialization exists' means. Even though this might seem unintuitive, I'll postpone that discussion to the last section ('What do I mean by 'a specialization exists'?') and will, for now, continue with the intuitive sense that 'specialization exists' means 'I can use it at this place in the code'.

Where I mention the standard, I refer to the C++17 standard [C++17], and I usually use GCC 12 and Clang 15 as compilers. See the sidebar on MSVC at the end for why I'm not using it in this article.

Testing for a specific function template specialization

First, the easiest part: Testing for one specific function template specialization. I'll use `std::swap` as an example here, though in C++17 you should of course use `std::is_swappable` to test for the existence of `std::swap<T>`.

Without much ado, my proposed solution is in Listing 1.

Let's unpack this: The two `exists` overloads are doing the heavy lifting here. The goal is to have the preferred overload when called with the argument `42` (i.e., the overload taking `int`) return `true` if and only if `std::swap<T>` is available. To achieve this, we must only make sure that this overload is **not** available if `std::swap<T>` does not exist, which we do by SFINAE-ing it away if the expression

```
decltype(std::swap<T>(std::declval<T&>(),
                      std::declval<T&>()))
```

is malformed.

```
struct HasStdSwap {
private:
  template <class T, class Dummy =
    decltype(std::swap<T>(std::declval<T &>(),
                          std::declval<T &>()))>
  static constexpr bool exists(int) {
    return true;
  }
  template <class T> static constexpr bool
    exists(char) { return false; }

public:
  template <class T> static constexpr bool
      check() {
    return exists<T>(42); }
};
```

Listing 1

```
struct HasStdHash {
private:
  template <class T, class Dummy =
decltype(std::hash<T>{})>
  static constexpr bool exists(int) {
    return true;
  }

  template <class T>
  static constexpr bool exists(char) {
    return false;
  }

public:
  template <class T>
  static constexpr bool check() {
    return exists<T>(42);
  }
};
```

Listing 2

You can play with this at Compiler Explorer [CompExp-1]. Note that we need to use `std::declval<T&>()` instead of the more intuitive `std::declval<T>()` because the result type of `std::declval<T>()` is `T&&`, and `std::swap` can, of course, not take rvalue references.

Testing for a specific class template specialization

Now that we have a solution to test for a specific function template specialization, let's transfer this to *class* templates. We'll use `std::hash` as an example here.

To transform the above solution, we only need to figure out what to use as default-argument type for `Dummy`, i.e., something that is well-formed exactly in the cases where we want the result to be `true`. We can't just use `Dummy = std::hash<T>`, because `std::hash<T>` is a properly *declared* type for all types `T`! What we actually want to check is whether `std::hash<T>` has been *defined* and not just *declared*. If a type has only been declared (and not defined), it is an *incomplete* type. Thus we should use something that does work for all complete types, but not for incomplete types.

In the case of `std::hash`, we can assume that every definition of `std::hash` must have a default constructor (as mandated by the standard for `std::hash`), so we can do Listing 2.

This works nicely as you can see at Compiler Explorer [CompExp-2]. This is how you can use it:

```
std::cout << "Does std::string have std::hash? "
          << HasStdHash::check<std::string>();
```

Lukas Barth is a software engineer who has been using C++ almost exclusively for a couple of years now. After completing his computer science PhD with a focus on algorithms in 2020, he now works at MENTZ on building a journey planner for public transport. You can contact him at contact@lukas-barth.net

If we want to stay with a type, we can use something unintuitive: the type of an explicit call of the destructor.

A generic test for class templates

If I want to put this into my template toolbox, I can't have a implementation that's specific for `std::hash` (and one for `std::less`, one for `std::equal_to`, …). Instead, I want a more general form that works for **all** class templates, or at least those class templates that only take type template parameters.

To do this, I want to pass the class template to be tested as a template template parameter. Adapting our solution from above, Listing 3 is what we would end up with.

This does still work for `std::hash`, as you can see at Compiler Explorer [CompExp-3], when being used like this:

```
std::cout << "Does std::string have std::hash? "
<< IsSpecialized<std::hash>::check<std::string>();
```

However, by using `Tmpl<Args...>{}`, we assume that the class (i.e., the specialization we are interested in) has a default constructor, which may not be the case. We need something else that always works for any complete class, and never for an incomplete class.

If we want to stay with a type, we can use something unintuitive: the type of an explicit call of the destructor. While the destructor itself has no return type (as it does not return anything), the standard states in `[expr.call]`:

> If the postfix-expression designates a destructor, the type of the function call expression is void; […]

So Listing 4 will work regardless of how the template class is defined[1] (changes highlighted in Listing 4).

Note that we use `std::declval` to get a reference to `Tmpl<Args...>` without having to rely on its default constructor. Again you can see this at work at Compiler Explorer [CompExp-4].

```
template <template <class... InnerArgs>
          class Tmpl>
struct IsSpecialized {
private:
  template <class... Args,
    class dummy = decltype(Tmpl<Args...>{})>
  static constexpr bool exists(int) {
    return true;
  }
  template <class... Args>
  static constexpr bool exists(char) {
    return false;
  }
public:
  template <class... Args>
  static constexpr bool check() {
    return exists<Args...>(42);
  }
};
```

Listing 3

1 With the notable exception of the template class having a private destructor.

As an aside, if you know of a way to extend this to templates taking non-type template parameters, please let me know!

```
template <template <class... InnerArgs>
          class Tmpl>
struct IsSpecialized {
private:
  template <class... Args,
    class dummy =
    decltype(std::declval<Tmpl<Args...>>()
    .~Tmpl<Args...>())>
  static constexpr bool exists(int) {
    return true;
  }
  template <class... Args>
  static constexpr bool exists(char) {
    return false;
  }
public:
  template <class... Args>
  static constexpr bool check() {
    return exists<Args...>(42);
  }
};
```

Listing 4

Problem: Specializations that sometimes exist and sometimes don't

The question of whether `SomeTemplate<SomeType>` is a complete type (a.k.a. 'the specialization exists') depends on whether the respective definition has been seen or not. Thus, it can differ between translation units, but also within the same translation unit. Consider this case:

```
template<class T> struct SomeStruct;
bool test1 =
  IsSpecialized<SomeStruct>::check<std::string>();
template<> struct SomeStruct<std::string> {};
bool test2 =
  IsSpecialized<SomeStruct>::check<std::string>();
```

What should happen here? What values would we want for `test1` and `test2`? Intuitively, we would want `test1` to be `false`, and `test2` to be `true`. If we try to square this with the `IsSpecialized` template from Listing 4, something weird happens: The same template, `IsSpecialized<SomeStruct>::check<std::string>()`, is instantiated with the same template arguments but should emit a different behavior. Something cannot be right here. If you imagine both tests (once with the desired result `true`, once with desired result `false`) to be spread across different translation units, this has the strong smell of an ODR-violation.

If we try this at Compiler Explorer [CompExp-5], we indeed see that this does not work. So, what's going on here?

The program is actually ill-formed, and there's nothing we can do to change that.

If you use Tmpl<T> at multiple places in your program, you must make sure that any explicit specialization for Tmpl<T> is visible at all those places

The standard states [C++17a]:

> If a template […] is explicitly specialized then that specialization shall be declared before the first use of that specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs; no diagnostic is required. […]

Of course the test for the availability of the specialization would 'cause an implicit instantiation' (which fails and causes SFINAE to kick in).[2] Thus it is *always* ill-formed to have two tests for the presence of a specialization if one of them 'should' succeed and one 'should' fail.

In fact, the standard contains a paragraph, [C++17c] that does not define anything (at least if I read it correctly), but only issues a warning that 'there be dragons' if one has explicit specializations sometimes visible, sometimes invisible. I've not known the standard to be especially poetic, this seems to be the exception:

> The placement of explicit specialization declarations […] can affect whether a program is well-formed according to the relative positioning of the explicit specialization declarations and their points of instantiation in the translation unit as specified above and below. When writing a specialization, be careful about its location; or to make it compile will be such a trial as to kindle its self-immolation.

Thus, as a rule of thumb (not just for testing whether a specialization exists): If you use `Tmpl<T>` at multiple places in your program, you must make sure that any explicit specialization for `Tmpl<T>` is visible at **all** those places.

