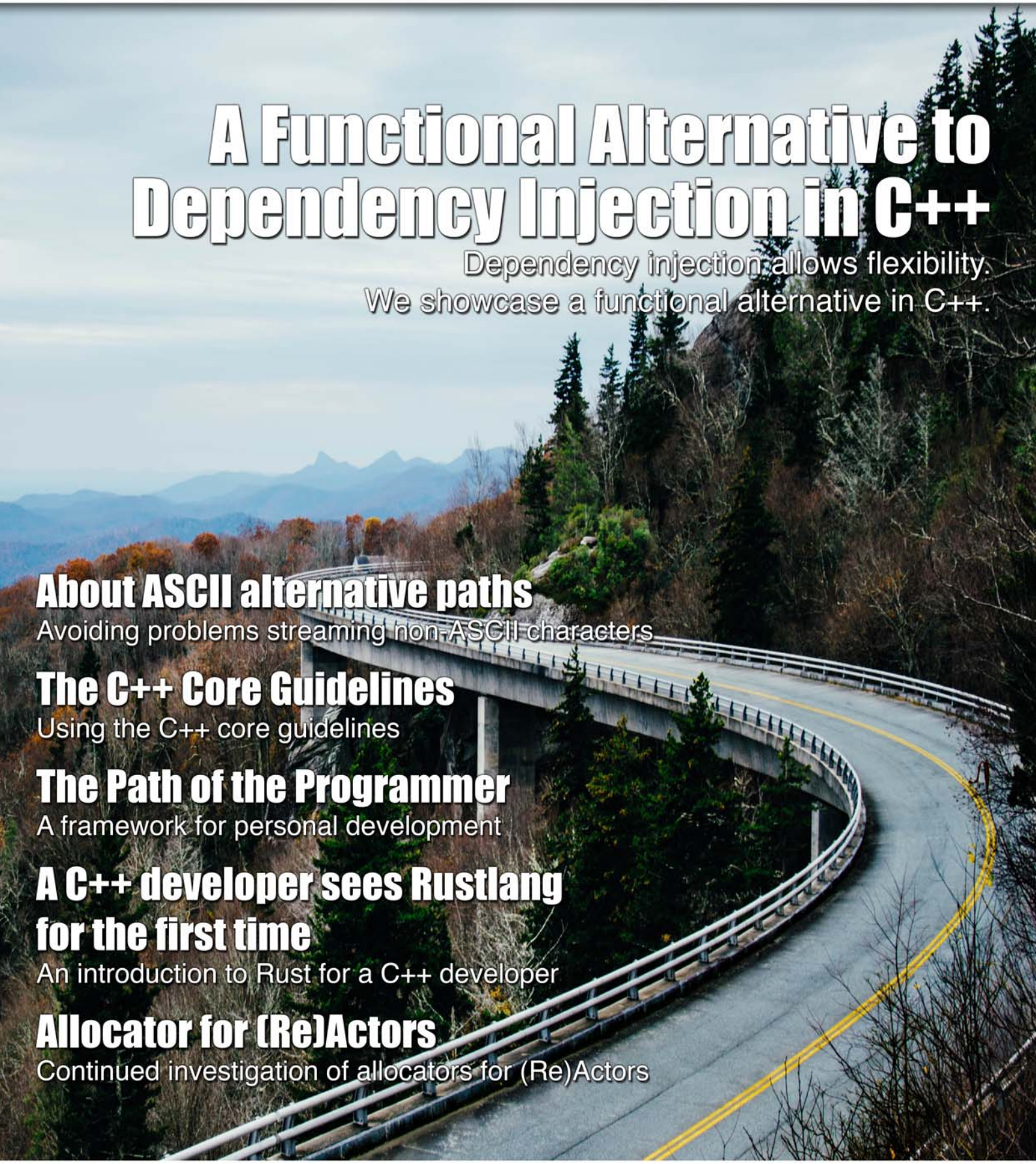
overload 140
AUGUST 2017 £3



A Functional Alternative to Dependency Injection in C++

Dependency injection allows flexibility. We showcase a functional alternative in C++.

About ASCII alternative paths

Avoiding problems streaming non-ASCII characters

The C++ Core Guidelines

Using the C++ core guidelines

The Path of the Programmer

A framework for personal development

A C++ developer sees Rustlang for the first time

An introduction to Rust for a C++ developer

Allocator for (Re)Actors

Continued investigation of allocators for (Re)Actors

A magazine of ACCU
ISSN: 1354-3172

"The magazines"

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

"The conferences"

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

"The community"

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

"The online forums"

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

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

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

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

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

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

Design: Pete Goodliffe

OVERLOAD 140

August 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 141 should be submitted by 1st September 2017 and those for Overload 142 by 1st November 2017.

The ACCU

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

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

Overload is a publication of the ACCU

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

Copyrights and Trade Marks

Gnomes and Misnomers

What's in a name? Frances Buontempo decides some names are better than others.

It is time, yet again, to attempt to write an editorial. Of course, 'editorial' is something of a misnomer. We are all aware that it should be the editor's opinion on a topical issue. I believe I have rigorously steered clear of this so far, despite being accused of sailing a bit close to the edge once in a while. As I consider what an editorial is, I notice it has several synonyms, including feature, commentary and write-up, yet I could only find one antonym; secondary source [PT]. This is not to be confused with secondary sauce, which has a very different flavour. Homophones, words that sound the same but mean something very different can be the cause of confusion or fun. Do not confuse your carets with carrots, or vice versa. More extremely, homonyms appear the same when spoken or written but mean different things. I saw a saw. I object to that object. Such ambiguity can lead to dreadful puns, hilarious jokes or beautiful prose. Or long meetings with fights over documentation, meaning and project or component names.

As programmers we are often stumped when searching for a good clear name for something. I have been re-reading *Jingo* by Terry Pratchett [Jingo], where we meet a character, Leonard de Quirm who bears more than a passing resemblance to Leonardo da Vinci. He has created a metal transport device that goes under water.

"Er... what is this thing called?" said Colon, as he followed the Patrician up the ladder. "Well, because it is submersed in a marine environment I've always called it the Going-Under-the-Water-Safely Device," said Leonard.

A footnote observes that "Thinking up good names was, oddly enough, one area where Leonard Quirm's genius tended to give up". How do we find names for our components, programs and projects? Some have long, relatively precise titles. *Microsoft Visual Studio 2017* springs to mind. Others are designed to be unsearchable on the internet, perhaps by accident; *Q, R, C* for starters. We also have many abbreviations and acronyms, which end up feeling like whole words, and we forget some people don't know these. I know what RC means in a list of versions of software downloads, but 'release candidate' may not spring to mind immediately for everyone. Some acronyms are more natural that others. Some are quite clearly backronyms, where words have been tortured into spelling another, or TISA if you will. Some spring from a central idea. Animals give us GNU, YACC, then Bison, and others. People choose Greek gods, comic book characters, mythology. Having a theme can get your imagination going. I wonder if it might sometimes alienate people though. If your projects were all named after My Little Pony ponies or Jane Austen characters, how would you feel? Or lesser known industrial bands? Or Old Testament prophets? Footballers? What do your choices say about you, your company, or open source project?

This is touched on obliquely in *REAMDE* [Stephenson]. Correct Reamde, not Readme; a deliberate word twist à la Skinny Puppy lyrics. In a fabled cryptocurrency mining fantasy world, where character names are predominantly in Western fonts, the Chinese hackers and crypto-coin miners use character names that fit Western fiction, not just because they can't type their letters into many Western apps. In the real word, such fantasy games might have groups of mythology or fantasy aficionados that hail from, usually, Western tales. A Chinese hacker in the book commented that he knew all about the mythologies, from comics and US films. Nonetheless a rival group had sprung up who preferred modern, bright colours and futuristic names. The party lines and allegiances might not lie where you would naively expect them. Tribal behaviour and exclusion is a topic for another day. The names we choose often say something about us, either individually or as a group.

We do frequently draw on mythology for inspiration, or at least for names. Viking and Norse history and mythology often provide. Take Bluetooth, a king and wireless technology, bringing things together [Bluetooth]. Consider Munin, one of Odin's ravens, whose name means memory, used for a monitoring tool [Munin]. We also see Gnome, the open source desktop or a 16th Century 'diminutive spirit', who lives underground [Gnome]. The Renaissance spirits don't stop there. We have Sprite: 'a smaller bitmap composited onto another by hardware or software' or 'legendary creatures such as elves, fairies and pixies' [Sprite]. I am sure you can think of many I have missed. If you are unfamiliar with the meaning of the names or the background they draw from, this probably won't detract from your ability to use or program with something so named. Be aware that most fairy-type creatures are troublemakers, as far as I can tell. We don't stop there. We also have daemons, running unobtrusively in the background. Or crashing or spewing to a log file. Not to be confused with demons or dæmons, who are benevolent, benign or downright bad, depending on which mythology you are reading. There are those who say most computer programs are benevolent, benign or downright bad as well. Jinn, djinn, or genie is another matter entirely, each inspiring many acronyms. Mine's a gin!

Along with names for programs and projects, we have some more abstract words to describe design approaches or management processes. These can help communication, but things tend to trend, dragging the uninitiated on-board. Many teams claim to be agile, even rigidly agile. True story. We find many misnomers, where a wrong or inappropriate name is used. A speaker, I forget who, once quipped that a class under consideration was more of an entire school then a class, since it had so many members. Using and misusing words can distort or help your

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.

understanding. Sometimes you just need to work side by side for a bit to gain a shared vocabulary. I had mentioned TDD a few times at work a while ago, and it was only when I sat with a colleague she realised I did **really** mean I wrote the test first. I was never sure what she thought I had meant previously when I said 'test first'. Obviously, it didn't mean 'test first' to her. Does your team have continuous integration, CI? Does it actually have several different branches that haven't been merged for months? Do the scripts run 'automatically, at the touch of a button'? Another true story. We filter what people say to us through our own experience and bias by default. Without the shared context of comics, films or coding, misunderstandings abound. Inconceivable, you might say. "You keep using that word, I do not think it means what you think it means," Inigo Montoya from the 1987 film *The Princess Bride* would retort. Meme-tastic. Claiming you are agile, or promote based on merit does not make it so. A log line claiming 'ERROR:' does not always mean an actual error. Class names frequently sprout meme-like parts; builder, factory, abstract. These should convey how they work but as we know ending up with an `AbstractFactoryBuilder` or similar is meaningless to the point of ridicule.

Aside from class names, consider namespaces or package names. How many times do we get a 'Utility' package or module? Or even worse 'Misc'. We often end up with a bucket of stuff since we are neither clear what to call it or if it's worth breaking things down into smaller related groups. On a code level, I often end up with a snaky mess of `if`s and `else`s when I haven't stopped to think about ways to define small, clear functions that would stop the rot. I am continually improving at deleting Boolean flags though. Sometimes you can't find a way to do something, or chose between options because you are doing it wrong. Being unable to name a class is a sign. If you're blocked, it's time to step away from the keyboard.

Many out-and-out misnomers exist. Does Excel excel at anything? How easy is it to get at data in Access? How often do people claim they have Big Data, which is less than a gigabyte (e.g. all of Shakespeare)? Do you have any Smart devices? Almost anything with 'giga' or 'nano' in its title falls in this category. Is your job title 'Developer' but you spend all day in meetings or grepping log files? In my keynote for this year's ACCU conference, I observed that machine learning is almost certainly a misnomer. The machines don't learn anything, but we sometimes do from the data analysis they perform. Furthermore, AI, artificial intelligence is not clearly defined, beyond a Turing test assessment; you'll know it when you encounter it. I have heard the question "Can't you make the AI more intelligent?" in relation to a variety of applications recently. I think this really means "Can't you write a different algorithm?" when thought about. Giving an amorphous collection of ideas or algorithms a single name allows them to be discussed easily, but frequently conveys an incorrect impression to people outside the subject area. Front ends and user interfaces provide no end of places for trouble to happen from un-thought-through wording. How many times have you been confronted by a message box with 'OK' or 'Cancel' on, when it's not ok and you're not sure what cancel will actually do. I have recently been working on 'plugins' for some security tools. One input plugin is more like a section of regex in an ini file. Beware technical terms. You might be disappointed. If you need a good book to get you thinking straight (and make you laugh) consider Randall Munroe's *Thing Explainer*. He strives to explain complicated stuff with pictures and a very small vocabulary. One review notes, "If you can't explain something simply, you don't really understand it."

Sometimes we pick names in the hope they will become true. Just setting up a unit test project might encourage people to add unit tests. If you are unfortunate, they may add code that isn't strictly a unit test, running off to a database and taking ages to complete. Clearly, 'unit test' is another technical term that can lead to fights for hours. Some tests are better than no tests, though. You have probably heard of nominative determinism: 'people tend to gravitate towards areas of work that fit their names.' [ND] Mr Baker, the baker, and so on. Many cultures attached a great significance to names. Children are given saints names, or called Hope or similar, as though you are imparting a magic power to them. Having never had kids, this is not a problem I've had to wrestle with. We do have a cat though, and he is called Vim; another area where technical and non-technical people might draw very different conclusions. You can't call a cat emacs, as far as I'm concerned. Well, not in our house.

Finally, we sometimes end up with names or words begging to be used for something. I am glad to hear that Covfefe [Covfefe] has been applied to an Act to preserve Mr Trump's tweets [Independent]. If you are stuck on a name for something, you might find you don't have one coherent thing to name, so need to refactor, carving things up differently. Or if you can't find a pronounceable acronym, perhaps you should draw on things you or your team love, including stories or food or drinks or something inspiring. If someone uses a term you think you understand, it might be worth clarifying to avoid talking at cross-purposes. Names can help to communicate, but can lead to misunderstanding and can convey a demographic which might make some feel excluded. Names give us power over things. We can talk about them more easily when we can identify them. We can also get in a muddle if the names are unclear. Cultural and contextual issues can add to our confusion. The essence of what we do is communication, to other programmers, rather than the machine; as Knuth said,

> Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. [Knuth]

References

[Bluetooth] https://en.wikipedia.org/wiki/Harald_Bluetooth and https://en.wikipedia.org/wiki/Bluetooth

[Covfefe] https://english.stackexchange.com/questions/391945/what-does-covfefe-exactly-mean

[Gnome] https://www.gnome.org/ or https://en.wikipedia.org/wiki/Gnome

[Independent] 'US politician introduces the Covfefe Act', Emily Shugerman, June 2017 http://www.independent.co.uk/news/world/americas/us-politics/covefe-act-trump-twitter-bill-introduced-democrats-stop-president-deleting-tweets-a7786676.html

[Jingo] *Jingo*, Terry Pratchett, 1997.

[Knuth] Literate programming, 1992 (taken from http://www.literateprogramming.com/)

[Munin] http://munin-monitoring.org/ or https://en.wikipedia.org/wiki/Huginn_and_Muninn or indeed https://en.wikipedia.org/wiki/Hugin_and_Munin_(Marvel_Comics)

[ND] https://en.wikipedia.org/wiki/Nominative_determinism

[PT] https://www.powerthesaurus.org/editorial/antonyms – though I didn't look very far.

[Stephenson] Neal Stephenson, *Reamde*, Atlantic Books, 2012.

[Sprite] https://en.wikipedia.org/wiki/Sprite_(computer_graphics) or https://en.wikipedia.org/wiki/Sprite_(entity)

The Path of the Programmer

Personal development is important. Charles Tolman provides a framework for looking at this.

