overload 170
AUGUST 2022 £4.50

Advancing the State of the Art for std::unordered_map Implementations

Joaquín M López Muñoz presents a new, fast version of unordered maps

saturating_add vs. saturating_int – New Function vs. New Type?

Jonathan Müller explores when and how to avoid integer arithmetic overflows

Don't Block Doors

Frances Buontempo uses cellular automata to show what happens if people stand in doorways

Lessons Learned After 20 Years of Software Engineering

Lucian Radu Teodorescu shares his reflections

Afterwood

Chris Oldwood considers what legacy we can leave to make life better for others

A magazine of ACCU
ISSN: 1354-3172

accu

professionalism in programming

- Monthly journals, available printed and online
- Discounted rate for the ACCU Conference
- Email discussion lists
- Technical book reviews
- Local groups run by ACCU members

Visit www.ACCU.org to find out more

August 2022

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Ben Curry
b.d.curry@gmail.com

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.co.uk

Balog Pal
pasa@lib.hu

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

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Cover photo by @wirestock on freepik.com.

Copy deadlines

All articles intended for publication in *Overload* 171 should be submitted by 1st September 2022 and those for *Overload* 172 by 1st November 2022.

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

4 saturating_add vs. saturating_int – New Function vs. New Type?
Jonathan Müller explores when and how to avoid integer arithmetic overflow.

7 Advancing the State of the Art for std::unordered_map Implementations
Joaquín M López Muñoz presents a new, fast version of unordered maps.

10 Don't Block Doors
Frances Buontempo uses cellular automata to demonstrate what happens if people stand in doorways.

12 Lessons Learned After 20 Years of Software Engineering
Lucian Radu Teodorescu reflects on lessons learned during his career.

16 Afterwood
Chris Oldwood considers what legacy we can leave to make life better for others.

The ACCU

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

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

Copyrights and Trademarks

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

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

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

Whodunnit?

Coding is a creative process. Frances Buontempo wonders how close it often gets to fiction.

Recently, I have been watching far too many murder mysteries on the television, so forgot to write an editorial. Some are more serious than others. We have several episodes of Midsomer Murders [IMDB-1] saved, which we dip into from time to time. This long running series is a typical 'whodunit' (pronounced "Who done it?") [Wikipedia-1], where someone is murdered near the start and you spend the next hour or so trying to decide who the murderer is. Over time, the series resorted to adding more and more murders and the plots became more and more silly, leaving you guessing what ridiculous plots twists may come next rather than trying to remember who died and who the murder might be. Other murder mysteries are available. A whodunit is very different to a thriller, in that the former goes backwards and forwards in time, filling in clues, while the latter usually moves forward in time, ramping up the sense of suspension. One encourages you to guess who is responsible while the other makes you wonder what happens next. I suggest both types of narrative crop up while we write or run code. Many other types of fiction can happen too as we attempt to create software.

The ACCU conference talks are now showing up on YouTube, giving me even more to watch instead of writing an editorial. Matthew Dodkin's talk [Dodkin22] encouraged us to expect the unexpected and think about what happens next. He talked about dolphin and bat detectors, requiring long deployments (months). You chuck them overboard in a remote location and wait, meaning you can't easily monitor remotely – and go back and get them later. This necessitates the need to minimize and handle failures and keeping a log of what happened somewhere accessible is useful. Matthew talked about various ways to 'handle' problems, including an error handler which does nothing, in effect ignoring problems, or sits in a `while` loop doing nothing. He also mentioned asserts and said you should "disable them in production code that needs to keep running, unless you are extremely confident about what happens next." Furthermore, he suggested using 'what happens next' as a useful thought-experiment for designing all your error handling functionality. In order to avoid any further plot spoilers, I'll say no more but leave you to watch the talk's recording. Other ACCU conference talks are available.

"What happens next?" fits the thriller genre better than a murder mystery, though the advice to leave accessible logs does chime with a crime investigation. When we try to figure out what went wrong in code, logs are often a first port of call. Of course, they tell us what happened previously rather than what's up next. Writing useful log files is a bit of an art form. Chris Oldwood has spoken and written about this on many occasions. In an article for Overload, he claimed log files rarely contain anything helpful and suggested ways to do better [Oldwood15]. His 2019 conference talk encouraged us to avoid the stream of consciousness style of logging, and add a little structure [Oldwood19]. Writing a stream of consciousness 'story' is one thing, and can be cathartic; however, trying to read it might not be so easy. James Joyce's *Ulysses* is often cited as an example of this style of writing and many people, myself included, who try to read it, give up. I lost track of who the characters were, and lost the plot, if there really is one. Wikipedia quotes an example on its 'Stream of consciousness' page [Wikipedia-2]:

> a quarter after what an unearthly hour I suppose theyre just getting up in China now combing out their pigtails for the day well soon have the nuns ringing the angelus theyve nobody coming in to spoil their sleep except an odd priest or two for his night office the alarmclock next door at cockshout clattering the brains out of itself let me see if I can doze off 1 2 3 4 5 what kind of flowers are those they invented like the stars the wallpaper in Lombard street was much nicer the apron he gave me was like that something only I only wore it twice better lower this lamp and try again so that I can get up early

Many log files read like this. Don't get me wrong, a tumble of word associations and ideas can be fun. I recently reminded myself of the 'lyrics' to *Eat, Sleep, Rave, Repeat* by Fatboy Slim. The words tumble and you only partially follow what's going on.

> And then this cat walked in
> You know, not like a cat
> Like a feline cat
> Like a real, like you know
> Like
> You know what I'm saying dog
> Like cats and dogs
> It was raining

Logs can aid us in detective work after the fact if they contain the right clues. If we can debug code, which wasn't the case for the bat and dolphin sensors, we can watch things play out in real time and that can take time too. Finding the best place to put a breakpoint is important. I've lost hours deep down inside a call stack when the problem was actually somewhere completely different. In some ways, this ends up like the tumbling lyrics or stream of consciousness style of writing. It's very easy to lose the plot. Always time-box debugging and try to find more efficient ways to work out who or what 'done it', like writing a unittest if you can. Sometimes a program crashes and gives you a useful core dump. This so-called post-mortem analysis tells you where to start looking and if often way quicker than stepping through every line of code. *Silent Witness* anybody? [IMDB-2]. You can also use diagnostic tools on live code in order to figure out why strange things are afoot.

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.

Before ever running the code, you might use a compiler along with static analysis tools. When faced with a wall of errors, it sometimes takes some investigation work to find one offending line. Have you ever resorted to a binary search taking smaller and smaller chunks of code out until you find the cause of the problem? Maybe you tried ctrl+Z to undo code until you got back to a clean state, or maybe you used version control? Hurrah for small commits between changes, otherwise you may have a very large search space! Playing detective is part of programming and leaving yourself a trail of easy to follow clues is a good idea. For unscripted languages, you can lean on the compiler to find problems and even code usage, by changing a name or type and so on [Feathers04]. Scripted languages may seem like another genre, but often don't have much of a plot either. However, the interpreter and static analysis tools can also help you avoid a potential crime scene.

There's often more to coding than just the source code. As mentioned, if you're kind to yourself, you'll be using version control. Not only does that help you undo any changes that broke things, but also keeps track of who did what, when. Unless you use git lie squash on the commits, of course, or change a username to 'deleted' when they leave, which is not unheard of. Many have a 'blame' command that displays which line was changed by whom and when. If you're trying to solve a coding crime, this can be useful. I've heard it claimed that blame might be a bit of a negative frame of mind, and 'praise' might be due sometimes. That's fair – not everything coding is problem fixing and troubleshooting. Sometimes, it's actually fun. You may be using some kind of work tracking system too, perhaps Jira. Jira, in and of itself, is fine. Left to my own devices, I'd use a simple Kanban board or just a TODO list, however keeping track of what needs doing is the important part. And yet many people complain about Jira. It is configurable, which, again, in and of itself is fine. You can even write stories, or epics, which is nice. But, and this is the reason for most of the complaints, Jira can be configured to a point of pain or even left on its defaults, forcing you to add various fields and tick many boxes to move a story through to the finale. Once a process becomes onerous, people get inventive and find work-rounds. If coding ends up as a form-filling Kafkaesque nightmare, and I don't mean the distributed event-streaming sort, I mean the writer Kafka's bureaucratic nightmare world. More screaming than streaming.

Sometimes 'whodunit?' doesn't actually matter. What's more important is how you are going to fix it. Fix the problem, not the blame, as the saying goes. Sometimes we may never find out whodunit, but that's OK. What's important is how we move on. It's all too easy to get caught up in gossip and rumour, or follow a hunch down a rabbit hole. If you're trying to release code or fix a bug, you need to keep your eyes on the prize and avoid being distracted (too much) by other things, like most of the UK government resigning or tweets about the *C++ on Sea* conference. Many places I have worked at use the pattern of ascribing the blame to the last person who left. As soon as someone starts asking "Who on earth wrote this code?", then reply is "So-and-so, remember them?" You are supposed to then say "So-and-so who?" and concentrate on the task in hand. The culprit isn't important, rather making progress is. Knowing who is responsible won't stop a repeat performance. You might end up in a non-fiction version of Groundhog day [IMDB-3] with variations of the same thing happening over and over again.

