overload 162
APRIL 2021 £4.50

Amongst Our Weaponry
Learn how to use records to define types in C#

Composition and Decomposition of Task Systems
Concurrency can be hard to get right, but tasks can help.

Chepurni Multimethods for Contemporary C++
Showcasing an approach that uses custom type identification and introspection.

<script>
A different look at some well-known plays, setting them in a programmer's world.

A magazine of ACCU
ISSN: 1354-3172

OVERLOAD 162

April 2021

ISSN 1354-3172

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Vincent Maret on Unsplash.

Copy deadlines

All articles intended for publication in Overload 163 should be submitted by 1st May 2021 and those for Overload 164 by 1st July 2021.

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

Keep up at the back

It's hard to keep up in a changing world. Frances Buontempo wonders how to deal with the constant state of flux.

I am not keeping up, as evidenced by yet another lack of editorial. Trying to keep on top of things can be a challenge. I didn't use my diary much last year. I'm sure I'm not the only one. I don't have one this year and sometimes forget what the date is. That is not a great way to be organized and prepare for conference talks, write editorials and the rest. Aside from day to day troubles such as knowing date or even day it is, keeping up, particularly in tech, is hard work. I have recently strayed into a world of AWS, terraform and Java, taking me well outside my usual world of C++, Python and C#. Learning new languages and tools is tremendous fun, but trying to cram loads of new things into my head sometimes leaves me feeling like my brain is full. How do you keep up with new technology, language changes and the like? I find ACCU very helpful, either being able to chuck an email at ACCU general, or chat with other members. Sometimes someone can be persuaded to write an article or give a talk, providing an executive summary of the salient points to give you a leg up.

Learning a new language, programming or human can be difficult. If you know one language well, this can cause confusion as you try to get your head round a second language. A little bit of knowledge can be dangerous. If words sound similar it's tempting to guess meanings. Pronunciation can have subtle differences. You need practice and help from a native speaker. Throwing yourself into an unfamiliar culture requires acceptance of new idioms, sentence orderings and much more besides. Translation is hard. Some software vendors offer "international" versions of their products, tending to mean non-English versions. Using the word international hints at an old bias here! Nonetheless, one to one translations of words don't always fit on menus and picking a direct translation of a word, say "bullet" when internationalizing a word processor may end up suggesting gunfire rather than spherical symbols [Lepouras99]. Even if people speak the same native language, different experience or roles can cause confusion. A BA I work with regularly devolves into laughter when I mention strings. He thinks of the Goon Show's Great String Robberies [GoonShow], whereas I am talking about a series of bytes containing some characters. "Oh, dear, dear. Oh, dear, dear, dear, dear, oh, dear" as a character says in that very sketch. I would tell him about words being numbers in my universe, but he's not ready for it yet. Different contexts change the semantics of symbols and words in human communication.

The same goes for programming languages. If you start out knowing how to write procedural code in C, there might be a temptation to write procedural code in every language. Good luck with that. The idea of a different context strays beyond the syntax and semantics of the language used. Code that behaves one way on one operating system may behave differently, or not even run on another. Line ending vary between platforms. Endianness is a thing. How big is an integer? Some many differences. Watch out for byte ordering marks when you try to open a file on Linux that a Windows machine happily opened via Python code. ProTip: set the encoding, once you have figured out what the encoding is. Finding that out is another story. The OS is one thing, but the device itself is another. I keep a blog [Buontempo]. I regularly get emails telling me the links are too close together on a mobile and other related problems. I recently discovered you can use developer tools to "toggle device" to see just how bad it might be on various different mobile phones etc. If I find the time I might try to sort this out. Maybe. First, I need to get my head round AWS lambdas, and why they can sometimes be slow in surprising ways. How do you find out what's going on when the code hits the metal when it's not your metal? I am experiencing a paradigm shift, and doing my best to keep up.

Learning new technology affects many people, not just programmers. Many people use smart phone or have a laptop who don't have a clue what's happening inside, which is fine. I recall struggling with keyboards and the "mouse" on a laptop the first time I tried to use one. You should see what happened when I try to use a touch screen for the first time – long finger nails complicate matters. The switch to online banking has upset some people. This forced new way of transacting, with little choice where local bank branches have shut, can cause upset. When people can't understand a new procedure that had become a simple task due to familiarity previously, they can feel and even become excluded. "It was simpler in the olden days" goes the cry. Truth be told, it probably wasn't; you learnt some new-fangled things way back when, forget the other stuff that was way more complicated and your memory is selective. Nonetheless, many people struggle with change. In my experience, many techy people have a life-long love of learning, so may tend to embrace change. This does sometimes leave us as IT support for family and neighbours, which can be a time consuming role. I personally love learning *some* new things. Happy to try new programming languages, find new ways of testing, make algorithms quicker and easier to understand. Bring it on. But, move the buttons in a new version of a word processor, IDE or similar and I might start muttering very loudly. Hide the buttons in a ribbons and make me find how to make the button appear, and I'll go back to my editor of choice and leave your new "improved" software closed for weeks. Give me a website with stealth scroll bars, those that only appear if I get my mouse in the right place, and I'll be tempted to curl from a prompt. As stated, I like learning *some* new things and I suspect I'm not alone in this. Other changes are annoying. I'll get used to it, then they'll move my buttons, again.

Some things will never change, right? Like University Challenge on television on a Monday night in the UK, so I can have a quiet night in and

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.

focus on something other than failing to write an editorial. It has been running since the sixties, though has suffered a long hiatus in the 1980s being revived in 1995 [Wikipedia]. The quiz show is open to students representing a university and, in the case of Oxford or Cambridge, a specific college. Despite mumblings of elitism, due to these special rules for "Oxbridge" and that they frequently win, I still enjoy watching and trying to answer the questions. Recently I have noticed I can't answer many questions on scientists or mathematicians. Why? I learnt the names mainly via the theorems and theories I studied and by reading some history of science books many years ago. Back then, most of the names I learnt were of white men, tending to live in Europe or the USA. University Challenge has started an annoying habit of asking about women. I can't name many female mathematicians or scientists. Why? Because I was only taught about the guys. I love that the students seem to know many of the answers. I hope teaching has changed to be more inclusive. I clearly need to keep up, by expanding my horizons a bit, and updating my inaccurate, biased view of history. From now on, if anything seems to only involve white guys, I am going to do a bit of digging, and find out whether this is really the case. If it is, I will ask what can be done about it. I realise Overload has an international audience and the white guy syndrome is not a universal plague. Some cultures remember great females or know their BAME superstars.

Does not knowing the women involved in STEM make me sexist? No. Would assuming a women can't be involved in STEM be sexist? Maybe. I have sometimes assumed women won't be involved in the deep technical stuff, based on my personal experience. I am often the only women on a team. If I meet another woman, I am often initially surprised if she can code in C++ or similar. I then notice my assumptions and move on to be delighted. I bet I am not the only one. In fact, I know I'm not. Recently Steve gave a talk about Records, having written the article for this issue of Overload (thanks Steve!). When we both joined the online Meetup, an organizer asked if I was a tester, as though me being able to code hadn't occurred to him. I understand and make a mistaken assumption as well far too often. Note to self: a better question is always, "What do you do?" You can hide your bias that way, and have a quiet word with yourself later. Your previous experience is no indicator of future performance. Your previous experience is your previous experience. Your future experience can and will be different. Everything changes eventually. Even you. Or me.

How do we keep up as things change? First, notice the change. If it's a moved button then give up (if you're like me). If it's more important, take time to learn. Read. Listen. Try new things. Don't be afraid. If you feel like you are falling behind, in terms of new technology or language features or anything tech, find someone to help you out. However young or old you feel, ACCU is here to help. Get in touch, write for us, join us: ask us anything. I'd love to see the study groups revived. These are open to ACCU members, and previously tended to work through a tech book, trying out coding exercises. Sharing together with others is a great way to learn. Learning isn't a race. You don't need to come out top of class, or win University Challenge. Slow and steady is fine. I frequently feel as though I struggle to understand new things when I compare myself to others. I have a deep sense of whether or not I have truly grokked a new idea or concept. This means I will often be muttering "But I don't get it" while others are forging ahead, possibly missing some subtleties that are confusing me silly. I need to remind myself about the hare and tortoise having a race in one of Aesop's fables. A hare can run fast. A tortoise cannot. And yet, the tortoise wins, because the hare, confident in his abilities, has a nap during their race. As the Old Testament book Ecclesiastes says, "The race is not to the swift." Do beware going down a rabbit hole though. Sometimes you need to time box a task and give up if you are being slower than a tortoise.

