overload 138

APRIL 2017 £3

Single Module Builds: The Fastest Heresy in Town

C/C++ "unity builds" are controversial. But they can make a significant difference to project build times.

(Not Really So) New Niche for C++: Browser!?

Runing C++ in a browser with Emscripten

Space Invaders in Elm

An overview of Elm: a functional language which compiles to JavaScript

Contractual Loopholes

How to stop compilers optimising away functions

All About the Base

Representing numbers presents many choices

A magazine of ACCU

ISSN: 1354-3172

JET BRAINS

A Power Language Needs Power Tools

We at JetBrains have spent the last decade and a half helping developers code better faster, with intelligent products like IntelliJ IDEA, ReSharper and YouTrack. Finally, you too have a C++ development tool that you deserve:

- Rely on safe C++ code refactorings to have all usages updated throughout the whole code base
- Generate functions and constructors instantly
- Improve code quality with on-the-fly code analysis and quick-fixes

ReSharper C++

Visual Studio Extension for C++ developers

CLion

Cross-platform IDE for C and C++ developers

AppCode

IDE for iOS and OS X development

Find a C++ tool for you
jb.gg/cpp-accu

OVERLOAD 138

April 2017

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael@accu.fi

Klitos Kyriacou
klitos.kyriacou@gmail.com

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Matthew Wilson
stlsoft@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 139 should be submitted by 1st May 2017 and those for Overload 140 by 1st July 2017.

Overload is a publication of the ACCU

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

4 Space Invaders in Elm
Ossi Hanhinen provides an overview of Elm.

7 Single Module Builds – The Fastest Heresy in Town
Andy Thomason shows how much difference unity builds can make to build times.

10 An Interview: Emyr Williams
Frances Buontempo interviews the *CVu* interviewer: Emyr Williams

12 (Not Really So) New Niche for C++: Browser!?
Sergey Ignatchenko demonstrates how to use Emscripten.

16 Contractual Loopholes
Deák Ferenc explores ways to stop compilers optimising away functions.

20 All About the Base
Teedy Deigh counts the ways to represent numbers.

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 Trade Marks

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from Overload without written permission from the copyright holder.

Breadth First, Depth First, Test First

You can approach a problem top-down or bottom-up. Frances Buontempo wonders if algorithms can help us choose the most appropriate direction.

Trying to decide what to write an editorial on is a continuing struggle. Should I do a deep dive into a subject I have been learning or thinking about recently? Should I cast my eyes over a wide panorama of topics that are on the horizon? This indecision has led to writer's block. A similar problem often happens when bug-hunting or developing code. Should you follow one path and get a complete feature working or have an overall shape or walking skeleton [Cockburn] first when writing code? Should you choose London-style versus Detroit-style, also referred to as outside-in, top-down or mockist style versus inside-out, traditional style test driven development [StackExchange]? It appears, as ever, to depend. There is no one true way, though Steve Freeman says on a Google groups discussion about the GOOS book "Do the stuff that you need to learn most about first" [Freeman]. Of course, some people choose a third option and don't test first, possibly don't even have tests at all and the really fool-hardy don't use version control. What about bug-hunting? Recently 'Cloudbleed' surfaced, a memory leak which appeared to corrupt some HTTP requests sent through Cloudflare [Cloudbleed]. They managed to track where the problem was very quickly and turn off the features, which stopped the problem across the board, but then required a deep-dive follow-up to fix the problem and clean up any cached sensitive data. Because they had services they could swap out quickly, it took under seven hours to deploy a full fix. This did require a team of people round the globe with their heads down checking the problem stopped when the services were stopped and that the fix did actually work. The approach of 'breadth-first', to stop things running across the board, then 'depth-first' deep dive to find the problem (a case of `==` instead of `<=` in some C code) succeeded. Spending time finding the bug while the problem continued would have been a mistake. This was supplemented with some fuzz testing from an InfoSec team to find similar problems in similar areas of their code. I am reminded of several demos I have seen finding Heartbleed in a very short period of time by using fuzzers [for example, Boeck]. Fuzzers are a form of 'random testing' [Fuzzing], often using genetic algorithms, to generate input cases which cause memory leaks or similar. They can be purely black-box or combined with some form of instrumentation to seek out problems. I frequently wonder how many bugs could be fettered out by trying fuzz testing on a code base. Are any readers regularly using these tools?

If you find yourself in a high-pressure situation, such as hunting a production bug, possibly with the added stress of it being pointed out on a late-night phone-call, it can be hard to keep your wits about you, and pull back from a depth-first plunge down a rabbit-hole of a call-stack guided by clues in a log file. Many technical people will follow one line of enquiry to its logical end, because a task-switch to look at something else feels like an interruption. It is important to build up an instinct of what is likely to be waste of time though, and perhaps jotting down a note, either in the form of your bug tracking system, a hand-written TODO list or just a post-it note, might be enough to pick up where you left off if you spot something else that might be related and feels more likely to be fruitful. If you don't find a way to keep track of what you have already visited, you might find yourself going round in circles. Another thing to try is time-boxing. Give yourself thirty minutes debugging, and not a moment more, just to get a feel for what might be going on. Don't allow yourself to have 'just five more minutes'. Stop. Step away from the keyboard and think. If you can't fix it, it might be best to just turn off some features, if you can do that, and get some sleep before continuing your investigation. A lack of sleep will mean you are not on top form. It can be more heroic, or at least sensible, to stop than plough on regardless. It might be worth adding more logging to get better clues, and in fact Cloudflare did add extra logging as part of their investigation. Having a good logging system that allows you to explore and aggregate logs easily is also a good thing. Having a bird's eye view with the possibility of a swoop down to dig in the depths if required is ideal.

In a calmer world, where you are not fire-fighting a production bug, but creating a new program the breadth-first or depth-first dichotomy still matters. I have previously been amused as I coded with colleagues who would either complete one feature first, not paying enough attention to how it might fit into the big picture or who would skip around from part of one thing to part of another and leave lots of functions marked 'Not implemented' or 'Todo'. It was often worse if I worked by myself, with no-one to hand to say, 'Hang on a minute'. I have got more disciplined at jotting a note on a list of what needs investigating next, allowing me to finish my current item, or at least writing a new test, which fails, to remind me of something distracting that will need doing, just not now. It is important to keep track of what you are doing and to learn when something does matter, but not at the moment, since it is a distraction from your current focus. Trying not to interrupt yourself is an important skill to learn. Just because you have thought of something doesn't mean you should always do it immediately. Conversely, there will be times where it will take as long to raise a Jira ticket, or similar, than it will to just do it on the spot. As ever, it depends.

The observation of breadth-first versus depth-first, of course, stems from two main approaches to iterating through a tree structure. Situations in life are often more like cyclic-graphs than trees, or even tangled string, so the analogy will not apply in all situations. Sometimes the only solution to a knotty problem is lateral thinking, or a giant sword which legend has Alexander the Great used to 'untie' the Gordian knot. Straining the

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

analogy, a breadth-first approach tends to use more storage, and I feel that my brain fills up as I walk across all the possible approaches and things that could be explored or discussed up front. I would rather park something under 'Any other business' and talk about it at the next meeting, or leave a function to be implemented later, or a test to get to pass. Later. I know I am easily distracted though. If you do choose a depth-first approach, you still have options. Should you adopt pre-order, in-order, or post-order? At this point I have taken the analogy too far now, I'm sure. The three approaches will enumerate the tree's content in different orders, so it will depend what you are trying to achieve if this is a pure algorithm question. Not all trees are binary. Not all trees are balanced. Not everything is black and red. Sometimes life is too short to conduct an exhaustive search anyway. Recently, computers playing Go have employed Monte-Carlo search trees [MCST]. This combines heuristics – guiding suggestions – with random sampling to explore more promising looking paths through the search space. It seems that there are times when it is better to randomly try something than to spend time enumerating all the possible approaches before getting things done. Brute force will only work if you have the space and time to enumerate all the options. This might not even be possible.

Besides impossibilities, choices and forks in the road can freeze us. If there is no obvious advantage in one path over another, how do you decide what to do? I watched some friend on social media discussing whether Blocky or Scratch is better for using to teach children to program. I suspect it will be hard to decide which is better. You often also see people asking which programming language they should learn. Sometimes you should just try something. If you really can't decide, toss a coin. It might be that your circumstances make one thing easier than another. If you have a friend who already knows a programming language and has tools set up for it, try that. To quantify which of a set of options is better requires a suitable metric. It can be worth spending time figuring this out, but it might not be. Try something, test it out and hold on to what is good.

Something similar can happen in mathematics. Kevlin Henney talked about Pythagoras' theorem at NorDevCon [NORDEVCON] in February. He observed there are several different proofs of this theorem, though was only taught one at school. I don't recall being shown a proof at school, though have subsequently read about some. Inspired by Kevlin, I have found a website [cut-the-knot] which gives 118 proofs. This may not be exhaustive. A reference to the Gordian knot again, however this is not my point. I have observed several mathematics lessons attempting to provide the pupils with some exemplars of right-angled and non-right angled triangles and encourage them to discover Pythagoras' theorem. This is a frustrating and boring thing to be subjected to, in my opinion. Furthermore, pupils with a mathematical bent are likely to correctly think these are just one or two examples and it proves nothing. There is nothing wrong with exploring one or two examples up front, to investigate the problem, but this does not prove in general what is going on. The mathematical test at this point is a compelling proof; some form of deduction or a formal proof by induction, rather than the equivalent of 'It works on my bit of paper'. If you employ test-driven development, are you just giving one or two examples in the hope that this proves your software works by extrapolation? In general, no. Some problems are more appropriately tested with properties, rather than one or two specific examples. Moving in a different direction, many people have written about why they don't accept the call to use test-first or TDD [for example, Reddit]. This might not mean no tests that can be run by machines, of course, however, many people do find some form of TDD useful for a variety of reasons. Matteo Vaccari wrote a blog called 'TDD is no substitute for knowing what you are doing' [Vaccari]. You are unlikely to discover Pythagoras' theorem by trying a few arithmetic combinations of the lengths of the sides of a variety of triangles. Vaccari says,

> it is not satisfying to use the tests in TDD as a crutch for constructing haphazard code that, with a kick here and a few hammer blows there, seems to work. The point of TDD is to **design** code; and a good design shows how and why a solution works… TDD does not work well when we don't know what we're doing.