Sometimes 'who did it' really does matter. I often try to look up a source for a quote and find various conflicting suggestions. It's good to be able to reference a source for a variety of reasons, but if you can't, you can't. If you can, you can leave clues for readers to follow, so they can draw their own conclusions. Some theorems, physical phenomena and computing ideas are named after the person who invented them, for example Pascal's triangle, the Higgs boson or the Liskov substitution principle. We do sometimes find that names are misattributed though. I was told about the Rutherford experiment at school, and later learnt he was the supervisor and the experiments themselves were performed by Geiger and Marsden [Wikipedia-3]. You could argue that science or knowledge is more important, but if someone's name is associated with an idea making them more well-known, this might skew our understanding of history, particularly if they didn't do it. In effect, this rewriting history is a lie. I already told you what I thought of git squash to rewrite history! Misappropriations happen, though we can strive to avoid making things worse. And it's lovely to ensure you credit someone if they have contributed or helped in some way. So, thank you to all our writers and the review team.

We've considered a few styles of fiction, and though programming isn't really fictional, it is a creative process and it may involve stories or actors (of Carl Hewitt *et al*'s formalism, rather than thespians [Hewitt73]). Sometimes real drama is involved if you have a prod outage or other catastrophic failure. Sometimes the code you are using seems somewhere between fantastical or a farce, littered with 'Here be dragons' or 'Wtf?!' comments on the way.

References

[Dodkin22] Matthew Dodkin, 'A Year In A Rainforest: Engineering For Survival' from the ACCU 2022' conference, available online at: https://www.youtube.com/watch?v=Zua4twRU2VU

[Feathers04] Michael Feathers (2004) *Working Effectively With Legacy Code*, Addison-Wesley

[Hewitt73] Carl Hewitt, Peter Bishop and Richard Steiger (1973) 'A Universal Modular Actor Formalism for Artificial Intelligence' *IJCAI*, available at: https://www.ijcai.org/Proceedings/73/Papers/027B.pdf

[IMDB-1] Midsomer Murders: https://www.imdb.com/title/tt0118401/

[IMDB-2] Silent Witness: https://www.imdb.com/title/tt0115355/

[IMDB-3] Groundhog day: https://www.imdb.com/title/tt0107048/

[Oldwood15] Chris Oldwood, 'Terse Exception Messages', *Overload*, 23(127):15-17, June 2015, available at: https://accu.org/journals/overload/23/127/oldwood_2110/

[Oldwood19] Chris Oldwood, available at https://www.youtube.com/watch?v=O6NJgcK6K7g

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

[Wikipedia-2] Steam of consciousness: https://en.wikipedia.org/wiki/Stream_of_consciousness

[Wikipedia-3] Geiger & Marsden experiments: https://en.wikipedia.org/wiki/Geiger%E2%80%93Marsden_experiments

saturating_add vs. saturating_int – New Function vs. New Type?

Integer arithmetic tends to overflow. Jonathan Müller explores when and how to avoid this.

Suppose you want to do integer arithmetic that saturates instead of overflowing. The built-in `operator+` doesn't behave that way, so you need to roll something yourself. Do you write a `saturating_add()` function or a new `saturating_int` type with overloaded `operator+`? What about `atomic_load(x)` vs `atomic<int> x`? Or `volatile_store(ptr, value)` vs `volatile int*`?

When should you provide functions that implement new behaviour and when should you write a wrapper type? Let's look at the pro and cons.

Writing a new function

If you want to have a saturating addition, just write `saturating_add(int, int)`; to load something atomically, just write `atomic_load(int*)`; to store something that isn't optimized away, just write `volatile_store(int*, int)`.

It's a simple, straightforward solution, and for some of you the post can end here. However, it isn't quite ideal.

Disadvantage #1: Can't re-use existing names/operators

The following code computes something with overflowing (undefined) behaviour:

```
int x = …;
int result = x * 42 + 11;
```

This is the same code, but using saturating behaviour:

```
int x = …;
int result =
  saturating_add(saturating_mul(x, 42), 11);
```

Which version is more readable?

As `operator*` and `operator+` already have meaning for `int`s, we can't use them for saturating arithmetic, we have to use functions. This means we lose the nice operator syntax and instead have to figure out nested function calls.

The problem can be solved at a language level. For example, Swift has `+` which raises an error on overflow and `&+` which wraps around on overflow. By defining new syntax, we don't need to resort to function calls. Of course, this is inherently limiting to users that don't work on the language itself, or it requires a language where you can define your own operators. But even Swift has no saturating operator and C++ doesn't have anything at all.

If we instead decide to write a new `saturating_int` type, we can overload `operator*` and `operator+` to implement the desired functionality (Listing 1), then code that performs saturating arithmetic looks almost identical to regular code, we just need to change the types:

```
int x = …;
auto result = int(saturating_int(x) * 42 + 11);
```

```
struct saturating_int
{
  int value;
  explicit saturating_int(int v)
  : value(v) {}
  explicit operator int() const
  {
    return value;
  }
  friend saturating_int operator+
    (saturating_int lhs, saturating_int rhs);
  friend saturating_int operator*
    (saturating_int lhs, saturating_int rhs);
  …
};
```

Listing 1

Disadvantage #2: Can't directly use generic code

This is really the same as the first disadvantage: as we have to invent a new name for the operation and can't re-use the existing one, generic code doesn't work out of the box. In C++, templates use duck-typing and they call operations based on syntax. If the syntax isn't available or doesn't do what we want, we can't use them.

For example, using our `saturating_add()` function, we can't use `std::accumulate` directly, as it calls `operator+`. Instead, we have to pass in a custom operation that calls `saturating_add`.

Disadvantage #3: Can't enforce behaviour

Suppose we want to control some sort of embedded peripheral (e.g. an LED) by writing to the special address `0xABCD`. The code in Listing 2 is buggy. As the compiler can't see anybody reading the `1` written to `*led`, it considers it a dead store that can be optimized away. The compiler has no idea that it has the additional side-effect of turning an LED on and needs to be preserved!

The correct fix is to use a volatile store, which tells the compiler that it must not optimize the store away. Let's suppose it is implemented by a hypothetical `volatile_store()` function (see Listing 3, overleaf). Now it works, but we have to manually remember to use `volatile_store()` as opposed to `*led` every time. If we forget, nobody reminds us.

```
const auto led =
  reinterpret_cast<unsigned char*>(0xABCD);
*led = 1; // turn it on
std::this_thread::sleep_for
  (std::chrono::seconds(1));
*led = 0; // turn it off
```

Listing 2

Jonathan Müller is a computer science and physics student at the RWTH Aachen University. In his spare time, he works on various C++ projects, and enjoys writing libraries (especially for real-time applications, where performance matters). You can contact him via his blog (foonathan.net) or Twitter (https://twitter.com/foonathan).

Suppose you want to do saturating arithmetic, but only sometimes; otherwise, you want overflow. As the behaviour is provided by types, you need to change types to change the behaviour

```
const auto led =
  reinterpret_cast<unsigned char*>(0xABCD);
volatile_store(led, 1); // turn it on
std::this_thread::sleep_for
  (std::chrono::seconds(1));
volatile_store(led, 0); // turn it off
```

Listing 3

In actual C++, where volatility is part of the pointer type, this isn't an issue: once we create a `volatile unsigned char*`, all loads/stores are automatically volatile and we don't need to remember it. By putting it in the type system, we can enforce the consistent use of a given behaviour.

Disadvantage #4: Can't store additional state

Suppose we want to write a generic function that can atomically load a value at a given memory address:

```
template <typename T>
T atomic_load(T* ptr);
```

On modern CPUs, implementing this function is straightforward if `sizeof(T) <= 8`. For `sizeof(T) == 16`, it becomes tricky, and for `sizeof(T) == 1024`, it is impossible, as there simply is no instruction that can load 1KiB of data atomically.

Yet `std::atomic<T>::load()` from the C++ standard library works for all `T`, as long as they're trivially copyable. How do they manage that?

One possible implementation can look like Listing 4. As they define a new type for atomic access, they can put additional members in there. In this case, a mutex to synchronize access. If all we have is a function that can't change the type, this isn't something we can do.

Writing a new type

So based on those disadvantages you decide to write a new type when you want to tweak the behaviour. A `saturating_int`, a `volatile_ptr`, an `atomic<T>`. It's a lot more boilerplate compared to the couple of free functions, but it's worth it, as you have the beauty of existing operators, the flexibility of adding additional state if necessary, and the safety guarantees the type system gives you.

However, the new situation isn't ideal either.

```
template <typename T>
class atomic
{
  T value;
  mutable std::mutex mutex;
public:
  T load() const
  {
    std::lock_guard<std::mutex> lock(mutex);
    return value;
  }
};
```

Listing 4

Disadvantage #1: Conversions everywhere

Suppose you want to do saturating arithmetic, but only sometimes; otherwise, you want overflow. As the behaviour is provided by types, you need to change types to change the behaviour:

```
int x = …;
saturating_int y = saturating_int(x) * 42;
int z = int(y) + 11;
saturating_int w = saturating_int(z) * 2;
```

For an `int`, this doesn't really matter, the compiler will optimize them away. But for bigger types? All of those conversions can add up and the poor CPU needs to constantly move stuff around.