The impetus for this talk[1] came out of a chat I had with a friend, where I was ranting – as I can do – about code, and then realized that of course it is easy to rant about other people's code. This prompted me to look back at my own experience. I started coding for a living back in 1980 – a fact that doesn't bear thinking about! – and have spent most of my career implementing high data rate video editing systems. Until recently I worked in a company that does TV and film effects and editing systems, working on a large C++ system of more than 10MLOC. I have now moved into the CAE sector.

This is quite a 'soft' talk and I will be following on from some points in the keynote (*Balancing Bias in Software Development*) [ACCU16] given by Dr. Marian Petre, although I will drop into some more grounded issues around video player pipeline design and some of the design issues that I have come across.

As I mentioned, I had a sense of frustration with the quality of what was getting produced in a commercial context, and frustration in terms of finding people who could make that switch from doing the actual coding and implementation to taking a more structural view. But though I started coding in 1980, it was not until 1995 that I can say I was actually happy with what I was producing. That is quite a sobering thought. OK, maybe I have the excuse that I did not really get into Object Orientation until 1985/6, and the Dreyfus brothers [Wikipedia_01] say it takes 10 years to become an expert in a domain, but even so…

I therefore want to delve into my own experience and try to understand why this takes so long. This is an issue, not so much about teamwork, but about what we could possibly do individually drawn from my own experiences with being a practitioner with large codebases.

In terms of my inspirations with regard to software architecture, Christopher Alexander of course is one, and there is one from left field. I got involved in starting a Steiner school for my children back in the 1990s and Steiner's epistemology, drawn from a foundation coming from Goethe, is actually quite relevant.

I will recap some of the points from my talk at ACCU2013 about 'Software and Phenomenology' [Tolman13], and my workshop in ACCU2014 about 'Imagination in Software Development' [Tolman14], but will be taking a slightly different slant on that content.

The path of the programmer

I want to start with some reflections on the path of the programmer as I have come to see it, borrowing an idea from Zen about the three phases on the path to enlightenment.

Charles Tolman earned a degree in Electronic Engineering in the 70s, and then moved into software; progressing through assembler to Pascal, Eiffel and eventually C++. He's now involved in large scale C++ development in the CAE domain. Having seen many silver bullets come and go, his interest is in a wider vision of programmer development that encompasses more than purely technical competence. You can contact him at ct@acm.org

There is the initial NOVICE phase where you are still learning about the tools you have at your disposal.

A lot of your thinking is going to be Rule Based since you are learning the steps you need to take to do the job. The complexity of your thought is generally going to be less than the problem complexity you are dealing with when you get into 'live' industrial work, and hence you are producing brittle code, and/or it is not doing all that is needed. Here you are aware of your own limits because you know you do not know things, but you are unaware of your own process. I am not here talking about team development process, I am talking about your own personal learning process.

This level is thus characterized by an undisciplined self-awareness. There is little self-awareness about your own limits, and the lack of knowledge about your learning process means what awareness you have is undisciplined.

The next phase is what I call the dangerous phase, the JOURNEYMAN phase. It was about 1984 when I was in this phase.

Here you have a better knowledge of tools, having learnt about many of the programming libraries available to you. But the trap here is that the Journeyman is so very enamoured of those tools, and this conforms to the upward spike in the confidence curve that Dr. Marian Petre talked about this morning (The Dunning-Kruger effect [Wikipedia_02]).

Here the problem is that you can get into Abstract thinking and this can lead you to having an overly complex view of the solution. Your thinking here is more complex than the problem warrants. It is quite possible that up to 80% of the code will never be used. Therefore you are unaware of your own thinking limits and this can lead to an experience of total panic, especially if you are working on larger systems. [About a quarter of the listeners raised their hand when I asked if anyone had ever experienced this] This conforms to the downward spike that occurs after the upward spike on the confidence curve.

One anecdote I have is the story of one rather over-confident colleague who was given responsibility for a project. The evening before the client was due to turn up for a demo he was still coding away. When I came into work the next morning there was a note on his desk saying 'I RESIGN'. He had been working through the night and didn't manage to get to any solution. Of course the contract was lost.

This highlighted the total lack of awareness about his own limits. In this phase I too remember having an arrogant positivity – "its just software", with the accompanying assumption that anything is possible. I had an undisciplined lack of self-awareness. Some people can stay in this phase for a long time, indeed their whole career and it is characterized by an insistence on designing and coding to the limit of the complexity of their thinking. This means, by definition, that they will have big problems during debugging because more complex thinking is needed to debug a system than was used in its creation.

1. ACCU2016: Talk on Software Architecture Design 1: The Path of the Programmer.

We have gone here from one undisciplined state of partial self-awareness to another undisciplined state of no self-awareness. Of course this could be seen to be a bit of a caricature but you know if you hit that panic feeling – you are in this phase.

The next phase is the MASTER phase. In the past I have hesitated to call it the Master phase, referring to it instead as the Grumpy Old Programmer phase!

Here we have a good knowledge of tools, but the issue that is different is that you will be using a Context Based thinking. You are looking at the problem you have got in front of you and fitting the tools to that problem. There is a strong link here with a practice when flying aircraft where you need to read from the ground to map, not the other way around. You must do it correctly because there have been a number of accidents where the pilots have read from the map to ground thus misidentifying their location.

It is the same with problem-solving. Focus on the problem, use the appropriate tools as you need them. It is interesting what Dr. Marian Petre said about how experts can seem as though they are novices – which is exactly what I feel like. Sometimes I look at my code and think "that doesn't really look that complicated". You bring out the 'big guns' when you need them, hopefully abstracted down under a good interface, but you know you need to keep the complexity down because there will be a lot of maintenance in the future, where you or others will have to reason about the code.

In this phase the software complexity is of the order of the problem complexity, perhaps a bit more because you will need a some 'slack' within the solution. At a personal level the major point here is that you are aware of your own limits because in the previous phase you have reached that panicked state.

One of the big things I have learnt through my career is the need to develop an inner strength and ability to handle this stressed state. For example there will be a bug. The client may panic. This is to be expected. The salesman may panic. Still possibly to be expected. As a developer if your manager panics too, you have a problem, because the buck will stop with you. Can you discipline your own thinking and your own practice so that you can calmly deal with the issue, regardless of how others are handling the situation? This is the struggle you can get in a commercial coding environment.

Implicit in this description is that you have developed a disciplined personal practice.

So in summary:

Novice

- Rule-based thinking
- Undisciplined
- Some self-awareness.

Journeyman

- Abstract thinking
- Undisciplined
- No (or very little) self-awareness.

Master

- Contextual thinking
- Disciplined
- Deep self-awareness.

References

[ACCU16] ACCU Conference 2016 https://accu.org/index.php/conferences/accu_conference_2016/accu2016_sessions#Balancing_Bias_in_Software_Development

[Tolman13] 'An Exploration of the Phenomenology of Software Development', https://www.slideshare.net/charlestolman/accu-2013-exploration-of-phenomenology-of-sw-development

[Tolman14] 'Imagination in Software Development' https://charlestolman.com/2014/04/21/accu2014-workshop-imagination-in-software-development/

[Wikipedia_01] Dreyfus model of skill acquisition, https://en.wikipedia.org/wiki/Dreyfus_model_of_skill_acquisition

[Wikipedia_02] Dunning-Kruger effect https://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect

A Usable C++ Dialect that is Safe Against Memory Corruption

Suitable allocators for (Re)Actors can speed things up. Sergey Ignatchenko continues his investigation in Allocator for (Re)Actors (Part 2).

We have this handy fusion reactor in the sky.
You don't have to do anything, it just works.
~ Elon Musk

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.

As we briefly discussed in Part I of this mini-series [NoBugs17], message-passing technologies such as (Re)Actors (a.k.a. Actors, Reactors, ad hoc FSMs, and event-driven programs) have numerous advantages, ranging from being debuggable (including post-factum production debugging), to providing better overall performance.

In [NoBugs17], we discussed an approach to handling allocations for (Re)Actors – and were able to reach kinda-safety at least in what we named 'kinda-safe' and 'safe with relocation' mode. Unfortunately, kinda-safety didn't really provide the Holy Grail™ of safety against memory corruptions. Now, we can extend our allocation model with a few additional guidelines, and as long as we're following these rules/guidelines, our C++ programs WILL become perfectly safe against memory corruptions.

#define (Re)Actors

To make this article self-contained and make sure that we're all on the same page with terminology, let's repeat the definition of what we're considering: (Re)Actors [NoBugs17].

Let's begin with a common denominator for all our (Re)Actors: a `GenericReactor`. `GenericReactor` is just an abstract class – and has a pure virtual function `react()`:

```
class GenericReactor {
  virtual void react(const Event& ev) = 0;
}
```

Let's define what we will refer to as 'infrastructure code': a piece of code which calls `GenericReactor`'s `react()`. Quite often this call will be within a so-called 'event loop' (Listing 1).

Let's note that the `get_event()` function can obtain events from wherever we want; anything from `select()` (which is quite typical for servers) to libraries such as *libuv* (which is common for clients).

Sergey Ignatchenko has 20+ years of industry experience, including being an architect of a stock exchange, and the sole architect of a game with hundreds of thousands of simultaneous players. He currently writes for a software blog (http://ithare.com), and translates from the Lapine language a 9-volume book series 'Development and Deployment of Multiplayer Online Games'. Sergey can be contacted at sergey.ignatchenko@ithare.com

Also let's note that an event loop, such as the one above, is certainly *not* the only way to call `react()`: I've seen implementations of infrastructure code ranging from one running multiple (Re)Actors within the same thread, to another which deserialized (Re)Actor from DB, then called `react()` and then serialized (Re)Actor back to a database. What's important, though, is that even if `react()` can be called from different threads, it MUST be called *as if* it is one single thread (if necessary, all thread sync should be done OUTSIDE of our (Re)Actor, so `react()` doesn't need to bother about thread sync regardless of the infrastructure code in use).

Finally, let's refer to any specific derivative from `GenericReactor` (which implements our `react()` function) as a `SpecificReactor`:

```
class SpecificReactor : public GenericReactor {
  void react(const Event& ev) override;
};
```

In addition, let's observe that whenever (Re)Actor needs to communicate with another (Re)Actor – adhering to the 'Do not communicate by sharing memory; instead, share memory by communicating' principle – it merely sends a message, and it is only this message which will be shared between (Re)Actors. In turn, this means that we can (and *should*) use single-threaded allocation for all (Re)Actor purposes – except for allocation of those messages intended for inter-(Re)Actor communications.

Rules to ensure memory safety

With (Re)Actors defined, we can formulate our rules to make our (Re)Actor code (`Reactor::react()` and all the stuff called from it) perfectly safe.

First, let's postulate that there are three different types of pointers in our program: 'owning' pointers, 'soft' pointers, and 'naked' pointers.

'Owning' pointers delete their contents in destructors, and within our rules, should comply with the following:

- an 'owning' pointer is a template, semantically similar to `std::unique_ptr<>`
- 'owning' pointers are obtained only from operator `new`
- copying 'owning' pointers is not possible, but moving them is perfectly fine
- there is no explicit `delete`; however, there *is* a way to assign `nullptr` to the 'owning' pointer, effectively calling `destructor`

```
std::unique_ptr<GenericReactor> r
  = reactorFactory.createReactor(...);
while(true) { //event loop
  Event ev = get_event();
    //from select(), libuv, ...
  r->react(ev);
}
```

Listing 1

our rules do NOT allow the creation of any pointers, unless it is a pointer to an existing on-heap object, or an on-stack object

and deleting the object. However, while the `destructor` will be called right away, implementation of our allocator will ensure that actual *freeing of the memory* will be *postponed* until the point when we're out of `Reactor::react()`. As we'll see below, it is important to ensure safety in cases when there is a 'naked' pointer to the object being deleted.

'Soft' pointers are obtained from 'owning' ones. Whenever we're trying to access an already deleted object via a 'soft' pointer (or create a 'naked' pointer from a 'soft' pointer which points to an already deleted object) – we are *guaranteed* to get an exception. 'Soft' pointers should comply with the following:

- a 'soft' pointer is also a template, somewhat similar to `std::weak_ptr<>`
- 'soft' pointers are obtained from an 'owning' pointer, or as a copy of an existing 'soft' pointer
- both copying and moving 'soft' pointers is ok
- 'soft' pointers can be implemented either using tombstones (with reference counting for the tombstones), or using the ID-comparison-based technique described in [NoBugs17].

'Naked' pointers are our usual C-style pointers – and are inherently very dangerous as a result. Apparently, we can still handle them in a safe manner, as long as the following rules are followed:

- our 'naked' pointers are obtained *only* from 'owning' pointers, from 'soft' pointers, or by taking an address of an existing on-stack object. This implies (a) that all pointer arithmetic is prohibited, and (b) that all casts which result in a pointer (except for `dynamic_cast<>`) are prohibited too.
- We *are* allowed to copy our 'naked' pointers into another 'naked' pointer of the same type; however, whenever we're copying a 'naked' pointer, we MUST ensure that the lifetime of the copy is not longer than the lifetime of the original pointer.

The most reliable way to enforce the 'lifetime is never extended' rule above is to say that all copying of 'naked' pointers is prohibited, except for a few well-defined cases:

- Calling a function passing the pointer as a parameter, is ok.

  NB: double-naked-pointers and references to naked pointers effectively allow to us to return the pointer back (see on returning 'naked' pointer below) – so assigning to such `*ptrs` should be prohibited.
- Creating an on-stack copy of a 'naked' pointer (initialized from another pointer: 'owning', 'soft', or 'naked') of is generally ok too.

On the other hand, the following constructs are known to violate the 'lifetime is never extended' rule, and are therefore prohibited:

- Returning 'naked' pointer(s). Instead, we'll need to return either the 'owning' or 'soft' pointer(s). Actually, if we think about it, we'll see that is not that much of a restriction. If we want to return a pointer to an on-heap object, 'soft' or 'owning' pointers are the way to go; and returning a pointer to our local stack is a Bad Idea™ anyway. This only leaves us with functions such as `strchr()`, which tend to return a pointer on an object which was passed to them as a parameter – but it is not difficult to find a different way to return this information (to implement an analogue of `strchr()` within our restrictions, we can always return an offset instead of the pointer).
- Assigning 'naked' pointers to members of on-heap objects (and any naked-pointer parameter *may* happen to point to the heap) is prohibited. This can be seen as a stronger version of our restriction from [NoBugs17], of '(Re)Actor state cannot have 'naked' pointers'; as an important side-effect which we'll rely on later, this means that as soon as we're out of `Reactor::react()`, there are no 'naked' pointers whatsoever.

Note that the respective lists of ways to create pointers are exhaustive; in other words: the ONLY way to create an 'owning' pointer is from operator `new` of the same type; the ONLY ways to create a 'safe' pointer is (a) from an 'owning' pointer of the same base type, or (b) as a copy of a 'safe' pointer of the same type; and the ONLY way to create a 'naked' pointer is from {'owning'|'soft'|'naked'} pointer as long as the 'naked' pointer doesn't extend the lifetime of the original pointer.

This implies prohibiting casting *to* pointers (and also prohibits C-style `cast` and `static_cast<>` with respect to pointers; however, implicit pointer casts and `dynamic_cast<>` are ok). Note that although casting *from* pointers won't cause memory corruption, it is not a good idea in general.