A generic test for function templates

The move from testing whether *one particular* class template was specialized for a type `T` to having a test for *arbitrary* class templates was pretty easy. Unfortunately it is a lot harder to replicate the same for function templates. This is mainly because we cannot pass around function templates as we can pass class templates as template template parameters.

If we want to have a template similar to `IsSpecialized` from above (let's call it `FunctionSpecExists`), we need a way of encapsulating a function template so that we can pass it to our new `FunctionSpecExists`. On the other hand, we want to keep this 'wrapper' as small as possible, because we will need it at every call site. Thus, building a struct or class is not the way to go.

C++14 generic lambdas provide a neat way of encapsulating a function template. Remember that a lambda expression is of (an unnamed) class type. Thus, we can pass them around as template parameter, like any other type.

Encapsulating the function template we are interested in (`std::swap`, again) in a generic lambda could look like this:

```
auto l = [](auto &lhs, auto &rhs) {
  return std::swap(lhs, rhs); };
```

Now that we have something that is callable if and only if `std::swap<decltype(lhs)>` is available. When I write 'is callable if', this directly hints at what we can use to implement our `FunctionSpecExists` struct – 'is callable' sounds a lot like `std::is_invocable`, right?

So, to test whether `SomeType` can be swapped via `std::swap`, can we just do this?

```
auto l = [](auto &lhs, auto &rhs) {
  return std::swap(lhs, rhs); };
bool has_swap = std::is_invocable_v<decltype(l),
  SomeType &, SomeType &>;
```

Unfortunately, no. [CompExp-6] Assuming that `SomeType` is not swappable, we are getting `no matching call to std::swap` errors. The problem here is that `std::is_invocable` must rely on SFINAE to remove the infeasible `std::swap` implementations (which in this case are *all* implementations). However, SFINAE only works in the elusive 'immediate context' as per paragraph 8 in section 17.8.2 (temp.deduct) of the specification [C++17d] . The unnamed class that the compiler internally creates for the generic lambda looks (simplified) something like this:

```
struct Unnamed {
  template <class T1, class T2>
  auto operator()(T1 &lhs, T2 &rhs) {
    return std::swap(lhs, rhs);
  }
};
```

Here it becomes obvious that plugging in `SomeType` for `T1` and `T2` does not lead to a deduction failure in the 'immediate context' of the function, but actually just makes the body of the `operator()` function ill-formed.

We need the problem (no matching `std::swap`) to kick in in one of the places for which the temp.deduct section of the specification [C17++c] says that types are substituted during template deduction. Quoting from paragraph 7:

> The substitution occurs in all types and expressions that are used in the function type and in template parameter declarations.

One thing that *is* part of the function type is a *trailing return type*, so we can use that. Let's rewrite our lambda to:

```
auto betterL = [](auto &lhs, auto &rhs)
    -> decltype(std::swap(lhs, rhs)) {
  return std::swap(lhs, rhs);
};
```

Now we have a case where, if you were to substitute the non-swappable `SomeType` for the `auto` types, there is an error in the types involved in the function type. And indeed, this actually works, as you can see on Compiler Explorer [CompExp-7] and in Listing 5 (overleaf).

I don't think that you can further encapsulate this into some utility templates to make the calls more compact, so that's just what I will use from now on.

2 This is explicitly stated in [C++17b].

```
auto betterL = [](auto &lhs, auto &rhs)
    -> decltype(std::swap(lhs, rhs)) {
  return std::swap(lhs, rhs);
};
constexpr bool sometype_has_swap =
  std::is_invocable_v<decltype(betterL),
    SomeType &, SomeType &>;
```

Listing 5

What do I mean by 'a specialization exists'?

I wrote at the beginning that it's not entirely clear what 'a specialization exists' should even mean. It is, of course, not possible – neither for class templates, nor for function templates – to check at compile time whether a certain specialization exists *somewhere*, which may be in a different translation unit. I wrote the previous sections with the aim of testing whether the class template (resp. function template) can be 'used' with the given arguments at the point where the test happens.

For **class templates**, I say a 'specialization exists' if, for a given set of template arguments, the resulting type is not just declared, but also defined (i.e., it is a complete type). As an example:

```
template<class T>
struct SomeStruct;

template<>
struct SomeStruct<int> {};
// (Point A) Which specializations "exist"
// at this point?
template<>
struct SomeStruct<std::string> {};
```

In this code, at the marked line, only the specialization for the type `int` 'exists'.

For **function templates**, it's actually a bit more complicated, since C++ has no concept of 'incomplete functions' analogous to 'incomplete types'. Here, I say that a specialization 'exists' if the respective overload has been *declared*. Take this example:

```
template<class T>
void doFoo(T t);

template<class T, class Dummy=
  std::enable_if_t<std::is_integral_v<T>,
  bool> = true>
void doBar(T t);
template<class T, class Dummy=std::is_same_v<T,
  std::string>, bool> = true>
void doBar(T t) {};

// (Point B) Which specializations "exist"
// at this point?
```

At the marked, line:

- For **any** type `T`, the specialization `doFoo<T>` 'exists', because the respective overload has been declared in lines one and two.
- The two specializations `doBar<std::string>` and `doBar<T>` for any integral type `T` 'exist'. Note that this is indenpendent of whether the function has been defined (like `doBar<std::string>`) or merely declared.
- For all non-integral, non-`std::string` types `T`, the specialization `doBar<T>` does 'not exist'.

This of course means that our 'test for an existing specialization' for functions is more of a 'test for an existing overload', and can in fact be used to achieve this. ■

This article was published on Lukas Barth's blog on 1 January 2023 and is available at: https://lukas-barth.net/blog/checking-if-specialized/

A note on MSVC and std::hash

In all my examples, I used GCC and Clang as compilers. This is because my examples for `std::hash` do not work with MSVC [CompExp-8], at least if you enable C++17 (it works in C++14 mode). That is because of this (simplified) `std::hash` implementation in MSVC's STL implementation [Microsoft]:

```
template <class _Kty, bool _Enabled>
struct _Conditionally_enabled_hash
{
  // conditionally enabled hash base
  size_t operator()(const _Kty &_Keyval) const
  {
  return hash<_Kty>::_Do_hash(_Keyval);
  }
};
template <class _Kty>
struct _Conditionally_enabled_hash<_Kty, false>
{
  // conditionally disabled hash base
  _Conditionally_enabled_hash() = delete;
  // *no* operator()!
};

template <class _Kty>
struct hash
  : _Conditionally_enabled_hash
  <_Kty, should_be_enabled_v<_Kty>>
{
  // *no* operator()!
};
```

This implementation is supposed to handle all integral, enumeration and pointer types (which is what `should_be_enabled_v` tests for), but the point is: For **all** other types, this gives you a defined, and thus complete, class – which does not have an `operator()`. I'm not sure why the designers built this this way, but that means that on MSVC, our testing-for-type-completeness does not work to determine whether a type has `std::hash`. You must *also* test whether `operator()` exists!

References

[C++17] The C++17 Standard: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf

[C++17a] Explicit specialization declaration: https://timsong-cpp.github.io/cppwp/n4659/temp.expl.spec#6

[C++17b] Implicit specialization: https://timsong-cpp.github.io/cppwp/n4659/temp.inst#6

[C++17c] Explicit specialization: https://timsong-cpp.github.io/cppwp/n4659/temp.expl.spec#7

[C++17d] Template argument deduction: https://timsong-cpp.github.io/cppwp/n4659/temp.deduct#8

[CompExp-1] Compiler Explorer (1): https://godbolt.org/z/MjTn6Y3Eo

[CompExp-2] Compiler Explorer (2): https://godbolt.org/z/6hv4rcTrc

[CompExp-3] Compiler Explorer (3): https://godbolt.org/z/qhso5vajj

[CompExp-4] Compiler Explorer (4): https://godbolt.org/z/8ocaWMTd9

[CompExp-5] Compiler Explorer (5): https://godbolt.org/z/5xMv6Mh16

[CompExp-6] Compiler Explorer (6): https://godbolt.org/z/jj4PjYG9n

[CompExp-7] Compiler Explorer (7): https://godbolt.org/z/MWjW7WT84

[CompExp-8] Compiler Explorer (8): https://godbolt.org/z/8bMhM5xhq

[Microsoft] MSVC STL implementation: https://github.com/microsoft/STL/blob/214e0143d1d2f7a1c5ca53a338ba3fbb657bdfa3/stl/inc/type_traits#L2177-L2204

Stack Frame Layout On x86-64

Stacks can have different layouts. Eli Bendersky describes the x86-64 layout in detail.