He talks though Peter Norvig's approach to writing a Sudoku solver [Norvig], observing there are two main approaches; depth-first and constraint based. His main point is you might still need to think first before diving in and writing some tests. You might need to learn some data structures and algorithms first. Alternatively you could explore the extent of the problem, then stop and revise or learn specific algorithms you need. Knowing some basics is a good thing. There are, however, many times where a random walk, in one form or another can be useful. Many financial pricing and risk models use stochastic calculus, or random stuff if you will, to produce useful results. Furthermore, a fuzzer is doing random stuff to explore the search space. A fuzzer using genetic algorithms is using randomness in conjunction with a fitness function. It is guided by randomness (and fitness) rather than fooled. There are many choices of paths through a problem. Each will have advantages and disadvantages, though having a good fitness function or clear goals can stop you wasting a lot of time. Keep track of where you've already explored, as many tree and graph algorithms do. Possibly try to find a solution, however suboptimal, then incrementally improve it, as many flow path algorithms do [for example an augmented path: Weisstein] Being slightly meta, being aware of the approach you are taking to problem solving is both interesting and can suggest alternatives. For example, getting round to writing an editorial. One day.

References

[Boeck] https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html

[Cloudbleed] https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/

[Cockburn] http://alistair.cockburn.us/Walking+skeleton

[cut-the-knot] http://www.cut-the-knot.org/pythagoras/

[Freeman] https://groups.google.com/forum/#!topic/growing-object-oriented-software/GNS8bQ93yOo

[Fuzzing] https://en.wikipedia.org/wiki/Fuzzing

[MCST] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

[NORDEVCON] http://www.nordevcon.com/nordevcon2017/

[Norvig] http://norvig.com/sudoku.html

[Reddit] https://www.reddit.com/r/programming/comments/kq001/testdriven_development_youve_gotta_be_kidding_me/

[StackExchange] http://softwareengineering.stackexchange.com/questions/166409/tdd-outside-in-vs-inside-out

[Vaccari] http://matteo.vaccari.name/blog/archives/416

[Weissstein] Weisstein, Eric W. 'Augmenting Path' From MathWorld – A Wolfram Web Resource http://mathworld.wolfram.com/AugmentingPath.html

Space invaders in Elm

Elm is a functional language which compiles to JavaScript. Ossi Hanhinen provides an overview.

I first learned about Elm in May 2015. I fell in love. I also wrote an article describing how to get started with the language by implementing the base for a game. Its title, 'Learning FP the hard way' [Hanhinen15], was supposed to be a joke of sorts, as Elm is in fact a very easy language to learn! I'm not sure how many people got that. ☺

Since then, I have used Elm in two separate customer projects, and it has definitely made my work better!

The recent update (0.17) meant a rather large shift in the way the language works, so I decided to revisit the original subject. So here it is: the base for a Space Invaders game in Elm 0.17!

You can find all of the code on GitHub, along with some setup instructions. [SpaceInvaders]

Elm, what is it again?

Elm is a "delightful language for reliable webapps." [Elm]. It is a functional language which compiles to JavaScript. You can explore it online at http://elm-lang.org/try. If you're interested in an overview, I gave a talk about the language, called 'Confidence in the frontend with Elm' at GeeCON 2016. [Hanhinen16]

Modeling the problem

First off, let's re-iterate what we want to achieve.

From the player's perspective the program should be like this:

- There is a ship representing the player near the bottom of the screen
- The player can move the ship left and right with corresponding arrow buttons
- The player can shoot with the space bar

And from the ship's perspective the same is:

- Ship has a position on a 1D axis
- Ship can have a velocity (positive or negative)
  - Ship changes position according to its velocity
- Ship can shoot

This gives us a definition of what the **`Model`** of our little program should look like:

```
type alias Model =
  { position : Float
  , velocity : Float
  , shotsFired : Int
  }
```

Ossi Hanhinen Ossi is building apps for browsers to run and users to enjoy at Futurice. He likes to constantly challenge his views on user interface programming, and has gravitated towards functional programming and strong typing with the Elm language. Ossi has started two customer projects using Elm, making Futurice one of the first commercial users of it. Contact him at ossi.hanhinen@futurice.com

This is an example of a data structure called Record. It is like a strongly typed and immutable cousin of the JavaScript object. Now, we have only defined the type of the data so far, so let's create a model to start from:

```
model : Model -- this is a type annotation
model =
  { position = 0
  , velocity = 0
  , shotsFired = 0
  }
```

What we have here is a simple value, or a constant. As everything in Elm is immutable, **`model`** will always be the same no matter what happens in the app. If we tried to redefine it, the compiler would simply complain that there are multiple definitions for the same name and the code would not compile.

Alright, moving on to moving the ship! I remember from high school that $s = v*dt$, or moved distance is the product of the velocity and the time difference. So that's how we can update the ship. In Elm, that would be something like the following.

```
applyPhysics : Time -> Model -> Model
applyPhysics dt model =
  { model | position =
    model.position +   (model.velocity * dt) }
```

The above is the way to update a record. We start off with the **`model`** as the base, but update the **`position`** as per the formula. Note that **`{ record | x = newX }`** creates a new record, as everything in Elm is immutable. We will never have to worry about affecting anyone else's state by accident. Even better, we can be certain no one else is affecting our state either!

The type annotation on **`applyPhysics`** says: given a **`Float`** and a **`Model`**, I will return a **`Model`**, but also: given a **`Float`**, I will return **`Model -> Model`**. For example, **`(applyPhysics 16.7)`** would actually return a working function to which we can pass a **`Model`**, and get the physics-applied ship as the return value. This property is called Currying and all Elm functions automatically behave this way. Currying is very useful in many cases, but that is a topic for another article.

We can update the other properties in the very same way (see Listing 1).

Using these little functions we can update all of our state, but we're missing something quite necessary: 1. View of the current state, and 2. getting input from the user and turning that into updates.

```
updateVelocity : Float -> Model -> Model
updateVelocity newVelocity model =
    { model | velocity = newVelocity }

incrementShotsFired : Model -> Model
incrementShotsFired model =
  { model | shotsFired = model.shotsFired + 1 }
```

Listing 1

Elm 0.17 brought a new way of reacting to changes: Subscriptions

Showing the state

Our game wouldn't be much use if it couldn't show the current state in some way. To keep things as simple as possible, let's just print the model as text. We can do it like so:

```
view : Model -> Html msg
view model =
  text (toString model)
```

Here, **`toString`** turns the **`model`** record into a readable String representation of it, and **`text`** from the Html package turns a String into an HTML text node. Pretty handy! Now the strange part here might be the return type of our **`view`** function: **`Html msg`**. We don't need to worry about it too much right now, but what the type annotation is saying is in essence: "I am returning some HTML, which may produce messages of the **`msg`** variety."

This will do for now, so let's move on to the interactive part!

Subscribing to user input

Elm 0.17 brought a new way of reacting to changes: Subscriptions. What we will do is this: we will subscribe to certain changes in the world, and when they happen, give the changes some names. We want to control the game by keyboard, so let's start by taking a look at the Keyboard package. It seems we want to listen for both pressing down on buttons, and letting go of them. With these, we can determine when the user is pressing down on a certain key. We will need something else as well: to keep updating the position of our ship, we need to have a somewhat steady rhythm of **`applyPhysics`** with the time difference! That we can get using the **`AnimationFrame.diffs`**. Bundling that up into code works like this, defining the messages in our program:

```
type Msg
  = TimeUpdate Time
  | KeyDown KeyCode
  | KeyUp KeyCode
```

Here we have a union type. For something to be considered a **`Msg`** in this module, it will have to be one of the above (**`TimeUpdate`**, **`KeyDown`** or **`KeyUp`**). Furthermore, the contents of e.g. **`TimeUpdate`** must be something that can be considered **`Time`**, and so on.

Okay, now let's declare the subscriptions we need, and name them with our newly-defined message types.

```
subscriptions : Model -> Sub Msg
subscriptions model =
  Sub.batch
    [ AnimationFrame.diffs TimeUpdate
    , Keyboard.downs KeyDown
    , Keyboard.ups KeyUp
    ]
```

This again will just return a subscription, or a **`Sub Msg`**. It doesn't do anything on its own, but we need it for the actual wiring part of our code (see Listing 2).

```
main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }
```

Listing 2

If you are really paying attention, you might notice that we have **`view`** and **`subscriptions`** done by now, but both **`init`** and **`update`** are still missing. Luckily we already have all the building blocks, so taking this home shouldn't be too much of a stretch anymore! In fact, **`init`** is so simple that we should get it out of the way right now.

```
init : ( Model, Cmd Msg )
init =
  ( model, Cmd.none )
```

That's all there is to it! Again, we can leave the **`Cmd`** stuff for later, but just as a primer: commands are the only way to have effects in an Elm program. What are effects? They are anything that can affect the world outside the app (such as posting something to the Internet), or whose value can vary between program runs (such as the current time, or random numbers). Here we don't need to do any commands, so we define it to be **`none`**.

Putting it all together: the update

All right, now's the time to make it all work!

Let's begin from the high level. The **`update`** function takes the incoming message and the old model, and returns the updated model along with possible commands. In this case we won't need any commands, but we still need to fulfil the contract with **`none`**s (see Listing 3).

Note that each of the possible **`Msg`** options is handled. If they weren't, the Elm compiler would catch the problem, which is pretty cool and impressive. Anyway, the **`TimeUpdate`** is nice and easy. We can simply use the **`applyPhysics`** function to get the updated model. For the

```
update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
  case msg of
    TimeUpdate dt ->
      ( applyPhysics dt model, Cmd.none )

    KeyDown keyCode ->
      ( keyDown keyCode model, Cmd.none )

    KeyUp keyCode ->
      ( keyUp keyCode model, Cmd.none )
```

Listing 3

When it comes to the packages, Elm 0.17 is still a bit of a work in progress

```
keyDown : KeyCode -> Model -> Model
keyDown keyCode model =
  case Key.fromCode keyCode of
    Space ->
      incrementShotsFired model

    ArrowLeft ->
      updateVelocity -1.0 model

    ArrowRight ->
      updateVelocity 1.0 model

    _ ->
      model
```

Listing 4

keypressing cases, I decided to split the handling into their own functions as well.

When it comes to the packages, Elm 0.17 is still a bit of a work in progress. So to make the keyboard handling a little nicer, I made a tiny helper module. There is a function that can turn a `KeyCode` into a `Key`, which is a simple union type. It only has the keys we need for this exercise now, but could easily be extended (see Listing 4).

The above should be pretty clear. Spacebar shoots once as soon as it is pressed down and doesn't do anything else. The arrow keys set the velocity of the ship when pressed down. Notice that we need an "otherwise" case, customarily denoted as _. This is because there are many other possible keys on the keyboard besides the ones we've covered.

How about the release part? See Listing 5...

```
keyUp : KeyCode -> Model -> Model
keyUp keyCode model =
  case Key.fromCode keyCode of
    ArrowLeft ->
      updateVelocity 0 model

    ArrowRight ->
      updateVelocity 0 model

    _ ->
      model
```

Listing 5

If the released key happened to be one of the movement keys, reset the velocity to 0, otherwise let's just keep the current model. Pretty straightforward, right?