Disadvantage #2: Different types

A `saturating_int` is not an `int`. Sure, you can provide a conversion operator to make them related, but this doesn't help in the case of `std::vector<saturating_int>` and `std::vector<int>`: they're entirely unrelated types.

Remember how I complained about having to pass `saturating_add` to `std::accumulate`? Well, if you start with a `std::vector<int>` as opposed to `std::vector<saturating_int>`, you're still out of luck. Your only option is to use C++20 ranges to provide a view that turns a `std::vector<int>` into a range of `saturating_int`. Or you just provide a custom operation.

A similar issue occurs when you decide to store a value somewhere. Do you store it as an `int`, as that's what it is, or as a `saturating_int` as that's how it's used? The types are different, you have to pick one.

The fundamental issue

There is a fundamental issue trade-off here we have to make: logically, we want to provide behaviour which is done by writing functions, but in the OOP model we need types to do it properly.

In C++, we always have this trade-off that we need to reason about. However, there are some hypothetical language changes that could be made to improve the situation.

> Disclaimer: They aren't serious proposals and don't work with C++ for multiple reasons.

Solution #1: Distinguish between 'layout' and 'type'

Right now, `int` and `saturating_int` are different types even though for the CPU they're essentially the same, only the function matters. So we can imagine that this underlying layout can be reasoned about in the language. C++20 already has the notion of 'layout compatible types' [cppreference], which matter for unions, let's build on top of that.

We can imagine a `layout_cast<T>(expr)` operator that changes the type of an object while keeping the layout intact:

```
int x = …;
auto y = layout_cast<saturating_int>(x);
```

Do you often need to mix different behaviours? Is it important that you can't accidentally forget the new behaviour? If so, write a new type.

This generates no assembly instructions, as nothing changes for the CPU, and it logically ends the lifetime of `x`. `y` is now a new object that lives at the same address as `x` and stores the same bit pattern, but has a different type. The only effect is a different overload resolution for its `operator+`.

This can then also be extended to containers:

```
std::vector<int> x = …;
auto y =
  layout_cast<std::vector<saturating_int>>(x);
```

Again, logically there is no difference between a bunch of `int`s and a bunch of `saturating_int`s, so the CPU doesn't need to do anything. Only the type has changed.

This allows us to change the behaviour without affecting actual runtime performance.

Solution #2: Packaging behaviour into a separate entity

Scala has an interesting take on the problem. Consider `std::accumulate()` again. It takes an additional operation that controls how 'addition' is performed as well as the initial value. Mathematically, that is called a Monoid [Wikipedia], it describes 'addition' as well as the identity of 'addition'. For `int`, that is `operator+` and `0`. However, it can also be `operator*` and `1`. As such, `std::accumulate()` accepts the range of input as well as the Monoid to use.

In Scala, the Monoid can be passed in a special way, as an implicit parameter. The example in Listing 5 is from their website [Scala].

We first define a `Monoid` as an interface that has addition and unit, we then implement it for strings and int, and write a generic function that sums a list. It accepts the Monoid as an implicit parameter which doesn't need to be passed on the call site. Instead, the compiler will search for the closest `implicit` value and pass that in.

The same principle can be applied to our problem as well. For example, we can define `overflowArithmetic` and `saturatingArithmetic` and then use something to indicate which one we want. This would then change the lookup of `operator+` and `operator*` in our algorithms accordingly.

Of course, this requires a way to easily specify a 'compile-time interface', like Rust has with traits. However, C++ decided against C++0x concepts, which makes it impossible to add something like that now.

```
abstract class Monoid[A] {
  def add(x: A, y: A): A
  def unit: A
}
object ImplicitTest {
  implicit val stringMonoid: Monoid[String] =
    new Monoid[String] {
  def add(x: String, y: String)
    : String = x concat y
  def unit: String = ""
}
implicit val intMonoid: Monoid[Int] =
    new Monoid[Int] {
  def add(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
def sum[A](xs: List[A])(implicit m: Monoid[A])
    : A =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail))

def main(args: Array[String]): Unit = {
  println(sum(List(1, 2, 3)))
    // uses intMonoid implicitly
  println(sum(List("a", "b", "c")))
    // uses stringMonoid implicitly
  }
}
```

Listing 5

Conclusion

Writing a new type to change the behaviour is strictly more powerful than writing a new function. As such, in situations where you have to write a new type (e.g. `std::atomic<T>`), the choice is easy.

In all other cases, it is a trade-off.

Do you often need to mix different behaviours? Is it important that you can't accidentally forget the new behaviour? If so, write a new type. Otherwise, write a function.

In an ideal world, where we have some way of decoupling layout from behaviour, this wouldn't be a problem. But we don't have that, so we have to live with trade-offs. Of course, we can also provide both versions. This is what Rust does with `wrapping_add` and `Wrapping<T>`. ■

References

[cppreference] std::is_layout_compatible: https://en.cppreference.com/w/cpp/types/is_layout_compatible

[Scala] Implicit parameters: https://docs.scala-lang.org/tour/implicit-parameters.html

[Wikipedia] Monoid: https://en.wikipedia.org/wiki/Monoid

This article was first published on Jonathan's blog (https://www.foonathan.net/2022/03/behavior-function-type/) on 30 March 2022.

Advancing the State of the Art for std::unordered_map Implementations

Unordered maps can be implemented in various ways. Joaquín M López Muñoz presents a new, fast version.

Several Boost authors have embarked on a project [Boost-1] to improve the performance of Boost.Unordered's implementation of `std::unordered_map` (and `multimap`, `set` and `multiset` variants), and to extend its portfolio of available containers to offer faster, non-standard alternatives based on open addressing.

The first goal of the project has been completed in time for Boost 1.80 (launching in August 2022). We describe here the technical innovations introduced in `boost::unordered_map` that makes it the fastest implementation of `std::unordered_map` on the market.

Closed vs. open addressing

On a first approximation, hash table implementations fall on either of two general classes:

- *Closed addressing* (also known as *separate chaining* [Wikipedia-1]) relies on an array of buckets, each of which points to a list of elements belonging to it. When a new element goes to an already occupied bucket, it is simply linked to the associated element list. Figure 1 depicts what we call the *textbook implementation* of closed addressing, arguably the simplest layout, and among the fastest, for this type of hash tables.
- *Open addressing* [Wikipedia-2] (or *closed hashing*) stores at most one element in each bucket (sometimes called a *slot*). When an element goes to an already occupied slot, some *probing* mechanism is used to locate an available slot, preferrably close to the original one.

Recent, high-performance hash tables use open addressing and leverage on its inherently better cache locality and on widely available SIMD [Wikpedia-3] operations. Closed addressing provides some functional advantages, though, and remains relevant as the required foundation for the implementation of `std::unodered_map`.

Restrictions on the implementation of std::unordered_map

The standardization of C++ unordered associative containers is based on Matt Austern's 2003 N1456 paper [Austern03]. Back in the day, open-addressing approaches were not regarded as sufficiently mature, so closed addressing was taken as the safe implementation of choice. Even though the C++ standard does not explicitly require that closed addressing must be used, the assumption that this is the case leaks through the public interface of `std::unordered_map`:

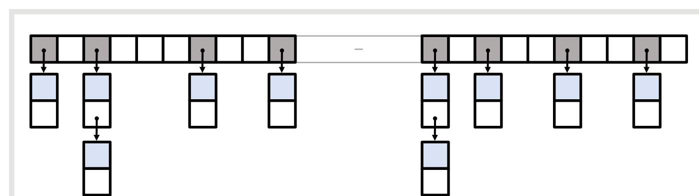


Figure 1

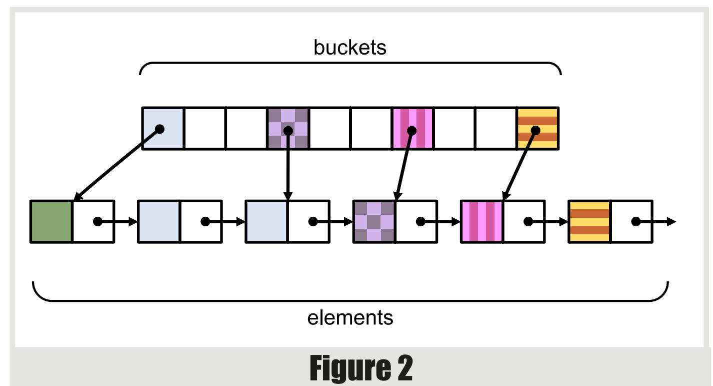


Figure 2

- A bucket API is provided.
- Pointer stability implies that the container is node-based. In C++17, this implication was made explicit with the introduction of `extract` capabilities.
- Users can control the container load factor.
- Requirements on the hash function are very lax (open addressing depends on high-quality hash functions with the ability to spread keys widely across the space of `std::size_t` values.)

As a result, all standard library implementations use some form of closed addressing for the internal structure of their `std::unordered_map` (and related containers).