A few months before writing this article, I wrote one called 'Where the top of the stack is on x86' [Bendersky11], which aimed to clear some misunderstandings regarding stack usage on the x86 architecture. The article concluded with a useful diagram presenting the stack frame layout of a typical function call.

In this article, I will examine the stack frame layout of the newer 64-bit version of the x86 architecture, x64.[1] The focus will be on Linux and other OSes following the official System V AMD64 ABI. Windows uses a somewhat different ABI, and I will mention it briefly in the end.

I have no intention of detailing the complete x64 calling convention here. For that, you will literally have to read the whole AMD64 ABI.

Registers galore

x86 has just 8 general-purpose registers available (**`eax`**, **`ebx`**, **`ecx`**, **`edx`**, **`ebp`**, **`esp`**, **`esi`**, **`edi`**). x64 extended them to 64 bits (prefix 'r' instead of 'e') and added another 8 (**`r8`**, **`r9`**, **`r10`**, **`r11`**, **`r12`**, **`r13`**, **`r14`**, **`r15`**). Since some of x86's registers have special implicit meanings and aren't *really* used as general-purpose (most notably **`ebp`** and **`esp`**), the effective increase is even larger than it seems.

There's a reason I'm mentioning this in an article focused on stack frames. The relatively large amount of available registers influenced some important design decisions for the ABI, such as passing many arguments in registers, thus rendering the stack less useful than before.[2]

Argument passing

I'm going to simplify the discussion here on purpose and focus on integer/pointer arguments.[3] According to the ABI, the first 6 integer or pointer arguments to a function are passed in registers. The first is placed in **`rdi`**, the second in **`rsi`**, the third in **`rdx`**, and then **`rcx`**, **`r8`** and **`r9`**. Only the 7th argument and onwards are passed on the stack.

The stack frame

With the above in mind, let's see how the stack frame for the C function in Listing 1 looks. The stack frame is in Figure 1.

So the first 6 arguments are passed via registers. But other than that, this doesn't look very different from what happens on x86[4], except this strange 'red zone'. What is that all about?

```
long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
  long xx = a * b * c * d * e * f * g * h;
  long yy = a + b + c + d + e + f + g + h;
  long zz = utilfunc(xx, yy, xx % yy);
  return zz + 20;
}
```

Listing 1

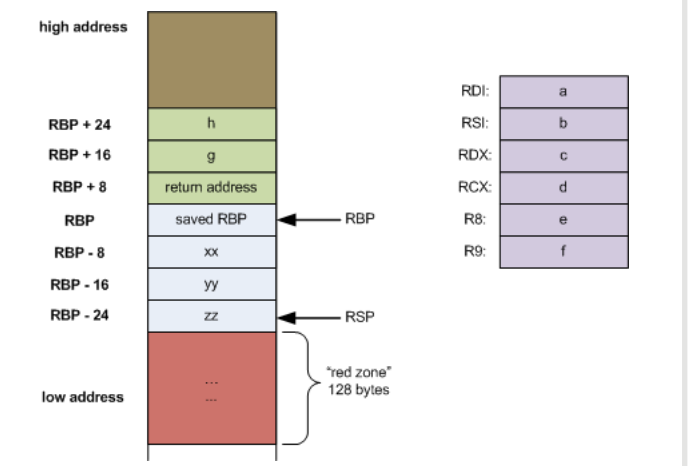

Figure 1

The red zone

First I'll quote the formal definition from the AMD64 ABI:

> The 128-byte area beyond the location pointed to by %rsp is considered to be reserved and shall not be modified by signal or interrupt handlers. Therefore, functions may use this area for temporary data that is not needed across function calls. In particular, leaf functions may use this area for their entire stack frame, rather than adjusting the stack pointer in the prologue and epilogue. This area is known as the red zone.

Put simply, the red zone is an optimization. Code can assume that the 128 bytes below **`rsp`** will not be asynchronously clobbered by signals or interrupt handlers, and thus can use it for scratch data, *without explicitly moving the stack pointer*. The last sentence is where the optimization lays – decrementing **`rsp`** and restoring it are two instructions that can be saved when using the red zone for data.

Eli Bendersky has been programming since the late 1990s. Most recently he's been working on the Go programming language at Google. You can contact him at eliben@gmail.com

1 This architecture goes by many names. Originated by AMD and dubbed AMD64, it was later implemented by Intel, which called it IA-32e, then EM64T and finally Intel 64. It's also called x86-64. But I like the name x64 – it's nice and short

2 There are calling conventions for x86 that also dictate passing some of the arguments in registers. The best known is probably **`fastcall`**. Unfortunately, it's not consistent across platforms.

3 The ABI also defines passing floating-point arguments via the **`xmm`** registers. The idea is pretty much the same as for integers, however, and IMHO including floating-point arguments in the article will needlessly complicate it.

4 I'm cheating a bit here. Any compiler worth its salt (and certainly **`gcc`**) will use registers for local variables as well, especially on x64 where registers are plentiful. But if there are a lot of local variables (or they're large, like arrays or structs), they will go on the stack anyway.

some compilers started omitting the base pointer for aggressive optimizations, thus shortening the function prologue and epilogue

```
long utilfunc(long a, long b, long c)
{
    long xx = a + 2;
    long yy = b + 3;
    long zz = c + 4;
    long sum = xx + yy + zz;

    return xx * yy * zz + sum;
}
```

Listing 2

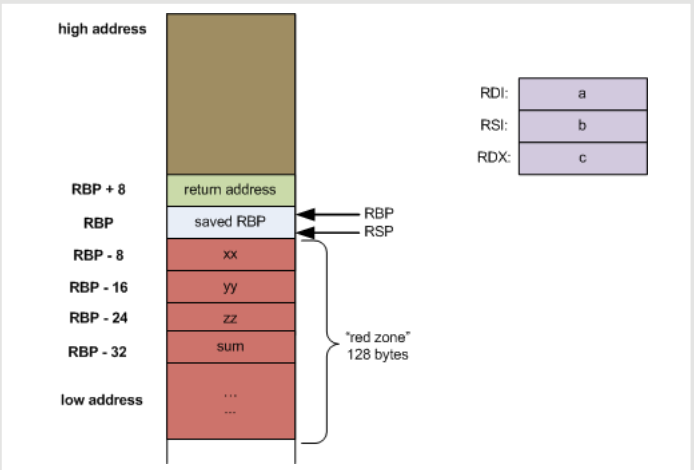

Figure 2

However, keep in mind that the red zone *will* be clobbered by function calls, so it's usually most useful in leaf functions (functions that call no other functions).

Recall how **myfunc** in the code sample above calls another function named **utilfunc**. This was done on purpose, to make **myfunc** non-leaf and thus prevent the compiler from applying the red zone optimization. Looking at the code of **utilfunc** (Listing 2), this is indeed a leaf function. Let's see how its stack frame looks when compiled with gcc (Figure 2).

Since **utilfunc** only has 3 arguments, calling it requires no stack usage since all the arguments fit into registers. In addition, since it's a leaf function, **gcc** chooses to use the red zone for all its local variables. Thus, **rsp** needs not be decremented (and later restored) to allocate space for this data.

Preserving the base pointer

The base pointer **rbp** (and its predecessor **ebp** on x86), being a stable 'anchor' to the beginning of the stack frame throughout the execution of a function, is very convenient for manual assembly coding and for debugging.[5] However, some time ago it was noticed that compiler-generated code doesn't really need it (the compiler can easily keep track of offsets from **rsp**), and the DWARF debugging format provides means (CFI) to access stack frames without the base pointer.

This is why some compilers started omitting the base pointer for aggressive optimizations, thus shortening the function prologue and epilogue, and providing an additional register for general-purpose use (which, recall, is quite useful on x86 with its limited set of GPRs).

gcc keeps the base pointer by default on x86, but allows the optimization with the **-fomit-frame-pointer** compilation flag. How recommended it is to use this flag is a debated issue – you may do some googling if this interests you.