This also implies that assigning the result of `new` to anything except an 'owning' pointer is prohibited.

Implementations for both 'owning' and 'safe' pointers should take into account that their methods *may* be invoked after their destructor is called (see discussion in (*) paragraph below); in this case, we'll either guarantee that no pointer to a non-existing object will be returned, *or* (even better) will throw an exception.

Note that for the time being, we do NOT handle collections and arrays; in particular, we have to prohibit indexed dereferencing (`a[i]` is inherently dangerous unless we're ensuring boundary checks).

That's it – we've got our perfectly safe dialect of C++, and while it doesn't deal with arrays or collections, it is a very good foundation for further refinements.

Proof sketch

The formal proof of the program under the rules above is going to be lengthy and, well, formal, but a sketch of such a proof is as follows.

First, let's note that our rules do NOT allow the creation of any pointers, unless it is a pointer to an existing on-heap object, or an on-stack object (the latter is for 'naked' pointers only). *NB: if we also want to deal with globals, this is trivial too, but for the time being let's prohibit globals within (Re)Actors, which is good practice anyway.*

As a result, there is no risk of the pointer pointing somewhere where there was never an object, and the only risks we're facing are about the *pointers to objects which did exist but don't exist anymore*. We have two types of such objects: on-stack objects, and on-heap ones.

For on-stack objects which don't exist anymore:

- To start with, only 'naked' pointers can possibly point to on-stack objects
- Due to our 'the lifetime of a 'naked' pointer never extends' rule, we're guaranteed that a 'naked' pointer will be destroyed *not later* than the object it points to, which means that we cannot possibly corrupt memory using it.

For on-heap objects which don't exist anymore:

- 'owning' pointers are inherently safe (according to our rules, there is no way to delete an object while an 'owning' pointer still points there)
- 'soft' pointers are safe because of the runtime checks we're doing every time we're dereferencing them or converting them into a 'naked' pointer (and throwing an exception if the object they're pointing to doesn't exist anymore).
- 'naked' pointers to on-heap objects are safe because of the same 'the lifetime never extends' rule and because of the postponing the freeing of memory until we're outside `Reactor::react()`. Elaborating on it a bit: as we know that at the moment of conversion from an 'owning' pointer or a 'soft' pointer to a 'naked' pointer, the object did exist, and the memory won't be actually freed until we're outside of `Reactor::react()`, this means that we're fine until we're outside of `Reactor::react()`; and as soon as we're outside of `Reactor::react()`, as discussed above, there are no 'naked' pointers anymore, so there is no risk of them dereferencing the memory which we're going to free.

(*) Note that via 'naked' pointers, we *are* still able to access objects which have already had their destructors called (but memory unreleased); this means that to ensure safety, those objects from supporting libraries which don't follow the rules above themselves (in particular, collections) *must* ensure that their destructors leave the object in a 'safe' state (at least with no 'dangling' pointers left behind; more formally: there should be a firm guarantee that *any* operation over a destructed object cannot possibly cause memory corruption or return a pointer which is not a `nullptr`, though ideally it *should* cause an exception).

Phew. Unless I'm mistaken somewhere, it *seems* that we got our perfectly safe dialect of C++ (without collections, that is).

Enter collections

[Enter Romeo and Juliet]

Romeo: *Speak your mind. You are as worried as the sum of yourself and the difference between my small smooth hamster and my nose. Speak your mind!*

Juliet: *Speak YOUR mind! You are as bad as Hamlet! You are as small as the difference between the square of the difference between my little pony and your big hairy hound and the cube of your sorry little codpiece. Speak your mind!*

[Exit Romeo]

~ Program in The Shakespeare Programming Language

As noted above, collections (including arrays) are not covered by our original rules above. However, it is relatively easy to add them, by adding a few additional rules with regards to collections.

First, we will NOT use the usual iterators (including pointers within arrays); instead, we're using 'safe iterators'. A 'safe iterator' (or 'safe range') is a tuple/struct/class/… which contains:

- An {'owning'|'soft'|'naked'} pointer/reference to the collection
- An iterator (or range) within the collection pointed out by the pointer above

The second rule about collections is that *all* the access to the collections (including iterator dereferencing) MUST be written in a way which *guarantees* safety.

For example, if we're trying to access an element of the array via our 'safe iterator', it is the job of the `operator*` of our 'safe iterator' to ensure that it stays within the array (and to throw an exception otherwise).

This is certainly possible:

- For arrays, we can always store the size of the array within our array collection, and check the validity of our 'safe iterator' before dereferencing/indexing.
- Then, as all the `std::` collections are implemented either on top of single objects or on top of arrays, rewriting them in a safe manner is always possible based on the techniques which we already discussed.
- On the other hand, more optimal implementations *seem* to be possible for specific collections. As one example, `deque<>` can be implemented without following the rules discussed above *within its implementation*, and simply checking range of the iterator instead. In another example, tree-based collections can be optimized too.

This way, whenever we want to use such a 'safe iterator'/'safe range', first we'll reach the collection (relying on our usual safety guarantees for our {'owning'|'soft'|'naked'} pointers), and then the collection itself will guarantee that its own iterator is valid before dereferencing it.

Different approaches to safety in infrastructure code and Reactor code

20% of people consume 80% of beer

~ Pareto principle as applied to beer consumption

An observation (*) above, as well as the discussion about optimized collections, highlights one important property of our Perfectly Safe Reactors:

> *we can (and often SHOULD) have different approaches to safety of the `Reactor::react()` and the rest of the code.*

This dichotomy between infrastructure code and Reactor code is actually very important in practice.

Infrastructure code (including supporting libraries such as collections, etc.) is:

- written once – and then stays pretty much unchanged
- usually relatively small compared to the business-logic stuff
- called over and over
- often fits into the 5% of the code which takes 95% of the execution time

In contrast, (Re)Actor code:

- contains business logic, which has a tendency to be changed several times a day
- as with any business logic, its code base can be huuuuge
- most of this code is called only occasionally compared to the Infrastructure Code
- 90% of it is glue code, which very rarely causes any performance issues

As a result, we can observe that for small, never-changing, and performance-critical Infrastructure Code, it is both feasible and desirable to provide safe highly-optimized versions (which may or may not follow our rules above in the name of performance). On the other hand, for (Re)Actor Code, formal safety is usually much more important than bare performance. This is especially so as, in the case of our rules, the expected performance hit is pretty much negligible: the only two runtime checks we're doing happen at 'safe' pointer to 'naked' pointer conversion (or at 'safe' pointer dereferencing), and at collection accesses; neither of them is expected to be noticeable (except in some very performance-critical code).

Generalizing this point further, we can split our code base into a small performance critical part (which we'll handle without our safety rules, but which is small enough to be scrutinized in a less formal manner), and a large performance-agnostic part (which we'll handle according to the safety rules above); however, in practice, these lines will be usually very close to the lines between Infrastructure Code and (Re)Actor Code.

One important thing to keep in mind when writing those Infrastructure objects which are intended to be called from (Re)Actors is ensuring that they're safe even after their destructor is called (as discussed in the (*) paragraph above). On the other hand, if our object follows our safety rules above, this will be achieved automagically.

All our rules are very local, which enables automated checks

One further very important property of our safety rules is that

they're very local.

Indeed, *all* the rules above can be validated within the scope *of one single function*. In other words, it is possible to find whether our function *f()* is compliant with our safety rules using function *f()* and only function *f()*.

This not only allows for simple code reviews, but also means that this process can be automated relatively easily. Implementing such a tool is a different story (and it is still going to take a while) but is perfectly feasible (well, as long as we find a tool to parse C++ and get some kind of AST, but these days at least Clang does provide this kind of functionality).

As soon as such an automated check tool is implemented, development will become a breeze:

- We separate our code into 'safe' code and 'unsafe' code (usually, though not strictly necessary, along the lines of the `(Re)Actor::react()`).
- For 'safe' code, such an automated check tool becomes a part of the build
- As a result, as long as 'unsafe' code is not changed (i.e. only 'safe' code is changed) there can be no possible regressions which can cause memory corruptions.

While this is not a real 'silver bullet' (nothing really is – in fact, the safety of theoretically safe languages also hinges on the safety of their compilers and standard libraries), this approach is expected to improve memory safety of the common business-level code by orders of magnitude (and even if your code is already perfectly safe, this approach will provide all the necessary peace of mind with regards to safety).

Conclusion

That's pretty much it – we DID get a perfectly usable C++ dialect which is also 100% safe against memory corruption and against memory leaks. BTW, if necessary our approach can easily be extended to a more flexible model which relies on semantics similar to that of `std:shared_ptr<>` and `std::weak_ptr<>`; while I am not a fan of reference-counted semantics (from my experience, reference counting causes much more trouble than it is worth – and simplistic 'owning' pointers are more straightforward and are perfectly usable for millions of LOC projects) – it is perfectly feasible to implement shared ownership along the same lines as discussed above; the only substantial twist on this way is that as `std::shared_ptr<>` (unlike our model above) does allow for circular references and resulting memory leaks, we will probably need to detect them (which can be done, for example, by running some kind of incremental garbage collection at those points where we're waiting for the input, sitting outside of `Reactor::react()`).

Phew. BTW, as the whole thing is quite complicated, please make sure to email me if you find any problem with the approach above (while I'm sure that it is possible to achieve safety along the lines discussed above, C++ is complicated enough we *might* need another restriction or two on this method). ■

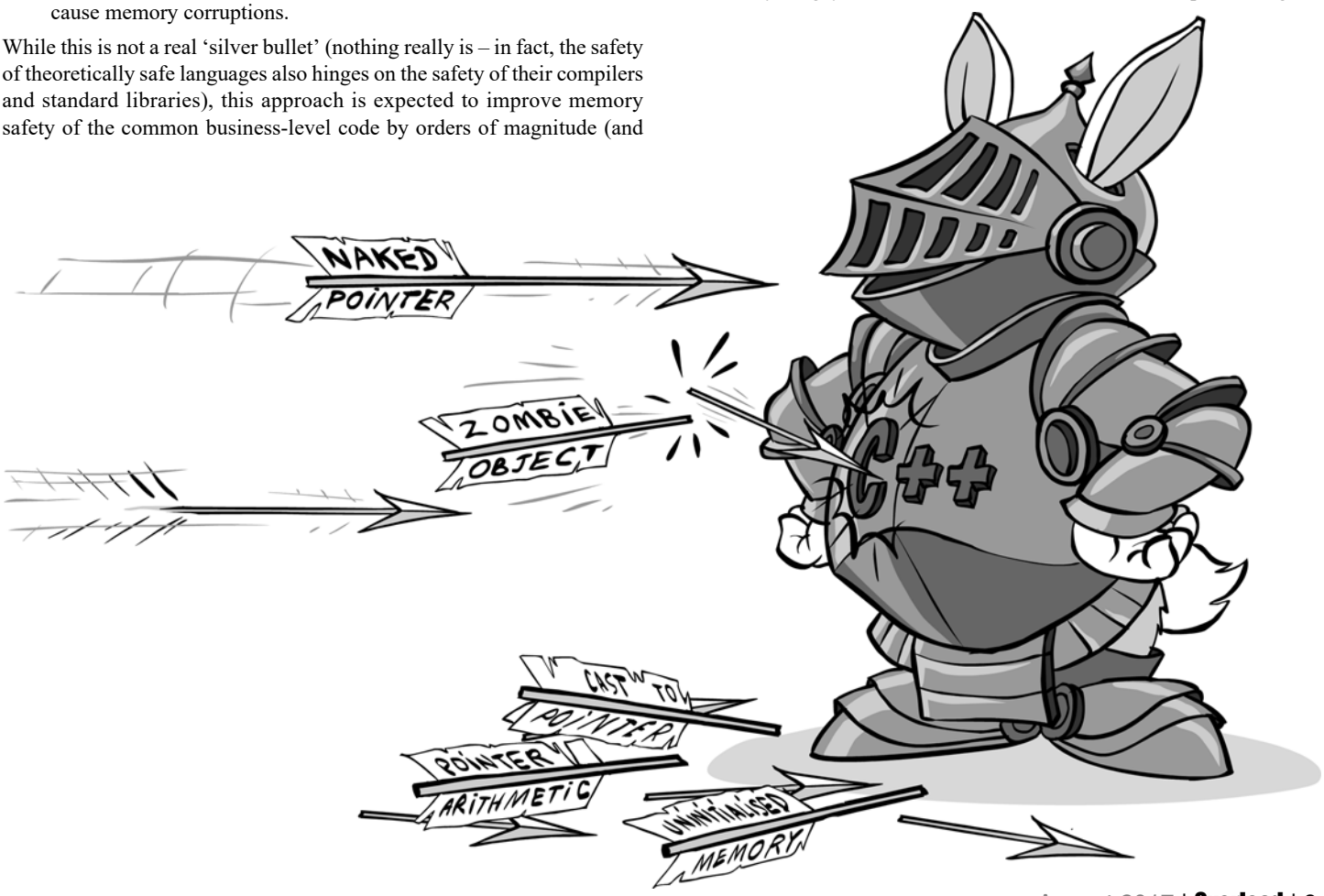
References

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

[NoBugs17] 'No Bugs' Hare, 'Allocator for (Re)Actors with Optional Kinda-Safety and Relocation', *Overload* #139, Jun 2017

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague



Metaclasses: Thoughts on Generative C++

Can you simplify C++ with minimal library extension? Herb Sutter shows how metaclasses could make this possible.

Herb recently blogged this note about a new ISO C++ proposal he and colleagues are working on, and we felt it would be of interest to Overload readers. Herb has kindly agreed to let us republish the blog post here as an article, and add the abstract from the current version of the proposal. -Ed.

I've been working on an experimental new C++ language feature tentatively called 'metaclasses' that aims to make C++ programming both more powerful and simpler. You can find out about it here:

- Current proposal paper: P0707R1 [Sutter17a]. I hope the first ten pages give a readable motivation and overview. (The best two pages to start with are 9 and 10, which probably means I need to reorder the paper…)
- Initial intro talk video: ACCU 2017 (YouTube) [Sutter17b]. This is the initial public presentation three months ago. Thank you to Roger Orr, Russel Winder, Julie Archer, and the other ACCU organizers for inviting me and for holding back the video until we could have the ISO C++ summer meeting in mid-July, so it could go live along with a report (herein) on the results of this feature's first presentation to the ISO C++ committee. And special thanks to Ina and Arvid, the two audience volunteers who graciously agreed to come on-stage to participate in a live mini UX study. There's a lot of subtle information in their nuanced reactions to the code examples; pay special attention when their responses are different or as their responses evolve.
- 'Incomplete and experimental' prototype compiler. The Clang-based prototype by Andrew Sutton is available as an online live compiler at cppx.godbolt.org, and as source at github.com/asutton/clang. It's incomplete but can compile a number of the examples in the paper (see the paper for example code links). Thanks to Matt Godbolt for hosting it on godbolt.org!

Please see the above paper and video to answer "what are metaclasses and why should I care?"

If you're the "show me code first, English later" kind of person, try the live compiler and these quick examples: interface, base_class, value (regular type), plain_struct (links are in the paper).

The rest of this article aims not to duplicate any information above, but to provide some context about the broader journey, and what I and others are attempting to accomplish.

A journey: Toward more powerful and simpler C++ programming

Phase 1: By using the existing language better

About five years ago, I started working on long-term effort toward making using C++ simpler and safer.