Coming as an additional difficulty, there are two complexity requirements:

- iterator increment must be (amortized) constant time,
- `erase` must be constant time on average,

that rule out the textbook implementation of closed addressing (see N2023 [López-Muñoz06] for details). To cope with this problem, standard libraries depart from the textbook layout in ways that introduce speed and memory penalties: for instance, Figure 2 shows how libstdc++-v3 and libc++ layouts look.

To provide constant iterator increment, all nodes are linked together, which in its turn forces two adjustments to the data structure:

- Buckets point to the node *before* the first one in the bucket so as to preserve constant-time erasure.
- To detect the end of a bucket, the element hash value is added as a data member of the node itself (libstdc++-v3 opts for on-the-fly hash calculation under some circumstances).

Joaquín M López Muñoz is a telecommunications engineer freelancing in product/innovation/technological consultancy for telco, TV, and IoT. He is the author of three Boost libraries (MultiIndex, Flyweight, PolyCollection) and has made some minor contributions to the standard, such as N3657 (heterogeneous lookup). Contact him at joaquin.lopezmunoz@gmail.com

We use the modulo by a prime approach because it produces very good spreading even if hash values are not uniformly distributed.

Visual Studio standard library (formerly from Dinkumware) uses an entirely different approach to circumvent the problem, but the general outcome is that resulting data structures perform significantly worse than the textbook layout in terms of speed, memory consumption, or both.

Boost.Unordered 1.80 data layout

The new data layout used by Boost.Unordered goes back to the textbook approach (see Figure 3).

Unlike the rest of standard library implementations, nodes are not linked across the container but only within each bucket. This makes constant-time **`erase`** trivially implementable, but leaves unsolved the problem of constant-time iterator increment: to achieve it, we introduce so-called *bucket groups* (top of the diagram). Each bucket group consists of a 32/64-bit bucket occupancy mask plus **`next`** and **`prev`** pointers linking non-empty bucket groups together. Iteration across buckets resorts to a combination of bit manipulation operations on the bitmasks plus group traversal through **`next`** pointers, which is not only constant time but also very lightweight in terms of execution time and of memory overhead (4 bits per bucket).

Fast modulo

When inserting or looking for an element, hash table implementations need to map the element hash value into the array of buckets (or slots in the open-addressing case). There are two general approaches in common use:

- Bucket array sizes follow a sequence of prime numbers p, and mapping is of the form $h \rightarrow h \bmod p$.
- Bucket array sizes follow a power-of-two sequence 2^n, and mapping takes n bits from h. Typically it is the n least significant bits that are used, but in some cases, like when h is postprocessed to improve its uniformity via multiplication by a well-chosen constant m (such as defined by Fibonacci hashing [Wikipedia-4]), it is best to take the n **most** significant bits, that is, $h \rightarrow (h \times m) >> (N - n)$, where N is the bitwidth of **`std::size_t`** and **`>>`** is the usual C++ right shift operation.

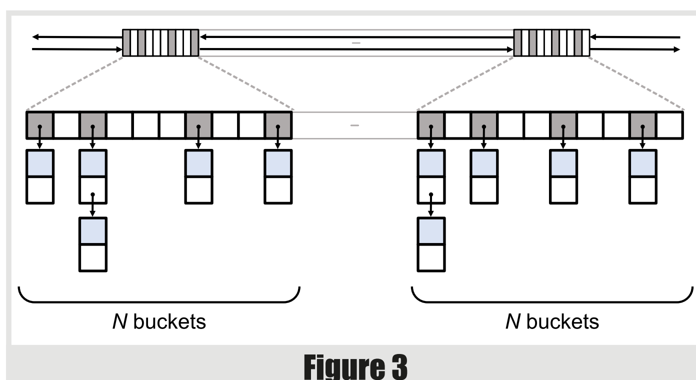


Figure 3

We use the modulo by a prime approach because it produces very good spreading even if hash values are not uniformly distributed. In modern CPUs, however, modulo is an expensive operation involving integer division; compilers, on the other hand, know how to perform modulo *by a constant* much more efficiently, so one possible optimization is to keep a table of pointers to functions $f_p : h \rightarrow h \bmod p$. This technique replaces expensive modulo calculation with a table jump plus a modulo-by-a-constant operation.

In Boost.Unordered 1.80, we have gone a step further. Daniel Lemire *et al.* [Lemire19] show how to calculate $h \bmod p$ as an operation involving some shifts and multiplications by p and a pre-computed c value acting as a sort of reciprocal of p. We have used this work to implement hash mapping as $h \rightarrow \text{fastmod}(h, p, c)$ (some details omitted). Note that, even though fastmod is generally faster than modulo by a constant, most performance gains actually come from the fact that we are eliminating the table jump needed to select f_p, which prevented code inlining.

Time and memory performance of Boost 1.80 boost::unordered_map

We are providing some benchmark results [Boost-2] of the **`boost::unordered_map`** against libstdc++-v3, libc++ and Visual Studio standard library for insertion, lookup and erasure scenarios. **`boost::unordered_map`** is mostly faster across the board, and in some cases significantly so. There are three factors contributing to this performance advantage:

- the very reduced memory footprint improves cache utilization,
- fast modulo is used,
- the new layout incurs one less pointer indirection than libstdc++-v3 and libc++ to access the elements of a bucket.

As for memory consumption, let N be the number of elements in a container with B buckets: the memory overheads (that is, memory allocated minus memory used strictly for the elements themselves) of the different implementations on 64-bit architectures are in Table 1 (overleaf).

Which hash container to choose

Opting for closed-addressing (which, in the realm of C++, is almost synonymous with using an implementation of **`std::unordered_map`**) or choosing a speed-oriented, open-addressing container is in practice not a clear-cut decision. Some factors favoring one or the other option are listed:

- **`std::unordered_map`**
  - The code uses some specific parts of its API-like node extraction, the bucket interface or the ability to set the maximum load factor, which are generally not available in open-addressing containers.

If you decide to use std::unordered_map, Boost.Unordered 1.80 now gives you the fastest, fully-conformant implementation on the market

Implementation	Memory overhead (bytes)
libstdc++-v3	16 *N* + 8 *B* (hash caching [GNU])
	8 *N* + 8 *B* (no hash caching)
libc++	16 *N* + 8 *B*
Visual Studio (Dinkumware)	16 *N* + 16 *B*
Boost.Unordered	8 *N* + 8.5 *B*

Table 1

  - Pointer stability and/or non-moveability of values required (though some open-addressing alternatives support these at the expense of reduced performance).
  - Constant-time iterator increment required.
  - Hash functions used are only mid-quality (open addressing requires that the hash function have very good key-spreading properties).
  - Equivalent key support, i.e. `unordered_multimap`/`unordered_multiset`, required. We do not know of any open-addressing container supporting equivalent keys.
- Open-addressing containers
  - Performance is the main concern.
  - Existing code can be adapted to a basically more stringent API and more demanding requirements on the element type (like moveability).
  - Hash functions are of good quality (or the default ones from the container provider are used).

If you decide to use `std::unordered_map`, Boost.Unordered 1.80 now gives you the fastest, fully-conformant implementation on the market.

Next steps

There are some further areas of improvement to `boost::unordered_map` that we will investigate post Boost 1.80:

- Reduce the memory overhead of the new layout from 4 bits to 3 bits per bucket.
- Speed up performance for equivalent key variants (`unordered_multimap`/`unordered_multiset`).

In parallel, we are working on the future `boost::unordered_flat_map`, our proposal for a top-speed, open-addressing container beyond the limitations imposed by `std::unordered_map` interface. Your feedback on our current and future work is very welcome. ■

Acknowledgements

The Boost.Unordered evolution project is being carried out by Peter Dimov, Christian Mazakas and the author. This work is funded by The C++ Alliance (https://cppalliance.org/).

References

[Austern03] ‘A Proposal to Add Hash Tables to the Standard Library (revision 4)’: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1456.html

[Boost-1] ‘Development Plan for Boost.Unordered’: https://pdimov.github.io/articles/unordered_dev_plan.html

[Boost-2] ‘Benchmarks’: https://www.boost.org/doc/libs/develop/libs/unordered/doc/html/unordered.html#benchmarks

[GNU] ‘Hash Code’ in *The GNU C++ Library Manual*, available at https://gcc.gnu.org/onlinedocs/libstdc++/manual/unordered_associative.html#containers.unordered.cache

[Lemire19] Daniel Lemire, Owen Kaser and Nathan Kurz, ‘Faster Remainder by Direct Computation: Applications to ompilers and Sofware Libraries’ from *Software: Practice and Experience* 49(6), available at https://arxiv.org/abs/1902.01961

[López-Muñoz06] ‘erase (iterator) for unordered containers should not return an iterator’: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2023.pdf

[Wikipedia-1] ‘Separate chaining’ in the topic ‘Hash table’: https://en.wikipedia.org/wiki/Hash_table#Separate_chaining

[Wikipedia-2] ‘Open addressing’ in the topic ‘Hash table’: https://en.wikipedia.org/wiki/Hash_table#Open_addressing

[Wikipedia-3] ‘Single instruction, multiple data’: https://en.wikipedia.org/wiki/Single_instruction,_multiple_data

[Wikipedia-4] ‘Fibonacci hashing’ in the topic ‘Hash table’ https://en.wikipedia.org/wiki/Hash_function#Fibonacci_hashing

This article was first published on Joaquín’s blog *Bannalia: trivial notes on themes diverse* (http://bannalia.blogspot.com/2022/06/advancing-state-of-art-for.html?m=1) on 18 June 2022.