Change brings new things but sometimes takes away the old. That can be hard to deal with. I have chucked out VHS videos because I can't play them. I left a tape player behind when I moved once. I regret this. So many demo tapes from so many friends' bands. But, you do need to let go sometimes. I had the honour of remotely attending Russel Winder's funeral remotely in March. Many ACCU people joined in too. Allow me to quote some lyrics from Fix You by Coldplay, a track used during the funeral:

> Tears stream down your face
> When you lose something you cannot replace

Change can be difficult. Russel will be sorely missed. He has left ripples on the internet, and many articles, including several in this very magazine. Despite my earlier complaints about a variety of annoying aspects of technology, the internet allowed me to join in. Technology can be an enabler. During the pandemic, Grayson Perry made a television programme called "Grayson's Art Club" [ArtClub]. The second series started recently. He's been encouraging people to submit art on a different theme each week, embracing people without formal training. I have found this delightful and encouraging. One stand out moment, was a young woman who had submitted a picture drawn on a tablet of some sort. She has restricted movement, I forget the specifics, making holding a pen or paintbrush impossible. Her submission had been drawn by moving her eyes and the technology translating that into brush strokes and lines. She said it gave her control and freedom, in a way that nothing else in her life did. Let's attempt to embrace change, celebrate the freedom great tech can bring and try to make sure no one gets left behind. Help everyone at the back keep up.

References

[ArtClub] Grayson's Art Club: https://www.channel4.com/programmes/graysons-art-club

[Buontempo] Buontempo Consulting (Fran's blog): https://buontempoconsulting.blogspot.com/

[GoonShow] The Goon Show: 'String Robberies': https://www.bbc.co.uk/programmes/b007jpkr

[Lepouras99] Lepouras, Giorgos and Weir, George R. S.: "It's Not Greek to Me: Terminology and the Second Language Problem", 1999, SIGCHI Bulletin, Association for Computing Machinery, 31.2 https://dl.acm.org/doi/pdf/10.1145/329657.329664

[Wikipedia] University Challenge: https://en.wikipedia.org/wiki/University_Challenge

Amongst Our Weaponry

Records in C# v9.0 and .Net 5.0. Steve Love explains a new way to define types in C#.

The release of .Net 5.0 in November 2020 was a major upgrade, bringing .Net Core and .Net Framework together under a single banner with a number of improvements, updates, and fixes. The release also represents an update to the C# language, and .Net 5.0 brings C# to version 9.0. One of the flagship features of C# v9.0 is the **`record`**, a new way to create user-defined types.

C# has supported classes and structs since it was introduced in the early 2000s, and the general concept of user-defined types goes back to the 1960s. A reasonable question to ask, then, is why does C# need a new way to create user-defined types?

In this article we'll look at what records are useful for, and how to use them in our own code. We'll contrast them with classes and structs, the other main ways of creating user-defined types in C#, with some common use-cases and examples. We will also look at some of the performance characteristics of records at a high level.

Let's begin with some examples of their main features.

What is a record?

Records are a light-weight way to define a type that has value semantics for the purposes of comparing two variables for equality. Here is the full definition of a simple record type:

```
public record Point(int X, int Y);
```

The **`Point`** type defined here is almost as simple as it could be. The record definition itself hides the fact that a **`Point`** has two **`int`** properties named **`X`** and **`Y`**, and a constructor taking two **`int`** parameters to initialize those property values. The syntax shown here is known as ***Positional Record***. The **`X`** and **`Y`** parameters in the record declaration correspond to properties of the same name and type in the **`Point`** record. How we create an instance of a **`Point`** and access its properties will be familiar to any C# developer:

```
var coord = new Point(2, 3);
Assert.That(coord.X, Is.EqualTo(2));
Assert.That(coord.Y, Is.EqualTo(3));
```

Those generated properties have no **`set`** accessors, so **`Point`** is *immutable*. Once we've created an instance of **`Point`**, its value can never be changed. This is in keeping with the common recommendation to make *all* value types, and value-like types, immutable. Some familiar examples include **`string`**, which is a class but has value-like semantics, and **`System.DateTime`**, which is a struct.

Equality

When we compare two record instances to determine if they're equal, their *values* are compared. For two **`Point`** variables, this means that both the **`X`** and **`Y`** properties match. This is similar behaviour to comparing two struct instances, but differs from most class types. We use reference variables to manipulate class instances on the heap. Two references compare equal if they refer to the same instance in memory.

The following code shows two **`Point`** values being compared for equality in different ways. Firstly, they're compared using **`==`** which gives a value-based comparison. Next, they're compared using **`ReferenceEquals`**, a method defined on the **`object`** base class that returns **`true`** if two variables refer to the same instance. Note that deliberately performing a reference comparison gives a different result from a direct equality check:

```
public record Point(int X, int Y);
var coord = new Point(2, 3);
var copy = new Point(2, 3);
Assert.That(coord == copy, Is.True);
Assert.That(ReferenceEquals(coord, copy),
  Is.False);
```

Two instances of the **`Point`** record compare equal if all of their respective properties compare equal, irrespective of whether they refer to the same object in memory. This is a defining characteristic of *value types*. More generally, two record instances compare equal if they are the same type and all their properties also compare equal.

Copying

Records are in fact *reference* types. They are allocated on the heap, are subject to garbage collection, and under normal circumstances are *copied* by reference. If we assign one **`Point`** variable to another, as shown below, we have two references to the same **`Point`** instance:

```
var coord = new Point(2, 3);
var copy = coord;
Assert.That(ReferenceEquals(coord, copy),
  Is.True);
```

There are times when we need to copy a value, but change only some of its properties. Records have an associated feature called ***non-destructive mutation*** which allows us to create a new instance from an existing one, but with some altered properties. When we assign one record variable to another, we can add a with clause to the assignment, as shown here:

```
var coord = new Point(2, 3);
var copy = coord with { Y = 30 };
```

In this example, **`copy`** is an independent instance of **`Point`**. It's a copy of the original **`coord`** record, except that it has a different value for the **`Y`** property. The **`X`** property of **`copy`** is taken from the corresponding **`X`** property of **`coord`**, and is unchanged in **`copy`**. Again, we can confirm this with a few tests:

```
Assert.That(coord.Y, Is.EqualTo(3));
Assert.That(copy.Y, Is.EqualTo(30));
Assert.That(coord.X, Is.EqualTo(copy.X));
Assert.That(ReferenceEquals(coord, copy),
  Is.False);
```

Since there are only two properties in this example, the benefit of using the **`with`** syntax in this way isn't immediately obvious. For records with several properties, however, this approach may be significantly more

Steve Love has been a professional programmer for over 20 years and is still finding new ways to be lazy. He can be contacted at steve@arventech.com

Records can only inherit from other records, so we can't inherit a record type from a class, nor can a class derive from a record

compact than the alternative of creating a new instance, and passing in a mixture of values and properties from an existing record to the constructor.

Deconstruction

We've examined *construction* of records so far, so in the interest of balance let's have a look at *deconstruction*. This is the process of capturing the component properties of a record into individual variables, like this:

```
var coord = new Point(12, 34);
var (x, y) = coord;
Assert.That(x, Is.EqualTo(coord.X));
Assert.That(y, Is.EqualTo(coord.Y));
```

Here, the `coord` variable is being *deconstructed* into the named variables `x` and `y`. We probably wouldn't use deconstruction directly like this; it's more useful when we call a method that returns a record instance but we want separate variables. Note that the names of the individual variables can be different to the property names in the record. We can use any valid variable name for those variables.

Sometimes we don't need to capture all the components because we're only interested in a subset of the values. Instead of creating a variable that is never used, we can use the underscore as a placeholder like this:

```
public Point ParsePoint(string coordinate)
{
  // ...
}
var (_, height) = ParsePoint("3,2");
Assert.That(height, Is.EqualTo(2));
```

In this example, only the *second* component of the record – the `Y` property – is copied to the named `height` variable. The placeholder, known as a ***discard***, tells the compiler to ignore the `X` property of the `Point` record. For records with more than two properties we can use the underscore identifier to discard multiple values.

String representation

Record types have a built-in consistent string representation available using the `ToString` method. This method is available to *all* types, since it's defined on the object base class. However, unless we override it for ourselves in classes and structs, it returns just the name of the *type*, qualified with its namespace.

Calling `ToString` on a record instance, however, returns just the type name along with the names and values of all of the properties, like this:

```
var coord = new Point(12, 34);
Console.WriteLine(coord);
```

This gives the output:

```
Point { X = 12, Y = 34 }
```

The `Console.WriteLine` method calls `ToString` to obtain the representation. We might also use string interpolation to embellish the output with the variable name like this:

```
Console.WriteLine($"{nameof(coord)} = {coord}");
```

Giving the output:

```
coord = Point { X = 12, Y = 34 }
```

Although logging is the obvious choice as a good candidate for uses like this, there are other potential benefits because the output is easily parsed to re-create an object, although records *don't* provide that facility for us – we have to write that ourselves.