Herb Sutter is chair of the ISO C++ committee and a programming language architect at Microsoft, and has been the author or co-author of a number of C++ features.

Abstract from P0707 R1

The only way to make a language more powerful, but also make its programs simpler, is by abstraction: adding well-chosen abstractions that let programmers replace manual code patterns with saying directly what they mean. There are two major categories:

> **Elevate coding patterns/idioms into new abstractions built into the language.** For example, in current C++, range-**`for`** lets programmers directly declare "for each" loops with compiler support and enforcement. Templates are a powerful parameterization of functions and classes, but do not enable authoring new encapsulated behavior.
>
> **(major, this paper) Provide a new abstraction authoring mechanism so programmers can write new kinds of user-defined abstractions that encapsulate behavior.** In current C++, the function and the **`class`** are the two mechanisms that encapsulate user-defined behavior. In this paper, **`$class`** metaclasses enable defining categories of **`class`**es that have common defaults and generated functions, and formally expand C++'s type abstraction vocabulary beyond **`class`**/**`struct`**/**`union`**/**`enum`**.

Also, §3 includes a set of common metaclasses, and proposes that several are common enough to belong in **`std::`**. Each subsection of §3 is equivalent to a significant "language feature" that would otherwise require its own EWG paper and be wired into the language, but here can be expressed instead as just a (usually tiny) library that can go through LEWG. For example, this paper begins by demonstrating how to implement Java/C# **`interface`** as a 10-line C++ **`std::`** metaclass – with the same expressiveness, elegance, and efficiency of the built-in feature in such languages, where it is specified as ~20 pages of text.

In the first phase, a small group of us – centered on Bjarne Stroustrup, Gabriel Dos Reis, Neil MacIntosh and Andrew Pardoe – pushed to see how far we could get with 'C++ as it is' plus just a few well-chosen library-only extensions, with a particular goal of improving type and memory safety. Bjarne, Neil, and I first publicly reported on this effort in the two CppCon 2015 plenary sessions 'Writing Good C++14' [CppCon15a] and 'Writing Good C++14… By Default' [CppCon15b]. The results of that work so far have manifested as the C++ Core Guidelines [CCG] and its support library, GSL [GSL], that adds a limited number of library types (e.g., **`span`**, now being standardized); and I led the Lifetime design in particular (available in the Guidelines /docs folder) which Neil and I and others continue to work on formalizing with the aim of sharing a 'draft' static analysis spec later this year.

One of the goals of this phase was to answer the question: "How much progress can we make toward simplifying the existing C++ language with only a few key library extensions?" The answer as I see it turned out to be: "Some solid progress, but probably not a major simplification." And so that answer led to phase two…

Phase 2: By evolving the language

Two years ago, I started to focus specifically on exploring ways that we might evolve the C++ language itself to make C++ programming both more powerful and simpler. The only way to accomplish both of those goals at the same time is by adding abstractions that let programmers directly express their intent – to elevate comments and documentation to testable code, and elevate coding patterns and idioms into compiler-checkable declarations. The work came up with several potential candidate features where judiciously adding some power to the language could simplify code dramatically.

Of those potential candidate features, metaclasses is the first major piece I picked to propose for ISO C++. We presented it for the first time at the summer ISO C++ meeting earlier this month, and it received a warm reception. [1]

There was (rare) unanimous support for pursuing this kind of capability, but also some concern about how best to expose it and specific design change feedback the committee wants us to apply to improve the proposal. [2]

We'll work to include in a revision for the November standards meeting as we start the multi-year process of vetting and refining the proposal. So this is good progress, but note that it (only) means encouragement to continue the experiment and see where it leads; it's far too early to talk about potential ship vehicles.

So do expect change: The proposal is still evolving, and it in turn assumes and builds on the static reflection proposal (P0578 et al.) and the compile-time programming proposal (P0633), both of which are actively evolving in their own right. Incidentally, one of the contributions of Andrew Sutton's prototype metaclasses compiler is that it is implementing those other proposals too(!), since the metaclasses feature needs them. The aim is to keep the latest compiler and the latest P0707 paper in sync with each other and with those related proposals, but there will doubtless be occasional drift in between syncs.

What's next

I'll talk about metaclasses more in my upcoming CppCon 2017 talk this September, and Andrew Sutton will also be giving two CppCon talks about metaclasses – one about implementing them in Clang, and one about using them for a real project.

This is just the beginning, and we'll see whether it all pans out and leads somewhere, but I hope you enjoy this exploration and I look forward to talking with many of you about it at CppCon this September. ■

Notes

1. I actually brought a smaller piece from this same work to the committee at the previous meeting, the winter meeting in Kona: P0515 (consistent comparisons), which proposes adding the three-way <=> comparison operator. P0515 is only about a minor feature, and not one of the most important things that can help improve C++, so normally I wouldn't have picked that piece to contribute first; but the committee was already continuing to actively discuss comparisons, so I cherry-picked it from my design work and contributed it since I had the design in my pocket anyway. Happily the committee liked what they saw and both EWG and LEWG accepted it, and it is now progressing well and on track to hopefully be voted into draft C++20 in the next meeting or two. Thanks to Jens Maurer and Walter Brown for the heavy lifting of writing the core language and library standardese wording, respectively, for that P0515 proposal.
2. The committee's design feedback was primarily about how to wrap up the transformation code: Instead of putting it inside a new 'meta' class-like abstraction, how about wrapping the same code inside a compile-time function-like abstraction that takes an input meta::type parameter and returns a generated meta::type return value? This doesn't affect the proposal's basic engine, just the shape of its steering wheel – for example, we could change the first line of each example metaclass definition from the class-like syntax

```
$class interface {
   constexpr {
     // … basically same code …
   }
};
```

to the decorator-function-like syntax

```
meta::type interface(const meta::type source) {
  // … basically same code …
};
```

where the latter has the advantage that it's easy to see that we're reading one type and generating another type. Interestingly, I think this dovetails with the mini UX study in the video where most of the difficulty the UX participants seemed to encounter was in understanding the $class syntax, not the metaclass bodies and not later using the metaclasses to author new types.

But we'll explore this and other options and validate/invalidate it with more experiments… and feel free to drop me a line (or comment on the original blog post [Sutter17c]) if you like one of these styles better, or perhaps another variation.

References

[CCG] C++ Core Guidelines http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[CppCon15a] https://www.youtube.com/watch?v=1OEu9C51K2A

[CppCon15b] https://www.youtube.com/watch?v=hEx5DNLWGgA

[GSL] GSL: Guideline support library http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-gsl

[Sutter17a] Metaclasses: Generative C++ P0707 R1 proposal paper at https://herbsutter.files.wordpress.com/2017/07/p0707r1.pdf

[Sutter17b] Metaclasses. Goal: Making C++ more powerful, and simpler at https://www.youtube.com/watch?v=6nsyX37nsRs&feature=youtu.be

[Sutter17c] Original blog post: https://herbsutter.com/2017/07/26/metaclasses-thoughts-on-generative-c/?platform=hootsuite

A C++ Developer Sees Rustlang for the First Time

Rust claims to run blazingly fast, prevents segfaults, and guarantees thread safety. Katarzyna Macias provides an introduction for a C++ developer.

I decided to learn a new modern language – the more exotic the better. But to be honest, it's not hard to impress me because throughout my studies and career I have only had contact with the most mainstream languages, like C, C++, Java, Python and Javascript.

At first, I planned to choose between Haskell, Clojure and Scala, but then I made a Twitter survey and I got several recommendations to try Rust. After a quick look at some examples … it looks weird enough. I'll take it!

I liked it at first sight: Rust may be the language that the world was waiting for! (Yes, I know – it is too early for me to have this much enthusiasm).

The documentation says:

> Rust is a systems programming language focused on three goals: safety, speed, and concurrency. It maintains these goals without having a garbage collector.

Looks great: it doesn't have a garbage collector and that gives it a big advantage over the languages that do have GC. It also introduces a new level of safety. It has a lot of other assets that I don't know about yet.

I don't think it will replace C++, because I don't think that any language will be able to do that within 10–20 years (don't blame me if I'm wrong). However, it may well become a proud neighbour of C++ on the shelf reserved for the most beautiful programming languages.

A first look at Rust

These are the things that caught my attention when I started to learn about Rust.

Variables are immutable

In Rust, every 'variable' – which is called a binding – is immutable by default. The binding declaration is shown in the example below. You can't reassign the value of `x` since `x` is `const`!

```
fn main ()
{
  let x = 5;
  x = 3; // This won't compile!
}
```

To make a binding mutable, add `mut` keyword: `let mut x = 5`.

This is different from C++, where you need to add the additional word to make a variable immutable. And I think the Rust solution is better. It's easy to forget or to skip the 'unnecessary' keyword out of laziness. Here, forgetting gives you the less-risky default.

Katarzyna Macias lives in Wroclaw, Poland. She works as a C++ software developer since 2013. Her main interests are new language features and telecommunication. Contact: kasia.macias@gmail.com

I have even heard a rumour that Bjarne (Stroustrup) once said that he would like to have such 'inverted const logic' in C++. But did he really say that? I don't know.

You can't use an uninitialized binding

Another safety improvement is the fact that you get a compilation error when you try to use an uninitialized binding. I also appreciate this feature. Why would anybody want to use uninitialized variables? It's an obvious error, so why not stop compilation when it happens?

Formatting correctness checked at compile time

The next protection for the careless developer: if you've ever had a crash in your application because you used the wrong number of arguments in your print function, you will value this. I have had this kind of problem and it was really hard to detect all the places in the code where the print was misformatted (our compiler gave no warnings for that – GCC fortunately does). To make it worse, the crashes were only sporadic. I would really prefer that the code would not compile as soon as the problem was introduced. And so it is in Rust: it won't allow you to compile incorrect formatting.

```
fn main ()
{
  // This won't compile!
  println!("The value is: {}");
}
```

Function declaration order does not matter

You can call functions before you declare them. This code is correct:

```
fn main ()
{
  print_number(5);
}

fn print_number(x: i32)
{
  println!("x is: {}", x);
}
```

This goes against the expectations of a C++ developer, but perhaps that is not important.

Returning without a return statement

This one is weird. The example below shows how to return a value from a function:

```
fn add_one(x: i32) -> i32 {
  x + 1
}
```

No semicolon, no `return` keyword… And a strange-looking arrow. It doesn't look very friendly at first sight.

If you want to ask how to return early… Good question. Here you should use **return**:

```
fn foo(x: i32) -> i32 {
  return x;

  // we never run this code!
  x + 1
}
```

Further, from the Rust book, we learn that:

> Using a return as the last line of function works, but is considered poor style.

Hmm. For me this syntax is odd, but I won't be discouraged by that.

My impressions of Rust

I feel very positive about Rust and I consider the additional safety as a great feature. I think the more checks that are done during the compilation time the better, as long as the compilation time doesn't exceed reasonable limits. Remember that the time you spend on waiting for your compilation may save you long hours of debugging.

I feel very curious about the possibilities for this language and I'm excited to learn more! ■

Portable Console I/O via iostreams

Portable streaming is challenging. Alf Steinbach describes how his library fixes problems with non-ASCII characters.

My Boost licensed **stdlib** header library [stdlib] applies some crucial fixes to the C++ implementation's standard library, and provides a (hopefully) complete set of wrapper headers that apply these fixes; some functionality used internally in the **stdlib** implementation; and a number of convenience headers for the standard library.

The most important fix, because it enables portability and reasonable functionality for beginners' programs, is of **char**-based text iostreams (e.g. **cout**) console i/o in Windows. **stdlib** installs special buffers in the standard iostreams that are connected to the console, and these buffers provide an UTF-8 view of the console. That means that portable ordinary **char** and **std::string** based code can present e.g. Norwegian and Russian text in the console, via **cout**, and can input international text from the user, via **cin**.

stdlib also provides an UTF-16 view of the console for **wchar_t** based i/o via the wide iostreams, such as **wcout**.

The UTF-16 view was functionality that essentially came for free, because it was base functionality needed for the UTF-8 view, and it means that in addition to supporting portable **char** based code **stdlib** also supports **wchar_t**-based pure Windows programs.

Here I discuss only this ***portable console i/o*** aspect of **stdlib** – the other **stdlib** stuff is also nice, but is not as significant.

```
[C:\my\dev\libraries\stdlib\writings\overload\code]
> type n\data\norwegian_russian_name.utf-16.txt
Pål Аркадий Jørgen Sæther

[C:\my\dev\libraries\stdlib\writings\overload\code]
> cl greeting.textbook_version.cpp /utf-8 /Fe"unfixed-textbook-hi"
greeting.textbook_version.cpp

[C:\my\dev\libraries\stdlib\writings\overload\code]
> chcp 65001 & unfixed-textbook-hi
Active code page: 65001
Hi, I'm the 日本国 кошка, what's your name? Pål Аркадий Jørgen Sæther
Pleased to meet you, P l         J rgen S ther!          ← Uh oh.
```

Figure 1

```
[C:\my\dev\libraries\stdlib\writings\overload\code]
> cl greeting.textbook_version.cpp /FI"stdlib/fix/console_io.hpp"
/Fe"textbook-hi"
greeting.textbook_version.cpp

[C:\my\dev\libraries\stdlib\writings\overload\code]
> textbook-hi
Hi, I'm the 日本国 кошка, what's your name? Pål Аркадий Jørgen Sæther
Pleased to meet you, Pål Аркадий Jørgen Sæther!     ← OK.
```

Figure 2

Goal: portable console i/o

The main goal with **stdlib** was to enable simple textbook style console based exploratory C++ programs, like the example in Listing 1.

```
#include <iostream>
#include <string>
using namespace std;

auto main()
  -> int
{
  cout << "Hi, I'm the 日本国 кошка, what's your name? ";
  string name;
  getline( cin, name );
  cout << "Pleased to meet you, " << name << "!" << endl;
}
```

Listing 1

A student should be able to type in his or her own non-English name into this program, and see it accurately presented back by the program, *also in Windows*. This goal is accomplished, modulo the Windows console windows' restriction to the BMP[1] part of Unicode.

Without a console i/o fix applied, Visual C++'s runtime library forwards the nullbytes that a Windows console window in UTF-8 mode (codepage 65001) produces for non-ASCII characters, i.e. yielding a **name** string with embedded nullbytes, which in the console window's presentation leaves blank areas (see Figure 1).

Using the Visual C++ 2017 compiler **cl** in Windows 10 and applying the **stdlib** i/o fix via the **/FI** option for a forced include gives the output in Figure 2.

This correct result is independent of the console window's active codepage, and is the same in the *nix world.

The **stdlib** i/o fix includes a convenience **#pragma** for Visual C++, setting the execution character set to UTF-8, for otherwise the execution

Alf Steinbach learned Basic and some 8080 assembly back in 1980. He's been a senior consultant with Kantega and Accenture, a lecturer at Nordland University, a vocational teacher, has contributed to various C++ FAQs, and helps administer the Facebook group 'C++ Enthusiasts'. He was awarded Microsoft's MVP in Visual C++ in 2012. Contact: alf.p.steinbach@gmail.com

1. The BMP, the *Basic Multilingual Plane*, is Unicode restricted to 16 bits, like in Unicode version 1 in 1991/1992. The 21-bit version 2 came in 1996. By that time Microsoft had committed to 16-bit Unicode. Unicode 2's UTF-16 encoding was designed to allow the existing 16-bit Unicode systems (various programming languages, + Windows) to just keep on working; a backward-compatible encoding. So most of Windows uses full UTF-16, but Windows console windows have a non-streaming API that restricts each character position to 16 bits. Hence if you output an UTF-16 surrogate pair (representing a Unicode code point outside the BMP, e.g. an emoji or an archaic Chinese glyph) to a Windows console window, you get two characters displayed, probably as "I didn't understand that" squares.