Don't Block Doors

You can build simulations using cellular automata. Frances Buontempo uses this technique to demonstrate what happens if people stand in doorways.

Being in lockdown for much of the last couple of years, many of us feel a bit anxious about being in crowded spaces. You've probably seen models of how a pandemic spreads; those are fairly common. For my presentation at the ACCU conference in April, I used cellular automata to create some models. Instead of modeling the spread of a disease, I started from first principles – people moving in space. If we wanted to see how an infection spreads through a crowd, we could extend this example.

In this article, you'll learn how to make a simple cellular automata (CA) model. CAs date back to the 1940s and were introduced by von Neumann [Wikipedia-1], among others. People had grand ideas of sending robots into space to mine precious metals and wondered if the robots could be autonomous and even repair or rebuild themselves. This need for robot autonomy lead to a simplified idea of minimal units or *cells* following instructions – CA were born. They have tended to remain a bit of a niche curiosity; however, CAs can be used to model aspects of biology from patterns on shells to fibroblasts. You can also use them to model fluid flows [Wikipedia-2]. Whatever your motivation, CAs are surprisingly simple to code up and fun to watch.

People moving in space

Let's build a simple CA to model people moving in space. We can then watch what happens and see if we learn anything. We'll model the world as a two-dimensional space containing some blobs. The blobs start inside a paper bag and can move around. As a bonus, if the blobs manage to get out of the bag, we've coded our way out of a paper bag, which is a useful skill. The blobs could represent conference attendees moving through an atrium, represented by the bag. Maybe the attendees are heading to the bar or outside. We could make many possible simulations, but the simplest is to see the space as a grid (Figure 1).

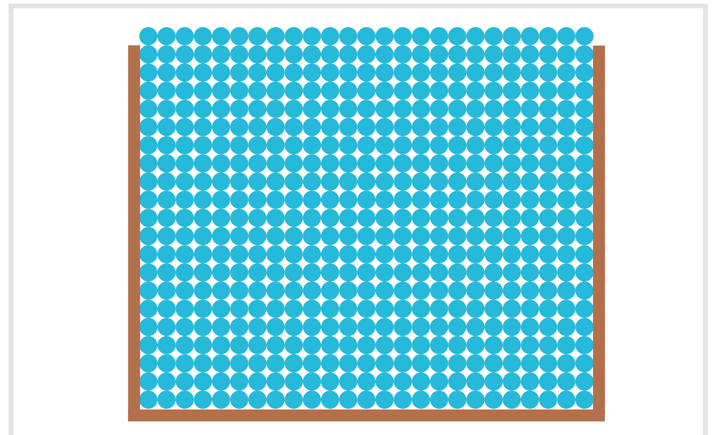


Figure 1

Each circle is a space a blob can occupy. All but the top row are in the paper bag/atrium – whatever we are modelling. The top row is outside the paper bag, simulating people moving outside an enclosed space, and arriving in the bar or another destination.

It's easy to vary the number of blobs or their starting positions, but imagine you have a row – maybe people who just left a talk or workshop (Figure 2).

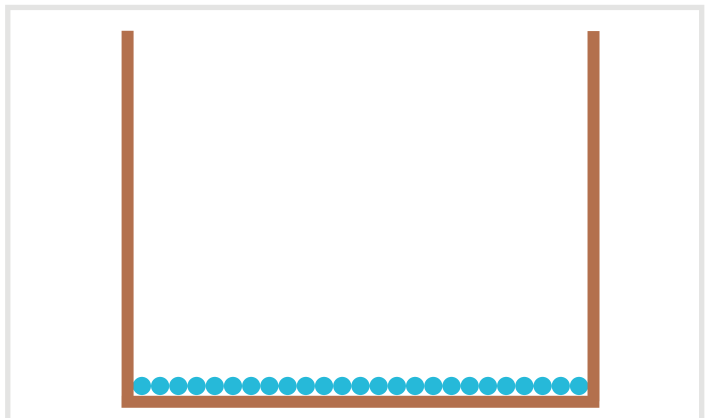


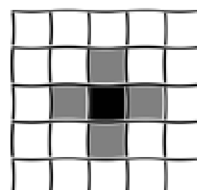
Figure 2

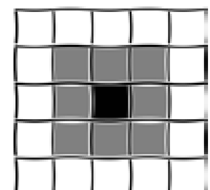
If they all walked forward at the same pace, we would have something very easy to code, but it wouldn't be much fun to watch and we wouldn't learn much. Instead, let's build stochastic cellular automata. Our blobs are the automata – they have agency and can therefore move. They move in the cells, so are cellular in that sense, rather than being living, breathing organisms. Finally, they are stochastic, in the sense that their movements are random.

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.

Starting positions and neighbours

Some cellular automata, such as Conway's *Game of Life* [Game-of-Life], have deterministic rules, though may initialize their grids at random. In this article, we've doing things the other way round: always placing the automata in the same starting position, but letting them make random moves. Rather than teleporting off to another cell, each automaton can only move only to an adjacent or neighboring square. In general, cellular automata tend to use one of two definitions of 'neighbour', von Neumann, or Moore.





Moore's schema includes the eight surrounding squares, up, down, left, right and the diagonals. Von Neumann's is the simplest, since it has only four, up, down, left, and right with no diagonals. Let's keep it simple.

Defining movement

So, we know where the blobs start, and where they can move to. Let's stop them from going through the sides or standing on each other, so the four neighboring cells may not be available. Let's also allow a blob to stay still if it chooses.

In order to make the cellular automata stochastic, we simply pick a possible, allowed move at random. For example, given a some possible moves, in C++ you form a new list of unoccupied positions:

```
std::vector<std::pair<int, int>> moves{{0, 0},
  {-1, 0}, {1, 0}, {0, 1}, {0, -1}};
```

Feel free to use a language of your choice.

The first number is a step left or right; -1 or 1 respectively. The second indicates up or down. {0, 0} means the blob stays still. Now, we could pick any of these options; however, if any move is equally likely, the blobs will amble around for a long time. If our attendees are aiming to escape the paper bag, we can encourage them to go up. Only one of the options involves up, {0, 1}, so making that combination more likely gives us what we need.

More complex options

You can do more sophisticated things, and then build all kinds of simulations. What we have done, in effect, is hard code what's called a static floor field. It's static because it doesn't change, and it's a floor field because, like a magnetic field or similar, it influences the blobs within it. In this case they tend to go up. We could ascribe weights or values to cells in the grid and use those instead, like this for a door or exit at the top. See Figure 3.

That scheme is over the top for this simple case, but would allow us to add obstacles and even have a dynamic floor field if we wanted.

An algorithm to move blobs

Now that we have things to move and know how to choose where they move, the simplest thing to do is let each blob move, one at a time. In theory, you could let them all move together, but you'd have to do something about blobs trying to move to the same grid square. If you make a grid class to keep track of where each blob is, the overall algorithm is as follows:

```
choose n (=25)
put n blobs in grid (at bottom of bag)
while True:
  draw bag
  for each Blob in grid:
    if blob in bag:
      move blob
  draw blobs
```

Blobs do gradually move towards the door. Success. However, since there is only one row at the top, they end up blocking the door and the later escapees get stuck – for quite a long time. For example, colouring the blobs in the bag cyan and changing them to magenta when they escape, you can see the escaped blobs partially blocking the door for the others (see Figure 4, and there's an animated version available [Buontempo22]).

A dynamic floor field, simulating the blobs noticing what's happening around them, might overcome this. After drawing the blobs in the loop, you could increase the field value of empty spaces. This change would encourage Blobs to move away from each other. Alternatively, adding several other rows so there's more space to move into might help too. This second solutions is akin to encouraging people not to block doorways. Something worth remembering if you've not been outside for a while!

Both options are simple to code up, but do tell us important things about crowd control and designing a space. The floor field simulates the directions people tend to move in. If you are trying to leave a conference

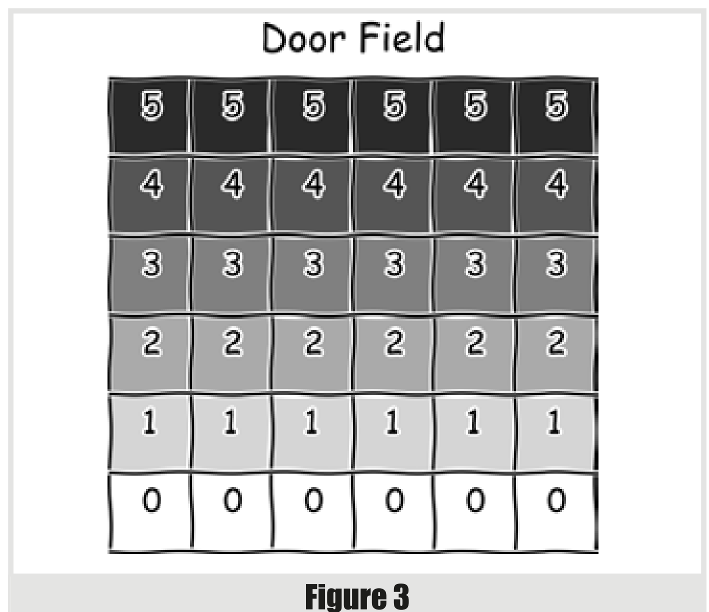


Figure 3

centre, an airport, or similar, there are exit signs, which are often lit up. The clear signage is meant to encourage people to move in a given direction, which clearly matters in the case of an emergency. Sometimes exits get blocked, so turning those exit signs off, and even introducing a one way system, might get people safely out of a building more quickly. You can then ask yourself *what-if* questions: what if we add more exit signs; try a one way system; and so on? You can use your simulation to measure the total time taken to evacuate everyone, or find out if some blobs get stuck or even trampled.