Inheritance

We can inherit one record type from another in exactly the same way as we can with classes, and the semantics are broadly the same as for class inheritance. For example, we can use a base-class reference to an inherited record, and we can cast from one type to another. In the following example, we derive a `Point3d` record from our `Point` record:

```
public record Point(int X, int Y);
public record Point3d(int X, int Y, int Z) :
  Point(X, Y);
var coord3d = new Point3d(2, 3, 4);
var coord = (Point)coord3d;
Point point = new Point3d(2, 3, 4);
var point3d = point as Point3d;
```

If we attempt to cast from a base type to a more derived type when the conversion isn't valid, we get an `InvalidCastException`, just as with classes:

```
var coord = new Point(2, 3);
Assert.That(() => (Point3d)coord,
Throws.TypeOf<InvalidCastException>());
```

Also in exactly the same way as we would with a derived *class*, we can use an instance of a derived record as an argument to a method expecting a base class record, as shown here:

```
public double LineDistance(Point a, Point b)
{
  // ...
}
var pointa = new Point3d(2, 5, 4);
var pointb = new Point3d(6, 8, 4);
Assert.That(LineDistance(pointa, pointb),
  Is.EqualTo(5));
```

Records can only inherit from other records, so we can't inherit a record type from a class, nor can a class derive from a record.

We can *seal* a record to prevent it from being inherited. Once again, the syntax is identical to that we'd use for a class, as shown here:

```
public sealed record Speed(double Amount);
```

It's common for value types to be `sealed` when they're modelled as classes. The built-in `string` class is a case in point, and *all* struct types are effectively implicitly sealed. Conceptually, values are fine-grained bits of information, often representing transient and even trivial bits of data. Values are different from *entities* most clearly in that value types place great importance on their *state*.

We determine if two values are equal according to the value they represent. This differs from entity types where they compare equal if they represent the same instance in memory

Value semantics

When we say that value types place great importance on their state rather than their identity, what we really mean is that we determine if two values are *equal* according to the value they represent. This differs from entity types where they compare equal if they represent the same instance in memory. This latter behaviour is often called *reference semantics*, where we can have more than one reference to an object instance.

Struct instances are all independent of each other. They have true value semantics in that we *can't* generally have two variables representing the same instance. Structs are copied *by value*, so when we assign one to another we get a whole new distinct instance. However, since two values compare equal if they have the same state, it makes no difference that they're distinct instances.

Records live in a middle ground. Under the covers, records are really classes and so instances are copied *by reference*. When we assign one record variable to another, we get a new reference to the same instance in memory, unless we explicitly ask for a copy using the `with` syntax.

However, records are *value like* in that when we compare them for equality, it's their *state* that's compared, not their identity.

This behaviour of comparing values instead of identities is much the same as for the `string` type. `string` is a class, and so is a reference type. Strings are copied by reference, so the contents of a string variable aren't usually copied. However, strings have *value-like* behaviour for the purposes of equality comparisons. The `string` class overrides the `Equals` method, and implements the `IEquatable<string>` interface, which defines a type-specific overload for the `Equals` method.

String also has an `operator==` definition, which overrides the behaviour of the built-in comparison with `==`. When we compare two string variables using either the `Equals` method or using `==`, we're determining if the two strings have the same *value*, whether or not they refer to the same string instance.

Equal by value

We can emulate value based comparison in our own classes by overriding the `Equals(object?)` method, implementing `IEquatable` for our type, and by providing both `operator==` and `operator!=`. There are some subtleties and potential pitfalls to be aware of in those implementations, including the need to handle `null` values correctly, and making sure we correctly handle any possible base class implementations. If we were to implement our `Point` type as a class, it might look something like Listing 1.

Note that if we override the `Equals(object?)` method, we also need to override `GetHashCode`. If we only override one or the other, we'll get a warning from the compiler. The reason it's important is that two objects that compare equal should also have equal hash codes. If we fail to observe this rule, we risk being unable to find objects that are used as keys in collections that depend on hash codes for lookup, such as `Dictionary` and `HashSet`.

```
public class Point : IEquatable<Point>
{
  public Point(int x, int y)
    => (X, Y) = (x, y);
  public int X { get; }
  public int Y { get; }
  public bool Equals(Point? other)
     => !ReferenceEquals(other, null) &&
        GetType() == other.GetType() &&
        X == other.X && Y == other.Y;
  public override bool Equals(object? obj)
     => Equals(obj as Point);
  public override int GetHashCode()
     => HashCode.Combine(X, Y);
  public static bool operator==(Point? left,
    Point? right)
    => left?.Equals(right) ??
       ReferenceEquals(right, null);
  public static bool operator!=(Point? left,
    Point? right)
    => !(left == right);
}
```

Listing 1

The overridden `GetHashCode` method in the class shown above might not be the most efficient implementation, but it does guarantee that if two instances of `Point` are *equal*, they will also definitely have the same hash code.

With record types, the *compiler* provides the implementations for each of those members. The code generated by the compiler takes all of the fields declared in the record into account to provide a *value-based* equality comparison. When we create our own record types, we're freed from the need to provide all of this boilerplate code just to be able to compare the values of two variables.

What about structs?

Instead of using a class, we can also model our own types using a `struct`. All structs derive implicitly from the `System.ValueType` class which provides the necessary overrides to give structs value semantics when we compare them for equality. In addition to the `Equals` method, `ValueType` also overrides `GetHashCode` in a way that ensures that equal instances have matching hash codes.

We might therefore choose to model our `Point` type as a struct like Listing 2.

This is significantly simpler than our class definition for `Point`, and only a little more verbose than the record version. There are limitations to structs, however.

The first thing to note is that we can't compare two struct instances with `==` unless we provide our own implementation of `operator==`. The implementation of that is straightforward enough, however, and with a

the derived class determines that the properties specific to it are equal, and if they are, it defers to the base class to perform its own comparison

```
public struct Point
{
  public Point(int x, int y)
    => (X, Y) = (x, y);
  public int X { get; }
  public int Y { get; }
}
```

Listing 2

matching `operator!=` it looks very much like the version for the class implementation (Listing 3).

Struct instances can't normally be `null`, but our implementations of `==` and `!=` here also cater for nullable `Point` values.

Much more significant are the implementations of the `Equals` and `GetHashCode` methods provided by the `ValueType` base class. Those implementations must cater for *every* possible struct type, and must therefore be very general. Structs can contain any number of fields, and there is no restriction on the types of those fields. How, then, can the base class implementation work correctly in all cases?

ValueType implementations

For `GetHashCode`, the answer is straightforward. The hash code for a value is calculated from the first non-null field in the struct. If there are no non-null fields, the hash code is `0`. This has the correct behaviour in that any two *equal* values will always have the same hash code. It's not necessarily the most *efficient* implementation, because two values can differ in all their other fields, but will have the same hash code if just the first fields are equal. This might slow down lookups requiring hash codes when we have large numbers of values to be compared.

The `Equals` method needs to be a bit more sophisticated, because comparing only the first field will not be correct in all cases. To determine if two values are equal, *all* the fields must be compared. In order for this to work for any value type, the implementation of `ValueType.Equals` uses reflection to discover the fields, and compares the two values by calling `Equals` on each field. See [Tepliakov] for more information on how `Equals` and `GetHashCode` are implemented.

Reflection is a wonderfully powerful tool used in a variety of circumstances, but one thing it most certainly is *not* is fast. Fortunately, there are optimizations that remove both the need for reflection and the restriction of calculating hash codes from only the first field. In fact, our `Point` struct would most likely benefit from this optimization because it has two `int` fields.

```
public static bool operator==(Point? left,
  Point? right)
    => left?.Equals(right) ?? !right.HasValue;
public static bool operator!=(Point? left,
  Point? right)
    => !(left == right);
```

Listing 3

Where a struct has only built-in integral type fields, the `Equals` method can perform a simple bit-wise comparison of two values, and `GetHashCode` uses bit-masks and bit-shifting on the raw memory representation to calculate a hash code very quickly.

The optimization gets disabled in a wide variety of relatively common cases, however. If a struct contains any field that's a reference, a floating-point value, or itself provides an override for either the `Equals` or `GetHashCode` methods, the slower algorithm must be used.

For the incorrigibly curious, the reference implementation of `ValueType.Equals` can be found in [Equals]. The key optimization is the call to `CanCompareBits`, and for the gory details (in C++), see [DotNetCoreRuntime].

The bottom line here really is that we need to override both `Equals` and `GetHashCode` for struct types if we need to be *sure* about the performance of the implementation. These methods are generated for record types by the *compiler*. There is no base-class implementation that needs to cater for every possible combination of fields. The code is injected *directly* into a record, almost exactly as if we'd hand-written it ourselves.

All structs are implicitly *sealed*, which means implementing equality for a struct is relatively straightforward. Records *can* inherit from other records, and this makes implementing equality more complicated. To see exactly why that is, let's look at a naïve implementation for a derived class.

Equality and inheritance

Earlier we saw a class called `Point` that had an override of the `Equals` method taking an `object` parameter, and a type-specific overload of `Equals`. Here is the `Point` class again, along with a `Point3d` class that inherits from it (see Listing 4).

The implementation of the `IEquatable` interface in each of these classes, that is the `Equals` method taking a `Point` or `Point3d` rather than `object`, follows Microsoft's advice on correctly defining equality for a class as shown in [MSDN2015]. For brevity, they're not *exactly* the same, but they are equivalent to those shown online.