In Windows, the limited byte streams are second or third class citizens, not the primary way to interact with consoles

character set would have had to be specified explicitly as UTF-8 in every compilation, like the **`/utf-8`** option in the first compiler invocation above. Visual C++ defaults to Windows ANSI encoding, which depends on the locale Windows is installed for. With g++ the execution character set default is already UTF-8.

The technical problem(s)

I hate to hear 'Less is more.' It's a crock of crap.
~ R. Lee Ermey, American soldier and movie star of *Full Metal Jacket* [Ermey]

The C and C++ standard libraries' unified view of console, pipe and file i/o as minimalist streams of bytes, works fine in the *nix world where C and C++ originated. But Windows is based on different ideas, ideas of more rich standard functionality – much richer standard functionality. And so, in Windows the limited byte streams are second or third class citizens, not the primary way to interact with consoles: the streams are evidently there as backward compatibility support for archaic pre-Unicode programs, because UTF-8 console input Just Doesn't Work™ for non-ASCII characters.

So, what happens if you tell a Windows console window to use UTF-8 encoding, by setting its active codepage to 65001?

As of Windows 10 byte stream output appears to work, but, down at the Windows API level, byte stream input of non-ASCII characters produces just nullbytes, as illustrated by a program that directly uses Windows' **`ReadFile`** and **`WriteFile`** functions (see Figure 3).

Additionally, Visual C++'s **`setlocale`** in Windows [Microsoft-a] explicitly does not support UTF-8. A possible reason is the C standard's requirement that a **`wchar_t`** "can represent distinct codes for all members of the largest extended character set specified among the supported locales" [C99]. For Windows' **`wchar_t`** type, from the early Unicode adoption, is just 16 bits, which with modern 21-bit Unicode is not enough for all members of an UTF-8 locale.

And in addition to the limited Windows support for UTF-8 in consoles, the C and C++ standard libraries fail to support UTF-8 text handling. There is no functionality for iterating over code points (which can be of a variable number of bytes); the functionality for **`char`** classification, such as the C library's **`isupper`**, only works for single bytes, i.e. when the UTF-8 character is in the ASCII subset; the C++ library's **`std::ctype::widen`**, which can deal with a string of encoding units, is rendered impotent for portable code by the fact that there's no UTF-8 locale in Windows, so there's no way to tell it that those bytes are UTF-8 encoded text; and so on, and on. AFAIK there's no solution that addresses all the issues.

However, the lack of C++ standard library support was not a showstopper for the *nix world's transition to UTF-8. In the late 1990s and early 2000s one simply let existing tools treat UTF-8 as extended ASCII text with occasional pass-them-right-through-please hey just ignore them high value bytes. Today, as of 2017, the *nix world appears to be all UTF-8 for

```
[C:\my\dev\libraries\stdlib\writings\overload\code]
> cl greeting.windows-api.cpp /Fe"hi_windows" /execution-charset:utf-8
greeting.windows-api.cpp

[C:\my\dev\libraries\stdlib\writings\overload\code]
> chcp 65001
Active code page: 65001                    ← UTF-8 codepage.

[C:\my\dev\libraries\stdlib\writings\overload\code]
> hi_windows
Hi, I'm the 日本国 кошка, what's your name? Pål Jørgen Sæther
19 bytes read: | 80 P | 0 □ | 108 l | 32   | 74 J | 0 □ | 114 r | 103
g | 101 e | 110 n | 32   | 83 S | 0 □ | 116 t | 104 h | 101 e | 114 r
| 13 □ | 10 □
Pleased to meet you, P!                   ← Uh oh.
```

Figure 3

text files, so that approach worked, and hence it can presumably also work for Windows.

Possible solutions

The missing functionality for text handling is offered by various 3rd party libraries, including IBM's open source ICU library [ICU], and Boost Locale, which is a **`char`**-based wrapper over ICU. The Boost Locale documentation notes that "The default character encoding is assumed to be UTF-8 on Windows" [Boost-a]. So evidently, an assumption of UTF-8 as the main text encoding on every platform, including in Windows, is not unheard of.

A mainly all UTF-8 approach for external text and for simple processing, with conversion to and from UTF-16 for e.g. use of ICU, seems to be where we're heading, also for Windows programs.

Anyway, to work with international text in Windows consoles, especially for beginners, it's practically necessary to

- change the default font for Windows console windows[2] to one that can display international characters, such as Lucida Console, or else use 3rd party console windows.

With that display fix in place one basically has three options for portable C++ code:

- use byte stream i/o with some fix applied in Windows, e.g. the standard library's byte streams with a restricted character set from a national codepage, or with conversion to/from internal UTF-8 such as provided by **stdlib**;
- use wide stream i/o (note: the standard library's wide stream i/o converts to and from external byte streams) with some platform-dependent fix applied, e.g. in Windows, using the standard library's wide streams with Microsoft's **`_setmode`** extension [Microsoft-b], or again using **stdlib**, and in the *nix world, with a suitable UTF-8 locale; or

2. To change the default font for a Windows console window, just right click the window title for a menu, and drill down into it

there seems to be no portable non-intrusive way to fix the encoding of the arguments of main in Windows

- use an abstraction that transparently adapts the encoding to the system, selecting between byte and wide stream i/o within the implementation of that abstraction, with an encoding unit type suitably defined for each system.

Some years ago, I saw adaptive encoding and i/o as a viable compromise between conflicting goals [Steinbach13].

One main problem with that approach, however, is that it's necessarily intrusive, e.g. requiring string literals wrapped in adaptive macro calls like `S("Hi")` and use of standard streams via adaptive references like `sys::out` for `std::cout`, so that

- the approach can't handle simple textbook example program code as-is, and hence
- existing code doesn't automatically benefit.

This is what **stdlib** addresses with its UTF-8 console i/o: it can handle textbook example program code as-is, and if existing code uses the C++ iostreams, then that code benefits automatically.

In contrast the **nowide** library [nowide], adopted in Boost [Boost-b] in June 2017, is an intrusive UTF-8 i/o approach, and thus, except that it handles ordinary narrow literals, it suffers from the drawbacks above.

The **nowide** web page refers to a 2011 blog posting of mine [Steinbach11] about Unicode in Windows console windows, which, incidentally, is how I became aware of **nowide**, some time after I started work on **stdlib**. In that article, I argued for leveraging Microsoft's `_setmode` extension, using wide text internally in the C++ program, and I referred to a 2008 blog posting by Microsoft's Unicode guru Michael Kaplan, titled 'Conventional wisdom is retarded, aka What the @#%&* is `_O_U16TEXT`?' [Kaplan08]. Both **stdlib** and **nowide** now go in the opposite direction, using narrow text internally in C++.

General comparison: adaptive versus stdlib versus nowide

The C++ core language is involved in two areas: string literals and process command line arguments, namely the arguments of `main`. Happily, with the all UTF-8 approach of **stdlib** and **nowide**, and with modern compilers' (especially now Visual C++'s) support for UTF-8 as the execution character set, one can just use ordinary narrow literals. Unfortunately, there seems to be no portable non-intrusive way to fix the encoding of the arguments of `main` in Windows, and so both libraries provide intrusive, portable means of obtaining UTF-8 encoded command line arguments.

Apart from that the **stdlib** library is based on only providing transparent *fixes* to the standard library implementation, and a minimum of new functionality, while the adaptive approach and the **nowide** library are based on providing *alternatives* to the core language and standard library in certain areas.

With **stdlib**'s goal of providing as little new functionality as possible, checking which of **stdlib** and other libraries provide the most features, would be mostly meaningless. But one can still compare general goals or ideals achievement for the libraries. For the adaptive approach, the table below just lists what will be generally true of any reasonable implementation of that approach.

```
#include <nowide/iostream.hpp>
#include <string>
using namespace std;

auto main()
  -> int
{
  nowide::cout << "Hi, I'm the 日本国 кошка, what's your name? ";
  string name;
  getline( nowide::cin, name );
  nowide::cout << "Pleased to meet you, " << name << "!" << endl;
}
```

Listing 2

Goal/ideal	Adaptive	stdlib	nowide
General			
Working narrow Unicode console i/o	n/a	Success	Partial
Working wide Unicode console i/o	n/a	Success	Failure
That it fails gracefully for bad data	-	Success	Failure
Support of coding			
Idiomatic `char` based learner's C++	Failure	Success	Success
No `<windows.h>` namespace pollution	-	Success	Success
Few or no explicit encoding conversions	Partial	Failure	Failure
Using textbook example code as-is	Failure	Mostly	Failure
Automatic benefit for existing code	Failure	Mostly	Failure
Support of building & other tool usage			
No large 3rd party library dependency	-	Success	Success
Header only library	-	Success	Failure
Tools, e.g. string display in debuggers	Success	Failure	Failure
Clean build with common compilers	-	Success	Failure

My 'partial' mark on **nowide**'s working is mainly due to its failure to remove carriage return characters from input in Windows (Listing 2). The result is in Figure 4.

This problem, plus a ditto problem with Windows' convention of using Ctrl Z as EOF marker, has probably already been fixed by the time you're reading this. But I was perplexed to discover that the library bungled input, which is so fundamental to what it's all about, after it had been approved for Boost. It's really strange.

```
Hi, I'm the 日本国 кошка, what's your name? Pål Аркадий Jørgen Sæther
!leased to meet you, Pål Аркадий Jørgen Sæther      ← Uh oh.
```

Figure 4

Both stdlib and nowide assume that main arguments on other platforms than Windows are UTF-8 encoded

```
#include <stdlib/all/basics.hpp>
using namespace std;
using stdlib::process::Command_line_args;

void cppmain( Command_line_args const& args )
{
  cout << args.size() << " command line arguments:\n";
  for( int i = 0; i < args.size(); ++i )
  {
    cout << setw( 2 ) << i << ": “" << args[i] << "”.\n";
  }
}

auto main()
  -> int
{ cppmain( Command_line_args{} ); }
```

Listing 3

With Visual Studio's debugger in Windows one can use the format specifier **`,s8`** on a watch of a raw C string to force UTF-8 interpretation of the bytes. However, with other presentations of narrow strings the VS debugger uses Windows ANSI, even when the program's execution character set is UTF-8, with gobbledygook as the result. This is the main tool support failure of **stdlib** and **nowide**, and it's one area where the adaptive approach would shine.

Hopefully, in the not distant future the Visual Studio debugger will gain some option to assume UTF-8, or maybe it will just pick up what the program's execution character set is, not to mention encoding information for each literal, and use that.

stdlib's not quite 100% success in supporting textbook example code is due to the following constraints:

- automatic conversion to/from internal UTF-8 for console i/o seems to not be portably possible for C **`FILE*`** i/o, and
- with both Visual C++ and MinGW g++ the arguments of **`main`** are (incorrectly) Windows ANSI-encoded even when the execution character set is UTF-8, and a transparent automatic fix appears to not be practically possible.

Command line arguments in stdlib versus nowide

Both **stdlib** and **nowide** assume that **`main`** arguments on other platforms than Windows are UTF-8 encoded. In Windows, they both use the **`GetCommandLineW`** API function to obtain the original UTF-16 encoded command line passed to the process, and **`CommandLineToArgvW`** to parse it into individual arguments. **stdlib** uses this info to provide a separate set of UTF-8 encoded original command line arguments, while **nowide** uses the info to replace the **`main`** arguments with UTF-8 encoded originals.

The intended default usage in **stdlib** (and what I hope for in some future C++ standard library support for this) is that a **`Command_line_args`** object should be default-constructed wherever command line arguments are needed, which supports use in e.g. the constructor of a namespace scope variable, or in some other function without access to the actual **`main`** arguments.

As of July 2017, default construction of **`Command_line_args`** is implemented only for Windows and Linux, but code that only needs to be portable to these two systems can look like Listing 3.

This can be made fully portable by replacing the **`main`** code with Listing 4... which, however, is not possible for the mentioned case of constructor for a namespace scope variable (without employing a time machine to check what the future call of **`main`** will have).

The **nowide** library offers only this latter restricted approach of passing the actual **`main`** arguments to a fixer object (see Listing 5).

Using the *nix world convention of representing the command line arguments as an **`int`** + **`char**`** pair makes it easy to use library functions based on that convention, such as **`getopt`**. With **stdlib** the **`Command_argv_array`** class offers this value pair. A key difference is that an instance of **stdlib**'s **`Command_argv_array`** is a copy of the argument string data, so that the data can be freely modified.

```
auto main( int const n, char** const arg_pointers )
  -> int
{ my::cppmain( Command_line_args::from_os_or_else_from( n,
   arg_pointers ) ); }
```

Listing 4

```
#include <nowide/args.hpp>
#include <nowide/iostream.hpp>
#include <iomanip>
using namespace std;

#ifdef _MSC_VER
#  pragma comment( lib, "shell32.lib" )
#endif
void cppmain( int const n_args, char** const args  )
{
  nowide::cout << n_args
               << " command line arguments:\n";
  for( int i = 0; i < n_args; ++i )
  {
    nowide::cout << setw( 2 ) << i << ": “"
                 << args[i] << "”.\n";
  }
}
auto main( int n, char** arg_pointers )
  -> int
{
  nowide::args a( n, arg_pointers );
  // At this point `n` and `arg_pointers` have been modified,
  // if necessary, so that they're now (1) UTF-8 and (2) in
  // Windows, unexpanded wildcards.
  cppmain( n, arg_pointers );
}
```

Listing 5

Neither stdlib nor nowide provide dedicated wildcard expansion functionality, but stdlib offers portable access to the C++17 filesystem library

```
auto main()
  -> int
{
  stdlib::process::Command_line_args const args;

  namespace fs           = std::filesystem;
  fs::path const
    program_path         = fs::absolute( args[0] );
  fs::path const
    program_folder_path = program_path.parent_path();
```

Listing 6

Note: with MinGW g++ and **nowide** the value of **n** above *can be reduced* by the declaration of the **nowide::args** variable, because MinGW g++ provides wildcard expansion of arguments, and the synthesized UTF-8 encoded arguments are not expanded.

Neither **stdlib** nor **nowide** provide dedicated wildcard expansion functionality, but **stdlib** offers portable access to the C++17 filesystem library, which combined with some regular expression matching can do the chore. However, that's quite complex machinery. E.g. with normal Windows filename wildcards a * doesn't match backward slashes (which a regular expression simple **.*** pattern does), and one has to deal with absolute and relative paths. I think wildcard expansion functionality properly belongs with the iteration ability of the filesystem library, and not with mainly a console i/o fix library. Alas, the filesystem library does not yet offer this functionality.

Using the C++17 filesystem library

Sometimes an executable has associated files such as configuration files and resource files, placed in the directory that itself resides in, or in some sub-directory there. Thus, sometimes one needs a path to the executable's directory. The 'current directory', the default origin for relative paths, can be and often is some other directory. Usually the current directory is initially the directory from which the program was launched this time, i.e. some arbitrary directory, anywhere. Since the current directory is used automatically, client code does not usually need its path for e.g. resolving command line filename arguments. But client code does, in general, need the path to the executable's directory.