Anyhow, one other 'novelty" the AMD64 ABI introduced is making the base pointer explicitly optional, stating:

> The conventional use of %rbp as a frame pointer for the stack frame may be avoided by using %rsp (the stack pointer) to index into the stack frame. This technique saves two instructions in the prologue and epilogue and makes one additional general-purpose register (%rbp) available.

gcc adheres to this recommendation and by default omits the frame pointer on x64, when compiling with optimizations. It gives an option to preserve it by providing the **-fno-omit-frame-pointer** flag. For clarity's sake, the stack frames showed above were produced without omitting the frame pointer.

The Windows x64 ABI

Windows on x64 implements an ABI of its own, which is somewhat different from the AMD64 ABI. I will only discuss the Windows x64 ABI briefly, mentioning how its stack frame layout differs from AMD64. These are the main differences:

1. Only 4 integer/pointer arguments are passed in registers (**rcx**, **rdx**, **r8**, **r9**).
2. There is no concept of 'red zone' whatsoever. In fact, the ABI explicitly states that the area beyond **rsp** is considered volatile and unsafe to use. The OS, debuggers or interrupt handlers may overwrite this area.
3. Instead, a 'register parameter area'[6] is provided by the caller in each stack frame. When a function is called, the last thing allocated on the stack before the return address is space for at least 4 registers (8 bytes each). This area is available for the callee's use without explicitly allocating it. It's useful for variable argument functions as well as for debugging (providing known locations for parameters, while registers may be reused for other purposes). Although the area was originally conceived for spilling the 4 arguments passed in registers, these days the compiler uses it for other optimization purposes as well (for example, if the function needs less than 32

5 Since inside a function **rbp** always points at the previous stack frame, it forms a kind of linked list of stack frames which the debugger can use to access the execution stack trace at any given time (in core dumps as well).

6 Sometimes also called 'home space'.

bytes of stack space for its local variables, this area may be used without touching `rsp`).

Another important change that was made in the Windows x64 ABI is the cleanup of calling conventions. No more `cdecl`/`stdcall`/`fastcall`/`thiscall`/`register`/`safecall` madness – just a single "x64 calling convention". Cheers to that! ■

Further reading

For more information on this and other aspects of the Windows x64 ABI, here are some good links:

- Official MSDN page on x64 software conventions – well organized information, IMHO easier to follow and understand than the AMD64 ABI document. [MSDN1]
- 'Everything You Need To Know To Start Programming 64-Bit Windows Systems' – MSDN article providing a nice overview. [MSDN2]
- 'The history of calling conventions, part 5: amd64' – an article by the prolific Windows programming evangelist Raymond Chen. [Chen04]
- 'Why does Windows64 use a different calling convention from all other OSes on x86-64?' – an interesting discussion of the question that just begs to be asked. [Stackoverflow]
- 'Challenges of Debugging Optimized x64 code' – focuses on the 'debuggability' (and lack thereof) of compiler-generated x64 code. [Microsoft09]

References

[Bendersky11] Eli Bendersky 'Where the top of the stack is on x86', published 4 February 2011 at https://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/

Unwind Tables

The unwind tables (for both the Itanium ABI and the Windows ABI) ensure exceptions (and debuggers!) are able to unwind the stack and recover the values of some local variables even without the chain of 'base pointer' registers.

The full details are pretty messy and fortunately few programmers need to navigate the tables themselves. For example, see: https://itanium-cxx-abi.github.io/cxx-abi/exceptions.pdf

[Chen04] Raymond Chen 'The history of calling convention...', published 14 January 2004, available at https://devblogs.microsoft.com/oldnewthing/20040114-00/?p=41053

[Microsoft09] 'Challenges of Debugging...' published 9 January 2009, available at https://learn.microsoft.com/en-gb/archive/blogs/ntdebugging/challenges-of-debugging-optimized-x64-code

[MSDN1] 'Overview of x64 ABI convention' published 22 April 2022 at https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions?redirectedfrom=MSDN&view=msvc-170

[MSDN2] 'Everything you need to know…' published 10 July 2019 (from an original published May 2006) at https://learn.microsoft.com/en-us/archive/msdn-magazine/2006/may/x64-starting-out-in-64-bit-windows-systems-with-visual-c

[Stackoverflow] 'Why does Windows64 use a different …' – question asked 13 December 2010, last answer 1 September 2022 at https://stackoverflow.com/questions/4429398/why-does-windows64-use-a-different-calling-convention-from-all-other-oses-on-x86

This article was published on Eli Bendersky's website on 6 September 2011 and is available at: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/#id8

And the winners are...

In *Overload* 172 and *CVu* 34.6, we invited you to vote for your favourite articles both in *Overload* and *CVu* (which is our sister publication for members). The results are in.

1st place

CVu: LOON: Line Oriented Object Notation (Pete Cordell, in *CVu* 34.4)

Overload: Compile-time Wordle in C++20 (Vittorio Romeo in *Overload* 169)

2nd place

CVu: Code Optimization (Part 1 & Part 2) (Pete Goodliffe, in *CVu* 34.3 and 34.4)

Overload: C++20 Benefits: Consistency With Ranges (Andreas Fertig in *Overload* 167)

Thank you to everyone who took the time to vote, and to those who wrote the articles. We can't offer a prize – just the mention here. A number of other writers got a vote, so if you wrote something for us, someone probably thoroughly enjoyed what you had to say.

If you're reading this online, the article titles (or parts, in the case of Code Optimization) link to the articles. *Overload* articles are publicly available, but you must be a member (and logged in) to access the *CVu* ones. If you're not a member yet, why not join?

Value-Oriented Programming

The research Val programming language uses value-oriented programming. Lucian Radu Teodorescu explores this paradigm.

Robert C. Martin argues that we've probably invented all possible programming languages [Martin11]. There are two dimensions we need to analyse programming languages by: syntax and semantic class. In terms of syntax, we've experimented just about everything; we've probably seen all types of syntax we can have.[1] If we look at the semantic class of a programming language – i.e., the paradigm that the language emphasizes – we don't have too many choices.

The approach that Robert C. Martin has on semantic classes is particularly interesting. He argues that a programming paradigm is best characterised by what it removes, not by what it adds; for example, structured programming is the paradigm that removes `goto`s from the language, constraining the direct transfer of control, which leads to programs that are easier to reason about. Imposing constraints on the possible set of programs that can be expressed in a language adds discipline to the language and can improve the language.

In this article, we will build on this idea and show that we haven't yet reached the end of the stack with the things we can remove from languages. We look at *value-oriented programming*, a programming paradigm that the Val programming language [Val] proposes, and how this paradigm can improve safety, local reasoning, and the act of programming.

Programming paradigms and their constraints

Let's start by looking at how Robert C. Martin describes mainstream paradigms in terms of their restrictions.[2]

Before doing that, let us issue a warning to the reader and soften some claims we will make. Whenever we say that a language *restricts the use of a feature*, we don't actually mean that the language completely forbids it; it's just that there is an overall tendency not to use that feature, even if the language allows it (directly or indirectly). In practice, languages don't strictly follow a single paradigm. Moreover, the idea of a programming paradigm is an abstraction that omits plenty of details; we can't have a clear-cut distinction between languages just by looking at their programming paradigms.

Modular programming is a paradigm that imposes constraints on the source code file size. The idea is to disallow/discourage putting all the source code of a program into one single file. One can have the same reasoning applied to the size of functions. Doing this will enable us, the readers of the programs, to understand the code more easily; we don't have to fit the entire codebase in our head, we can focus on smaller parts.

Lucian Radu Teodorescu has a PhD in programming languages and is a Staff Engineer at Garmin. He likes challenges; and understanding the essence of things (if there is one) constitutes the biggest challenge of all. You can contact him at lucteo@lucteo.ro

1 For any syntax, there is also semantics associated with it; Robert C. Martin seems to conflate the two notions and just talks about syntax.

2 Some of the readers might disagree with some of these characterisations; as much as possible, I tried to stick to Marin's exposition of semantic classes (programming paradigms).

Structured programming can be seen as a discipline imposed on direct transfer of control. We don't use `goto` instructions, but rather construct all our programs of sequence, selection, or iterations. This allows us to easily follow the flow of the program, and manually prove the correctness of the code (when applicable).[3] Structured programming also allows (to some degree) local reasoning about the code; this is something of great interest for the purpose of this article.