The key points here are that the derived class determines that the properties specific to it are equal, and if they are, it defers to the base class to perform its own comparison. The base class checks that both values being compared are *exactly* the same type before also comparing its individual properties. The type check is required to catch the following comparison:

```
var point = new Point(2, 3);
var point3d = new Point3d(2, 3, 4);
Assert.That(point.Equals(point3d), Is.False);
```

Here we're comparing a `Point` variable with an instance of `Point3d`. The `Equals` method actually being used here is the one defined on the `Point` base class. The `point3d` variable will be implicitly cast to a `Point`. The comparison fails the type check in `Point.Equals` because

Records can inherit from other records as long as the base record isn't sealed. The key is in how equality is implemented for records.

```
public class Point : IEquatable<Point>
{
  public Point(int x, int y)
    => (X, Y) = (x, y);
  public int X { get; }
  public int Y { get; }
  public bool Equals(Point? other)
    => !ReferenceEquals(other, null) &&
       GetType() == other.GetType() &&
       X == other.X && Y == other.Y;
  public override bool Equals(object? obj)
    => Equals(obj as Point);
  // ...
}
public class Point3d : Point, IEquatable<Point3d>
{
  public Point3d(int x, int y, int z)
    : base(x, y) => Z = z;
  public int Z { get; }
  public bool Equals(Point3d? other)
    => !ReferenceEquals(other, null) &&
       Z == other.Z && base.Equals(other);
  public override bool Equals(object? obj)
    => Equals(obj as Point3d);
  // ...
}
```

Listing 4

the run time types of the two objects being compared aren't exactly the same.

Even though the `X` and `Y` properties match in both objects, the two objects don't have the same value. A `Point3d` instance has an extra property named `Z` that will not be considered by the base class `Equals` method.

We wouldn't usually *directly* assign a derived type to a base class reference like this. It would more usually occur when we call a method taking parameters of the base class type.

Base class comparisons

Standing in for a real method taking parameters of `Point` type in this example is a simple method named `AreEqual` (Listing 5).

In this example, we create two `Point3d` instances that differ in their `Z` property. We confirm they do indeed compare *not* equal when we call the `Equals` method. On the last line we call the `AreEqual` method, which takes two parameters of the base class type.

This test *fails* because the call to `AreEqual` actually returns *True*. This time, both objects are exactly the same type, and neither one is `null`. More than that, their `X` and `Y` properties both match. However, the comparison of `Z` properties *never happens* when the objects are compared using their base class type.

```
bool AreEqual(Point left, Point right)
{
  return left.Equals(right);
}
var p1 = new Point3d(2, 3, 1);
var p2 = new Point3d(2, 3, 500);

Assert.That(p1.Equals(p2), Is.False);
Assert.That(AreEqual(p1, p2), Is.False);
```

Listing 5

If we change the `AreEqual` method to take `object` parameters instead of `Point`, the test will pass, because `object.Equals` is a virtual method call. However, in keeping with the advice given on the MSDN, the type-specific *overload* of `Equals` is *not* virtual. When we use a `Point` variable to call the `Equals` method, the `Point` implementation will be called, irrespective of whether the variable actually refers to a more derived type.

We can resolve this problem by making `Point.Equals` virtual, and adding an override for it to the `Point3d` class. There are some subtleties to doing this, however, and it's very easy to get wrong.

Records, as we noted earlier, can inherit from other records as long as the base record isn't sealed. Moreover, records behave *correctly* with inheritance and don't exhibit the problems demonstrated here. The key is in *how* equality is implemented for records.

Compiler-generated Equals

The code generated by the compiler to implement equality diverges from that recommended in [MSDN2015] – quite rightly, since that implementation isn't sufficient, as we've demonstrated. Let's begin with the base type `Point`. Again, for the sake of brevity, the code in Listing 6 isn't exactly the same as that created by the compiler, but its equivalent.

There are two things of note here. The first is the synthesized `EqualityContract` method. This is used in the `Equals` method to confirm that both the invoking object and the argument are exactly the same type. It replaces the call to `object.GetType` for this purpose.

The `GetType` method is available to any type, but it's a *non*-virtual method that involves a native system call. The `EqualityContract` method *is* virtual, but makes use of the `typeof` operator which is evaluated *at compile time*. The result of both `GetType` and `EqualityContract` under these circumstances is identical, but `EqualityContract` uses information available to the compiler, whereas `GetType` calculates the required `Type` to return at run time.

The second thing to note is that the type-specific implementation of the `Equals` method is itself virtual. The importance of this becomes apparent when we look at the equivalent code in the derived `Point3d` class.

Inheritance and virtual methods work well for entity types where we want to customize or embellish the behaviour of a base class

```
public class Point : IEquatable<Point>
{
  public Point(int x, int y)
    => (X, Y) = (x, y);
  public int X { get; }
  public int Y { get; }

  protected virtual Type EqualityContract
    => typeof(Point);

  public virtual bool Equals(Point? other)
    => !ReferenceEquals(other, null) &&
       EqualityContract == other.EqualityContract
       && X == other.X && Y == other.Y;
  public override bool Equals(object? obj)
    => Equals(obj as Point);
  // ...
}
```

Listing 6

```
public class Point3d : Point, IEquatable<Point3d>
{
  public Point3d(int x, int y, int z)
    : base(x, y) => Z = z;
  public int Z { get; }

  protected override Type EqualityContract
    => typeof(Point3d);

  public sealed override bool Equals(Point? other)
    => Equals((object?)other);
  public virtual bool Equals(Point3d? other)
    => base.Equals(other as Point) && Z == other.Z;
  public override bool Equals(object? obj)
    => Equals(obj as Point3d);
  // ...
}
```

Listing 7

Inheriting Equals

Listing 7 is the equivalent code for `Point3d` that derives from the `Point` type.

Not only does `Point3d` provide its own type-specific implementation for the `IEquatable` interface, it also *overrides* the base class's type-specific `Equals`. The override invokes the `Equals` method taking `object?` as its argument. This in turn resolves to the `Point3d.Equals(object?)` method, which attempts to cast its parameter to a `Point3d`.

We should also note that the type-specific implementation of `Equals` is *sealed* in the `Point3d` class. This means that if we were to inherit from `Point3d` – for the sake of the argument let's call it `Point4d` – that more derived type *cannot* override that method. Sealing a method has the effect preventing a derived type from further customising the implementation of it, but the method is still available for more derived types to call. Our potential `Point4d` type could still override the `Equals(Point3d?)` method, however.

Testing Equals for records

There are other minor differences between our `Point.Equals` implementation and that shown previously, but the main point is that if we were to model our `Point` and `Point3d` types as classes, there is quite a lot of boilerplate we need to provide in order for equality to work correctly.

Using records to model these types saves a great deal of code that would otherwise have to not only be written, but tested too. We previously saw a test for equality for our original class implementation of `Point` and `Point3d` that failed. Here it is once more:

```
bool AreEqual(Point left, Point right)
{
  return left.Equals(right);
}
var p1 = new Point3d(2, 3, 1);
var p2 = new Point3d(2, 3, 500);

Assert.That(p1.Equals(p2), Is.False);
Assert.That(AreEqual(p1, p2), Is.False);
```

Where `Point3d` is a record that inherits from a `Point` record, this test now passes. There is more than just equality to consider when we inherit from a value type, however.

Style over substitutability

Although the compiler generates code to correctly perform an equality comparison for records that inherit from one another, it *can't* generate code for any of the other operations we might need to implement. For example, if we wanted to implement the `IComparable` interface for our `Point` and `Point3d` types, we'd have to implement it ourselves.

Would it make sense for us to compare a `Point3d` instance to determine if it was *less than* an instance of a `Point`? What about the other way around? What compromises might we have to make?

Inheritance and virtual methods work well for entity types where we want to customize or embellish the behaviour of a base class. We also get the benefit of *substitutability* between the base type and derived type. An instance of a derived type can be used anywhere a base type reference is needed. This allows us to write code in terms of a base type that can be used seamlessly by objects that inherit from that base type.

Entities are the higher-order objects in our designs. They usually represent the persistent information about a system, and the processing of that information in collaboration with other entities. Identity is often important for entities, because we often need to use a specific instance. By contrast,

```
public readonly struct Point3d :
IEquatable<Point3d>
{
  public Point3d(int x, int y, int z)
    => (xy, this.z) = (new Point(x, y), z);
  public int X => xy.X;
  public int Y => xy.Y;
  public int Z => z;
  public bool Equals(Point3d other)
    => xy.Equals(other.xy) && z == other.z;
  public override bool Equals(object? obj)
    => obj is Point3d other && Equals(other);
  public override int GetHashCode()
    => HashCode.Combine(xy, z);
  public static bool operator==(Point3d? left,
     Point3d? right)
    => left?.Equals(right) ?? !right.HasValue;
  public static bool operator!=(Point3d? left,
     Point3d? right)
    => !(left == right);
  private readonly Point xy;
  private readonly int z;
}
```

Listing 8

values place *no* importance on identity. One value is as good as any other value with the same *state*.