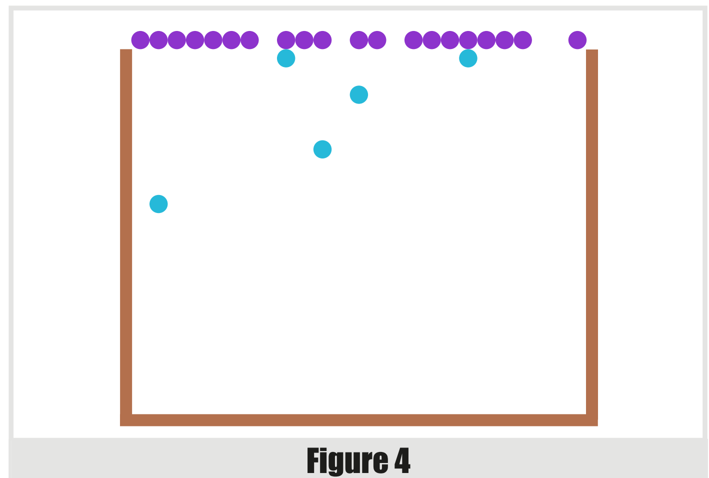


Figure 4

CAs are loads of fun. This stochastic cellular automata reminds us not to stand in doorways, but you can use CAs to design a space for safety – and so much more! Have a play with cellular automata. Try some *what-if* questions. For example, add some obstacles and see what happens. ■

References

[Buontempo22] Door blocking CA: https://www.youtube.com/watch?v=wlsbg5q0hO0

[Game-of-Life] Game of Life: https://playgameoflife.com/

[Wikipedia-1] Von Newmann cellular automaton: https://en.wikipedia.org/wiki/Von_Neumann_cellular_automaton

[Wikipedia-2] Lattice gas automaton: https://en.wikipedia.org/wiki/Lattice_gas_automaton

This article was first published online on *Pragmatic Programmers* on 15 June 2022 (https://medium.com/pragmatic-programmers/dont-block-doors-e38e7affbf56). You will find more information in Fran's ACCU Conference presentation. You may also enjoy *Genetic Algorithms and Machine Learning for Programmers*, published by The Pragmatic Bookshelf.

Lessons Learned After 20 Years of Software Engineering

It's good to sit back and reflect from time to time. Lucian Radu Teodorescu does just that and reports back.

15th of August 2002 – the date I started working as a professional software developer. I was still 18 at that time. I have been in this profession more than half my life. Enough time to hope that one can learn a thing or two about what it means to be a software engineer. This article explores 20 lessons that I learned (or I wish I had learned better) during my 20-year career.

Far be it from me to provide a list of clear guidelines for young software engineers based on my professional experience. But I do believe this could be a nice occasion to share a list of items that are hard to master; and, yes, I must confess that I am not an expert in most of the items below. However, I do believe that the sooner a software engineer connects with these items, the easier it is to acquire the needed skills. These items are not necessarily the end goal; they can be the means for improving oneself to be a better software engineer. An aspect on which I am continuing to work, as despite over-used clichés, I consider that learning must be a never-ending process.

1. Reading more software engineering literature

Software engineering should be based on science. Science is based on knowledge.[1] Knowledge is best obtained through reading books or journals. Reading about software engineering is thus essential for being a good professional.

Repeatable experiments are key to good science, but not every scientist and engineer needs to repeat all the relevant experiment; we should know the 'state of the art' in our field and not reinvent the wheel.

This is important, especially in our era. We are now bombarded by too many shortcuts for getting to needed information. There are wikis, small articles, small YouTube videos, and countless tweets. All of these seem to give the audience condensed information. That can be helpful sometimes, but it is not proper knowledge. To properly assimilate information and transform it into knowledge, one often needs to know the context around that information and to be able to fully reason about it.

For example, saying that QuickSort is $O(n \log n)$ on average misses a lot of the context where QuickSort can be used and what the guarantees are about it. One may need to know that the worst case is $O(n^2)$, that there are tricks to improve its performance, that it can be faster than other $O(n \log n)$ sorting algorithms, or that it is usually slower than an insertion sort for small number of elements, etc.

With all the benefits of being able to access information quickly, we tend to lose a lot of the things that form knowledge. New media simply doesn't allow the authors to expand on the context for the information they try to convey.

Looking from another perspective, literature can be more trustworthy than various blogs found on the internet. Good publishing houses have thorough review processes and try to keep high standards for everything they publish. Of course, books can sometime contain errors/misinformation, and getting quick access to information is sometimes good enough, but the general idea holds in many contexts.

2. Read more literature, in general

Reading about software engineering is good, but reading general literature (i.e., fiction) should not be ignored either. It can help a lot in self-development. In a way, it allows one to live more than one life. It promotes the development of one's understanding skills, it makes one much more capable of mastering various languages, and it contributes significantly to building knowledge in general.

The human mind does not work like a set of drawers (or like a hard disk) in which we put information in different compartments, disallowing any interaction between them. It's more like a complicated web of interactions between different parts. If a programmer looked at the brain, they would probably describe it as the biggest spaghetti code that ever existed.

Acquiring knowledge in one domain helps in other domains too. Learning to learn and to reason will help software engineering a lot, as our field can be described as applied epistemology (see below).

For example, looking at how many attempts we had before we sent people to the moon can provide good insights on how software engineering needs multiple iterations and refinement to solve complex problems.

3. Context is king

During my 20 years of professional activity, I often searched for good solutions that could be applied to all types of problems. For example, trying to search for the best programming paradigm, the best style of writing programs, the best way to approach concurrency, or the best way to architect a software system. But all my attempts were in vain; for each of these subjects, there isn't a general best solution.

All solutions depend on the context of the issue they try to address. Changing the context of the best solution can make the solution pretty bad compared to other solutions.

For example, there are cases in which monoliths are superior to microservices, despite the popular trend of moving from monolithic applications to microservices.

"It depends!" is often the right answer.

1 The word science comes from Latin *sciencia* which means knowledge. Nowadays, we often define science as the application of scientific method; this is a method of acquiring knowledge that involves making hypothesis (or models), driving predictions based on these hypotheses, and then verifying the predictions against reality. However we look at it, knowledge is at the heart of science.

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

4. Everything is a tradeoff

Even if the context of a problem is well understood, there isn't such a thing as a perfect solution. Every solution has downsides. I believe that part of our role as software engineers is to recognise these tradeoffs and provide the best solution, the one that best matches the goals of the project.

We often improve performance of a system by degrading its modifiability. Similarly, security is usually improved at the expense of usability. We frequently want to reduce coupling in an application, but the complete absence of coupling means that the two components cannot be used together; we need to have a tradeoff.

At the organisation level, there is always a tradeoff between being too conservative or being too progressive. A conservative organisation works with known tools and can be more predictable; however, it typically stays behind in the innovation game. A progressive organisation will use the latest version of each tool (even beta versions), and can innovate faster; on the other hand, it needs to spend too much time learning new things, that are then thrown away. The organisation must have a compromise between the two extremes.

I often say that, if you can't argue on both sides on a technical topic, you are probably just confused.

Drawing a parallel from philosophy, Aristotle founds his ethics on the principle of the golden mean, which is a tradeoff between two extremes [Aristotle]. He says:

> *virtue is a mean, […] a mean between two vices,*
> *the one involving excess, the other deficiency*

For example, Aristotle argues that the virtue of courage is the mean (to be read tradeoff) between cowardliness and recklessness.

And maybe this is just a small example of how general literature can help with new perspectives on software engineering. The truth is that we ought to properly see the tradeoffs with each solution we provide.

Maybe it's also worth mentioning that tradeoffs change over time. Well, to be more exact, it's not the tradeoff that changes over time, but the value that we associate with the alternatives involved in the tradeoff. In any case, the change over time is often important to be remembered; see also item 9 ('Document the decisions'). [2]

5. Software Engineering is applied epistemology

I learned this fact in the last few years from Kevlin Henney [Henney19], and it entirely changed my view of software engineering.

It may sound a bit precious to begin with, but once one thinks more about it, it makes sense. In our field, the main problem is not how to write code that machines understand, but rather to write code that humans understand; to be able to keep track of all the different parts of a complex system, to reason about it, and to ensure that the system grows according to the expectations. The main bottleneck is our mind, not the typing speed.