Object-oriented programming is a paradigm that adds restrictions on the use of pointers to functions and indirect transfer of control. In languages like C, to achieve polymorphic behaviour, one would typically use function pointers. In OOP, one would hide the use of function pointers inside virtual tables, which would be implementation details for class inheritance. This will make polymorphism easier to use, safer (fewer needs of casts) and it also allows us to easily implement dependency inversion. In turn, dependency inversion allows us to have loose coupling between our modules, making the code easier to reason about.

Functional programming can be thought as a paradigm that imposes discipline on assignment of variables. If the assignment is not allowed, all values are immutable, which leads to function purity. Function purity allows easier reasoning about the functions (i.e., function results depend solely on their input arguments), improves safety, allows a set of optimisations that are not typically available with impure functions (removing unneeded function calls, adding memoisation, reordering of function calls, etc.) and allows code to be automatically parallelised.

We can see a pattern here: we restrict the ability to write certain types of programs, we get some guarantees back from the language, and these guarantees can help us write better programs. Removal of features can be beneficial. To make a parallel, this is similar to governments that make laws to prevent certain (unethical) things to happen, but the removal of something that was previously allowed will make the society a better place.

In general, switching from one programming paradigm to another is hard, as we have to change our mental model for reasoning about code; things that are common practice in one paradigm may be restricted in another. Both the strategies and the patterns that we use must change. That is why I prefer the syntagm *programming paradigm* rather than *semantic class* when discussing these categories of programming languages.

Before looking at what restrictions the value-oriented programming paradigm instills, let's first look at the main benefits of its restrictions: safety and local reasoning.

Safety in programming languages

There is a lot of confusion around the word *safety* in the context of programming languages, so I will spend some time to define it for this article. I will follow the line of thought from Sean Parent [Parent22].

3 The ideas of structured programming go beyond this point; structured programming also emphasizes the use of abstractions, decomposition of programs and local reasoning [Dahl72]; but we will ignore these for the purpose of this article.

Ideally, we want all our programs to be strongly safe; but, unfortunately, strong safety is much harder to achieve than simple safety,

All programs consist of operations/commands; we will represent such an operation as `C`. Following Hoare formalism [Hoare69], we can associate preconditions (noted as P) and postconditions (noted as Q) to these operations. Thus, the programs can be represented as sets of triplets of the form $\{P\}\texttt{C}\{Q\}$. For a correct program, in the absence of any errors, for all operations in the program, if the preconditions P hold, then, after executing `C`, the postconditions Q will also hold.

For example, in C++, if `i` is a variable of type `int`, then for the operation `i++` we can say that the precondition is that `i` is not the maximum integer value and that the postcondition is that the new value of `i` is one greater than its initial value.

Let us look at all the possible cases there can be when we execute the operation in $\{P\}\texttt{C}\{Q\}$ (see Table 1).

Possible scenarios when executing operations

Scenario	What should happen
P and Q both hold	the execution is well-formed; the program is correct with respect to this operation
P holds, but Q doesn't hold	`C` needs to report an error, ideally explaining what led to the error; otherwise, the program is buggy
P doesn't hold	`C` can terminate the program (ideal case), it can produce unspecified behaviour, or it can result in undefined behaviour

Table 1

If all the operations in a program are in the first scenario (i.e., both preconditions and postcondition hold) then we say the program is well-formed and correct. Typically, it is impossible for programming languages to guarantee that all the code expressible in the language falls under this scenario.

The second scenario lays the emphasis on correctness in the presence of errors.[4] The program accepts the fact that there might be errors that can lead to failure to satisfy postconditions for all operations, but it's important for these errors to be handled correctly. For example, it may be impossible for a program to make a network call to a server if the network cable is unplugged; if that is correctly treated as an error, the program is correct (assuming that everything else is also treated appropriately). It is outside the scope of this article to delve into what it means to have correct error reporting, but this is not something hard to do (see [Parent22] for details). The bottom line is that error handling is the key to program correctness.

The last case is the one in which we are trying to execute an operation, but the preconditions don't hold. This case is about the safety of the language. The only way in which preconditions may fail is when the program is invalid (i.e. there is a bug); so safety in a programming language concerns its response to invalid programs.

We call an operation *safe* if it cannot lead to undefined behaviour, directly or indirectly. If we have an operation that may corrupt memory, it can lead to crashes while executing the operation, or the program may crash at any later point, or the program may execute the code of any other arbitrary program, or the program may continue to function normally — the behaviour is undefined, so that operation is unsafe.

We call an operation *strongly safe* if the program terminates when executing it if the preconditions of the operation don't hold. That is, programs consisting of strongly safe operations will catch programming failures and report them as fast as they can.

Operations that are *safe* but not *strongly safe* may result in unspecified behaviour. The operation can result in invalid values or can execute infinite loops. For example, the implementation of a square root function that is just *safe* can produce invalid values for negative numbers or can loop forever. Unlike undefined behaviour, though, unspecified behaviour is bounded by the normal rules of the language: no matter what else happens, the same program is still executing when that behaviour completes.

Ideally, we want all our programs to be *strongly safe*; but, unfortunately, *strong safety* is much harder to achieve than simple *safety*, because *safety* is a transitive property, while *strong safety* is not. An operation cannot contain undefined behaviour if it consists of a series of operations that cannot have undefined behaviour, which makes *safety* transitive. On the other hand, we can violate the preconditions of an operation without violating any of the preconditions of the operations it consists of.

The non-transitivity of *strong safety* is illustrated by the C++ code in Listing 1. In this example, all the operations are *safe*, thus calling `floorSqrt` cannot lead to undefined behaviour. We have a precondition that the given argument must be positive. But passing a negative number to this function will not invalidate the preconditions of any operation in the function.[5] All the operations inside the function have their preconditions

```
// Precondition: x >= 0
unsigned int floorSqrt(int x) {
  if (x == 0 || x == 1)
    return x;
  unsigned int i = 1;
  unsigned int result = 1;
  while (result <= x) {
    i++;
    result = i*i;
  }
  return i-1;
}
```

Listing 1

4 Errors are not bugs, just like throwing exceptions is not considered buggy behaviour. A program can be correct, i.e., bug-free, in the presence of errors.

5 In C++, overflows of unsigned values are well defined and use modulo arithmetic.

Locally detectable undefined behaviour can be easily fixed... Globally detectable undefined behaviour is trickier to deal with because it cannot be easily detected.

```
void my_push_back(vector<MyObj>& dest,
    const MyObj& obj) {
  dest.push_back(MyObj{});
  dest.back().copy_name(obj);
}
// caller:
vector<MyObj> vec = generate_my_vector();
my_push_back(vec, vec[0]);
```

Listing 2

hold, but overall, the precondition doesn't hold. In this case, passing a negative value as an argument can led to an infinite loop.

To conclude, *strong safety* is hard to achieve systematically, but we can achieve *safety* systematically. A programming language should aim at allowing only safe programs (unless explicitly overridden by the programmer).

To become *safe*, a language must add restrictions on the operations that can lead to undefined behaviour. Operations that lead to undefined behaviour in C++ are the ones that contain: memory safety violations (spatial and temporal), strict aliasing violations, integer overflows, alignment violations, data races, etc.

Locally and globally detectable undefined behaviour

Undefined behaviour can be of two types, depending on the amount of code we need to inspect in order to detect it:[6]

- *Locally detectable*, if we can detect it by analysing the operation in question, or surrounding code. Examples: integer overflow, alignment violations, null-pointer dereference, etc.
- *Globally detectable*, if we need to analyse the whole program to detect safety issues. Examples: most memory safety violations, data races, etc.

Locally detectable undefined behaviour can be easily fixed in a programming language. The language can insert special code to detect violations and deal with them. Globally detectable undefined behaviour is trickier to deal with because it cannot be easily detected. In an unsafe language, one needs a certain discipline to ensure that this kind of undefined behaviour cannot occur.

Listing 2 shows a C++ function that, at first glance, seems perfectly fine. But, on the caller side, we use the function in a way that will cause undefined behaviour. The **`push_back`** call might need to reallocate memory, move all the objects contained in the vector and might invalidate all the pointers to the original objects; if the given object is part of the original vector memory, then we would be accessing an invalid object.