The benefits of inheritance are much *less* clear for values, which is the reason that structs don't – indeed cannot – take part in inheritance relationships. It's also the reason that value-like classes such as `string` are sealed. Substitutability doesn't work so well for values; it's not fair to say that a `Point3d` is substitutable for a `Point` because they have different values, and the value is what really matters for a value type.

Has-A versus IS-A

Inheritance is commonly employed to *re-use* the characteristics of a type and build on it. When we derive a type from a non-abstract base, such as when inheriting `Point3d` from `Point`, we're really inheriting the implementation. Substitutability between types works best when the implementation *doesn't matter*. What we really want is to represent the same *interface*.

More formally the distinction is between *class* inheritance and *type* inheritance. By deriving a `Point3d` from a `Point` we're using class inheritance. In order to make it work correctly, we must alter the interface.

However, a much simpler solution would be to discard the inheritance altogether, and simply have `Point3d` *contain* an instance of a `Point`. We get all the benefits of re-using the *implementation* of `Point`, but have none of the difficulties of substitutability. Furthermore, we'd make both classes `sealed` and the implementations of both would be more straightforward. Perhaps even better, we make them structs instead of classes.

Consider the struct in Listing 8.

Here we have a `Point3d` type modelled as a struct that *contains* an instance of a `Point` as a field. We have no need to consider the case where a base class parameter might really be a `Point3d` because that's not possible. The only overridden methods are those necessary to provide the basic equality and hash code calculations from the `object` base class.

We can't use an instance of `Point3d` anywhere that a `Point` is needed. We might provide an explicit conversion – or *projection* – to a `Point` that could be used to invoke a method expecting `Point` variables. In all other respects, the behaviour of this struct matches all the expected behaviour from a `Point3d` that inherits from a `Point`.

The one possible objection to this is that structs are copied and passed *by value*, whereas records are copied and passed *by reference*. Since a `Point3d` contains an instance of another struct, we might expect its performance to suffer as a result of needing to copy the whole instance rather than just the reference.

```
const int N = 50000000;
const int Filter = 10000;

var source = Enumerable.Range(0, N)
  .Select(i => new Point3d(0, 0, i % Filter))
  .ToList();
var unique = source.ToHashSet();

Assert.That(unique.Count, Is.EqualTo(Filter));
Assert.That(unique.Contains(new Point3d
  (0, 0, Filter - 1)), Is.True);
```

Listing 9

As with all such questions, we must invoke the wisdom, or at least the objectivity, of a performance profiler.

Performance of structs and records

Our `Point3d` struct doesn't do much other than being a value. Similarly, the most important aspect of the record equivalent is its value. Therefore the most obvious thing to compare between the two is how equality is implemented. Just as important as the `Equals` implementation is the `GetHashCode` method. We should, then, measure the performance characteristics of both methods.

One simple way to do that is to employ a `HashSet`, which will use `GetHashCode` to determine where to look for a key, and then use `Equals` to determine an exact match. A hash set is a *unique* collection of keys, so a useful test would be to attempt to introduce duplicate keys so that we can be sure a full lookup of a value takes place.

The following simple test creates a list of `Point3d` objects, and we deliberately introduce duplicate values. We use the `source` list to populate a `HashSet` using the `ToHashSet` method, which simply discards any values that have already been added to the collection. (See Listing 9.)

The number of elements is intentionally *very* large in order to scale-up the relative cost of each method call to make the differences observable. All the following results were obtained by profiling a test using the `dotTrace` profiler from JetBrains (https://www.jetbrains.com/profiler/) using a straightforward wall-clock time report. In each case, the test was profiled using a Release build.

Profile results

Figure 1 contains the results from running this test using our `Point3d` record, which inherits from a `Point` record. the same test was profiled using our `Point3d` struct, which *contains* an instance of a `Point` struct. The results are also in Figure 1.

The headline time shows that the test using structs took not much more than half the time of the test using records. Note that the `ToHashSet` call is somewhat slower for records, but calls to `Equals` and `GetHashCode` are *much* slower than for structs. In fact, the cost of `Equals` for the struct type doesn't even register, which means the JIT compiler probably inlined the code.

The `Equals` method for records is relatively expensive owing to the number of virtual method calls it makes, in this case to the `EqualityContract` method.

The remainder of the time difference between the struct and record versions is most likely down to the fact that the struct instances are copied by value, but for the records, only the references are copied from the source to the hash set. The difference of ~200ms is negligible really, considering the huge number of elements we were using.

However, copying by value versus copying by reference has another, less obvious implication, which goes some way towards explaining the significant difference in the cost of the call to `ToList`.

The impact of the managed heap

Records are reference types, allocated on the heap, and are subject to garbage collection in the same way that class instances are. We deliberately introduced duplicate values in our source list, and when the `ToHashSet` method discards those duplicates they become *unreachable*, and so are eligible for garbage collection. Struct instances are *never* individually garbage collected, they simply go out of scope when they're no longer needed.

Adding such a large number of elements to the list would certainly put some pressure on memory, and very likely use up enough space to cause several garbage collections. We can see this by digging into the `ToList` call (see Figure 2).

The cost of the garbage collection here isn't objects actually being collected, it's most likely the cost of tracing references to each object to determine if they *can* be collected.

In fact, since we're putting so much pressure on memory here, it's likely that even the discarded objects stay in memory for much longer than necessary because they'll survive successive garbage collections caused by the huge number of memory requests being made.

All of which demonstrates that while copying objects by reference might be cheaper than copying by value, the associated cost of inhabiting the managed heap can offset that benefit and even overwhelm it.

Summary

The new record types in C# v9.0 provide us with a very compact way of defining value-like types without the need to manually write all the boilerplate code to perform equality correctly. The syntax we've explored in this article relates to *positional records*, which is the most compact representation that allows the compiler the greatest flexibility to generate code on our behalf.

We can choose to write our own version of almost any of the methods generated by the compiler if we wish. The exceptions to this are that we can't provide our own `operator==` or `operator!=`. If we want to customize the behaviour of equality, we need to write our own type-specific `Equals` method for the type. The compiler-generated `operator==` just forwards to the `Equals` method anyway.

Any method we write ourselves prevents the compiler from synthesizing its own version; it simply uses the version we provide.

However, since the compiler provides efficient and *correct* implementations for each of those methods, there seems to be little benefit in writing our own. If we feel the need to have more control over equality, we may as well just use a struct. Where we just need a simple representation of a value, records work very well and the associated facility of non-destructive mutation with the `with` keyword is a very useful way of handling those values.

Just because we *can* inherit one record from another, doesn't mean that we *should*. Values in general make poor parents, and so records, like structs and other value-like types such as `string`, should be sealed to prevent further derivation.

We also need to understand that records really are classes under the hood; when we create a record type, the compiler injects a *class* definition for us. Records are therefore reference types, and so live on the managed heap. This means they are garbage collected, and we might therefore consider using a struct anyway if we're very sensitive to performance.

An overview of records in C# v9.0, and more detail on what methods the compiler provides can be found at [MSDN2020]. ■

References

[Equals] https://referencesource.microsoft.com/#mscorlib/system/valuetype.cs,22

[DotNetCoreRuntime] https://github.com/dotnet/runtime/blob/01116d4e145d17adefc1237d55b1e3574919b1c1/src/coreclr/vm/comutilnative.cpp#L1738

[MSDN2015] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/how-to-define-value-equality-for-a-type

[MSDN2020] https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#record-types

[Tepliakov] https://devblogs.microsoft.com/premier-developer/performance-implications-of-default-struct-equality-in-c/

Records

```
► 5.52%   Hashset_of_records  •  5,622 ms  •  TestRecords.Hashset_of_records()
  ► 3.00%   ToList  •  3,055 ms  •  System.Linq.Enumerable.ToList(IEnumerable)
    2.51%   ToHashSet  •  2,554 ms  •  System.Linq.Enumerable.ToHashSet(IEnumerable)
    ► 0.47%   Equals  •  478 ms  •  Point3d.Equals(Point3d)
    ► 0.35%   GetHashCode  •  357 ms  •  Point3d.GetHashCode()
```

Structs

```
► 2.95%   Hashset_of_structs  •  3,002 ms  •  TestStructs.Hashset_of_structs()
    2.28%   ToHashSet  •  2,325 ms  •  System.Linq.Enumerable.ToHashSet(IEnumerable)
    ► 0.09%   GetHashCode  •  94 ms  •  Point.GetHashCode()
  ► 0.66%   ToList  •  677 ms  •  System.Linq.Enumerable.ToList(IEnumerable)
```

Figure 1

```
  3.00%   ToList  •  3,055 ms  •  System.Linq.Enumerable.ToList(IEnumerable)
    2.76%   <Hashset_of_records>b__13_0  •  2,809 ms  •  <Hashset_of_records>b__13_0(Int32)
      1.60%   [Garbage collection]  •  1,633 ms
      <0.01%   [Thread suspended]  •  5.8 ms
    ► <0.01%   Point3d..ctor  •  5.7 ms  •  Point3d..ctor(Int32, Int32, Int32)
```

Figure 2

Composition and Decomposition of Task Systems

Concurrency can be hard to get right. Lucian Radu Teodorescu demonstrates how tasks can help.