However, the C++17 filesystem library

- provides the generally not needed current directory path, *fs*`::current_path()` – where *fs* denotes `std::filesystem` – and
- does not provide the often crucial executable's directory path.

Happily, the first process command line argument, the first argument of `main`, is in practice a relative or absolute path to the executable. This is not formally guaranteed, but in practice it's nearly always so. Ideally then, to determine a path to the executable's directory, code like this should be sufficient (see Listing 6).

But run the program from a directory where the relative path to the executable's directory contains non-ASCII characters[3], and then this

```
[C:\my\dev\libraries\stdlib\writings\overload\code]
> π\кошка
!Unable to open "C:\my\dev\libraries\stdlib\writings\
overload\code\ÏЄ\data\blueberry-π.txt".
```

Figure 5

```
auto main()
  -> int
{
  stdlib::process::Command_line_args const args;

  namespace fs = std::filesystem;
  fs::path const program_path =
    fs::absolute( fs::u8path( args[0] ) );
  fs::path const program_folder_path =
    program_path.parent_path();

  // Here "df" is short for "data file".
  auto const df_path =
    program_folder_path / "data" /
    fs::u8path( "blueberry-π.txt" );
```

Listing 7

simple, natural and (assuming the first argument of `main` actually refers to the executable) formally correct code, fails (Figure 5).

What's going on here?

Running from the executable's directory would work because with this code the name of the executable, passed to *fs*`::absolute()`, is then effectively a dummy – any filename-like string would do.

But running it from the parent directory involves a non-ASCII character, π, in the path, which is served correctly, as UTF-8, to *fs*`::absolute()`. Here things go haywire because, as of July 2017, the Visual C++ and MinGW g++ implementations of the C++17 filesystem library *ignore* the execution character set and instead assume that narrow strings are and should be Windows ANSI encoded… Since Windows ANSI is a country-specific encoding choice the result **ÏЄ** can even be different on other machines.

It's trivially easy to check if the execution character set is UTF-8, and these implementations lay down the rules from scratch, with no frozen history constraining them. So, as I see it, the behaviour is really not excusable. Unfortunately, as far as I know there's no way that **stdlib** can fix this functionality transparently.

Until all common implementations of the C++17 filesystem library conform to the standard one therefore has to be very careful about always explicitly specifying UTF-8 in code using the filesystem library, by e.g. using the *fs*`::u8path` factory function (see Listing 7).

3. Using the name "cat", expressed as Russian "кошка", for an executable that lists the contents of a multi-language text file, is a weak pun. It was the best I could do.

the continued existence of this fundamental level failure of the filesystem library implementations, so very far into the game, appears perplexing, bewildering, inexplicable

… and the other way by using e.g. the *fs*`::path::u8string` conversion function:

```
string const dfp_utf8 = df_path.u8string();
```

In the first example `"data"` contains only ASCII characters and can therefore be served raw to the filesystem machinery, but `"blueberry-π.txt"` is decidedly non-ASCII so that it must be manually tagged as Unicode via a call to *fs*`::u8path`.

As with the **nowide** library's incorrect console input operation in Windows, the continued existence of this fundamental level failure of the filesystem library implementations, so very far into the game, appears perplexing, bewildering, inexplicable. But hopefully both the Visual C++ and the MinGW g++ implementations will be fixed. And, as Jerry Pournelle used to put it, Real Soon Now™.

The workarounds, the extra care and explicitness, is all that's needed with Visual C++. However, with MinGW g++ 7.1 and earlier the workarounds run into another filesystem implementation bug. For the MinGW g++ 7.1 implementation of *fs*`::u8path` can only handle UTF-16 encoded wide strings…

Happily, **stdlib** provides a transparent fix for that.

But, that fix must be explicitly requested, by defining `STDLIB_FIX_GCC_U8PATH`, because it's function template specializations that at least in theory won't necessarily build for a later or earlier version of the compiler, though this code may still work and may be necessary also for such versions. (See Figure 6.)

In passing: internally this fix uses `stdlib::wide_from_utf8` and `stdlib::utf8_from`, which are among the library implementation features that are made available via **stdlib**'s public interface.[4]

The fix is not needed in the *nix world. In the *nix world *fs*`::u8path` converts the argument to `std::string` with no encoding change. And so, for example, in Ubuntu, using g++ 6.3.0, the code compiles and works fine without the fix.

Just as MinGW g++ 7.1's *fs*`::u8path` punts on implementing an UTF-8 → UTF-16 conversion in Windows, with MinGW g++ 7.1 an *fs*`::path` argument to a file iostream constructor is not supported, though it's required by C++17. The lack of *fs*`::path` argument is problematic because g++'s default standard library implementation doesn't support wide string argument[5], either, and a narrow string path argument is assumed to be Windows ANSI encoded. And yes, that's even with UTF-8 execution character set.

There are three main solutions where portable Unicode paths are required:

```
[C:\my\dev\libraries\stdlib\writings\overload\code\π]
> set FIXES=-D STDLIB_USE_EXPERIMENTAL_CPP17 -D STDLIB_FIX_GCC_U8PATH

[C:\my\dev\libraries\stdlib\writings\overload\code\π]
> g++ filesystem.cpp %FIXES% -lstdc++fs

[C:\my\dev\libraries\stdlib\writings\overload\code\π]
> a
Contents of "C:\my\dev\libraries\stdlib\writings\overload\code\π\data
\blueberry-π.txt":
¶ Every 日本国 кошка likes Norwegian blåbærsyltetøy.
¶ Yummy! :)
```

Figure 6

- Only C++17-compatible compilers.

  This means not using MinGW g++, or not testing parts of the code with MinGW g++, or waiting until MinGW g++'s filesystem and iostreams library implementations are fixed.

- Pure ASCII alternative paths.

  Windows supports, although not completely and not for all Windows 'technologies', alternative pure ASCII paths. These are called short paths. The **stdlib** library provides a more robust abstraction, a best effort mostly readable native encoding narrow path, as `stdlib::char_path()` & friends.

- Custom iostream class.

  If one controls the file opening code, then better replace e.g. `std::ifstream` with a custom iostream class that supports *fs*`::path` or wide string argument, or best, that directly and portably supports UTF-8 encoded narrow string argument. The **nowide** library provides that as `nowide::ifstream` & friends. Such a class can also relatively easily be implemented in terms of `__gnu_cxx::stdio_filebuf<char>`.

Alternative ASCII paths were the basis of the MinGW g++ fix employed in the early Boost Filesystem, version 2 [Boost-c], but it was discontinued with no alternative fix in version 3, apparently deferring that fix to standardization. The original filesystem TS suggested that iostream constructors in Windows implementations should support the Visual C++ extension of wide character path argument. With C++17 we additionally have iostream constructors accepting *fs*`::path` directly, except that – the problem – as of this writing, MinGW g++'s default standard library implements neither.

Figure 7 is an example of a pure ASCII alternative path in Windows.

For readability and to preserve as much information as possible, especially for a name of a file to be created, `stdlib::char_path()` provides a Windows ANSI path, not a pure ASCII path, where it retains (transcoded) those items of the original Unicode path specification that can be encoded exactly as Windows ANSI (Figure 8).

Where an item can't be represented exactly as Windows ANSI and doesn't have an alternative ASCII name, `char_path` replaces any non-ANSI character with `stdlib::ascii::bad_char`, ASCII 127. I assume

4. ATTOW these conversion functions are limited to UTF-16 for wide text, e.g. they can't (properly) handle emojis in the *nix world. I intend to remove that limitation, but must do one thing at a time.
5. C++17 §30.9.1/3 requires wide string filename argument support for iostreams implementations on systems with wide native paths. Prior to C++17 this was a Visual C++ extension of the standard library.

```
// Here “df” is short for “data file”.
auto const   df_path    =
  program_folder_path / "data" / fs::u8path( "blueberry-π.txt" );
string const dfp_utf8   =
  df_path.u8string();
string const dfp_native =
  stdlib::char_path( df_path ); ← Using char_path.

ifstream f{ dfp_native };
if( f.fail() )
{
  cerr << "!Unable to open “" << dfp_utf8 << "”" << endl;
  cerr << "!using path “" << dfp_native << "”." << endl;
  return EXIT_FAILURE;
}
```

Listing 8

that this is often the desired behaviour: deferring path validity checking to the file opening code, and just using the path with replacements if it works, e.g. for display, or for creating a file. In contrast, **`stdlib::char_path_or_x`** throws a **`std::runtime_error`** exception if the Unicode path can't be represented exactly.

The design intention is to use **`char_path`** by default, e.g. for portably passing narrow paths to 3rd party library code, and as a not quite 100% but mostly Just Good Enough™ workaround/fix for filesystem-challenged implementations, like Listing 8.

Here, the UTF-8 path is used in the failure reporting instead of just outputting the *fs*`::path` directly, because while MinGW g++ 7.1 curiously does support that it adds simple ASCII quotes and duplicates every backslash, sort of happily sabotaging things.

As mentioned, the newly adopted-in-Boost **nowide** library provides streams that can be opened with UTF-8 encoded paths. And for file

```
[C:\my\dev\libraries\stdlib\writings\overload\code\π]   ← Original Unicode.
> for %f in ("%cd%") do @echo %~sf
C:\my\dev\LIBRAR~1\stdlib\writings\overload\code\0CA6~1 ← Alternative pure ASCII.
```

Figure 7

```
C:\my\dev\libraries\stdlib\writings\overload\code\0CA6~1 ← Result from char_path.
```

Figure 8

opening code that one controls, using an alternative file iostream implementation solves the availability problems of Windows ASCII alternative paths. For the code above, with the standalone variant of **nowide**, this solution entails just adding a

```
#include <nowide/fstream.hpp>
```

replacing **`ifstream f{ dfp_native };`** with

```
nowide::ifstream f{ dfp_utf8 };
```

and removing the **`dfp_native`** lines, and that's all.

With this approach, one uses each library for what it's good at.

Invalid-as-UTF-8 bytes, how, what?

Narrow text bytes that are invalid as UTF-8 can occur due to a number of possible reasons, e.g. just passing raw **`main`** arguments to **`cout`**, or doing conversion from wide text to the narrow encoding of the user's native locale, which in Windows cannot be UTF-8.

When this happens, it's in my opinion best if it doesn't stop output of further text, or indeed, of the text containing the bad bytes.

stdlib just replaces each bad byte with ASCII 127, **`DEL`** (see Listing 9). The result of the **stdlib**-based code is in Figure 9 – it works the same with g++.

The corresponding **nowide**-based code is in Listing 10 and the result of the **nowide**-based code is in Figure 10 (overleaf).

```
#include <stdlib/all/basics.hpp> // Among the fixes,
                                 // locale is set to "".
using namespace std;

auto some_library_func()
  -> wstring
{ return L"Pål Jørgen Sæther"; } // No Russian,
                                 // so it fits in CP1252.
auto main()
  -> int
{
  wstring const wname = some_library_func();
  string name( 256, '#' );
  // This is a Wrong Thing™ to do,
  // but existing code might do it:
  auto const n_bytes =
    wcstombs( &name[0], wname.data(), name.size() );
  if( n_bytes == -1 ) { return EXIT_FAILURE; }
  name.resize( n_bytes );
  cout << "Pleased to meet you, " << name << "!" << endl;
  cout << "Have a nice day! :-)" << endl;
}
```

Listing 9

```
[C:\my\dev\libraries\stdlib\writings\overload\code]
> cl invalid_utf8_bytes.stdlib.cpp /Fe"stdlib_bytes"
invalid_utf8_bytes.stdlib.cpp

[C:\my\dev\libraries\stdlib\writings\overload\code]
> stdlib_bytes
Pleased to meet you, P□l J□rgen S□ther!
Have a nice day! :-)                    ← OK.
```

Figure 9

Summary

There are currently two C++ libraries for UTF-8 console i/o in Windows: the author's **stdlib**, and the **nowide** library recently adopted in Boost. With **stdlib**, existing textbook code can work for Unicode console i/o in Windows, and since it's a header only library it's easy to use for novices. With **nowide** there is separate compilation, which can be a barrier to novices, and one's code must be modified to explicitly use the **nowide** functionality, which also means that existing, unmodified code doesn't benefit from **nowide**.

```
#include <nowide/iostream.hpp>
#include <locale.h>                // setlocale
#include <stdlib.h>                // wcstombs, EXIT_FAILURE
#include <string>                  // std::(wstring, string)
using namespace std;

auto some_library_func()
  -> wstring
{ return L"Pål Jørgen Sæther"; }  // No Russian, so it fits
                                  // in CP1252.
auto main()
  -> int
{
  wstring const wname = some_library_func();
  string name( 256, '#' );
  // This is a Wrong Thing™ to do,
  // but existing code might do it:
  setlocale( LC_ALL, "" );
  auto const n_bytes =
    wcstombs( &name[0], wname.data(), name.size() );
  if( n_bytes == -1 ) { return EXIT_FAILURE; }
  name.resize( n_bytes );
  nowide::cout << "Pleased to meet you, " << name
               << "!" << endl;
  nowide::cout << "Have a nice day! :-)" << endl;
}
```

Listing 10

As of this writing, console input just didn't work correctly with **nowide** – it included carriage return characters in input lines.

The **nowide** library's **`nowide::ifstream`** (& family) can be very useful as a workaround for MinGW g++'s current filesystem library implementation deficiencies, when one controls the file opening code. The corresponding **stdlib** fix **`stdlib::char_path`** is based on Windows' alternative ASCII names, which is easy to use and supports 3rd party library functions such as with OpenCV. It's guaranteed to work for a path that can be represented exactly with Windows ANSI encoding, plus this approach has worked for general Unicode existing paths on all the myriad local Windows systems that the author has used. I.e. it's not a perfect fix, but simple and usually Good Enough™. ■

References

[Boost-a] At http://www.boost.org/doc/libs/1_48_0/libs/locale/doc/html/default_encoding_under_windows.html

[Boost-b] Boost acceptance of NoWide: https://lists.boost.org/boost-announce/2017/06/0516.php

[Boost-c] Referred to in a 2011 discussion between the Boost Filesystem creator Beman Dawes and the author, titled 'Making Boost.Filesystem work with GENERAL filenames with g++ in Windows (a solution), at https://lists.boost.org/Archives/boost/2011/10/187282.php

[C99] C99 §7.17/2 (I used the N1256 draft, roughly C99 + TC1 + TC2 + TC3, for the quote).