Now it should work! ■

References

[Elm] http://elm-lang.org/

[Hanhinen15] Learning FP the hard way: Experiences on the Elm language https://gist.github.com/ohanhi/0d3d83cf3f0d7bbea9db#learning-fp-the-hard-way-experiences-on-the-elm-language

[Hanhinen16] https://speakerdeck.com/ohanhi/confidence-in-the-frontend-with-elm-1

[SpaceInvaders] https://github.com/ohanhi/elm-game-base

Kindly republished from Ossi's blog:
http://ohanhi.github.io/base-for-game-elm-017.html

Courses:

Moving Up to Modern C++
An Introduction to C++11/14/17 for experienced C++ developers. Written by Leor Zolman.
3-day, 4-day and 5-day formats.

Effective C++
A 4-day "Best Practices" course written by Scott Meyers, based on his Legacy C++ book series. Updated by Leor Zolman with Modern C++ facilities.

An Effective Introduction to the STL
In-the-trenches indoctrination to the Standard Template Library. 4 days, intensive lab exercises, updated for Modern C++.

Live on-site C++ Training
by Leor Zolman

Mention ACCU and receive the U.S. training rate for any location in Europe!

www.bdsoft.com • bdsoftcontact@gmail.com • +1.978.664.4178

Single Module Builds – The Fastest Heresy in Town

Unity builds can be controversial. Andy Thomason shows how much difference they can make to build times.

We have been building our C++ projects pretty much the same way since the early 80s and maybe it is time to change. Back then, no one could imagine the need for more than 640k of RAM and so C and C++ modules needed to be small enough to fit in memory. In the C world, we would build modules with a single C function or a group of related functions, carefully keeping the limit to under a thousand lines or so as any more than this would cause the memory to page, causing the build time to shoot up. Linking, too, was a problem as file systems were quite slow and so we needed to build libraries which were archives of object files to minimise link times.

To understand the build process in depth let us look at the layers that we must go through to make an executable file.

Preprocessor	Reads include files, expands macros.
Lex/Parse	Generates tokens for uninstantiated functions
C++ Semantics	Template expansion, class instantiation.
Intermediate Representation (IR)	Platform independent assembly-like pseudo language for high level optimisation.
Code Generation (CG)	Platform specific pseudo language for low level optimisation.
Object files/Static libraries	Platform specific binary code plus debug information (Dwarf/PDB)
Executable/ Dynamic library	Linker-generated code ready to run.

This is pretty much the same stack as the original C compilers, with the exception of the C++ semantics. Most of the compile time comes from the fact that `#include` will typically read hundreds of thousands of extra lines per module, no matter how careful you are with the includes. In addition to this, the full code generation path is executed for many functions defined in header files but only one version of the binary will make it to the executable. Worse than that this is the amount of work needed to generate debug information for every class included, with multiple versions of debug information for each instantiated template class.

Traditional C++ builds

In most C++ projects declarations are made in `.h` files with classes defined separately using `::` in `.cpp` or `.cc` files. Some functions are inline in the class, but usually only short ones. The reason for this is largely historical, and some developers prefer not to have function definitions in classes because it clutters the simplicity of their class definitions. However, as we shall see this comes at a very high price in terms of build time and code generation (see Listing 1).

Unity builds

In the games industry, many large engines use 'unity builds'. A unity build reduces the number of modules in a compilation and has a significant impact on build time. They work by constructing `.cpp` files that look like Listing 2.

```
toaster.cpp:

#include "bread.h"
#include "toaster.h"

bread &toaster::eject() {
}

bread.cpp:

#include "bread.h"
bread::bread(flour &f, water &w) {
}
```

Listing 1

```
my_unity_build_1.cpp:

#include "renderer.cpp"
#include "ui_elements.cpp"
#include "gameplay_code.cpp"
#include "character_AI.cpp"

my_unity_build_2.cpp:

#include "file_io.cpp"
#include "cat_dynamics.cpp"
#include "wobbly_bits.cpp"
#include "death_ray.cpp"

...
```

Listing 2

The source code in the `.cpp` files still looks the same with the exception that it must be 'clean', ie. no static or global variables and no anonymous namespaces. Also it is important to avoid `using namespace` globally as this can cause some problems.

The result is a build that contains fewer modules. Because every module includes pretty much the same header files, despite your best efforts, then it turns out that it takes about the same time to compile a module regardless of its complexity. Fewer modules mean less duplication of debug information and common functions, less code generation, more optimisation opportunities as compilers can inline any function in the module.

Unity builds made a significant difference to large game projects, which often have more than ten thousand source files. But the logical assumption

Andy Thomason Andy worked for Sony Computer Entertainment on the Playstation compilers. He now teaches game programming to aspiring developers and runs a consultancy analysing scientific data. Contact Andy at a.thomason@gold.ac.uk

link time tends to dominate for projects with thousands of modules

is that the larger the modules, the slower the build would be for an individual change to a file. In practice, in many cases, this turns out not to be true because the link time tends to dominate for projects with thousands of modules and will easily exceed the time taken to compile a single module, thus the incremental compile time is also improved by unity builds.

I demonstrated a unity build of Clang at LLVM 2014 that reduced the compile time from over an hour to twenty seconds and incremental builds to five seconds, but there was fierce resistance to this concept despite the 200× speedup. I had to make over a thousand edits to the codebase to make it build and Clang makes extensive use of static variables and anonymous namespaces, as well as having some namespace name clashes.

Beyond unity builds: single module compilation

One thing that C++ did for us was to allow us to define small functions in the class definition itself. As compilers got better, a new kind of library emerged: the header-only C++ library. Many of the libraries in Boost, for example, are header only and this has the huge advantage that we do not need to build and distribute a binary of the library and so can run the code on any platform with a modern compiler.

JavaScript is also an example of header-only code. When we load a web page, we compile all the JavaScript in a single module – there is no concept of linking in JavaScript, and yet this works well and is accepted practice.

A single module is the ultimate unity build with only one module to build, no linking and source-only library distribution. Advantages are very fast build times in most circumstances, clearer code and very fast link and rebuild times in builds dominated by links. Disadvantages include the effort to re-write libraries to be header only, a 'brick wall' response when compilers consume all available DRAM and start paging to SSD and a potentially worse response to very complex template metaprogramming.

How about circular references? In single module builds, if class A refers to class B and vice versa, it is still necessary to separate declarations from definitions as in traditional builds. This is because C++ processes classes in the file scope in the order they are seen in the file. To do this we generally use `.inl` files, which use the `::` operator and the `inline` keyword. Using the `inline` keyword means the defined functions are created in `linkonce` sections rather that the regular sections and as a result then the library can be used in multiple modules. The key is to declare the classes from leaf to root so that forward references are minimal. Another method is to use only template classes or to declare all classes inside a `struct` which defers the evaluation of the function definitions. It would be possible to change the language to allow classes to be declared in any order, a small change now as compilers keep everything in memory.

So why not build C++ programs as a single module? We know that the build time is very good and the code generation is close to optimal. The answer currently is that there are very few header-only libraries and there is no infrastructure of header-only libraries that we can call on to build our code. There is no packaging mechanism for header-only libraries and finding them on GitHub for example is something of a lottery.

Using the Unity build method helps to bridge the gap somewhat, for example Bullet Physics, TinyXML, LUA (see table) and other libraries can be converted to header-only form without too much effort, but the effort to 'clean' the builds and remove static variables can be daunting.

But if you can build your code in a single module, the performance of the build and the generated code can improve by several orders of magnitude.

In an ideal world, compilers would be multithreaded, but even Clang, which is fairly modern, is stubbornly single threaded and the design is not likely to accommodate multithreaded compilation. This means that there is a lower limit on module compile time that will be with us for some time.

Include vs import

Java and C# have both adopted the header-only style as the designers realised that separate declarations and definitions were unnecessary. Java and C# also dropped the `#include` mechanism in favour of an 'import' mechanism. C++ is acquiring an import mechanism of its own and we hope that it will improve build times when it makes it into mainstream compilers. There have been many proposals, however, and all are different.

Daveed Vandevoorde's original 2006 proposed paper is here:
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2073.pdf

Daveed is VP of engineering at Edison Design Group, whose front end we used for the Playstation 3 and Vita compilers. Many well-know compilers are based on EDG. Incidentally, EDG is a well-structured C library and builds as a single module unity build in less than a second.

Microsoft have a module mechanism in VS2015:
https://kennykerr.ca/2015/12/03/getting-started-with-modules-in-c/

```
#include <stdio.h>
module dog;
export namespace dog {
  void woof() {
    printf("woof!\n");
  }
}
…
import dog;
```

Clang also is working up to module support:
https://clang.llvm.org/docs/Modules.html

This involves a module map which maps headers to modules.

```
module std [system] [extern_c] {
  module assert {
    textual header "assert.h"
    header "bits/assert-decls.h"
    export *
  }
}
…
import std.assert;
```

We can see that single module builds ... can improve rebuild times by up to three orders of magnitude

Results are an average of three runs

Number of classes	Traditional rebuild time	Traditional build time (single change)	Traditional build exe size (bytes)	Single module build/ rebuild time	Speedup	Single module exe size (bytes)
10	6.12s	1.93s	12800	2.14s	2.87x	11776
100	41.12s	2.36s	14336	2.06s	19.92x	11776
1000	401.66s	4.77s	30208	2.87s	139.88x	11776

Table 1

Whilst modules improve the structure of C++ programmes, will they improve build times? Much will depend on the implementation.

Synthcode

To help demonstrate the benefits of single module compilation, I've created a simple python script in a project called **synthcode**. (https://github.com/andy-thomason/synthcode)

This script generates a synthetic C++ project with a variable number of classes. It offers the choice of a single module build (one file per class) or a traditional multi-module build (two files per class) so that we can benchmark the two against each other.

The generated classes aggregate other classes and call a single function on each aggregated member. A more sophisticated script could generate more complex behaviour but even this simple method is quite revealing of build and code performance. Table 1 shows the results on Windows using using `cmake -G "Ninja"` (Ninja is a high-performance build tool using multiple threads.)

Note that the traditional build time scales roughly linearly with project size but the build time for a single module change also goes up due to link times. On GCC builds the link time can increase to an hour or more on debug builds thanks to the DWARF information replicated in all the object files.

Libraries that can be built header-only

Some libraries that can be built header-only even though not designed to do so.

- bulletphysics.org/

  Bullet is Erwin Coumans' excellent physics library. As a multi-module build, it takes about a minute to build; as a single module build, it takes around a second.
- https://github.com/leethomason/tinyxml2

  TinyXML is a lovely little XML parser that is widely used. As it has very few files, a single module build does not make it much faster, but it does make it portable as no binary is required.
- https://www.lua.org/

  LUA is a small script language that is widely used in the game industry. It is written in C, but with a little hacking it will run as a header-only library.

The Single module build time grows much more slowly regardless of the number of classes compiled although in a real scenario thousands of big and complex classes could slow it down to a few tens of seconds.