Probably the most important method that we apply when designing software is *decomposition*; this comes in the same package as *composition*. All the rest are irrelevant if we cannot decompose software. Even the highly acclaimed *abstraction* is insignificant compared to this pair.

Indeed, if we cannot decompose the system into multiple independent parts, then we could only apply the abstraction to the whole software. Thus, we would have the whole system and the abstracted whole system. There would be no other part that could benefit from using the abstraction instead of the whole system. So, the abstraction would be completely useless.

When one needs to solve a complex problem, one applies *decomposition* to break that problem into multiple parts that are easier to conceive, understand and develop. Thus, a software system is *decomposed* into multiple subsystems/components. I cannot think of any software system, even the most simple ones, that cannot be decomposed into simpler systems. That's just how our world is.

Ideally, the subsystems of a system can be developed in parallel, at least to a certain degree. Then, it's important that those subsystems can be put together, so that we can form the whole system.

Let's take a simple example, using functional decomposition and composition. Let's assume that for a given number n we want to compute $f(n)=n^2+1$. To solve this problem, we can decompose it in two smaller sub-problems: square a number, and add 1 to a number. That is, we reduce the initial problem to solving two other problems: computing $sqr(n)=n^2$ and $inc(n)=n+1$. After implementing these smaller functions, we need to combine them in the following way: $f(n)=inc(sqr(n))$, or, more commonly in mathematics: $f=inc \circ sqr$.

Decomposition and composition are the two faces of the same coin. One is useless without the other. For example, it's useless to decompose a software system into multiple components, if we can't compose the smaller components to form the larger system. They are so tied together that I cannot resist the urge to quote Heraclitus:

> The road up and the road down is one and the same.

This article aims to show that by using tasks, one can achieve good decomposability and composability, and thus tasks can be used as building blocks for concurrency.

Threads and locks are not composable

We all know that programming using raw threads and locks (in general, synchronisation primitives) is hard. We have understandability problems, we have thread safety problems, and most often we have performance problems. But one important inconvenience of this concurrent programming style is that raw threads and locks are not composable. [Lee06]

Lucian Radu Teodorescu has a PhD in programming languages and is a Software Architect 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

```
template <typename T>
class ValueHolder {
public:
  using Listener = std::function<void(T)>;
  void addListener(Listener listener) {
    listeners_.push_back(std::move(listener));
  }
  void setValue(T newVal) {
    curValue_ = std::move(newVal);
    for ( lstner : listeners_ )
      lstner(curValue_);
    }
private:
  std::vector<Listener> listeners_;
  T curValue_;
};
```

Listing 1

To ensure proper functioning of a subcomponent, one needs to understand the threading constraints of the adjacent subcomponents; that is, from what threads the subcomponent will be called, and what are the locks that are held while making these calls.

If one component holds a lock while calling the other, we need to be very sure that the other component will not call the first component back in a call that would require the same lock. And we need to make sure we are always taking all the locks in the same order, even if we don't have visibility of the other components. Also, components cannot assume that some APIs will be called from a known set of threads; a component will never know the threading used by adjacent components.

There are ways to protect against these types of safety problems, but they typically fall into two categories:

1. They are not general enough to be applied to all types of problems (i.e., they are ad hoc).
2. They typically degrade overall performance (they involve adding locks and restrictions).

To better illustrate this, let's revisit the simple example that Edward Lee used [Lee06]. Consider a single-threaded implementation of the observer pattern, as shown in Listing 1. If this is run inside a component that only calls `addListener` and `setValue` from a single thread, then this would be ok. But if another component tries to call this from a different thread, then we have a potential race condition; and, it's somehow normal to expect other components to maybe call this from different threads.

To fix this, one would add locks around the critical code, something similar to what is shown in Listing 2. This may work in certain cases, but not in others; depending on how listeners are implemented, this can lead to a deadlock. In general, it is bad practice to call unknown functions while holding a lock.

Another potential fix may be to extract the calling of the listeners from the lock, as shown in Listing 3. Now, even this version may have some

One can speak of decomposition when separating out complex processes into multiple threads. But, each time we do that, we encounter the two usual problems: safety and performance

```
template <typename T>
class ValueHolder {
public:
  using Listener = std::function<void(T)>;
  void addListener(Listener listener) {
   std::unique_lock<std::mutex> lock{bottleneck_};
   listeners_.push_back(std::move(listener));
  }
  void setValue(T newVal) {
   std::unique_lock<std::mutex> lock{bottleneck_};
   curValue_ = std::move(newVal);
   for ( lstner : listeners_ )
     lstner(curValue_);
  }
private:
  std::vector<Listener> listeners_;
  T curValue_;
  std::mutex bottleneck_;
    // Yes, this is a bottleneck
};
```

Listing 2

problems, but in general, most people would agree that this is a good solution.

If one can draw the conclusion that, in a threads and locks approach, to be able to interoperate with other components, one needs to encapsulate the threading behaviour, hardening the threading assumptions. In general, cross-API calls need to be made without holding any lock, and all the incoming calls need to be assumed that can be coming from any thread.

This is a simple example. The problems can only amplify for larger and more complex systems.

The composition for threads and locks systems is ad hoc, with effort needed to prevent safety issues, and, in general, with performance issues. It is hard to obtain good composability within systems based on threads and locks.

```
void setValue(T newVal) {
 std::vector<Listener> listenersCopy;
 {
   std::unique_lock<std::mutex> lock{bottleneck_};
   curValue_ = std::move(newVal);
   listenersCopy = listeners_;
 }
 for ( lstner : listenersCopy )
   lstner(curValue_);
}
```

Listing 3

Decomposition with threads and locks

Decomposition is seldom associated with threads and locks. This is mainly because there are no general rules for decomposition.

One can speak of decomposition when separating out complex processes into multiple threads. But, each time we do that, we encounter the two usual problems: safety and performance. To solve the safety problem, we typically need to add more locks. As we know, adding more locks typically downgrades performance.

There is also another performance problem caused by too many threads. One cannot simply create numerous threads if one has a limited number of cores. Assuming CPU-intensive work, the optimal performance is obtained when the number of threads is equal to the number of cores. Thus, decomposing in terms of threads seems like a bad idea.

Bottom line, decomposition with threads and locks is, at best, ad hoc.

Composability of task systems

Compared to the classical threads and locks approach, task-based systems compose better. The composability advantages of tasks systems can be summarised as follows:

- no extra protection needed at the interface level (i.e., adding a component to an existing component does not compromise its safety)
- no safety issue
- no loss in performance

We will analyse all these point one at a time, but before doing that let's introduce some formalism. For a given component c, let $T(c)$ be the set of all tasks that can be generated/executed in that component. Also, following the notation from [Teodorescu20b], we denote by $t_1 \rightarrow t_2$ the fact that task t_2 depends on task t_1 being executed (*dependency* relation), and by $t_1 \sim t_2$ the fact that tasks t_1 and t_2 cannot be executed in parallel (*restriction* relation).

For representation simplicity, we also assume that whenever there is a dependency relation $t_1 \rightarrow t_2$, there is also a restriction relation $t_1 \sim t_2$. After all, if one task depends on another task, the two tasks cannot run in parallel.

With these two relations, we can define the set of constraints for a component:

$$R(c) = \{(t_1 \rightarrow t_2) \mid \forall t_1, t_2 \in T(c), t_1 \rightarrow t_2\} \cup \{(t_1 \sim t_2) \mid \forall t_1, t_2 \in T(c), t_1 \sim t_2\}$$

In addition, we also denote by spanw(t) the action of spawning a task t, and by $S(c)$ the set of all the points in the component c where we spawn a task.

With these defined, then the three sets $T(c)$, $R(c)$, and $S(c)$ completely define the task system for the component c.

Now, if we want to compose two components and into a bigger system , then the following would apply:

$$T(c_+) = T(c_1) \cup T(c_2)$$
$$R(c_+) \supseteq R(c_1) \cup R(c_2)$$
$$S(c_+) \supseteq S(c_1) \cup S(c_2)$$

In plain English, the super-component contains all the tasks, relations and spawns of the two subcomponents, plus some more relations or spawn that might appear from the integration. The set of tasks is always the union of the two sets corresponding to the two sub-components; a task can be created either by one sub-component or by the other. But, when composing the two sub-components we may need to add relations between tasks coming from different sub-components. Similarly, we need to allow the code from one component to spawn a task from a different component.

There is an important assumption we are making: the set $R(c)$ for a component is maximal. That is, if two tasks cannot be run in parallel, there must be a restriction connecting these two tasks (either direct or by transitivity). We do not consider subsets of $R(c)$ which are not maximal, and two tasks are not executed in parallel for other, accidental, factors (i.e., some functions are never called into a particular order).

Lemma 1 (inner safety). When composing two or more task-safe components into a larger one, the internal safety of each component is not affected. i.e., one does need to add extra protection within any of these components to make the composed system safe.