6 We use the term 'detect' here in a broader context; we don't mean that the compiler or generated runtime code would be required to perform analysis to identify those scenarios (that might lead to the halting problem). If a person looking at the code can identify the potentiality of safety issues, we say that the undefined behaviour is detectable.

```
void remove_by_name(vector<Person>& persons,
                    string_view name) {
  std::erase(persons.begin(), persons.end(),
    name); // C++20
}
// caller
const Person& to_fire =
  worst_performer(employees);
remove_by_name(employees, to_fire.name());
```

Listing 3

This is clearly a bug and it cannot be detected only by looking at the function that has the undefined behaviour. We need to also look at all the possible ways this function is called.

Listing 3 shows another case in which we access invalid memory. Here, in a similar way, we take a **`string_view`** object from an object that is part of our vector. The ownership of the actual string data belongs to the **`Person`** object, which belongs to the vector. But, while utilising the **`string_view`** object we are changing the vector, possibly deleting the underlying object. Invalid memory access is expected.

Listing 4 shows a slightly strange code: we indirectly change the content of a container while traversing the container. While this example may be somehow contrived, one can find it in large code-bases under different forms. We are trying to access an object with some predicate, and because of the complexity of the code we are unaware of the fact that the predicate may actually change the original object.

All these examples have a common problem: looking locally at the code, we are assuming that objects we are operating with have certain properties, without realising that the codebase will indirectly change those properties. We might assume that a reference will point to a valid object, that the object referenced by a constant reference will not change or that the object, if it's in a valid state, will remain in that state. All these assumptions cannot be guaranteed by the compiler, just by looking at the surrounding context. We need to perform analysis on the whole program to spot potential safety issues.

The language is not strong enough to provide us guarantees that would enable us to reason locally. It's similar to how the use of **`void*`** casts and **`goto`**s is discouraged, even though we can write good programs with them – these features require extra discipline to ensure the code is correct.

```
vector<MyObj> objects;
bool my_pred(const MyObj& obj) {
  if (obj.is_invalid() && !objects.empty()
      && objects.front() == obj) {
    objects.erase(objects.begin())
  }
  return obj.can_be_selected();
}
auto it = find_if(objects.begin(), objects.end(),
  my_pred);
```

Listing 4

Local reasoning

The previous examples suggested that maintaining local reasoning is hard, even within the bounds of structured programming, if we have mutation on top of reference semantics. In the presence of references, two objects might be connected in ways that cannot be properly deduced by reasoning locally.

Local reasoning lowers our cognitive burden when writing and analysing code. And, as we know that our mind is the main bottleneck while programming, it is probably making sense to conclude that local reasoning as one of the most important goals in software engineering. Therefore, programming languages should aim at ensuring local reasoning.

We call it *spooky action at a distance* when local reasoning is broken because shared state has been unexpectedly mutated.

Value-oriented programming

In this article, what we call *value-oriented programming* is a programming paradigm in which first-class references are not allowed. The language might *use* references under the covers for efficiency reasons, but these are not exposed as first-class entities to the programmer. In contrast to pure functional programming where first-class references are also not present, value-oriented programming allows mutation.

Forbidding first-class references has the following consequences:

- spooky action at a distance cannot happen anymore
- the *law of exclusivity* is imposed [McCall17], which guarantees exclusivity of access when performing mutation
- aggregation is eliminated; to be replaced by composition.

In this paradigm, emphasis is on the use of value semantics across the language. All types in such a language should behave like `int`; one should see all objects as values, like `int` values.

A first formalisation of this paradigm can be seen in [Racordon22a].

Law of exclusivity and spooky action at a distance

Surprisingly, we can eliminate all undefined behaviour detectable only globally (probably the worst class of safety issues) if we impose just one restriction on the programming language: whenever an object is mutated, the code that does the mutation needs to have exclusive access to the object. This is called the *law of exclusivity* [McCall17]. This law is a fundamental part of value-oriented programming.

This law has two important consequences:

- while reading an object, nobody can change it
- while mutating an object, nobody can read or change it.

If we start looking at an object, and we know that the object is valid, there is nobody else that can invalidate the object while we are looking at it. Mutation of the object would require us to stop looking at the object.

While trying to change the object, we don't affect other code that might look at the same object; there can't be such code under the *law of exclusivity*.

In other words, there cannot be any *spooky action at a distance* [Racordon22a, Abrahams22a, Abrahams22b, Abrahams22c Racordon22b]. Nobody can indirectly change an object while we are looking at it. Eliminating *spooky action at a distance* greatly improves local reasoning; I see this as a considerable improvement on some core ideas from Structured Programming [Hoare69].

Coming back to safety, if we start with a valid object, the only way to break the validity of that object is in the local code (as all the mutation to the object is done locally). That is, if a set of preconditions for an object were true at some point, there is no distant code that can invalidate these preconditions; local code is the only one responsible for the evolution of the validity of those preconditions. And because local memory safety issues can be handled easily by the language, we can construct a safe language. Formally, there are details that need to be discussed to reach this conclusion, but I hope the reader will understand the intuition behind this. For more information, please see [Racordon22a].

We just argued that global (memory) safety issues cannot be present if the *law of exclusivity* is applied. Moreover, the language can prevent local safety issues, either by detecting them during compilation or by adding runtime checks for potential unsafe operations. That means all our operations can be safe. And, as safety composes (as we argued above), we get guarantees that the whole program is safe (i.e., without undefined behaviour). Thus, this enables us to build programming languages that are *safe by default*.

We've just covered the memory safety issues, but we haven't touched threading safety issues. Let's now complete the picture.

Thread safety

Adding threading to C++ applications is a big source of unsafety and frustration. Actually, in her 2021 C++ Now talk [Kazakova21], Anastasia Kazakova presents data showing that in the C++ community, Concurrency safety accounts for 27% of user frustration; this is the highest source of frustration, and it accounts for almost the double of the next source of frustration. Our question is now whether the model we are discussing implies thread safety or not. In other words, can we have data races in this model or not? [7]

In his concurrency talks, Kevlin Henney often presents the diagram reproduced in Figure 1. If we look at whether the data is mutable or not, and if the data is shared or not, we can have 4 possible cases. Out of the 4 quadrants, the one in which the data is both mutable and shared is problematic; if we don't properly add synchronisation, we get thread safety issues. And, almost by definition, synchronisation is something with global effects; we cannot locally reason about it.

In the world of value-oriented programming, where the *law of exclusivity* applies, we cannot be in the synchronisation quadrant. If we have a mutable object, the *law of exclusivity* doesn't allow us to have that object shared – we would be in the top-left quadrant. If we are looking at an object that is shared, then the only possibility is that the object is immutable – the bottom-right quadrant.

Thus, under the *law of exclusivity* we can be in 3 of the 4 quadrants, but not in the synchronisation quadrant. This means that this model doesn't require explicit synchronisation and cannot lead to data race issues. Data races occur when one thread is trying to update a value that another thread is reading; this case is completely forbidden in our model.

Value-oriented programming allows us to be *concurrently safe by default*.

Whole-part relations

Many programming languages use reference semantics as their underlying model. With our model, we move away from reference semantics towards value semantics (more precisely, *Mutable Value*

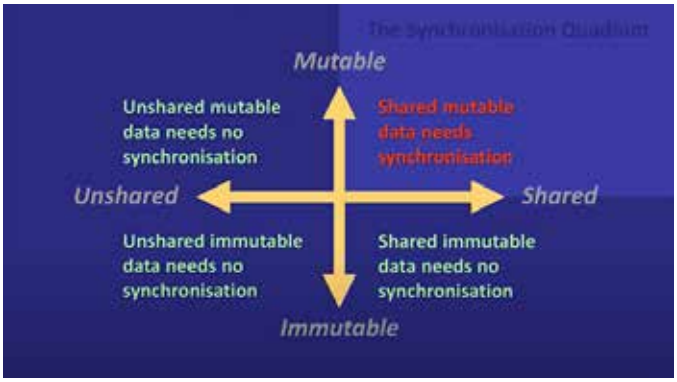

Figure 1

7 Deadlocks and livelocks are not safety issues by our definition, so we won't cover them here.

Semantics [Racordon22a]). We can no longer directly encode arbitrary graphs in data structures, while allowing selective mutation to the nodes of the graphs. To use Sean Parent's words, we are no longer allowed to have incidental data structures [Parent15].

This description may be too dense, so I'll attempt to describe this from a different perspective. In a UML class diagram, one can associate two objects *A* and *B* by using composition or by using aggregation. In reference-oriented languages, they are usually represented by the same code, although semantically they are different.