Thus, our main concern should be to organise knowledge in a structured and meaningful way. That is applied epistemology.

Taking this together with the previous item, it makes sense to say that software engineering can be closer to philosophy than we might naively acknowledge.

6. Software engineering != programming

For many years, I described myself as being a professional programmer. I don't do that anymore. We are (or should be) software engineers, not programmers. I found that there needs to be a big distinction between software engineers and programmers.

Similarly to the distinction between a carpenter and a mechanical engineer, and to the distinction between an electrician and an electrical engineer, we must have a distinction between a programmer and a software engineer. The main job of the engineers is to design, while the non-engineers typically just execute. The non-engineers might design small-scale things, but they don't have a structured approach to design.

I believe that being good software engineers really entails the following:

- basing our decisions on knowledge
- having a structured approach to design
- using empirical methods
- using iterations to improve knowledge

As Mary Shaw remarks in *Progress Toward an Engineering Discipline of Software* [Shaw15], one of the purposes of good engineering is to allow regular persons to do what previously could only be done by virtuosos.

And, let's not forget that engineers should solve real problems. That may seem obvious, but oftentimes we spend a lot of effort solving problems that we don't have (overgeneralization or problem misunderstanding). Talking to the customer for a couple of minutes can make the difference between implementing what the customer really wants and what we assume they want.

7. Apply knowledge and good reasoning in day-to-day work

When we design software in a certain way, we should not do it just because it feels right. We should make informed decisions, and we should be able to defend our decisions.

For example, we might design a certain system with a few classes that follow the *clean code* doctrine. The code looks right to us, and it looks right to our peers. But we should also be able to explain why that's the case. We should probably be able to explain that our system has low coupling, high cohesion, and it follows the SOLID principles.

As another example, if we decide to use a NoSQL database solution for a web service, we should have made a comparison with SQL databases and be able to show why the chosen solution is superior. Just using NoSQL because it is trendy is not a good argument. See also item 4 ('Everything is a tradeoff').

This type of reasoning needs to be applied to the important parts when designing a system, but it also has to be applied in the day-to-day job. This can be a hard thing to do, but that's the mark of a good engineer. Whenever one writes a new function, changes a class, uses an algorithm, it must follow the same knowledge-based reasoning approach. This process may not need to be shared with other engineers, but it has to happen in the head of the person writing the code.

8. Know the implications of the design

Having a design that solves a particular set of concerns is not enough. The design must perform well for any other concerns that might apply to the software system.

This phrasing is a bit abstract, so let's take an example. One might design a software sub-system to process some images. The business rules might be complex, so the design will focus on meeting those constraints. But, even if performance is not an explicit constraint of the problem, the engineer needs to be aware of the performance implications of the new design.

If one chooses an algorithm to solve a problem, one has to know the conditions in which the algorithm can operate, the performance characteristics, and the potential difficulties that using the algorithm might entail.

If one chooses a particular database engine, one must understand the performance characteristics, the functional capabilities, the scalability implications, the costs for using that database, and, of course, the development cost for using that database.

Every decision has a set of implications, and these should be well understood when taking the decision.

---

2 Or maybe this paragraph was just the result of my mind trying to argue both sides of the argument.

9. Document the decisions

Software development is applied epistemology, so one can consider it to be a large collection of decisions. The rationale behind the decisions might be forgotten with time. Furthermore, not all the engineers working on the project were involved with these decisions. Thus, it's important to document the decisions.

When documenting these decisions, one should document the context in which the decision was taken, the alternatives considered, tradeoff analysis between different alternatives and the impact of the decision.

The context is essential; if the context changes, then the decision may not be valid anymore. If we don't document the context, we never know when a decision is not applicable anymore.

While writing too much documentation can be a big burden, good judgement needs to be taken to document just the things that make sense to be documented. In my personal experience, I would document all architectural decisions, and other decisions that may not be intuitive, or for which the context may not be immediately apparent.

10. Find a way to explain technical details to newcomers

Explaining complex technical details to newcomers is an essential skill for software architects and a good-to-have skill for any software engineer. One should be able to explain technical details to new engineers on the team, to seasoned engineers that were not part of the design, to quality engineers, to management and sometimes to end users. All these stakeholders should have the same understanding of the problem/solution.

One can consider this to be a communication ability. But probably what's more important is the skill of abstracting out the unimportant details and focusing on the important aspects. Having good abstraction skills is required to be a good engineer, and tailoring your explanation to your audience helps develops those skills.

11. Design the system for others to understand it

In our profession, we often work in teams. We don't just have to build personal knowledge on the system we are designing, but instead we should build collective knowledge about it. This means that it's often more important for others to understand a design than the person doing the design.

And, considering the amount of information we subject our brain to, we soon start to forget why and how we designed the system the way it is. We often become the 'other'.

A good approach when designing a software system is to ask whether a Jon Doe can easily understand the design. If the answer is *yes*, then we probably did a good job designing.[3] If the answer is *no*, we should consider what needs to change to make it easier for another person to understanding it. Sometimes this means changing the design, sometimes it means documenting important aspects of the design, and sometimes this requires having discussions with other people to understand the pain points. Regardless of the solution, we should make sure the design can be easily understood by other people.

12. Testing is essential

Just to be clear, this item doesn't say that we need a Software Quality department in addition to the software engineers. Any software engineer needs to spend considerable amount of time performing testing activities.

I'm not saying this because I'm a sold TDD fan. I'm saying this because testing is a core aspect of engineering. As part of being empirical, we must consider that our hypotheses are wrong, and try to test them. The more empirical we are, the more testing we are doing.

There are multiple ways of testing that we can employ in software engineering (unit testing, integration testing, load testing, etc.), and typically more than one testing method is used in a given project. The mix of testing methods depends on the specifics of the software projects. But regardless of the type of project, performing the required testing is the mark of being a good engineer.

13. More solving problems than writing code

Our job as software engineers is to solve software problems. Not to write code. Sometimes, removing code is the best solution for a certain problem. Sometimes, changing the configuration solves the problem. Other times, we can just prove that the problem is just apparent, and that this is actually the best behaviour for the users (in the given context). Occasionally, it's just making sure that other engineers/teams have the required information to implement a simple solution on their end.

A typical engineer writes about 300 lines of code per day. If we just consider the typing part, this can be done in less than 5 minutes. That is, about 1% of the total work time for a software engineer. We need to be aware that the other 99% is dedicated to solving problems.

One important part of that 99% is design, but there are other activities that need not be neglected: communicating with others, writing documentation, analysing empirical data, creating models to better understand the consequences of a certain design, exploratory experimentation, etc.

14. Knowing the algorithms and data structures

Coming back to what it means to be a good software engineer, we need to base our work on prior knowledge. Knowledge about algorithms and data structure constitute a significant part of software engineering's body of knowledge.

Knowing algorithms and data structures is like knowing the vocabulary of a language. The better one knows the vocabulary, the better one can communicate in that language; better capture the intended meaning, and be more precise.

15. Innovate on the small items

We tend to associate innovation with significant changes in our industry, like the launch of the iPhone, the launch of iPad, the advancements in deep learning, etc. But innovative products are not built out of thin air. They often require a culture of innovation; this culture is typically built on small-scale innovations.

Innovation is improving a product or a process compared to a *de facto* standard.

I frequently give the following example: if the common practice in a team is to get all the emails into a single inbox folder, adding rules to automatically move some emails to dedicated folders can be a small-scale innovation. Previously, one had to manually sort emails, and after a certain number of emails received daily, this can become a burden, and a source of defocus. If all the emails on a particular subject would go to a dedicated folder, then the person is freed of some manual work, and, moreover, can be more focused on reading emails.

If one gets in the habit of improving small processes, then one gets the innovation habit. Sooner or later, this person will participate in larger innovations.

16. Learn by doing and do by learning

Learning is quintessential to software engineering. After all, we argued that we are doing applied epistemology; moreover, we are constantly building new things, increase complexity, and adopt new technologies. I don't believe that there will be a point in the career of a software engineer in which one can stop learning.

Reading books, watching YouTube lectures is a way of learning. But one needs also applied learning. Thus, one needs to learn by doing. After all, engineering has to be empirical.

However, I believe that the opposite can also be true. We can do spectacular stuff by learning. If we develop good learning methods, if we apply sound empirical processes, a learning experience can also lead to

3 Here, we just focus on the human understating aspect of the design. We take for granted the fact that the design meets all its goals (functional, quality attributes, constraints).

good software. Approaching new topics with intense curiosity increases our creativity. This can lead to innovation.

17. Learn from mistakes to get things right

It's hard to not make mistakes in our field. Be it estimates that are too optimistic, consequences that were not anticipated, unintentional bugs, or something else, we all make mistakes. One should not be afraid to make mistakes, as the mistakes are key to progress.

One problem with trying very hard to avoid mistakes is to enter analysis-paralysis; that is, to continuously delay the point at which the solution is considered ready. This increases the time needed to build software a lot, and cannot eliminate mistakes. If we are entirely ignoring the severity of the mistakes, having a failure rate of 5% with a speed of 100 features per year is more profitable than having a failure rate of 1% with just 10 features completed per year. In the first case, the engineer delivers 95 good features in a year, while in the second case they deliver just 9.9 good features per year (on average).