Let's assume that there are two tasks t_1, t_2 within one component c_1, so that, when composing c_1 with c_2 into a super-component c_+ we get a safety issue. To get a safety issue in c_+ we need to be missing a dependency or restriction relation between the two tasks; formally $(c_1 \sim c_2) \notin R(c_+)$. But because $t_1, t_2 \in T(c_1)$, the missing dependency or restriction relation must also be part of $R(c_1)$.

However, based on the above assumption, the set $R(c_1)$ is maximal. Together with the fact c_1 that is tasksafe, this means that we cannot be missing any relation from $R(c_1)$.

We reached a contradiction. Therefore, we cannot have two tasks belonging to one component that, when joined with other components will generate a safety issue. This means that adding an extra component to an existing component we do not compromise the safety of the first component.

Q.E.D.

Lemma 2 (overall safety). Composing two components that are task-safe, will make the resulting supercomponent also task-safe, assuming that the tasks have only local effects.

Before proving this lemma, let us discuss what do we mean by "tasks have only local effects". The tasks in a component should affect only the resources owned by that component and should not have global effects. Of course that if two components have tasks that affect the global resources, then running two of such tasks in parallel, one from each component, might create a safety issue.

Let us prove this by contradiction. In order for the super-component to be unsafe, then there must be two tasks t_1 and t_2 that cannot be safely run in parallel, yet the composed system allows it. One alternative would be for the tasks to belong to the same component; but this cannot be true because of the previous lemma. Therefore, the only other alternative is for one task to be from a component and the other from the other component. But the tasks have only local effects. Thus, a task from one component cannot affect in any way the other component, so it cannot be unsafe to run it in parallel with another task from another component. This cannot be the case.

If neither of the two alternatives are plausible, it means that we cannot find two tasks that can run in parallel in the composed system, so the composed system is task-safe.

Q.E.D.

The reader should note that there are cases in which tasks have global effects. This doesn't necessarily mean that the safety of the super-system is compromised; it means that when composing the two components we need to add constraints (dependencies or restrictions) between tasks belonging to different components. This is precisely the reason why we said that the set of relations for the super-component can contain elements that are not in the any of the sets of relations for the sub-components.

Lemma 3 (inner performance). The task execution throughput of a component does not decrease when it's composed with other components, assuming there are no extra constraints added for the tasks belonging to that component, and that there are enough hardware resources.

Let $q_p(c)$ the maximum task execution throughput for component c, on a system with p cores: how many tasks are executed in a unit of time. This throughput is directly affected by the duration of the tasks and by the available parallelism. We assume infinite parallelism, and also, for simplicity let's not bother with the duration of tasks and assume they are all equal – we are concerned with the general dynamics, rather than specific timings that the tasks might have.

If there would be no restrictions, set ($R(c)=0$), then all the tasks can be executed in parallel, so $q_\infty = |T(c)|$. The only way to reduce this is to add constraints on the tasks. So, the throughput is a function of the restrictions set: $q_\infty(c) = q_\infty(N(c), R(c))$.

Adding other components without adding extra restrictions will keep $R(c)$ unchanged. Thus, all parameters being equal, the throughput remains the same. Q.E.D.

Please note that, in practice, the actual throughput is smaller than the maximal throughput. We never have all the tasks ready to start, and thus we only achieve a faction of this maximal throughput. The more tasks we spawn, the closer we get to the maximal throughput. Therefore, it may happen that, whenever other components are adding more spawns to the initial component, its throughput might actually grow.

On the other side, composing components with non-local tasks might require adding more constraints between tasks of different components. The more constraints, the less our throughput will be. A well-designed component, with very few tasks that have global effects will tend not to suffer from this problem.

Lemma 4 (overall performance). The total throughput of a system comprised of two components that have tasks with local effects will be the sum of the throughputs of the two components, assuming enough hardware resources.

The components have local tasks, so there will be no extra restrictions added to the overall system. Thus, taken independently the two components will not degrade their throughput. Now, considering that we have enough hardware resources, the tasks from two components can run completely in parallel. Therefore, the total throughput will be the sum of the individual throughputs. Q.E.D.

In practice, when the amount of parallelism is limited, the throughput will also be limited by the hardware constraints.

Again, the same discussions about tasks with global effects and about real-world throughput and maximal throughput apply here.

Also, as argued in [Teodorescu20a], one needs to be fully aware that in a real-world system there is always some indirect contention between the tasks, which may affect performance.

Theorem (composability of tasks systems). Components that have tasks with local effects can be composed in larger systems without any loss of safety or performance.

This follows directly from the above lemmas. Q.E.D.

Tasks systems will not have the same problems that the classical threads and locks approach would have.

Decomposing tasks into sub-tasks

So far, we've shown that tasks systems compose really well. If one has two components that are using tasks, they can easily be composed into a larger system while maintaining safety and performance. This is essential for a bottom-up approach. The questions that we are trying to answer in this section is whether is as easy to decompose task systems into smaller sub-systems; that is, how easy would it be to have a topdown approach to concurrency with tasks?

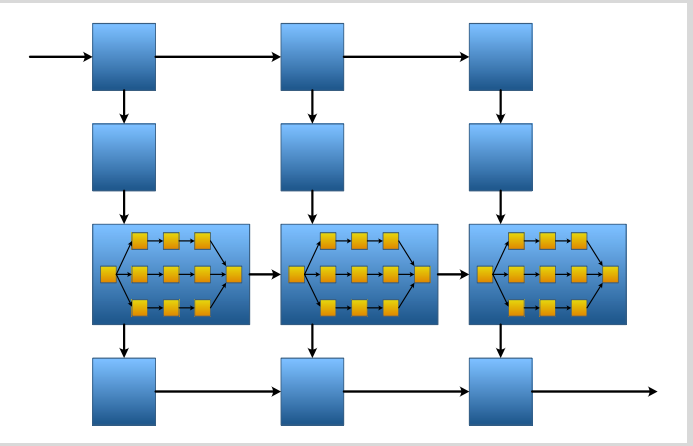


Figure 1

If we assume that the system is composed of generic tasks, then we can always have a partition of the tasks and relations, and have the decomposition around that partition. But if we have certain concurrent abstractions (e.g., a pipeline) that generate tasks then it might be harder to partition the tasks.

Also, we might need to take a large task and divide it into smaller tasks, so that they can be potentially run in parallel (under certain constraints). This may be relatively easy if the tasks are hand-made, but it may be more complicated if the tasks are generated by a concurrent abstraction (e.g., a pipeline).

Let us take a motivating example and try to fix all these cases. Let's assume that we have a high-level pipeline processing in our component, one stage of that pipeline can be subdivided into smaller tasks. This actually comes from a real-world problem that I tried to solve recently.

Figure 1 shows a diagram of the problem. We have a pipeline with 4 stages, and the third stage contains more processing than the others, and can be potentially decomposed into smaller tasks. The amount of parallelism generated by the pipeline is relatively small, especially because the third stage needs to be executed in order. Thus, we would take great performance advantage from breaking a task from the third stage into multiple smaller tasks.

The problem

The way that the pipeline abstraction is constructed, at each stage the body of the task is executed, and when that execution is finished, the pipeline checks to spawn the next tasks (next stage and the same stage of the next line). Given a task functor, the pipeline wraps it into another function that executes this task termination logic.

This composability of custom stage logic with pipeline logic is done inside a single task, in the single stack of execution and on a single thread. Thus, one cannot simply inject multithreaded execution on that thread / stack context.

A first attempt

One way of getting around this is by using the *fork-join* pattern [McCool12] [Robison14] [Teodorescu20c]. In Concore [concore], one can implement it similar to the code shown in Listing 4. One can spawn a lot of sub-tasks attached to a `task_group` and then wait for that `task_group` object, waiting on all these tasks to be complete.

From a high-level perspective, this gives us what we need: we can create sub-tasks inside a pipeline stage, and parallelise the stage more. The wait operation is going to be a busy wait, so, in terms of throughput we are ok, assuming that we have enough tasks to be executed.

But this solution can introduce high latencies, and may not work well when we don't have too many tasks to be executed, like in our case. The problem is that the `wait()` call will try to execute any task that it can, hoping that the tasks from the group will finish early. It provides no guarantee that the tasks spawn in the same context will be executed first. Thus, we can arrive

```
void third_stage(LineData& line) {
  auto grp = concore::task_group::create();
  // Create some sub-tasks
  concore::spawn([&line]{ f1(line); }, grp);
  concore::spawn([&line]{ f2(line); }, grp);
  concore::spawn([&line]{ f3(line); }, grp);
  // Ensure that all the tasks are executed
  // before continuing
  concore::wait(grp);
}
```

Listing 4

in a situation that we execute all the other tasks before we can actually execute the tasks that we need. If we don't have enough tasks in the system, we may actually drain the tasks from the system, leading to reduced throughput.

For example, in our pipeline problem, such a `wait()` call may execute all the stages/lines that can be executed before the stage is completed. This will essentially reduce the entire parallelism to the completion of one task.

This solution is relatively simple and can work in certain cases, but it will not solve all the needs for decomposition.