[Ermey] Quoted from https://www.brainyquote.com/quotes/quotes/r/rleeermey464853.html

[ICU] The International Components for Unicode library, available at http://site.icu-project.org/

[Kaplan08] Still available at http://archives.miloush.net/michkap/archive/2008/03/18/8306597.html

[Microsoft-a] Quoting Microsoft's documentation of **`setlocale`**: "If you provide a code page value of UTF-7 or UTF-8, **`setlocale`** will fail, returning **`NULL`**." ATTOW that documentation was available at https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/setlocale-wsetlocale

[Microsoft-b] **`_setmode`** docs at https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/setmode

[Microsoft-c] Windows API function GetShortPathName documentation, at https://msdn.microsoft.com/en-us/library/windows/desktop/aa364989(v=vs.85).aspx

[nowide] The NoWide library is available at http://cppcms.com/files/nowide/html/index.html

[stdlib] The **stdlib** library is available at https://github.com/alf-p-steinbach/stdlib

[Steinbach11] 'Unicode part 1: Windows console i/o approaches', at https://alfps.wordpress.com/2011/11/22/unicode-part-1-windows-console-io-approaches/

[Steinbach13] 'Portable String Literals in C++', *Overload* #116, August 2013, available at https://accu.org/index.php/articles/1842

ASCII Alternative Paths

In the *nix world, **`stdlib::char_path()`** just returns the argument converted to UTF-8 if necessary, and in Windows it uses the following algorithm to return a best effort readable ANSI path:

let R (the result) be an empty string.
for each item in the Unicode path:
  if the item is ASCII *then*
    append it to R.
  else if it converts exactly to Windows ANSI *then*
    append the converted item to R.
  else if it has an alternative ASCII name *then*
    append the alternative ASCII name to R.
  else if character substitution is permitted *then*
    convert the item to ANSI, possibly with substitutions.
    append this possibly inexact ANSI text to R.
  else
    fail by throwing a **`std::runtime_error`**.

The order of checking is crucial to not needlessly discard information.

If you want to implement this yourself, then do note that the short very Unicody **π** as a path item is left as is by Window's main API function for this, **`GetShortPathName`**, presumably because **π** is so short. It's quite perplexing. For, while ASCII alternative paths are a very nice feature indeed, who needs a transformation of Unicode paths to still Unicode unreadable ultimate shortness with cryptic digit sequences, tildes and uppercasing thrown in here and there? I can't think of any need for that. It appears to be just silly.

Happily the **`FindFirstFile`** API function does give a pure ASCII alternative for that **π**, on a Windows installation and filesystem that supports short paths. And it apparently works fine in general, but only on one single path item, namely the last.

Problems include that short filenames in principle can be turned off via a registry setting (though it's unlikely, considering that they e.g. appear in registry values), that short filenames can be somewhat cryptic (it's easy to expand them back though), and that the documentation [Microsoft-c] states that they're not available with three Windows 'technologies', namely SMB 3.0 Transparent Failover (TFO), SMB 3.0 with Scale-out File Shares (SO), and Cluster Shared Volume File System (CsvFS), which I read as network drives (?).

```
[C:\my\dev\libraries\stdlib\writings\overload\code]
> set NOSILLY=/D _CRT_SECURE_NO_WARNINGS

[C:\my\dev\libraries\stdlib\writings\overload\code]
> cl invalid_utf8_bytes.nowide.cpp nowide_impl.obj /utf-8 %NOSILLY%
/Fe"nowide_bytes"
invalid_utf8_bytes.nowide.cpp

[C:\my\dev\libraries\stdlib\writings\overload\code]
> nowide_bytes
                                        ← No output at all.
[C:\my\dev\libraries\stdlib\writings\overload\code]
> echo Process exit code = %errorlevel%.
Process exit code = 0.← But the program didn't exit early, output was attempted.
```

Figure 10

A Functional Alternative to Dependency Injection in C++

Dependency injection allows flexibility. Satprem Pamudurthy showcases a functional alternative in C++.

Functional programming languages have certain core principles: functions as first-class citizens, pure functions (immutable state, no side-effects) and composable generic functions. C++ is not a pure functional language – we cannot impose immutability constraints on a function, for instance – but that is alright. Most real-world applications have side effects such as writing to databases and I/O, and thus cannot be written exclusively using pure functional constructs. With the addition of variadic templates, generic lambdas, perfect forwarding and the ability to return lambdas from functions, C++'s functional credentials are the strongest they have ever been. While OOP is the most popular paradigm in C++, by introducing elements of functional programming into our designs, we can create highly modular, extensible and loosely coupled components. In this article, I propose an alternative to dependency injection that uses functions to allow object behaviors to be configured at runtime.

Dependency injection

The basic building block of OOP in C++ is a class. A class encapsulates data and methods operating on that data. The behavior of an object is defined by its methods and how they manipulate the object's state. Some objects require the use an external service (a dependency) to implement some of their behavior. Dependency injection is a technique for decoupling the client of a service from the service's implementation [Wikipedia-a]. If the client object were to directly create an instance of the service, it would introduce a hard-coded dependency (strong coupling) between the client and the service implementation. The client object would have to know the exact type of the service, making it impossible to substitute a different implementation of the service at runtime. In dependency injection, we define an interface for the service and the client accesses the service's methods through the interface. Code external to the client is responsible for creating an instance of the service and injecting it into the client. The injection of the service can be done at construction, or post-construction through setter methods. We can now configure the behavior of the client by substituting different implementations of the service interface. Consider the example in Listing 1.

The `Customer` class has two dependencies:

- `ICustomerDatabaseService`
- `IOrderDatabaseService`

It uses the `ICustomerDatabaseService` to get or update the customer's profile, and the `IOrderDatabaseService` to load information about past orders. The `Customer` class should not and does not concern itself with where this information is actually stored or even

Satprem Pamudurthy works in the financial services industry and has been programming professionally for over 10 years. His main tools are C++ and Python but he will use anything that lets him get the job done. In the past, that has meant Java, C# and even VBA. You can reach him at satprem@gmail.com.

```
class ICustomerDatabaseService {
public:
  virtual ~ICustomerDatabaseService() { }
  virtual void
    deleteCustomer(const CustomerId&) = 0;
  virtual CustomerProfile
    getProfile(const CustomerId&) const = 0;
  virtual void updateProfile(const CustomerId&,
    const CustomerProfile&) = 0;
};
class IOrderDatabaseService {
public:
  virtual ~IOrderDatabaseService() { }
  virtual Orders
    getPastOrders(const CustomerId&) const = 0;
  virtual void enterNewOrder(const CustomerId&,
    const Order&) = 0;
};

class Customer {
public:
  Customer(const CustomerId& id,
    std::shared_ptr<ICustomerDatabaseService>
      pCustomerDb,
    std::shared_ptr<IOrderDatabaseService>
      pOrderDb)
  : id_(id)
  , pCustomerDb_(pCustomerDb)
  , pOrderDb_(pOrderDb)
  {
  }
  CustomerProfile getProfile() const
  {
    return pCustomerDb_->getProfile(id_);
  }
  void
    updateProfile(const CustomerProfile& profile)
  {
    pCustomerDb_->updateProfile(id_, profile);
  }
  Orders getPastOrders() const
  {
    return pOrderDb_->getPastOrders(id_);
  }
private:
  CustomerId id_;
  std::shared_ptr<ICustomerDatabaseService>
    pCustomerDb_;
  std::shared_ptr<IOrderDatabaseService>
    pOrderDb_;
};
```

Listing 1

The only thing we require of the service methods is that they are callable. We do not require the use of inheritance or any other technique that entails strong coupling amongst service methods.

whether it is even stored anywhere – we might have constructed mock implementations of the services. We can also use the DECORATOR pattern to extend the behavior of a service. The DECORATOR pattern is an object-oriented design pattern that allows us to add behavior to an object at runtime [Wikipedia-b]. A decorator is a special implementation of the service interface that forwards calls to an inner service implementation while executing code around the forwarded calls. Consider the class in Listing 2, which traces all calls to an order database service.

Tight coupling in inheritance

In our example, what does creating a new service implementation entail? For starters, you need to define a new class, and each concrete service class must implement every service method. When extending an existing implementation, you need to define a new class even if you only need to extend one of the service methods. Put another way, the unit of abstraction and extension in object-oriented programming is a class. Implementation inheritance also creates strong coupling between base and derived classes,

```
class TracingOrderDatabaseService
  : public IOrderDatabaseService {

public:
  explicit
    TracingOrderDatabaseService
    (std::shared_ptr<IOrderDatabaseService>
    pInner) : pInner_(pInner)
  {
  }

  virtual ~TracingOrderDatabaseService()
  {
  }

  Orders getPastOrders(const CustomerId& id)
    const override
  {
    std::cout << "Getting past orders";
    return pInner_->getPastOrders(id);
  }

  void enterNewOrder(const CustomerId& id,
    const Order& order) override
  {
    std::cout << "Entering new order";
    pInner_->enterNewOrder(id, order);
  }

private:
  std::shared_ptr<IOrderDatabaseService> pInner_;
};
```

Listing 2

because the derived class has access to all of the base class's public and protected data and methods. For an in-depth discussion of the various types of inheritance and their implications, please refer to John Lakos's presentation on inheritance [Lakos16a]. The video of his presentation is available on the ACCU YouTube channel [Lakos16b].

OK, so can we solve this problem by using the Interface Segregation Principle (ISP) [Wikipedia-c], whereby we define finer role interfaces instead of a fat interface (we can still have a single class implement multiple role interfaces)? Yes, but only for the time being. Interfaces tend to accumulate methods over time, and each new method requires changes down the inheritance tree, which brings us back to square one. Dependency injection can also create unintended dependencies between the `Customer` and the services. All public service methods are visible to every method of the `Customer` class, and there is nothing preventing the `Customer` class from using any of them. In the example above, the `Customer` class has access to the `enterNewOrder()` method, and even though it does not use it now, we cannot guarantee that it will not do so in the future. It is good practice to assume that every available method will be used. To quote David L. Parnas's influential paper on design methodology, *a good programmer makes use of the available information given him or her* [Parnas71]. Unintentional and hidden dependencies increase complexity and drastically affect maintainability of the code. We need a solution that allows us to better manage dependencies amongst code components.

A functional approach to configurable objects

Let us introduce functional programming into the mix and re-think the design of the `Customer` class. The illustration below uses a utility class I put together called `RuntimeBoundMethod`. It is a callable template class that stores a function object (as an `std::function`) whose first argument is a reference to the type containing the `RuntimeBoundMethod` (similar to the implicit 'this' in member functions). It takes a reference to the containing object in its constructor and passes it along to the stored function. This allows us to call a `RuntimeBoundMethod` as we would a member function. We can also specify the const-ness of the bound method with respect to the object containing the `RuntimeBoundMethod`. The code for this class (Listing 3) is available on github [Pamudurthy].

We still have the `Customer` class but instead of service interfaces, the `Customer` now depends on service methods. The only thing we require of the service methods is that they are callable. We do not require the use of inheritance or any other technique that entails strong coupling amongst service methods. Each service method can be bound (i.e. injected) independently of the other methods. Just as with role interfaces, it could very well be that a single class implements multiple service methods but that is entirely transparent to the `Customer` class. The entity that wires the `Customer` and the service methods together gets to decide exactly which service methods the `Customer` is able to use. Thus there are no unintended dependencies between the `Customer` and the services. But it is not all roses. If we forget to bind any of the service methods, we will

get a nasty surprise at runtime. This is also true of interface-based dependency injection when using setter methods to inject dependencies post-construction. We can avoid creating incomplete objects by requiring that all dependencies be provided at construction.

Extending function behaviors

The unit of abstraction, extension and composition in functional programming is a function. Just as we use decorator classes to extend the behaviors of an object, we can use decorator functions to extend the behavior of a function. C++ provides a powerful and concise syntax for writing generic functions that can forward arguments to another function. The decorator in Listing 4 adds a trace message before calling an inner function, while perfectly forwarding its arguments to that function.

```
class Customer {
public:
  explicit Customer(const CustomerId& id)
  : id_(id)
  {
  }

  const CustomerId& id() const
  {
    return id_;
  }

  RuntimeBoundMethod<const Customer,
    CustomerProfile> getProfile { this };
    // 'const' method
  RuntimeBoundMethod<Customer, void,
    const CustomerProfile&> updateProfile
    { this };
  RuntimeBoundMethod<const Customer, Orders>
    getPastOrders { this }; // 'const' method

private:
  CustomerId id_;
};

int main()
{
  CustomerId id{ 1 };
  Customer customer{ id };

  // bind service methods to the customer
  customer.getProfile = [](const Customer& self)
  {
    auto id = self.id();
    CustomerProfile profile;
    // populate the profile for id
    return profile;
  };

  customer.updateProfile = [](Customer& self,
    const CustomerProfile& profile) {
    // commit the new profile to storage
  };

  customer.getPastOrders =
    [](const Customer& self) {
    auto id = self.id();
    Orders orders;
    // load order details from storage for id
    return orders;
  };

  auto orders = customer.getPastOrders();

  CustomerProfile profile;
    customer.updateProfile(profile);

  return 0;
}
```

Listing 3

```
template<typename Method >
void addTraceMessage(Method& method,
  const std::string& traceMessage)
{
  method = [=](auto& self, auto&&... xs) {
    std::cout << traceMessage << std::endl;
    return
      method(std::forward<decltype(xs)>(xs)...);
  };
}
```

Listing 4

We would add tracing to a service method as follows:

```
addTraceMessage(customer.getPastOrders,
  "Getting past orders");
auto orders = customer.getPastOrders();
  // prints a message before calling
  // the inner function
```

Again, because we are dealing with functions and not interfaces, we are able to add tracing only to the service methods we are interested in.

A caveat about runtime behavior configuration

When using techniques that allow us to configure the behavior of an object at runtime, the intended behavior of an object cannot simply be deduced from its type. Instead, you will need to understand how the object has been wired at and after construction. This requires some adjustment on part of the programmer when it comes to code analysis and debugging, and this remains true even when using a functional approach.

Final thoughts

C++ is not a pure functional language, but ultimately programming paradigms are not so much about language features as they are ways of thinking about component and system design. Thinking functionally will allow us to build highly modular designs that are easy to compose and extend. Object-oriented and functional programming can coexist and C++ allows us get the best of both worlds – we can use classes to encapsulate entities, and function objects to define and extend their behaviors. ■

References

[Lakos16a] Proper Inheritance, John Lakos at https://raw.githubusercontent.com/boostcon/cppnow_presentations_2016/master/00_tuesday/proper_inheritance.pdf

[Lakos16b] Proper Inheritance, John Lakos, ACCU 2016 at https://www.youtube.com/watch?v=w1yPw0Wd6jA

[Parnas71] *Information distribution aspects of design methodology*, David L. Parnas, 1971

[Pamudurthy] RuntimeBoundMethod.hpp at https://github.com/spamudurthy1520/FunctionalCPP/tree/master/source

[Wikipedia-a] Dependency Injection at https://en.wikipedia.org/wiki/Dependency_injection

[Wikipedia-b] Decorator Pattern at https://en.wikipedia.org/wiki/Decorator_pattern

[Wikipedia-c] Interface Segregation Principle at https://en.wikipedia.org/wiki/Interface_segregation_principle

About the C++ Core Guidelines

The C++ core guidelines are a useful summary of C++ best practice. Andreas Fertig shows their use.

In 2015 at CppCon, Bjarne Stroustrup announced the C++ Core Guidelines [CCG] in his opening keynote. These guidelines are a summary of best practices intended to overcome known traps and pitfalls of the C++ programming language. All the rules are grouped into several sections ranging from philosophy, interfaces and classes to performance and concurrency. The project is designed as an open source project, which should ensure that it will improve over time along with the language itself.