The other key aspect of making mistakes is the impact of these mistakes. One needs to make sure that their impact is kept under control. Thus, when implementing a feature, the engineer must assess and track the most important risks of the feature. If these are kept under control, the possible mistakes have small impact.

Mistakes are not entirely negative. They tend to help us learn faster. Empiricism is based on the fallibility of hypothesis; we make a prediction, and if that turns out to be false, we learn something new. Mistakes prove that our model is wrong, which often leads to improving our model and our understanding. This was the key to success in natural science, and can be the key to success in software engineering as well.

In short, if their impact is controlled, making mistakes fast can lead to good software and improved knowledge.

18. Being skeptical with everything, including self

Engineering is based on science, and according to Karl Popper, science needs to be built on falsifiable propositions. That is, we should assume that all propositions can be false. We should be skeptical about all the predictions we make about our software.

Being skeptical allows us to incorporate failure in our processes. As previously discussed, this allows us to continuously improve.

But just being skeptical isn't enough. We also need to measure key aspects of the software we are building. Without this measuring, we are blindly navigating a territory full of traps and dead ends. Continuous measurement and frequent iterations allows us to improve our models and build quality software.

Looking at the moon-landing for example, we did not achieve success at the first attempt. There were several dozen missions – some of them successful, some of them failures – that led us to expand our knowledge. With each mission, we learned something new, so that incrementally we build the right technology to accomplish the moon-landing goal. The key behind the iterations was a large dose of skepticism. We had to assume that we might be wrong, to carry on an experiment.

This skepticism should also apply to yourself. You are also fallible, and you should acknowledge this. Be your number one critic. Spotting your mistakes first is extremely beneficial for your personal growth, and it also gives others less chance to criticise you.

19. Own biases are problems that need to be managed

Continuing on the previous item, you should acknowledge your biases. You have to be an attentive observer of self, and then work out what your biases are. Knowing your biases allows you to compensate them so that you get the best out of you.

Personally, I'm an introvert. Although it doesn't come naturally, I learned to force myself to communicate even when I would rather just run away. I was also fearful of making mistakes; I learned that I should try more often, and that I should engage others to criticise me as early as possible so that I don't get the overall feeling that I completed something just to find out it was a mistake. Being too optimistic in estimating simple tasks is also a bias that I have (for complex tasks, I usually overestimate).

Think of your biases as problems that need to be tackled even if, most probably, you will not be able to eradicate them. Then apply empirical methods to improve yourself.

20. Never being done

People have claimed for centuries that physics is almost done. And yet, we continue to have big revolutions in physics.

Civil engineering (and its precursors) is an old discipline. By now, we expect that most of the things have already been tried, and somehow there is not much left for us to do in civil engineering. And yet, now and then we have innovation in this field.

We can't expect software engineering to be done soon. After all, this is about complexity, and we cannot find a way to simplify all the complexity involved.

We should not stop learning, experimenting, and improving ourselves. ■

Acknowledgement

During my 20 years as a software engineer, I have met many people who contributed in a significant and positive manner to my professional development. I always will be grateful for what they taught me, directly or indirectly. Still, I want to take this opportunity to express my appreciation for the very first person who, in a country which at that time did not encourage young students to take jobs, being rather reluctant to this idea, believed in my passion and offered me my first job. Thank you, Gina, for giving my career a wonderful start.

References

[Aristotle] Aristotle, *Nicomachean Ethics*, translated by W. D. Ross, http://classics.mit.edu/Aristotle/nicomachaen.mb.txt

[Henney19] Kevlin Henney, 'What Do You Mean?', *ACCU 2019*, https://www.youtube.com/watch?v=ndnvOElnyUg

[Shaw15] Mary Shaw, 'Progress Toward an Engineering Discipline of Software', *GOTO 2015*, https://www.youtube.com/watch?v=lLnsi522LS8

Advertise in C Vu & Overload

80% of readers make purchasing decisions, or recommend products for their organisations.

Reasonable rates. Flexible options. Discounts available to corporate members.

Contact ads@accu.org for info.

Afterwood

What's your legacy? Chris Oldwood considers what we can leave to make life better for others.

As I write, the UK is going through political turmoil. The current Prime Minister (PM) has finally stepped down as leader of his party and so we're in for a new PM in the coming months. Given the various shenanigans that have gone on over the last couple of years, it's unlikely that history will look kindly on him, although some of the more recent changes in policy feels like it comes straight from Orwell's 1984 so maybe we'll never unearth the truth.

Much like the last two Prime Ministers, my own tenure is shorter than a normal employee because I'm a freelancer. One consequence of moving on frequently is that I get to reflect on the 'legacy' that I'm leaving behind for the team. Hopefully, unlike the last few PMs, my parting gifts are looked upon with more affection.

Outside the world of software development, the term 'legacy' has a far more positive meaning. It's commonly used to convey some asset (money or property) that's left as part of a will. By and large, receiving a legacy is A Good Thing and relished by the recipient. Conversely, in the world of software, it's more generally used as a pejorative – inheriting a legacy codebase is more likely to be met with derision. If we take the extreme view put forward by Michael Feathers in his seminal book *Working Effectively with Legacy Code,* any code without tests is considered legacy code. Ergo, if you're not practising TDD then you're producing legacy code with every keypress until you pay off your debt by adding a test. When the PM said he "got Brexit done", what I heard was "it's code complete" – metaphorically, Brexit feels like a monster codebase with no tests.

Closely related and suffering a similar disparity in persona is the notion of inheritance. Finding out you have a long-lost, obscenely rich relative who you have inherited a massive estate from is the stuff which dreams are made of. In contrast, the thought of receiving a massive inheritance in a codebase fills one with dread. In typical programming fashion, the term is overloaded and not all inheritance is the work of the devil. *Implementation* Inheritance and deep class hierarchies are frowned upon due to the tight coupling they introduce, whereas *Interface* Inheritance is lauded for helping to loosen unnecessary coupling. In this work of fiction, the Open/Closed Principle is your overly literal uncle responsible for the quagmire of classes you find yourself wading through.

Refactoring parts of a codebase to make it both easier to understand and, more importantly, to change, is definitely the kind of the legacy we should all look to give and receive. George Orwell warned us in 1984 about people who had a habit of rewriting history, but I'd hope he would approve of its use to simplify code. Unlike in 1984, where any evidence of the past was eradicated, we have the wonders of version control to allow us to see how we got to the new state of affairs and why. Of course, version control tools come with their own problems and I'm sure Orwell would have plenty to say about squashed commits and rebasing.

For sure, improving the quality of the product's production code is a rewarding legacy to pass on, but for me it's also the easiest to justify and therefore also perhaps the least contentious. Personally, I look to improving those aspects of the software delivery process which are a little harder to instigate (often for political reasons) or less valued by others, but only because they may not realise what a difference it can eventually make.

Automated builds are far more common these days, but also being reliable and easy to reproduce locally is still a gap that frequently needs plugging. Before the rise of 'DevOps' as a more formal role, it was left to the developers to try and lash something together in and around their other duties. Much like testing, the ability to build and deploy the product took a back seat, and so taking on that 'poisoned chalice' feels like a challenge worth tackling. Making it easy to go from 'works on my machine' to 'also works on the production machine' really helps the flow. Sadly, the rise of so called 'continuous integration' products has meant that I now see 'broken on the build server' coupled with 'can't reproduce on my machine' because the developers can no longer build, test, and deploy the product locally in the same way as the 3rd party product does it. Closing this gap always pays dividends in the end when it really matters.

Another task which rarely gets any TLC is documentation. The emphasis is traditionally on trying to make the code as readable as possible to make comments redundant. However, as Grady Booch likes to say, "the code is the truth, but it's not the whole truth". Matters of architecture and design, such as the rationale cannot be reflected in the code, only the outcome of the decisions. Similarly, even if you represent your platform as code, forcing someone to mentally derive the overall architecture by reading your Terraform scripts is not a pleasurable experience. Adding Architecture Decision Records and, say, a C4 model of the system goes a long way to helping newer team members understand the journey. Also having a wiki is a great start. What's even better though is having some actual content in it. In general, I feel that developer documentation appears to be in the same state that developer tests were a decade ago – they were rarely found and, if they did exist, were poorly written. Giving a wiki some structure, content, and style so that it can be browsed as well as searched is probably my other preferred contribution that I hope makes my successors nod in approval.

Unlike the departing PM, I do not see my legacy as a medium for getting my ego massaged. My legacy is not some kind of monument to be revered; on the contrary, it's more like the scaffolding that surrounds it enabling the team to work in safety. If it goes unnoticed that's not a bad thing, if anything that's an even bigger compliment as it means it's not become a source of friction. ■

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

CARE about code?

passionate about programming?

Join ACCU

www.accu.org

ACCU

professionalism in programming

- Monthly journals, available printed and online
- Discounted rate for the ACCU Conference
- Email discussion lists
- Technical book reviews
- Local groups run by ACCU members

ACCU is a not-for-profit organisation.

Become a member and support your programming community.

www.ACCU.org