We want to keep the composition relationship but restrict the use of aggregations. Composition is also called a *whole-part* relationship.

Using only whole-part relationships, all the objects in a program form a forest (set of disjoint trees). Changing one object cannot change an entirely different object (that is not a super- or sub-object of the object we are changing). This means that the impact of changing objects is always local, and we can fully reason about object change locally. I cannot stress enough the importance of this.

Looking at the ownership property, we can infer the memory safety of the program, if all the relationship between objects are compositions. If object *A* needs object *B* to function properly, then it will contain object *B*; but *B* cannot be destructed before *A*, so, *A* will always have a valid instance. If we think about it, this will rule out invalid memory access.

Please note that, for efficiency reasons, implementations of objects may share storage, either by making it immutable or by using copy-on-write to make it immutable-when-shared. But the language behaves *as if* there are no aggregation relationships.

Emphasis on value semantics

Under the *law of exclusivity*, disallowing spooky action at a distance and with objects using only whole-part relationships, all objects behave as values. They are very similar to values of an `Int`.

Reference semantics disappears: one cannot have an object that is changed indirectly by mutation in some other object. Object identity becomes less relevant in the face of object equality; we only care about the *value* of an object.

By this logic, value-oriented programming can give us the same guarantees as functional programming. But, the model is more relaxed than in functional programming. We can mutate variables under the *law of exclusivity*, and thus value-oriented programming is more expressive than functional programming.

Compared to functional programming, our paradigm has several advantages:

- allows expressing some problems more efficiently
- allows expressing some problems in imperative terms, which can be more natural for programmers (i.e., thinking in terms of postconditions and being more mathematical is often considered harder than just thinking in terms of a sequence of actions); using operation sequence we can avoid the mental gymnastics of functional composition.

The reader may consult [O'Neill09] for an example of a problem that, when written in functional languages, is not necessarily efficient nor easy to understand.

Changing our mental model

Like we mentioned at the beginning of the article, a programming paradigm always comes with a change in the mental model when building programs. One simply cannot write and reason about programs in the same way between two different programming paradigms. The main reason is that the programming paradigm restricts the use of certain constructs. And, each time the programmer would want to use those constructs, it needs to take a step back and devise a new strategy that avoids using them.

In value-oriented programming, we cannot share mutable objects. And, not surprisingly, this is widely used in object-oriented programming (and, in general, in many forms of imperative programming).

Let's take an example. Let's imagine that we are compiling a program, and we want to store the program information (syntax tree, type information, and links between nodes) in some kind of a graph. In OOP, many would probably create a `Node` base class, and create a class hierarchy from it. Then, one would create the possibility of child nodes (most likely by having nodes directly linked in other node classes), and the possibility of nodes to reference other nodes (similar to weak pointers). By doing so, the programmer would create what Sean Parent would call an incidental data structure [Parent15].

Value-oriented programming would prevent the user from directly expressing such a structure. Let us try to illustrate how one can model this. The whole program can be modelled by a `Program` class. This class can have ownership of all the nodes we need to create (i.e., use whole-part relations); for example, one can store all the nodes in an array inside this class. The *children* and the *reference* relationships can be built using indices in the entire collection of nodes. We have a completely equivalent data structure, but built using only whole-part relationships. Accessing related nodes from a given node is an operation outside any given node, so nodes cannot mutate other nodes. Mutating one node requires the mutation ability of the entire `Program` object, and we cannot mutate two nodes at the same time. The mutating logic, as it needs to be external to the nodes, can be reasoned locally. Because of local reasoning, the value-oriented model is arguably better compared to its object-oriented alternative.

Let us now take a more complicated example. Let's assume that we want to build a shared cache component. And, to simplify the exposure of the problem, let's also assumed that the cache that can be accessed from multiple threads. By definition, a cache can update its state every time one wants to read something from it. That is, we have a potential mutation for every call made to the cache. And, because this is a shared cache, we need to have shared access to a mutable object. This is forbidden by our model.

There is no way of implementing this program in a pure value-oriented programming style. One needs to get outside the paradigm to implement such a cache.

The bottom line is that we can't simply jump on value-oriented programming and expect our journey to be effortless. We have to adjust our mental model for it.

The Val programming language

Val [Val] is a programming language created by Dimi Racordon and Dave Abrahams that is probably the first programming language that is focused, at its core, on call value-oriented programming. Val aims to be *fast by definition*, *safe by default*, *simple* and *interoperable with C++*.

Val is based on Swift. One can argue that Swift made the first steps towards value-oriented programming; Val takes some core ideas from Swift further and cleans up the semantics to fully resonate with value-oriented programming.

Swift encourages using value semantics [Abrahams15], but it doesn't go all the way through to remove reference semantics. While structures uphold value semantics, classes in Swift follow reference semantics; closures (with mutable captures) also follow reference semantics.

Val doesn't have structures and classes as Swift does; in Val there is one way of defining structures and that follows value semantics. From this point of view, one can argue that Val is simpler than Swift.

Swift aims at being a safe language (i.e., remove the presence of undefined behaviour); it does that by adding runtime checks when generating code. Val inherits the safety principle from Swift but does this in a better way. Because of the guarantees of value-oriented programming, Val has more guarantees about the preconditions of the operations, so it can eliminate some of the runtime checks.

For some people, Val somehow appears as a successor language to C++; see also my previous article [Teodorescu22] (although not necessarily positioned this way by language authors). Val's efficiency aim and the goal to be interoperable with C++ put it in that space. But, looking at the main programming paradigms in the two languages, the two languages seem to operate in different spaces.

There appears to be a big difference between Val and other languages that target to be C++ language successors (i.e., Carbon or Cpp2). Val is the only language that proposes a paradigm shift. The other languages will operate in the same paradigm as C++; i.e., in the words of Robert C. Martin, they only propose syntactic changes. With those changes alone, it's hard to get additional guarantees from the language, and therefore it's hard to full fix the safety of the language and to ensure local reasoning.

Comparing Val to Rust, they both seem to fix the safety issues. That is because Rust's borrow checker is also compatible with the *law of exclusivity*. But there is a big difference on how the same results are achieved. In Rust one uses reference semantics and manually annotate objects to express lifetime guarantees.

In Val the programmer cannot use reference semantics. Aggregation is forbidden, and whole-part relationships are used to express connections between objects. To recognize different ways of handling objects, Val has four parameter passing conventions (let, inout, sink and set) [Abrahams22d]. In Val all copies are explicit.

There are cases in which the programmer wants to exit the bounds of value-oriented programming. One trades safety guarantees with expressive power. Going outside of safety guarantees of the language is not a bad thing, and it doesn't mean that the code is unsafe; it just means that the programmer takes full ownership of guaranteeing safety. Similar to Rust's `unsafe` construct, Val aims to provide a mechanism that allows programmers to exit the bounds of *value-oriented programming* [Evans20].

Conclusions

This article explored value-oriented programming, a new programming paradigm proposed by Dimi Racordon and Dave Abrahams with the creation of the Val programming language.

In value-oriented programming, first-class references are forbidden; everything operates under the *law of exclusivity*, which allows exclusive access to an object while mutating it. What is called *spooky action at a distance* (i.e., indirect mutation) is, consequently, also forbidden. In this model, all the relationship between objects are whole-part. In other words, we ban aggregation in the favour of composition.

We impose restrictions on programming languages to gain some guarantees. In our case, the restrictions imposed by value-oriented programming will allow us to improve on local reasoning, and to avoid a big class on safety issues (the rest of the safety issues being simpler to solve). As cognitive power is the main bottleneck in software engineering, improving on local reasoning may have a big impact on software engineering.

Val programming language is the first programming language that has value-oriented programming (as we defined it) at its core. It is an experiment that allows us to explore the boundaries of this paradigm.

Will the Val language succeed? Will the value-oriented programming be highly used? We don't know; and at this point, I don't think it matters that much. We've found a new programming paradigm, and I feel that it's our moral duty to explore this new programming paradigm. Only after exploring it, we can decide to completely drop it, or to build all future programming languages based on it. ■

Acknowledgements

The author would like to thank Dave Abrahams and Dimi Racordon for reviewing the article and providing numerous suggestions to improve it.