We are also not using template metaprogramming which performs functional style programming in the compiler. Every operation of template metaprogramming can consume millions of cycles in the compiler and each template expansion can create large amounts of debug information. This can be very useful, however, if properly controlled.

The module compile time is largely dependent on the number of lines pulled into the project by #include. In this case I am including the following with each class which contributes about 110,000 lines to the build:

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <cstdint>
#include <future>
```

Code size, an indication of optimisation, in the single module builds is consistently low but in the multi module builds it is high. In GCC builds this can lead to multi-gigabyte executable sizes and hour-long link times.

Conclusion

We can see that single module builds not only can improve rebuilds of C++ projects but can improve rebuild times by up to three orders of magnitude. Of course it is not likely that it will be widely adopted as most projects are legacy projects which cannot be updated and C++ literature abounds with illustrations of traditional programme layouts. However, for new projects it would be wise to consider it as an option.

Another very useful side effect of fast compile times is that we could follow the javascript route and distribute code solely as source. A three second compile time for a thousand class application is less that the loading time for even a modest JavaScript application. C++ scripting in web pages would be a very nice thing with libclang embedded in the browser. More palatable than asm.js in many ways.

Heresy? Undoubtedly. New ideas, especially pragmatic ones, take time to become mainstream. 100 times build speed improvements are a compelling argument however. ■

An Interview: Emyr Williams

CVu has been running a series of interviews. Frances Buontempo interviews the interviewer, Emyr Williams.

First, introduce yourself.

I'm Emyr, I've been a professional developer for eight years or so, I've coded in Java, Python, C++ and I've been an ACCU member for the last four years. I have a keen interest in all things space related, I'm an avid reader, and my favourite Science Fiction book is *The Martian*.

When did you join ACCU?

I joined the ACCU in 2014, I think.

Why?

I attended the ACCU Conference for the first time in 2013, and met some awesome people there, who were passionate not just about C++, but about software development in general. And they were always willing to answer some inane question a C++ newbie had for them. I wanted to be part of such an organisation.

Has joining the ACCU been worth it?

Yes, without question.

Can you expand on that? What do you get out of it that makes it worth £45 a year?

Well, you get a discount on the ACCU Conference, which is always worth it, you also get two great publications in *CVu* and *Overload*, but more than that, it's the sense of you belonging to a body of like-minded people who are passionate about being professional software developers. And it's the community spirit that I enjoy as well. It's a close knit group despite having people all over the world. I've never felt I can't approach someone with a question I have.

What was the first thing you wrote for us?

The first thing I wrote was an interview with Bjarne Stroustrup, who I met at the ACCU conference in 2013, I happened to bump in to him and I rather cheekily asked if he'd be willing to be interviewed for my blog, which ended up working out slightly different.

How did getting something published feel?

Quite cool actually, if a little weird. I've not been in a publication before.

What tech talks have you given?

I've given some talks at work on moving to C++ 11, and I've given a couple of lightning talks at the main conference. I'm not confident enough yet to give a full technical talk, but I am building up to that.

Do you conduct interviews face to face or electronically?

I mostly interview via e-mail, the exception was the interview I did with Scott Meyers, I was fully intending on doing the interview via e-mail, but Scott suggested a face to face interview as I was on a training course he was running in London, which was pretty awesome if not a little scary. Especially as it's someone I look up to and have read most of his books. I admit I was a bit star struck, but it was an awesome experience.

What inspired you to interview people for *CVu*?

I originally intended to interview people for my blog, which chronicles how I'm trying to become a better programmer. I'd just managed to get Bjarne Stroustrup to agree to be interviewed, and I was telling Pete Goodliffe about it. He promptly encouraged me to talk to Steve about writing for *CVu*, and that's how that started.

You mention 'Becoming Better'. Is that an oblique reference to Pete's writings? I dimly remember you giving a lightning talk about this. In fact, I found this link: https://www.slideshare.net/welshboy2008/becoming-better-a-two-year-journey Why this name for your blog?

The first time I was at the ACCU main Conference, Pete Goodliffe hosted a talk called 'Become a better programmer', which had a panel of people, I can't recall the names of the panellists now, but it challenged me to improve as a programmer as I'd been on autopilot, not really pushing myself. And I thought I'd start blogging about it. I checked with Pete that he was ok with me calling my blog Becoming Better, and he was great and said he was ok with it. He's also challenged me on some stuff over the years, and two years later, he asked me to do a lightning talk to show how I'd got better over the year, so it was pretty cool.

In what ways have you changed?

What's changed in me? Well, I'm more keen to try other languages, whereas I only wanted to do C++. So I've learned to code in Javascript, Python and Java, even if they weren't my first choices. For example, I found that coding in Javascript helped me to understand what the practical use for a lambda function was. I've learned that my mind works best when given real world examples of something, rather than abstract examples. I also read a lot more than I used to now, and I try to make sure that I have time in my day to read, whether it be the current book I'm reading, or a blog post. I've also learned to widen my skill set as well, for example the last few weeks I've been doing mainly sysadmin tasks at work, in which I learned to use Ansible and Bamboo.

How do you think of the questions?

I often look up the people I'm interviewing so I don't ask pre-canned questions. There are some generic ones of course, but the questions I asked Scott Myers for example, would be very different to the questions I asked Kate Gregory. I think you need to show that you've done some research, otherwise it shows a lack of effort on your part.

How did the questions to Scott Meyers differ from those for Kate Gregory?

I'm not sure I can give a satisfactory answer to that question if I'm honest. I basically base my questions on stuff I've read from Scott

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

I mainly ask on the ACCU Mailing list for volunteers to give talks. I'm also keen on getting more technically diverse speakers as well

or Kate, and their respective backgrounds. I also have a group of colleagues who are also ACCU members and I ask them "What would ask X if you could do so...?"

How do you find people?

It's usually through a blog post I've read, or someone I look up to, and sometimes it's "Wow, wouldn't be cool if I could interview them?" sort of thing. I've also had people suggest folk to interview as well.

If a reader wants you to interview them for *CVu* what should they do? Or thinks of someone for you to interview?

Feel free to ping me an e-mail (egwilliams2002@googlemail.com), or ping me on twitter (@welshboy2008). I'm usually around

Why did you volunteer to run the Bristol group?

Ha ha, well I don't think I stepped back fast enough, I think Nigel had a hand in that. I knew Ewan was stepping down after doing an amazing job, then Nigel suggested I take it over. I wasn't too sure, but I thought I'd give it a go. Ewan was graceful enough to let me see how I got on by arranging a few evenings and work as a double act, a sort of long handover if you will.

How do you find speakers?

I'm not sure I've found the best way for that yet, at the moment, I mainly ask on the ACCU Mailing list[1] for volunteers to give talks. I'm also keen on getting more technically diverse speakers as well, so for example I'm hoping a friend of mine who runs a small ISP in rural Scotland will be able to come and give a talk on setting up an ISP but nothing's confirmed yet.

When will you give a longer talk? What will it be about?

I'm not sure at the moment. I tried to do a technical talk a year or so ago at a local ACCU group which didn't go very well. Mainly due to a lack of preparation on my part, or rather rushed preparation. And it shot my confidence a bit. I do get nervous when I give technical talks because I overly worry about people's reactions. I will do a technical talk at some point, just not sure when at the moment.

If you could summarise the benefit you've got from being a ACCU member in a sentence, what would you say?

It's a great community of like-minded people from all walks of life, from all corners of the globe who are passionate about professionalism in software, and you'll be made to feel welcome. ■

1. https://accu.org/index.php/mailinglists

Join the ACCU

- Reduced member-rate for the annual conference
- Member-only discussion lists
- Study groups
- Book reviews
- Become part of a great community, meeting developers at local groups
- Printed copy of *CVu* and *Overload* (depending on membership package) delivered to your door
- Online access to back issues of both *CVu* and *Overload*

visit www.accu.org for details

(Not *Really* So) New Niche for C++: Browser!?

How do you run C++ in a browser? Sergey Ignatchenko demonstrates how to use Emscripten.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Hare, and do not necessarily coincide with the opinions of the translators and *Overload* editors; also, please keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented an exact translation. In addition, the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

[In chess annotation,] '!?'... usually indicates that the move leads to exciting or wild play but that the objective evaluation of the move is unclear
~ Wikipedia

For quite a long while, C++ had been losing popularity; for example, as reported in [Widman16], in 2016 it got 7% less of the listings on Dice.com compared with a year earlier; and according to [TIOBE17], from the C++ Golden Age in 2004 till 2017, the C++ share fell from ~17% to a measly 6%.

As all of us (as in, 'hardcore C++ fans') know </tongue-in-cheek>, this has nothing to do with the deficiencies of C++; rather it is related to an observation that the time of downloadable clients (which was one of the main C++ strongholds) has changed into the time of browser-based clients – and all the attempts to get C++ onto browsers were sooo ugly (ActiveX, anyone?) that this didn't really leave a chance to use C++ there.

Well, it *seems* that this tendency is already in the process of being reverted:

> *C++ can already run on all four major browsers – and moreover, it has several all-important advantages over JavaScript, too.*

And this – not too surprisingly – is what this article is all about.

A word of warning: please do NOT expect any revelations here; this article is admittedly long overdue – and quite a few people know MUCH more than I can fit here (and MUCH more than know myself). Still, given the lack of such overviews intended for those of us who haven't tried it yet, I am sure that such an article has its merits. In the article, I will try to provide a very high-level overview of *Emscripten*, of the technologies involved, of the performance which can be expected, of the APIs which can be used – and what we can gain from using it.

JavaScript to the rescue!

Attempts to get C++ on browsers were continuing all the time (such as (P)NaCl), but all of them were platform- (and/or browser-)specific, and (as a result) were very problematic for browser deployments. However, help for the C++ side of things has come from exactly the same rival which has been stealing the browser show for all these years – from JavaScript. It wasn't easy, and took several all-important (and IMO ingenious) pieces of the puzzle to make it useful.

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko and Dmytro Ivanchykhin using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He currently holds the position of Security Researcher and writes for a software blog (http://ithare.com). Sergey can be contacted at sergey@ignatchenko.com