Behind this set of rules are some different ideas because, as Bjarne mentioned in his talk: "We all hate coding rules" [Sommerlad16]. Often, the rules are written by people with weak experience with the particular programming language. They tend to keep the use of more complex features to a minimum, with the aim of preventing misuse by less experienced programmers. Both concepts are invalid for the Core Guidelines, which are written by experts in the field targeting programmers with C++ experience. The aim of the Core Guidelines is to assist people in using C++ effectively, which implies transitioning legacy C++ code towards modern C++, using C++11 or newer. The guidelines also focus on the language itself and using its power. For example, enable the use of static code analysis by expressing your intent in the language while leaving comments only for documentation. Consider this example:

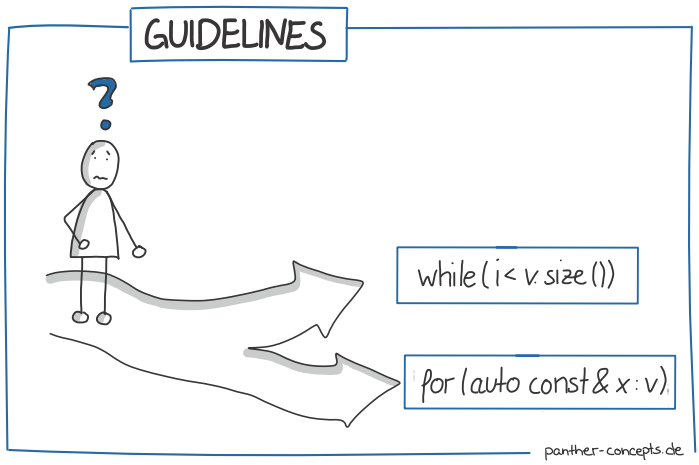
```
int i = 0;
while( i < v.size() )
{
  // do something with v[i]
}
```

By using modern C+, this can be transformed into something like this:

```
for(auto const& x: v)
{
  // do something with x
}
```

(Example taken from [Sommerlad16].)

There is a big difference between these two code fragments. In the first, a new variable comes into scope, and several problems can occur if it is reused. For example, it can lead to an out of bounds access Furthermore, there is the potential of getting the array access of **v** wrong by adding +1 to the loop variable. Tampering with this variable in other ways is also possible. These are typical mistakes, nothing somebody will do wrong on purpose. In the modern version, there is no need for an additional variable. It is also clear that the author is only interested in the objects of a vector. No modification will take place, hence the **const** reference. Since we are interested in all elements in the vector for the loop this is clearest. Last but not least, the modern version is much clearer to the compiler and static analysis tools. The compiler for instance, will not allow compilation if a write access to **x** takes place. A static analysis tool can understand that this is a way of iterating over the whole set of vector elements. You can improve it even more by using functions from **std::algorithm** like **std::replace** or **std::find**.



When using the C++ Core Guidelines, C++ becomes a little different. In Bjarne's words: "Within C++ is a smaller, simpler, safer language struggling to get out" [Stroustrup15]. This means that all the rules in the Core Guidelines work with a modern C++ compiler[1]. No additional extensions are required, albeit there are assisting libraries to facilitate using the rules. Let's look at some of the rules.

Signalling failure

There is:

> I.10: Use exceptions to signal a failure to perform a required task.

That's a rule I struggle with. The standard library uses exceptions as the main failure signalling mechanism. It fully denotes the word 'exception'. We do not expect such an event, so it's reasonable to throw an exception at this point. This also leaves the return value for returning a value in case the function was successful. My struggle here is, if nobody expects something why should anyone catch it? A ball thrown at you unexpectedly can hurt a lot because you were not ready to catch it. Well, in case of C++, it's like you are fast enough to duck. Then, at least you do not get hurt, but what about the others? You can let an exception which was caused by a function your code called pass to whoever called you. Now, the next higher function in the call stack has to deal with it. This pattern can continue until we hit **main**. Then, the program will terminate. Let's say somebody within the call stack *does* catch the exception – now what? There is often no good choice. In the layers above, nobody knows which call triggered the exception and how to react to it. A horrible scenario for embedded systems which are somewhat critical! There may

Andreas Fertig holds an M.S. in Computer Science from Karlsruhe University of Applied Sciences. Since 2010, he has been a software developer and architect for Philips Medical Systems with a focus on embedded systems. He also works as a trainer and develops various Mac OS X applications. Andreas' online presence is https://www.AndreasFertig.Info

1. There is an exception when it comes to guidelines involving concepts.

the choice is either to return the error code and pass the actual return value into the function as a pointer or reference parameter, or vice versa

be millions of lines of code out there which can throw an exception, but I prefer not to. I use my freedom to *not* pick this rule for me.

What can we do instead? A solution I have come across multiple times is the following

```
int SomeFunction(int param1, double param2,
  int* outValue)
{
  //...
}
```

Let us suppose the returned value uses the full range of its data type; then, there is no space left to squeeze in the error code. Now, the choice is either to return the error code and pass the actual return value into the function as a pointer or reference parameter, or vice versa. Both are suboptimal.

The guidelines provide an alternative in section I.10: "using a style that returns a pair of values".

```
auto [val, error_code] = do_something();
if (error_code == 0)
{
  // ... handle the error condition
}

// ... use val
```

It uses structured bindings which are available in C++17. This allows us to return a struct and directly assign variables to the members. The resulting code is much clearer and robust compared to the variant shown before.

However, there is another alternative: **`std::optional`** (see Listing 1).

```
std::optional<std::string> GetUserName(int uid)
{
  if( uid == 0 )
  {
    return "root";
  }
  return {};
}

void UsingOptional()
{
  if( auto str = GetUserName(0) )
  {
    std::cout << *str << "\n";
  }

  auto fail = GetUserName(1);
  std::cout << fail.value_or("unknown") << "\n";
}
```

Listing 1

We can ask the optional object whether or not it contains a value, meaning it can be used in a boolean expression. In case you would like to skip all those checks, you can invoke it with **`value_or()`** and pass a value which is used when the object does not contain a valid object. Pretty neat.

Safe and modern array passing

Let's move on to another item:

> I.13: Do not pass an array as a single pointer.

This aims to solve a popular problem we can often see in the wild. For example, in the safe version of string copy:

```
char* strncpy(char* dst, const char* src,
  size_t n)
{
  // ...
}
```

Wow, how safe is that? We have a single **`size_t`** parameter. To which value does it apply? Alright, it enables us to write code like this:

```
strncpy( dst, src, MIN(dstSize, srcSize) );
```

Honestly, does this code look good to you? Writing **`MIN()`** over and over again? How many mistakes can still be made? Rule I.13 is about getting rid of code like this. Instead, there is the template class **`span`** which uses the power of templates to deduce the size of the object. You can also cut it down to just a slice of the array. The resulting object can be queried for its size, hence the chances for discrepancies are reduced by a lot. It is one object containing data and size. An improved string copy function would look like this:

```
span<char> strcpy(span<char> dst,
           span<const char> src)
{
  // ...
}
```

If you pay close attention, you will notice that we no longer need to check for null pointers in **`strcpy`**.

No raw pointers

Another rule is

> R.10: Avoid **`malloc()`** and **`free()`**.

Together with

> R.11: Avoid calling **`new`** and **`delete`** explicitly

it aims to reduce the use of uncontrolled memory allocation, with the goal of preventing memory leaks. In C++ with objects, **`malloc`** and **`free`** do nothing good for us. They are legacies from C. The guideline tells us to avoid **`new`** and **`delete`** in their naked form as well. In modern C++, the use of so called raw pointers, pointers without an owner, are discouraged. To handle resource management better, allocated memory should belong to an owner: some object which takes care of the lifetime of the memory. In modern C++, we have several kinds of managing pointers: **`unique_ptr`**, **`shared_ptr`** and **`weak_ptr`**. Helper functions like

The concept of the library is to provide ready-to-use functions which enforce the idea of the C++ Core Guidelines, and increase the safety and correctness of a program

`make_unique` are available to assist us create such a pointer without writing `new` ourselves. Afterwards, the smart pointers take care of the allocated owned memory.

In case none of those managing pointers matches your needs, fallback to `owner<T>`. The idea behind it is to state the ownership of a simple pointer. In a perfect world, all `owner<T>` instances would be a managing a pointer like `unique_ptr`. When we are not there, `owner<T>` can be helpful for static analysis. Pointers which are not owning must not free memory. On the other hand, owning functions must free memory as soon as they go out of scope.

A library for the guidelines

For the best support of the C++ Core Guidelines, there is a library called 'Guideline Support Library' (GSL). Microsoft provides an implementation of it under the MIT licence hosted in github [Microsoft]. The concept of the library is to provide ready-to-use functions which enforce the idea of the C++ Core Guidelines, and increase the safety and correctness of a program.

There are simple things in it like `at()`. This tiny template function provides a bound-checked way of accessing built-in arrays like char buffer[1024].There are also places for things we did wrong for a long time: `narrow_cast`; again a template function which mimics the style of a C++ cast. Under the hood it checks whether the narrowing will lose signedness or results in a different value. Many of the checks are run-time checks. However, it is a way of letting static analysers know what you intend to do, and in doing so, there is a chance finding bugs before run-time.

In many ways, the GSL is similar to the boost [Boost] library. For years, boost has driven some new ideas and language improvements by letting the community try it out and decide if an idea is useful, all without compiler or standards changes. Some improvements of boost have found their way into recent C++ standards. The GSL may do the same for the community. In fact, they managed to get the first item of the GSL into the shiny new C++ standard C++17: `std::byte` [C++].

Summary

In summary, the C++ Core Guidelines try to encourage using modern C++. There is word on the street that they contain too many rules which at some point overlap. Still, they are a comprehensive collection of possible mistakes which can be avoided. Consider looking at the C++ Core Guidelines for ideas of how to write modern C++ and, of course, pick the items you consider valuable for your project. Also have a look at the GSL (multiple implementations are available) as it helps you write safer and more robust code. ■

Acknowledgements

Thanks to Peter Sommerlad who reviewed draft versions of this article. His comments contributed to substantial improvements to the article.

Artwork by Franziska Panter from panther concepts, https://panther-concepts.de

References

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

[C++] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0298r2.pdf

[CCG] https://github.com/isocpp/CppCoreGuidelines

[Microsoft] https://github.com/Microsoft/GSL

[Sommerlad16] http://wiki.hsr.ch/PeterSommerlad/files/ESE2016_core_guidelines.pdf

[Stroustrup15] https://github.com/CppCon/CppCon2015/blob/master/Keynotes/Writing%20Good%20C%2B%2B14/Writing%20Good%20C%2B%2B14%20-%20Bjarne%20Stroustrup%20-%20CppCon%202015.pdf

Afterwood

Have you ever broken prod? Chris Oldwood reminds us to fix the problem not the blame.

As a child. I struggled with honesty; often shooting from the hip meant I never really considered the consequences of my actions and so found myself facing the ire of my parents more frequently than I should have. It was never anything serious but I began to find it easy to explain why anything that went wrong was never entirely my fault. I suspect the innocence of childhood gave me the benefit of the doubt more often than I deserved too. Fortunately, my parents did a good enough job that my moral compass ensured I never strayed far from the straight and narrow.

Software development has always appeared more forgiving than many other jobs. In my early days as a programmer there were a million different reasons why things never went smoothly. For example, the very environment we worked in was brittle – you would routinely have to restart 16-bit Windows and pray your hard disk was still intact. I've lost many hours trying to debug even user-mode applications that would cause the earlier versions of Windows to crash. Then you have compilers with code generation bugs which you needed to work around by writing the same code in a different way. And then there's the language itself with its many traps and pitfalls for the unwary programmer that resulted in 'undefined behaviour' (UB) which you often found out the hard way. Add to this the ambiguities of natural language leading to poorly specified requirements and you'll find it's fairly easy to avoid having the finger pointed directly at you as the root cause of most problems.

My safe little bubble eventually burst one afternoon courtesy of a support incident at a large financial organisation. I was genuinely surprised when a colleague quietly asked me why I had just rebooted our system's main servers in the production environment. He confirmed that he'd double checked the security logs and, yes, my login was in there as being the instigator of the machine restarts, which seemed pretty conclusive. It didn't take long before I realised what had happened and how the mistake had been made. Yes, I had a pretty good clue where things had gone wrong but ultimately the mistake was mine. My stomach churned as I waited for the fallout. You often hear tales about how people have been marched straight off the premises for misconduct and can't help but wonder if there's a grain of truth somewhere in those friend-of-a-friend stories. Being a contractor and fairly new to the company didn't feel like it was exactly going to help my cause either.

Of course, nothing happened. In retrospect, my mistake was insignificant compared to the many others that occurred around me and, whilst there was a loss of service as everything slowly came back up and recovered, the actual loss to the business was probably less than the time taken to work out what it would have cost. Naturally the first code change I made straight after was to my custom admin tool so that machine names starting with a P and D were more easily discernible – bright red for the former, something contrasting for the latter. Oh, and it popped up a warning message too, for good measure.

The notion of holding a post-mortem is not a new one although the emphasis on it being a blameless post-mortem seems to have gained more recognition in recent years. My earliest recollection of the idea of post-mortems (outside the medical ones on TV shows like *Quincy*) was through the embedded software column in *Dr Dobbs Journal*, written by Ed Nisley. NASA had started making their post-mortems publicly available and so Ed would publish extracts in his column along with some additional commentary. I've never worked in that kind of environment but certainly marvelled at the ingenuity it creates working within such constraints. It would be easy to laugh at some of the extremely costly and yet seemingly trivial mistakes they've made in the past but the main take-away for me was always that if the super-smart people at somewhere like NASA couldn't get this software development lark right all the time, then what chance did I stand?

Reading about the recent major outage at a British airline and the apparent scapegoating of a system administrator reminded me of some of my own little mishaps and how they had been dealt with. I've clearly been fortunate enough to have worked within teams where the level of trust and respect both within the team and around it are sufficiently high that the occasional mistake is dealt with appropriately. One can only assume that senior managers and shareholders are after a scalp when something of that magnitude goes awry and therefore it takes real courage to stand up and blame the process that let this happen rather than single out the person who was likely the victim of a weak process.

The prime directive, which is read out at the start of a team's retrospective, is a very clear statement which attempts to try and make the team comfortable so they can get to the business of improving the process they use, rather than blame the people themselves for their actions. Martin Fowler has suggested in the past (based on the work of Pat Kua) how important reading this statement has become, as repeatedly hearing it could potentially change the culture of the team though the physiological effect known as priming.

This also ties in very nicely with 'psychological safety' which Google's recent project Aristotle [NYTimes] managed to bolster with some qualitative data. This is nothing entirely new though as 'safety' also features in the lower layers of Maslow's Hierarchy of Needs, which he published back in the late 1940's and has no doubt been the subject of many other research projects in the intervening years. What probably caused Google's project Aristotle to surface on my (and many other programmers') radar was no doubt down to the workforce studied.

The TL;DR of that research appears to be that we are at our happiest when we work with nice people, although I'm sure it's highly disingenuous to try and distil it to such a simplistic outcome. I don't think I'll ever truly get over the small amount of fear I feel when administering a production system but I also think that may be a healthy attitude, to some degree, to ensure I'm not reckless through complacency. Either way, as long as any fear we do feel is one of our own self-restraint and not out of a lack of process, and subsequent retribution, then our productivity will remain at its highest. ■

Reference

[NYTimes]https://www.nytimes.com/2016/02/28/magazine/what-google-learned-from-its-quest-to-build-the-perfect-team.htmllearned-from-its-quest-to-build-the-perfect-team.html

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's enterprise grade technology in plush corporate offices. He also commentates on the Godmanchester duck race and can be easily distracted via gort@cix.co.uk or @chrisoldwood