References

[Abrahams15] Dave Abrahams, Protocol-Oriented Programming in Swift, WWDC15, 2015, https://www.youtube.com/watch?v=p3zo4ptMBiQ

[Abrahams22a] Dave Abrahams, A Future of Value Semantics and Generic Programming (part 1), C++ Now 2022, https://www.youtube.com/watch?v=4Ri8bly-dJs

[Abrahams22b] Dave Abrahams, Dimi Racordon, A Future of Value Semantics and Generic Programming (part 2), C++ Now 2022, https://www.youtube.com/watch?v=GsxYnEAZoNI&list=WL

[Abrahams22c] Dave Abrahams, 'Values: Safety, Regularity, Independence, and the Future of Programming', *CppCon 2022*, https://www.youtube.com/watch?v=QthAU-t3PQ4

[Abrahams22d] Dave Abrahams, Sean Parent, Dimi Racordon, David Sankel, 'P2676: The Val Object Model', https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2676r0.pdf

[Dahl72] O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Structured Programming, Academic Press Ltd., 1972

[Evans20] Ana Nora Evans, Bradford Campbell, Mary Lou Soffa, Is Rust used safely by software developers?, 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), 2020

[Henney17] Kevlin Henney, Thinking Outside the Synchronisation Quadrant, ACCU 2017 conference, 2017, https://www.youtube.com/watch?v=UJrmee7o68A

[Hoare69] C. A. R. Hoare, An axiomatic basis for computer programming. Communications of the ACM. 12 (10): 576–580, 1969.

[Kazakova21] Anastasia Kazakova, Code Analysis++, CppNow, 2021, https://www.youtube.com/watch?v=qUmG61aQyQE

[Martin11] Robert C. Martin, The Last Programming Language, 2011, https://www.youtube.com/watch?v=P2yr-3F6PQo

[McCall17] John McCall, Swift ownership manifesto, 2017. https://github.com/apple/swift/blob/main/docs/OwnershipManifesto.md

[O'Neill09] Melissa E. O'Neill, The genuine sieve of Eratosthenes, Journal of Functional Programming 19.1, 2009

[Parent15] Sean Parent, Better Code: Data Structures, CppCon 2015, https://www.youtube.com/watch?v=sWgDk-o-6ZE

[Parent22] Sean Parent, Exceptions the Other Way Around, C++Now 2022, https://www.youtube.com/watch?v=mkkaAWNE-Ig

[Racordon22a] Dimi Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, Brennan Saeta, Implementation Strategies for Mutable Value Semantics, https://www.jot.fm/issues/issue_2022_02/article2.pdf

[Racordon22b] Dimi Racordon, Val Wants To Be Your Friend: The design of a safe, fast, and simple programming language, CppCon 2022, https://www.youtube.com/watch?v=ELeZAKCN4tY&list=WL

[Teodorescu22] Lucian Radu Teodorescu, The Year of C++ Successor Languages, Overload 172, December 2022, https://accu.org/journals/overload/30/172/overload172.pdf#page=10

[Val] The Val Programming Language, https://www.val-lang.dev/

Afterwood

Meetings come and go. Chris Oldwood asks us to meet him halfway.

I'm only a few days back into work at the dawn of a new year and I've already chalked up a couple of meetings. This particular one is the overrun for the planning meeting we held yesterday, and it looks like we'll need at least one more to finish things off. My colleague made a nervous remark about how long it's taken to cover all the topics we had on the list, and so I made light of the situation by pointing out that I've spent longer than this in sprint planning meetings that are only looking at the next two weeks. In our case, we're trying to look ahead at the next 12 months, so I'm hardly surprised we have plenty to discuss, not least because most of the people are new to the team so there's a plethora of history and context to fill in.

After spending the last three years filling the void between Development and Operations, acting more like a service desk, I'm quite pleased to slow the pace down and have the opportunity to sink my teeth into something meatier. Not long before Christmas, my diary only contained two meetings a week, and one of them was only held on alternate weeks which probably sounds like nirvana to some. The change in teams has brought with it a couple of extra regular entries in the diary for catch-ups but hardly anything arduous. Some kind of 'daily stand-up' has been a part of my professional routine for so many years now it felt really odd when it wasn't there.

The employees at Shopify started the new year with what many see as a belated Christmas present – all regular meetings of more than two people were cancelled, along with a cooling off period before they can be re-added. The premise, according to their CEO, is that many meetings "are a bug" and the underlying cause should be fixed instead. In his tweet, he cited trust, clarity, and missing APIs as common 'root' causes. The article I read suggested they were also cancelling all meetings on a Wednesday (but didn't say if that was to allow more time for sports activities like at school), while Thursday was reserved for any large meetings. In essence ,they want to put more time back in the hands of people to get on with actually building stuff.

When it comes to hot takes about meetings my personal favourite is this tweet from some years ago:

> Meetings are complete distraction from coding. Without them, I could work without interruption on solving the wrong problems. ~ @raganwald

To be sure, it was a flippant remark, but there's more than a grain of truth in there. I've been witness to at least two occasions where someone has gone off on a tangent for a couple of weeks and churned out a large body of code that was either far in excess of what was needed to solve the problem, or created more problems than we already had. Unlike a suggestion in a meeting which can often be reasoned about and either improved upon or dismissed in a fast timeframe, reverting a whole ton of somebody's code after they've sweated over it for hours is a much tougher prospect as their loss is likely to be so much greater. In both cases I had the unenviable job of breaking the bad news to them that we needed to revert their efforts as there was considerably more technical debt than credit.

In some way I'm not surprised this happens. Back in the late noughties, Daniel Pink famously ascribed autonomy, mastery, and purpose towards what drives people to be motivated. The aspect of autonomy can lead us to take on far more than we are equipped to handle by ourselves. Like free-speech, it doesn't mean you get to do what you want without consequences – you still need to be accountable for your actions. The intent was to avoid micro-managing people, but not at the cost of removing collaboration altogether. Instead, autonomy puts the power to collaborate back in the hands of the producer so they have more control over when and where it happens. However, if you're going to adopt a more 'trust, but verify' approach you don't want to leave it too long before doing some 'verifying', lest the waste starts to accumulate and course corrections become ever more costly. Ideally, you want a balance between giving out creative freedom whilst also ensuring they have access to the kinds of people they will want to draw upon because they provide insights in a positive manner. The best kind of people often go out of their way to have their work and opinions challenged as they treat failure as an opportunity for learning. ('Egoless Programming' was first coined by Gerry Weinberg way back in the early 1970s in *The Psychology of Computer Programming*.)

My take on the drastic actions by Shopify is not to stifle collaboration, as that has to happen somewhere, but to try and eliminate some waste by avoiding unproductive meetings or removing unnecessary people from the equation. For me, an unproductive meeting though is often a smell that there is some confusion that needs resolving, which I guess is what the CEO of Shopify is getting at, but I'd rather do that quickly face-to-face than have to experience the tortuous route of a protracted email exchange or pull request with a number of comments to rival *War & Peace*. If you're in Rumsfeld territory facing unknown unknowns you'll want to navigate out of there as quickly as possible to at least the safer harbour of known unknowns.

I remember a project manager getting tetchy once because we went around in circles in a few design meetings apparently arguing about a particular concept. He saw it as disruptive, but I saw the conflict as a sign that we were missing something fundamental, and we were, we just didn't discover what it was until an iteration or two later. When it did finally emerge all the confusion made sense, along with the realisation that we needed a much broader view of the business than many of our existing users had had up to that point.

As I write we're just about to pass the third anniversary of the first Covid lockdown in the UK, along with the sudden switch to long-term remote working for many businesses. The consequential rise in asynchronous communication via the written word and video recordings will have been liberating for some employees more used to a classic 9 to 5 schedule, rebalancing the work-life scales for them in the process. But meetings can, and should have a social element too, even if just for a minute or two here and there, and so I hope we don't throw the proverbial baby out with the bathwater in the stampede to deprecate face-to-face contact. ■

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology from ~~plush corporate offices~~ the comfort of his breakfast bar. He has resumed commentating on the Godmanchester duck race but continues to be easily distracted by messages to gort@cix.co.uk or @chrisoldwood

ACCU 2023

Conference 19 – 22 April 2023
Pre-conference workshops 17 & 18 April 2023

REGISTRATION NOW OPEN! Visit https://www.accuconference.org/

Follow @ACCUConf
Tweet #ACCUConf