```
function Vb(d) {
  d = d | 0;
  var e = 0, f = 0, h = 0, j = 0, k = 0, l = 0,
    m = 0, n = 0, o = 0, p = 0, q = 0, r = 0, s =
0;
  e = i;
  i = i + 12 | 0;
  f = e | 0;
  h = d + 12 | 0;
  j = c[h >> 2] | 0;
  if ((j | 0) > 0) {
    c[h >> 2] = 0;
    k = 0
  } else {
    k = j
  }
  j = d + 24 | 0;
  if ((c[j >> 2] | 0) > 0) {
    c[j >> 2] = 0
  }
     ...
}
```

Listing 1

Piece I – asm.js

In 2013, so-called *asm.js* was released. Essentially, *asm.js* is just a very small subset of JavaScript, intended to simulate good old assembler. If we take a look at a real-world *asm.js* program (not hand-written, but compiled from C++), we'll see something along the lines of Listing 1 [Resig13].

As we can see, it is nothing like your usual high-level JavaScript, which deals with DOM and high-level `onclick` handlers. Instead (except from the `if` statements and function declarations) it directly translates into what we'd usually expect from an assembler language.

On taking a closer look, we can observe the following elements of more-or-less typical assembler in the code above:

- registers (implemented as JavaScript `var`s)
- ALU operations (via JavaScript doubles, but converting them into `uint32_t` all the time via `| 0`)
- Ability to access memory (as one huge array; in the example above – `c[]`)
- Control operations (`if` and `function`)

Well, that's pretty much *all* we need to get the full-scale assembler rolling.☺

For our current purposes, we don't really want to go any deeper, but hopefully I've managed to describe the idea behind *asm.js*: essentially, it is pretty much a simulator of *a strange CPU* with *a strange instruction set*. In other words, *asm.js* did NOT try to simulate any existing instruction sets (and doing so would make it fatally inefficient).

> *Instead, asm.js has invented* ***its own instruction set****, which can be still seen as an instruction set of a CPU,* ***at least from the point of view of a C++ compiler****.*

Piece II – LLVM/Emscripten

The above observation has made it possible to write a back-end for the LLVM compiler, and this back-end has allowed the generation of *asm.js* out of our usual C++ (some restrictions apply, batteries not included). Moreover, such a compiler is not only possible, but it exists and is working: it is *Emscripten*.[1]

Actually, the *asm.js* in the example above has been generated by *Emscripten*. Using *Emscripten* is indeed rather simple:[2] we just take our existing standard-compliant and not-using-platform-specific-stuff C++ code (hey, you DO write your code as cross-platform and standard-compliant, don't you?</wink>), and compile it into *asm.js*. As long as our code is just 'moving bits around', it works near-perfectly (and what will happen when we need to interact with the rest of the world, we'll discuss in the 'APIs' section below), producing *asm.js* code which looks similar to the example above.

Piece III – optimizations for asm.js

When looking at all the stuff above, a very natural scepticism goes along the lines of "Ok, this compiled piece of [CENSORED] stuff MAY work correctly, but how slow it is going to be???" And here is the point where the third piece of the C++-to-*asm.js* puzzle comes in. I'm speaking about *asm.js*-specific optimizations.

The thing is that with *asm.js* being *this* simple and restricted, it becomes possible to optimize it during a JIT compile. That's it – we *can* have our cake (write in C++) and eat it (run it in *asm.js* with a reasonable speed) too!

As of now, all the four major browsers (in alphabetical order: Chrome, Edge, Firefox, and Safari[3]) – at least *try* to optimize for *asm.js*. Results vary, but currently, most of the time, we're speaking about a less than 2× performance degradation of *asm.js* compared to native C++ (say, compiled with Clang) [Zakai14]. While comparisons with native C++ are difficult to find (which BTW does make me to raise an eyebrow), the few resources available *seem* to support this claim (see, for example, [AreWeFastYet17]). BTW, Firefox results listed by the link are of special interest – in fact, *it manages to keep the performance of asm.js within a mere 20% of the 'native' performance* – and while we cannot *rely* on such performance (hey, we don't want to be *restricted* only to Firefox users), it still serves as an indication of what it is possible to achieve (well, if enough effort is spent on it).

BTW, one important property of *asm.js* is that

> *As asm.js is a strict subset of JavaScript – it will run even if there is no special support for asm.js in browser.*

Sure, without special support *asm.js* will be pretty slow – but if we're speaking about 'glue code', it still *may* fly even with *asm.js* support being unavailable/disabled.

Restrictions

While *Emscripten* provides a full-scale and very usable environment, there are certain limitations due to the need to run from within browser. When you're ready to go ahead with *Emscripten*, make sure to read [Emscripten.Porting]; the following is only a very short summary of the *Emscripten* restrictions and capabilities.

APIs

The most annoying restriction of *Emscripten* is (arguably) related to the provided APIs. First of all, we can use pretty much all the C++ standard libraries which don't need to interact with the system – and that's including STL (phew). `boost::` libraries are not explicitly supported, but there are reports that some of them can be compiled too (not without some associated headaches); most of the header-only `boost::` libraries are *expected* to work with *Emscripten* 'out of the box' (no warranties of any kind, batteries not included).

As noted above, libraries which interact with the rest of the world are a different story. Contrastingly, in general, all the stuff which we'd need to use on the client is present in the APIs; in particular, the following APIs are supported:

- Network support (`libc`-style, non-blocking only(!))
- File system access
- Graphics (OpenGL ES – though it is better to restrict yourself to WebGL-friendly subset, as I've heard that emulation of the rest kinda suxx)
- Audio, keyboard, mouse, joystick (SDL)
- Integration with HTML5 (DOM, some of the events – including device orientation, touch, gamepad, etc.)

Threads and main loop

Due to the *Emscripten* runtime being run on a top of the JS engine, threading in *Emscripten* is quite limited from the point of view of a C++ developer.

First of all:

> *Unless we're speaking about 'Workers', everything within our app happens within a single 'browser main loop'*

In practice, this means a few things:

- Our app MUST adhere to the 'event processing' model (i.e. if our code blocks for a while, it means that the whole page is blocked).
  - APIs are built in a way to help us with this; in particular, network access being non-blocking only, is a Good Thing™ from this perspective.
- If we have our own infinite loop (event processing loop, game loop, simulation loop, etc.), we'll need to break it and re-implement it on top of the browser main loop. It is NOT *as bad* as it sounds – see [Emscripten.BrowserMainLoop] for details
- Handling replies to asynchronous calls (such as replies to our requests which are coming from the server-side) can be a headache. For an overview of non-blocking handling techniques in C++ (though *not* taking *Emscripten*-specifics into account), see [NoBugs16] and [NoBugs17].

Personally, I do NOT think that this is really restrictive; in other words, I am arguing to write the code in such an event-driven manner (which I like to name '(Re)Actor-style') in any case, even when there is no *Emscripten* in sight. Very briefly – considering I have been arguing that having thread sync at app-level is evil for years now (see [NoBugs10] and [NoBugs15]) – going for a bunch of event-driven (Re)Actors exchanging messages is a Good Thing™.

Using multiple cores

While I am all for event-driven single-threaded processing, I am the first one to admit that there are situations when one single thread (and as a result, a single CPU core) is not sufficient to do whatever we need to do. Which means that we do need a way to use multiple cores.

However, being able to use multiple cores, DOES NOT necessarily imply the need to go into traditional mutex- and atomics-ridden untestable nightmare. Rather, we can have more than one separate event processor

1. There are alternative compilers (formerly Mandreel, now cheerp) which compile C++ not into *asm.js*, but into other subtypes of compliant JavaScript; we'll see in a jiff why compiling into *asm.js* is so important.
2. After the usual jumping through the hoops to get stuff installed
3. Well, actually – WebKit

a.k.a. (Re)Actors (in *Emscripten*-speak, additional (Re)Actors – that is, beyond the original one running within the 'browser main loop' – are called 'workers') and exchange messages with them. It provides several benefits compared to classical mutex-based shared-state synchronization models:

- There is no need to think about thread sync when programming.
  - While it comes at the price of headaches related to handling non-blocking calls, I am arguing that – in those scenarios when we need to handle intervening events anyway – non-blocking single-threaded handling is the least evil; for more discussion, see [NoBugs17].
- Each of the (Re)Actors is deterministic. This, in turn, enables several all-important improvements (from testability and replay-based testing, to production post-mortem analysis), see [NoBugs17] for a detailed discussion.
- This approach is Shared-Nothing and, as a result, it scales near-perfectly (though see the note below). This phenomenon (and problems with scaling shared states) is well-known; very briefly, each and every shared state (in other words, every mutex) carries a risk of becoming a very serious contention, causing severe degradation of scalability; moreover, in quite a few cases you may find that 90% of all your processing happens under one of the mutexes, which means that regardless of the number of cores, you cannot possibly scale more than to 1.1 core.
  - As discussed in [NoBugs17], the only case which I know when pure (Re)Actors-exchanging-messages are not scaling well is when we have a big unbreakable state with lots of calculations performed over it at the same time. This *can* be solved (and was solved for an AAA game Client too) without departing too much from the event-processing (Re)Actor-based ideology (using what I call (Re)Actor-with-Extractors). However, at the moment, (Re)Actor-with-Extractors is not supported by *Emscripten*, so there *may* be some issues on this way.
- (Re)Actor-based systems tend to exhibit very good performance. Discussion of performance advantages of event-driven systems over thread-synced ones is well beyond the scope of this article, but very briefly, it boils down to the costs of thread context switches (which can take anywhere between 10K and 1M CPU cycles(!)), and event-driven systems tend to have *much* fewer of these switches. From a completely different point of view, there is a reason why event-driven non-blocking systems (such as nginx) tend to beat blocking systems (such as Apache) performance-wise.

Pthread support

In theory, *Emscripten* has support for pthreads. However, the support is experimental – and moreover, it is *Firefox-only*. This, of course, makes its use for serious projects a non-starter; however, my rant about pthreads goes deeper than that:

> *Even in the long run, I would prefer support for (Re)Actor-with-Extractors to support for pthreads.*

Sure, having full-scale pthreads, we can implement (Re)Actor-with-Extractors ourselves; however:

- I have no idea how difficult it will be to push pthreads into *all* the browsers (from what I've seen, it can easily become an insurmountable task). (Re)Actor-with-Extractors should be easier to implement (while providing all the safety guarantees – and testability too).
  - In addition, at least in some cases, (Re)Actor-with-Extractors *may* happen to be more efficient (it depends on specifics of pthreads implementation under each of the browsers, but in general, it might easily happen)
- Enabling pthreads would bring us back into dark ages of massive usage of mutexes – and as you may have noticed, I am a very strong opponent of mutex-based thread sync at application level. I prefer to keep my code clean in this regard.

64-bit int and 32-bit float issues

As of now, the only numeric data type in JavaScript is 64-bit float; in addition, some operations (mostly bitwise ones) return 32-bit integer (which always fits into 64-bit float). As a result, any operations which are neither 64-bit float nor 32-bit integer are not 100%-efficient in *asm.js*. In particular:

- 32-bit floats need to be processed as 64-bit floats, which is rather slow compared to native 32-bit floats
- 64-bit integers need to be simulated from 2 of 32-bit integers, which is pretty slow too.

There are some proposals to deal with it (see, for example, [Zakai14]) but as far as I know, these slowdowns still apply, so if you're after best-possible performance, you need to keep them in mind.

Practical uses

As noted above, I haven't used *Emscripten* for a serious project (yet). However, quite a few projects were reported as compiled and working, including:

- Game Engines(!)
  - UE3 (reported to be ported in 4 days)
  - UE4
  - Unity

    Unity is quite an interesting beast when it comes to its use of *Emscripten*; as it uses C# at the app-level, it first re-compiles C# parts into C++ using IL2CPP compiler, and then uses *Emscripten* to compile it into *asm.js*. You won't believe it – but it does work. ☺
- Games
  - Quake 3
  - Doom
  - OpenDune
- Libraries/Frameworks
  - OpenSSL
  - SQLite
  - Pepper (via pepper.js)
  - Quite a few of Qt demos

For a much more comprehensive list of ports and demos, please refer to [Emscripten.PortingExamples].

Competition: NaCl/PNaCl

An alternative way of running C++ code on browsers, is NaCl/PNaCl by Google. It serves pretty much the same noble purpose of running C++ on the browser, however, it has the BIG problem of being restricted to Chrome. As (a) no other browser has followed suit, and (b) as Chrome market share, while it grew to about 60%, has slowed down its growth in 2016, I do NOT think that NaCl/PNaCl is a viable option (except for some very narrowly defined scenarios) – especially when comparing it to *Emscripten*+*asm.js*.

Moreover, I've got a *feeling* (no warranties of any kind) that Google itself has realized futility of (P)NaCl and has slowed down development as a result; overall, my *wild guess* is that in a few years from now, (P)NaCl will be quietly abandoned in favor of *asm.js* (and Google is already working on support for *asm.js* optimizations) or in favor of WebAssembly (see below).

As a result, while the only thing which is certain is that nothing is certain yet, if faced with the task of developing/porting a new C++ Client for browser, I would clearly prefer *Emscripten*+*asm.js*.

Oh, BTW – if you already have a (P)NaCl client, there is a library pepper.js, which aims to provide a migration path from (P)NaCl to *Emscripten*; while I didn't try it myself – well, it *seems* to be worth trying.

Ongoing development: WebAssembly a.k.a. wasm

As a next step in this development (and to compensate for certain problems such as *asm.js* parsing times on mobile devices), an alternative representation – known as WebAssembly or *wasm* – is being actively worked on.

The idea is to use (give or take) the same C++ source code as already can be used to compile into *asm.js*, and to compile it to a very different assembler (*wasm*). Then *wasm* will be loaded into the browser, where it will be JIT-compiled and then executed.

There *seems* to be quite significant momentum behind *wasm* – but as of now, it is too early to tell anything specific. What matters though is that

> *As app-level developers, we do NOT really care much whether it is asm.js or wasm which wins in the end. Rather, we can use asm.js right now, and hope that we won't need to change our programs* ***too much*** *when re-compiling them into wasm* ***(when/if it is widely available)***

Whether these hopes will stand in reality, we'll see, but as of now, it is IMNSHO by far the best option we have to try pushing our C++ Clients into browsers.

Practical uses: porting downloadable clients to the web

Well, it is all this stuff is certainly technically exciting, but what can we get from it in practice? Most importantly,

> *we can port our (well-written-enough) C++ Clients to the web.*

Until two or so years ago, there was no way to port an existing downloadable Client into a web app. In other words, whatever we were doing with our C++, we weren't able to avoid download and at least some warnings about how malicious our code can be from the browser – and this was the point where our potential users were dropping out the most.

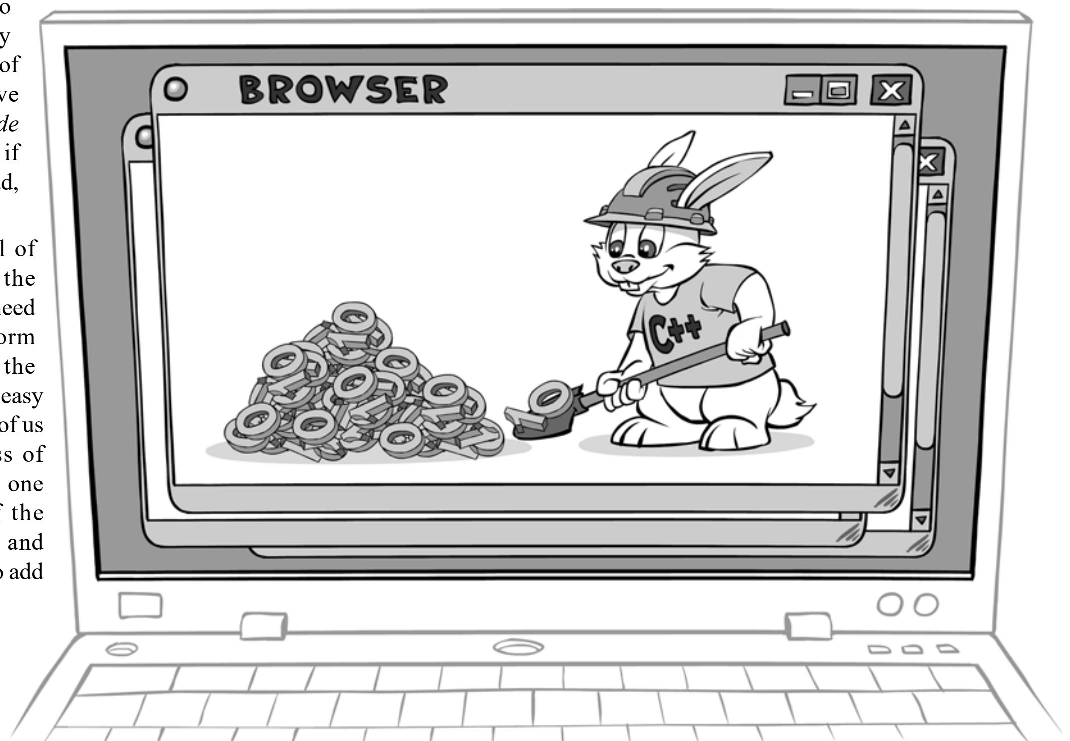
So, for a long while, when deciding how to develop our Client,

> *we were facing a tough choice: either to develop it in JS-only (losing all the bells, whistles, and performance of C++ development) – or to have it in C++ but at the cost of dropping those users who don't want to download.*

With *Emscripten* and *asm.js*, these problems are gone. We *can* have our C++ cake and eat it on browsers too.

In addition, such an option opens a door for some things that are not really widely used yet – such as creating live demo versions which can be viewed in-browser without the need to download and install them; it looks very promising for reducing drop-out rates of potential customers (as showing a live demo tends to work *orders of magnitude* better then showing a screenshot, and if we can get live demo without download, we have a clear winner).

Of course, to achieve this holy grail of multi-platform clients with one of the platforms being 'web browser', we'll need to re-learn how to write cross-platform programs (and apparently, with all the vendor efforts to lock us in, it is not an easy feat), but as soon as we do it (and some of us were doing it all the way regardless of *Emscripten*), we will be able to have one single C++ code base over all of the following: desktops, phones/tablets, and web (with AAA gamedevs being able to add consoles to the mix too). ■



Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

References

[AreWeFastYet17] AreWeFastYet, https://arewefastyet.com/#machine=28&view=single&suite=asmjs-apps

[Emscripten.BrowserMainLoop] Emscripten Contributors, Emscripten Runtime Environment#Browser main loop, https://kripken.github.io/emscripten-site/docs/porting/emscripten-runtime-environment.html#browser-main-loop

[Emscripten.Porting] Emscripten Contributors, Porting, https://kripken.github.io/emscripten-site/docs/porting/index.html

[Emscripten.PortingExamples] Emscripten Contributors, Porting Examples and Demos, https://github.com/kripken/emscripten/wiki/Porting-Examples-and-Demos

[Loganberry04] David 'Loganberry', Frithaes! – an Introduction to Colloquial Lapine!, http://bitsnbobstones.watershipdown.org/lapine/overview.html

[NoBugs10] 'No Bugs' Hare, Single-Threading: Back to the Future?, *Overload* #97/#98

[NoBugs15] 'No Bugs' Hare, Multi-threading at Business-logic Level is Considered Harmful, *Overload* #128

[NoBugs16] 'No Bugs' Hare, Asynchronous Processing for Finite State Machines/Actors: from plain event processing to Futures (with OO and Lambda Call Pyramids in between), http://ithare.com/asynchronous-processing-for-finite-state-machines-actors-from-plain-events-to-futures-with-oo-and-lambda-call-pyramids-in-between/

[NoBugs17] 'No Bugs' Hare, upcoming book *Development & Deployment of Multiplayer Online Games*, Vol.II, chapter on (Re)Actors, current beta available at Leanpub and Indiegogo

[Resig13] John Resig, Asm.js: The JavaScript Compile Target, http://ejohn.org/blog/asmjs-javascript-compile-target/

[TIOBE17] TIOBE Index (February 2017), http://www.tiobe.com/tiobe-index/

[Widman16] Jake Widman, The Most Popular Programming Languages of 2016, https://blog.newrelic.com/2016/08/18/popular-programming-languages-2016-go/

[Zakai14] Alon Zakai, NATIVE SPEED ON THE WEB. JAVASCRIPT & ASM.JS, http://kripken.github.io/mloc_emscripten_talk/sloop.html#/

Contractual Loopholes

Compilers can optimise away functions you may want to time. Deák Ferenc explores ways to stop this happening.

Recently, I attended a course held by Andrei Alexandrescu (of *Modern C++ Design* and *C++ Coding Standards* fame) about C++ optimization. Some of the concepts that were suggested during the lecture have opened up a series of questions, which have lead to some research and finally the creation of this article, in which we explore the delicate interface of the compilers in regard to concepts of real life. We'll look the way a compiler digests information and some of the measures it takes in order to assure highest performance of delivered application so vital for most systems. But mostly we will try to find loopholes in the contract covering the interfacing of the compiler in relation to the real life notion it represents.

When we programmers look at a source, with or without exhaustive experience in certain fields, some patterns will emerge, mental maps will be charted and some conclusions will be drawn from our in-mind analysis of the few lines observed. Using the familiar building blocks of the language (such as conditions, loops, jumps, etc) we will automatically recognize situations that certain pieces of code will provide us with, and we will act accordingly. However, when an optimizing compiler looks at the code, it sees something completely different.

It is extremely difficult, or might not even be possible, to fully understand all the operations performed by the compiler while generating optimized code, due to our very different view and approach of the same problem. The transformation of the source code into the generated binary code goes through a sequence of steps (each deserving a dedicated chapter in the big book of compiler implementation) and from the various intermediary representations, taking into consideration a series of settings, the final result may emerge having a wide variety of embodiments due to settings to optimization, environment and target system.

The background

During the course, while circumstantiating various techniques regarding C++ optimizations Alexandrescu was talking about measuring the time some operations take, how your compiler might disregard all your hard work while optimizing the final code (because the only code you should benchmark and optimize is Release code), and finally how to trick the compiler into actually performing what you want it to do, regardless of optimizations settings.

And let's admit it: the best compilers available nowadays are the C++ compilers, which are among the most advanced ones available – in today's performance oriented world they generate possibly the fastest and most efficient code which satisfies the requirements of lots of platforms and systems. And they have really advanced mechanisms of providing you with the fastest code. They can calculate during compile time complex operations of known data (there go your carefully handcrafted unit tests with vigorously chosen constant data), they can see if you ever intended the use of a function and if not they... just don't call it if they detect it does not affect other data, they ... can do a lot of unimaginable things to make your application faster.

Deák Ferenc Ferenc has wanted to be a better programmer for the last 15 years. Right now he tries to accomplish this goal by working at FARA (Trondheim, Norway) as a system programmer, and in his free time, by exploring the hidden corners of the C++ language in search for new quests. fritzone@gmail.com

```
long some_operation_you_want_to_measure(
  const char *a, int l)
{
  long c, n = 1;
  for (c = 0; c<l ; c++)
  {
    if(a[c] > '9') return 0;
    n +=  n * (a[c] - '0');
  }
  return n;
}

int main()
{
  unsigned counter = 1000000;
  while(--counter > 0)
  {
    some_operation_you_want_to_measure("234" , 3);
  }
}
```

Listing 1

Let's consider the code in Listing 1.

In the context above, when the code is compiled with high optimizations (-O3, -O2 for gcc/clang) enabled, the body of the function `some_operation_you_want_to_measure()` will be present in the generated code (unless you have told the compiler that this is a `static` method in which case it will ... miraculously disappear in the void) but the only thing that the `main` will do is:

```
main:                          # @main
     xor     eax, eax
     ret
```

This, however, might change once you change the optimization settings or the compiler. http://gcc.godbolt.org provides an excellent platform for comparing the output of various compilers when applied to the same source (as a side note: both clang and gcc are able to calculate the result of the function above being 60 and directly feeding it into the required place if we assign it to a variable which is printed).

In order to avoid this uncertainty, Alexandrescu has suggested the following solution: "Use I/O combined with a non-provable false test". His solution is a short function, like Listing 2.

And from this point on, we use the syntax in Listing 3.

As per [Alexandrescu], the code in Listing 1 will be treated by the optimizer in a fashion that guarantees the call (or calculation, as we have

I have started looking for 'contractual loopholes' … specifically researching situations where the compiler could have been clever enough to determine that a condition will always evaluate to false

```
template <class T> void doNotOptimizeAway (
  T&& d )
{
  if ( getpid () == 1 )
  {
    const void *p  = & d ;
    putchar (* static_cast < const char *>( p ));
  }
}
```

Listing 2

```
unsigned counter = 1000000;
while(counter-- > 0)
{
  doNotOptimizeAway(
    some_operation_you_want_to_measure("234" , 3)
  );
}
```

Listing 3

observed for certain simple situations) of the **some_operation_you_want_to_measure** and it will assure you the code will not be 'optimized away'. What the code above does is the following: it defines a function which will act as a placeholder, it is never optimized away (due to the fact that it contains an I/O operations which are never optimized away), but regardless does nothing (because no user-land process in a sane system can have their PID equal to 1, the value (un)officially being reserved to the **init** process).

And this unprovable false was the spark for this article. I have started looking for 'contractual loopholes' in the standard libraries and functions, specifically researching situations where the compiler could have been clever enough to determine that a condition will always evaluate to false, and use this knowledge while generating code.

The falses

The C and C++ standard libraries contain a huge number of functions, providing abstractions of real life concepts which easily can be translated into computer code. However, due to particularities of computing systems, these translations on certain occasions are unable to fully model the real life situations, thus providing me with the required attack surface.

Playing with time

There are several functions and structures related to time retrieval in the C++ standard libraries we can find in **<ctime>**, and its counterpart **C** library's **<time.h>**. One of them is the **localtime** function, which populates a **tm** structure with human readable values of the current time.

```
struct tm
{
  int tm_sec;   /* Seconds.[0-60](1 leap second)*/
  int tm_min;   /* Minutes.[0-59] */
  int tm_hour;  /* Hours.  [0-23] */
  int tm_mday;  /* Day.    [1-31] */
  int tm_mon;   /* Month.  [0-11] */
  int tm_year;  /* Year - 1900.   */
  int tm_wday;  /* Day of week.  [0-6]    */
  int tm_yday;  /* Days in year. [0-365]  */
  int tm_isdst; /* DST.          [-1/0/1] */
};
```

Listing 4

```
time_t now;
struct tm *tm;
now = time(0);
tm = localtime (&now);
if(tm->tm_sec >= 62)
{
  const void *p  = & d ;
  putchar (* static_cast < const char *>( p ));
}
```

Listing 5

As per [n1256], Listing 4 is the list of its fields (and the declaration was taken from my **time.h**).

As the comment says, all of these fields are restricted to meaningful values; however, no-one can stop me writing code like Listing 5.

Now, we humans know that in reality this will never happen (just like checking if the hour is **>24**); however, the compiler currently is not clever enough and the suggested feature of contracts in the **C++17** standard currently has no coverage for this situation [CppContracts], thus the compiler will happily generate the following code (showing the output from clang 3.9.1, since this came up with the cleanest code – the comment is from me):

```
xor     edi, edi
call    time
mov     qword ptr [rsp], rax
mov     rdi, rbx
call    localtime
cmp    dword ptr [rax], 62 ; <-- This is the
jl      .LBB0_3              unnecessary comparison
```

However, all compilers I have tested with this piece of code provided me with the same result: a comparison which can never be true.

Wondrous world of mathematics

There is an exhaustive library of mathematical functions available in **C++** and **C** (found in **<cmath>** or the corresponding **<math.h>** header files)

which can take almost all basic mathematical concepts and translate them into corresponding function calls providing the expected result. One of these is the `sin()` functions which computes the sine of the argument (given in radians). If no error occurs, the sine of the argument is returned, the value is in the range [-1 .. +1] as per the definition of the trigonometrical function. So, from a human's point of view it really makes no sense to check for values `>1`. However, again, the compiler is not clever enough to realise this, so the following call:

```
if ( sin( *(reinterpret_cast<float*>(&d)) )
  > 1.0f )
{
  const void *p  = & d ;
  putchar (* static_cast < const char *>( p ));
}
```

will happily generate code like:

```
movss   xmm0, DWORD PTR .LC1[rip]
mov     QWORD PTR [rsp+8], 60
call    sinf
ucomiss xmm0, DWORD PTR .LC0[rip]
jbe     .L2
```

where the number `60` is the value precalculated by the compiler (as presented above) however the call to the `sin` and the comparison after (`ucomiss` = Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS) are present in the generated code.

The code above was generated by gcc 6.3. I've had the same result with gcc 6.2 and 6.1. The gcc 5.x series produces also similar code (not exactly this, but a bit longer using a different set of assembly commands) which is identical to the code generated by gcc 4.9.x however the gcc 4.8 and 4.7 series did not generate any code for this senseless `sin` check which behaviour was also manifested by clang 3.9, 3.8. Basically, all clangs up to 3.0. Microsoft's Visual C++ compiler generated code, which was very similar to the code generated by gcc 5.x family with the check inside.

The interesting part came, however, when I increased the complexity of the `some_operation_you_want_to_measure` function. I have added the following line

```
if(n > 24) n-= n* a[c];
```

in the body of the `for` loop ***after*** the line `n +=  n * (a[c] - '0');`. Suddenly the compilers which did not take into consideration my senseless juggling with sine started paying attention to it and all of them generated some code to handle it. However, if I put the new line ***before*** the existing line nothing changed, the behaviour was the same as without the line. Seems that I have really stepped on the toes of some optimizers.

After several experimentation stages with the function `some_operation_you_want_to_measure`, I have drawn the conclusion that the more complex the function I want to measure gets, the least possible is for the call to the senseless `sin` to be optimized away. And when the complexity of the function `some_operation_you_want_to_measure` has reached a stage when the compiler was not able to automatically calculate its result during compile time, the senseless sine check was always called. I leave to the imagination of the reader to try to envision what other misconducts can be done with the function from the mathematical library.

The null-terminated byte strings

Not a very widely known name, but there is a header file `<cctype>` which has it roots in C's `<ctype.h>` containing all kind of functions which can be used to manipulate... well, null terminated byte strings. You can find in there a wide range of functions for checking various properties of characters (in the likes of is this an UPPERCASE character, is this a digit, etc...).

Among them is the very useful `tolower(int)` function which, as its name suggests, will convert its argument (an `int`, representing a character) to its lower case equivalent using character conversion rules as per the current C locale. The function, however, will return its unmodified argument if no lowercase version is available in the current C locale. And its counterpart `toupper`, which behaves the same way, but it converts the argument to an uppercase character if possible.

We can exploit this, in order to create another unprovable false. Let's consider:

```
if ( toupper('2') == 1 )
{
  const void *p  = & d ;
  putchar (* static_cast < const char *>( p ));
}
```

Again, the compiler does not know that the uppercase of the character `2` does not exist, due to specification `toupper` will return `2` and this is again a false which cannot be optimized away.

Contracts

The **C++17** standard supposedly will include a notion of contracts which will be like a run time assertion for allowing checks for validity of input arguments or other dependants, but as per the latest (as of February, 2017) available draft [n4618] I have found no mention of them. Due to the nature of some of the falses from above, these would be very easily identifiable at the compilation time and the compiler could take steps in order for them to not to behave as erratic as today.

For example let's consider the `sin` function. Currently one of the declarations it has is the following:

```
float sin( float arg );
```

which says nothing to the compiler about the nature of the function, ie: that its return value is always between -1 and +1 (inclusive).

Let's revise the declaration of the `sin` function, empowered with the concepts of contracts using the syntax from [CppContracts] (please note, the code below is not valid C++):

```
float sin( float arg )
  [[ensures: return <= 1 && return >= -1 ]];
```

Due to the almost documentation like nature of a contract, the compiler instantly knows that the following

```
sin( *(reinterpret_cast<float*>(&d)) ) > 1.0f
```

will always be false and can generate proper code in order to achieve maximum performance and speed.

There is just a tiny little problem with the code above: `return` is not part of the upcoming C++ contracts specification, so the declaration above is just a small dream that might come true one day.

Conclusion

The list of the presented falses is just a subset of all those that exist out there, when you shall look specifically for them you shall find even more, from various libraries, through operating system specific data, to miscellaneous functions. I just wanted to follow up on [Alexandrescu], list a few more currently in existence and offer a theoretical way to mitigate the not so serious threat they represent. Once the compilers fully support the contracts, I am sure a new set of improved C++ libraries will come in existence which will help the compilers in their everyday job. ■

Appendix 1

The techniques presented above, however, have a minor downside to them: Strictly speaking the measurement of time variations will include a tiny fraction consumed by the function call together with the performed operations, which introduces variable distortions in the performance measurements thus degrading the quality of the data we receive. In order to achieve constant time and minimum overhead we can use the following technique: we always call the function we want to measure via a volatile function pointer (Listing 6). This introduces only a few extra bytes in the code (see Listing 7), and by using it we do not depend on calling external functions with all their side effects, and indeed the calls are performed as we would expect them to be. Also and interesting fact is, that in this case the compiler is not able to do the precalculation of the result of the

```
using ptr_fn =long(const char*, int);

ptr_fn* volatile pfn = some_op;

unsigned counter = 1000000;
while(counter-- > 0)
{
  pfn("234" , 3);
}
```

Listing 6

```
mov     QWORD PTR [rsp+8], OFFSET FLAT:some_operat
ion_you_want_to_measure(char const*, int)
.L10:
mov     rax, QWORD PTR [rsp+8]
mov     esi, 3
mov     edi, OFFSET FLAT:.LC0
call    rax
```

Listing 7

function, nor will it inline the called function into the body of the calling function.

Appendix 2

In order to have a base of comparison for the variations in the generated code, I have created the test application also in two other programming languages with syntax very familiar to C and which also compile into binary code: Go and D.

Listing 8 is the Go version of the same program. The corresponding assembly code generated was (only showing the interesting parts, since Go compiled an executable file of 1.1Mb) is in Listing 9.

The same application in D is shown in Listing 10, and the generated code, which was compiled with optimizations on (-O) (the final executable was 594Kb long, I took only the relevant parts) is in Listing 11.

Although this is not necessarily related to the article, regardless it's interesting to see how various compilers handle the same situation.

Acknowledgements

I have referenced frequently [Alexandrescu] through the article, and with his permission I have re-used code written by him which was presented at the course. It was awesome.

```
package main
func
some_operation_you_want_to_measure(a string,l int)
  int {
  var n int = 1
  for c := 0; c<l ; c++ {
    if(a[c] > '9') {
      return 0
    }
    n = n + n * int(a[c] - '0')
  }
  return n
}
func main() {
  var counter int = 1000000
  for counter > 0 {
    some_operation_you_want_to_measure("234" , 3)
    counter --
  }
}
```

Listing 8

```
    mov rax,0xf4240
    nov OWORD PTR [rsp+0x20],rax
    cmp rax,0x0
    jle loc_00461oea
1oc_004010b5:
    lea rbx,[rip+0x711c4]
    mov QWORD PTR [rsp],rbx
    mov QWORD PTR [rsp+0x8],0x3
    mov QWORD PTR [rsp+0x10],0x3
    call main.some_operation_you_want_to_measure
    mov rax,QWORD PTR [rsp+0x20]
    dec rax
    mov QWORD PTR [rsp+0x20],rax
    cmp rax,0x0
    jg loc_004010b5
1oc_004016ea:
    add rsp,ox28
    ret
```

Listing 9

Regarding Appendix 1: it came into existence during the review phase of the article when one of the reviewers draw my attention towards the distorted time measurements and kindly suggested the solution to mitigate this problem. Thank you!

References

[Alexandrescu] Fastware: The art of optimizing C++ code, Kongsberg, 2017 January

[CppContracts] http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4415.pdf

[n1256] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

[n4618] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4618.pdf

```
long  some_operation_you_want_to_measure(const
char *a, int l) {
  long c, n = 1;
  for (c = 0; c<l ; c++) {
    if(a[c] > '9') return 0;
    n +=  n * (a[c] - '0');
  }
  return n;
}
int main() {
  int counter = 1000000;
  while(--counter > 0) {
    some_operation_you_want_to_measure("234" , 3);
  }
  return 1;
}
```

Listing 10

```
    mov ebx,0xf423f
loc_00427d53:
    lea rsi,[rip+0x26806]
    mov edi,0x3
    call
_DZtt34some_operation_you_want_to_measureFxPaiZL
    dec ebx
    test ebx,ebx
    jg loc_00427d53
```

Listing 11

All About the Base

Representing numbers presents many choices. Teedy Deigh counts the ways.

A programmer is nothing without numbers. And there are so many to choose from! Unfortunately, because of the taxation of representation – integers are bitly stunted to ints and floating-point numbers aren't real – a programmer has fewer to play with than the average (and sometimes mean) mathematician. This can leave some programmers with an inferiority complex, a problem that is both real and imaginary.

Many mathematicians, however, fear both numbers and meaningful variable names. Programmers, on the other hand, revel in – and overflow – different number representations and bases and delight in the full range of identifiers from the unidentifiable to the VeryExplicitlyMeaningfulThisIsAnAdjectivalPhraseServingAsANominal.

To make life a little more interesting and less inferior, here is a brief list of some bases you should know and employ, presented in hash order:

binary is the base line of code, its very bits: on or off, 1 or 0, yes or no, true or false, fact or alternative fact, to be or not to be, do or do not... there is no try – which goes some way to explaining the popularity of Boolean and Booleanish return values in languages without exception handling, leaving readers with the frisson of uncertainty as to whether 0 means success or failure or quite possibly both.

unary is the fundamental counting system of the universe – a million things is represented by a million things, not seven. Getting your hands on this original digital base is easy: one finger is 1, two fingers is 2 – or 11_1 – three fingers is three – or 111_1 – and so on. The simplicity of this system means that you can be free and easy with your choice of digit, i.e., instead of 1 you can use a pebble, a dash or a solipsistic I. It is also easy to generate in code. For an integer *n*, the following Python expression is a unary converter:

```
('I' * n)
```

You can also give it a bash with

```
printf %"$n"s | tr ' ' 'I'
```

ternary does not simply give us the operator of champions – Elvis is not dead... albeit conditionally – it gives us the three-valued logic of comparisons (-1, 0, +1) and uncertainty: SQL's `TRUE`, `FALSE` and `NULL`; JavaScript's `"false"`, `false` and NaN; the it-compiles-but-does-it-work yes/no/maybe.

octal is the base of the Unix demigods. It has been said that it is better to `fseek` forgiveness than to ask permission, which perhaps explains why octal has been progressively marginalised and relegated to file permissions. New Testament C makes it clear that 8 and 9 are not octal digits, whereas old testament C was somewhat looser and more permissive, anticipating Postel's law ('Be liberal in what you accept, and conservative in what you send') and the Liskov Substitution Principle (LSP, which is not to be mixed up with LSD... although that might explain a few things).

nonary is of surprisingly little use to the programmer, except as an off-by-one overflow for octal. It should be noted that 9 is not a nonary digit. It is listed here simply to highlight the common existential confusion between nonary and...

nunnery, which is a number base that highlights deep questions of existence and non-existence in black and white (and wimples). For example, Hamlet's to-be-or-not-to-be binary quandary takes place in what is known as the nunnery scene. When it comes to representation of nunnery in code we can turn to Nietzsche for implementation inspiration:

> If you gaze long into a `void`, the `void` also gazes into you.

decimal is widely understood and in common use among the general population. It is, therefore, of little interest to the programmer.

hexadecimal is how binary wants to be organised and what octal always wanted to be. Use it everywhere, especially when people are expecting decimal.

duodecimal is often touted as a geeky base which should be used in place of decimal. While this is certainly a point in its favour, it's a claim that doesn't meet all of its challenges. For example, base 12 is often touted as being better than decimal because 12 has so many divisors – 1, 2, 3, 4, 6, 12 – which means that in the interval 1..12 half of the integers are divisors. 50%? Not bad. Applying the same analysis to binary – 1, 2 – and unary – 1 – however, gives 100% coverage.

Advocates of legacy measurement systems also advocate duodecimal, noting that *uncia*, the Latin for twelfth part, gives us *inch* and *ounce*. Hence, 12 inches per FPS foot, 16 ounces to an avoidupois pound (a hexadecimal wannabe if ever I saw one) and 20 fluid ounces to an imperial pint (easily expressed in a bi-denary system). Clearly, there are some issues of legacy and implementation to be worked out before they're ready to present this one.

dewey decimal is an arcane numerological system for organising libraries. Forget packages and namespaces – or, more popularly, spaghetti inheritance and obfuscators – the priesthood of librarians has developed a far more obscure approach to structuring and deploying information systems.

There is also a parable of number systems, as told in *Silent Running*, which begins with ternary (Huey, Dewey and Louie), is reduced to binary (Huey and Dewey), then finally unary (Dewey).

bi-quinary is a much overlooked and classic digital system. Bi-quinary coded decimal is based on pairs of fives - hold out your hands - and can be found in proper abacus systems and early computers, such as the Colossus. It has been said that it is awkward to use and difficult to convert to and from. On the other hand, given a prior unary conversion, it has also been `sed`:

```
s/IIIII/V/g
s/IIII/IV/
s/VV/X/g
s/VIV/IX/
s/XXXXX/L/g
s/XXXX/XL/
s/LL/C/g
s/LXL/XC/
s/CCCCC/D/g
s/CCCC/CD/
s/DD/M/g
s/DCD/CM/
```

Teedy Deigh has no idea who Megan Trainor is – or, indeed, what any kind of trainer is. Teedy codes by night and counts imaginary sheep by day. Basically, she makes stuff up using numbers.

JOIN THE ACCU!

You've read the magazine, now join the association dedicated to improving your coding skills.

The ACCU is a worldwide non-profit organisation run by programmers for programmers.

With full ACCU membership you get:

- 6 copies of *C Vu* a year
- 6 copies of *Overload* a year
- The ACCU handbook
- Reduced rates at our acclaimed annual developers' conference
- Access to back issues of ACCU periodicals via our web site
- Access to the *mentored developers projects*: a chance for developers at all levels to improve their skills
- Mailing lists ranging from general developer discussion, through programming language use, to job posting information
- The chance to participate: write articles, comment on what you read, ask questions, and learn from your peers.

Basic membership entitles you to the above benefits, but without Overload.

Corporate members receive five copies of each journal, and reduced conference rates for all employees.

How to join

You can join the ACCU using our online registration form. Go to **www.accu.org** and follow the instructions there.

Also available

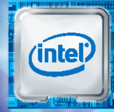
You can now also purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Design: Pete Goodliffe

INTEL INSIDE



FASTER APPLICATIONS OUTSIDE

CREATE FASTER CODE, FASTER

Reach new heights on Intel processors and coprocessors with new standards-driven compilers, award-winning libraries and innovative analyzers.

Intel Parallel Studio XE Composer Edition for Fortran Win Commercial Licence (SKU: 349062)

£634.99

QBS Software Ltd is an award-winning software reseller and Intel Elite Partner.

To find out more about Intel products please contact us:

020 8733 7101 | enquiries@qbssoftware.com
www.qbssoftware.com/parallelstudio