A better solution

Let us derive a general solution that will not have this performance problem.

First, going back to the theory from [Teodorescu20b], we argued that a task system implementation needs to have some special logic in two places to function properly. The first one is at the task creation, and the second one is at the completion of the task. We are going to focus on the latter one.

If one defines a task as a wrapper over `std::function<void()>`, then the logic that happens on task completion must be manually encoded in the body of the functor. For many structures that are built on top of these tasks (pipeline, task serializers, task graphs, etc.) each time the user pushes a task into it, another wrapper functor is created that contains the original task logic, plus the continuation logic.

If f_n is the function that needs to be executed at the stage n, then the pipeline actually constructs a task that executes the wrapper function $w_n(f_n, c_n)$, ensuring that after executing the user task the pipeline advances. Ignoring the error handling code, we can say that w_n is just executing c_n after f_n.

Traditionally, c_n is executed within the same stack, and on the same thread as f_n, but if we look more carefully, this doesn't need to be true. We can call c_n from a different thread, a different context; as long as we will call it, the pipeline can advance. This starts to resemble the continuation pattern [Teodorescu20c].

Let us formalise this, prove that it can actually solve our problem and prove that it can be applied generically to similar problems.

Instead of a task $t(f)$ that takes a user-supplied functor to be executed, we will introduce a new type of task of the form $w(f,c)$. The execution of an old-style task was defined as $execute(t(f))=f()$. For the new task, we define execution as $execute(w(f,c))=f();c()$ – that is, first execute the functor f then the functor c. All the other elements of the task system (spawning, constraints, etc.) remain the same.

Lemma 5 (equivalence). A task system using new tasks of the form $w(f,c)$ can be used to model all the problems that can be solved with old task types $t(w)$, with the same performance.

The proof follows directly from the realisation that we can always construct the new tasks with an empty continuation functor, such as:

$$w(f,0)=t(f),\forall f$$

Q.E.D.

After this equivalence lemma, we will use the new set of tasks in our higher-level concurrency abstractions. We will consider that all the user-supplied functors will be placed in the part untouched, while the code needed to make the abstraction work we will put in the part, without mixing

the user-supplied functors with the functors needed for realising the constraints of the abstraction. A concurrency abstraction that has this division, and that does not impose any restrictions on which threads the continuations are called will be called from now on a *continuation-based concurrency abstraction*. One can make abstractions like pipeline, the task serializers, task graphs, and a continuation abstraction [Teodorescu20b] as continuation-based concurrency abstractions.

Any continuation-based concurrency abstraction runs independently of the user-supplied functors (if they do not try to interact with the abstraction itself). Thus, regardless of the user-supplied functions, a pipeline will have the same execution pattern. Focusing on the continuations will allow us to assess the properties of the abstraction itself; in particular we are interested is seeing if various transformations will keep the abstraction running.

A key point in how we defined these continuation-based concurrency abstractions is that if the continuations are executed in the same order (possible from different threads), all the properties of the abstractions will remain the same. Thus, if one moves the execution of a continuation from a thread to another thread, the abstraction will still function.

Lemma 6 (task decomposition). A continuation-based concurrency abstraction that executes a task $w_0(f_0,c_0)$ can be transformed so that it executes an arbitrary graph of tasks $G=\{w_1, w_2, \ldots,w_n\}$(that ends its execution with a task w_n) instead of task w_0, without changing the functioning of the continuation-based concurrency abstraction.

The key here is to exchange the execution of w_0 with the execution of G, and ensure that c_0 is called at the end of the execution of G. If we can do that, then, based on what we observed above, the continuation-based abstraction will function the same.

For this, we will define $c_n' = c_n;c_0$ (call c_n then c_0) and (replace the continuation in w_n), and $G'=\{w_0, w_1, \ldots,w_n'\}$ (replace the last task from G). We also define $f_0' = execute(G')$ and $w_0' = (f_0',0)$.

Now, if we exchange w_0 with w_0', the continuation c_0 is still called when everything else finishes to execute. This means that we can safely exchange w_0 with w_0' in our continuation-based abstraction, without affecting the functioning of the abstraction. Q.E.D.

Theorem (task decomposition generality). For any concurrency abstraction that can be turned into a continuation-based concurrency abstraction, one can decompose tasks into smaller tasks (possible running in parallel), without affecting the overall properties of the abstraction.

This follows directly from the above lemmas. Q.E.D.

Listing 5 shows how this technique can be applied in concore[1], to avoid the blocking-wait from Listing 4.

```
void third_stage(LineData& line) {
  auto* cur_task = concore::task::current_task();
  auto cur_cont = cur_task->get_continuation();
  // Create the final task, with the current
  // continuation
  concore::task t_final{f_final, {}, cur_cont)};
  // Clear the continuation from the current tasks
  cur_task->set_continuation({});
  // Create some more tasks
  concore::spawn([&line]{ f1(line); });
  concore::spawn([&line]{ f2(line); });
  concore::spawn([&line]{ f3(line); });
  // we assume that after these tasks are executed,
  // t_final will be called
}
```

Listing 5

1. Continuation-based concurrency abstractions are just in infancy in concore. The syntax might change in the near future.

We have now a good way to take a top-down approach to our concurrency: use the most appropriate (continuation-based) concurrency abstraction, then, where more concurrency is needed, decompose a task into smaller tasks.

Conclusions

This article explores composition and decomposition of task-based systems. If classical threads and locks systems are hard to compose properly, with tasks this seems to be an easy task (pun intended).

With threads and locks, composition can easily lead to safety problems. To protect against these safety issues, one must always work at the interfaces between components and strengthen the protection. This is typically done by adding more locks, and this can downgrade the performance.

With task systems, we show that, for most cases, good composability is achieved without sacrificing safety and performance, and without any additional effort. The cases that need special attention are the cases in which tasks from one component have a global effect, and they are not safe to run in parallel with tasks from another component.

A good design would limit the number of tasks with global effects, so the amount of extra work, and the performance degradation would be limited.

Then we turn our attention to the decomposition of tasks. We show that, higher level concurrency abstractions based on continuations can easily be used to with task decomposition. We can break down a larger task into smaller ones, without affecting the functionality of the abstraction from which the task belonged to.

This is crucial as it would allow us to approach concurrency in a top-down manner. One first finds the right abstraction at the higher level, then decomposes different parts of that high-level abstraction into smaller one. This process can be repeated until we reached the desired level of concurrency.

Concurrency design is no longer dependent on the details of implementation, it doesn't need to be a bottom-up approach. One can design the concurrency of the application upfront, in a top-down, like any other software design activity.

Concurrency design doesn't need to be a hard thing. ■

References

[concore] Lucian Radu Teodorescu, Concore library, https://github.com/lucteo/concore

[Lee06] Edward A. Lee, The Problem with Threads, Technical Report, 2006, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf

[McCool12] Michael McCool, Arch D. Robison, James Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*, Morgan Kaufmann, 2012

[Robison14] Arch Robison, A Primer on Scheduling Fork-Join Parallelism with Work Stealing, Technical Report N3872, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3872.pdf

[Teodorescu20a] Lucian Radu Teodorescu, Refocusing Amdahl's Law, *Overload* 157, June 2020, available online at: https://accu.org/journals/overload/28/157/teodorescu_2795/

[Teodorescu20b] Lucian Radu Teodorescu, The Global Lockdown of Locks, *Overload* 158, August 2020, available online at: https://accu.org/journals/overload/28/158/teodorescu/

[Teodorescu20c] Lucian Radu Teodorescu, Concurrency Design Patterns, *Overload* 159, October 2020, available online at: https://accu.org/journals/overload/28/159/teodorescu/

Chepurni Multimethods for Contemporary C++

Multimethods can be implemented in various ways. Eugene Hutorny showscases an approach using custom type identification and introspection.

Multiple dispatch, or multimethods, is a feature of some programming languages in which a function or method can be dynamically dispatched based on the run-time (dynamic) type or, in the more general case, some other attribute of more than one of its arguments [Wikipedia-1]. The need for such a feature appears, for instance, in software architectures where numerous classes of objects interacts with each other in a way, specific for each pair. The C++ language did not directly provide such feature on the language level. It is possible to use `std::visit` in conjunction with `std::variant` to achieve multimethods (see [Filipek18], [Mertz18]). This article will consider an alternative approach, based on custom type identification and introspection facilities, letting us explore a variety of modern C++ techniques and providing flexibility

This study does not advocate a solution that would fit all use cases. Instead, it focuses on practical solutions for different use cases. The author encourages readers to experiment with the proposed solutions and tune or rework them for their particular needs.

Introduction

Existing solutions

Since a need for multiple dispatching is as old as the world of programming, quite a few solutions for C++ have been created (see [Bettini], [LeGoc14], [Loki], [Pirkelbauer07], [stfairy11], [Shopyrin06a], [Shopyrin06b], [Smith03a], [Smith03b]). However, none of them has a chepurni look (chepurni, Ukrainian чепурні – neat, clean, deft). Undoubtedly, chepurness is a subjective category. In the author's opinion, a chepurni solution is simple to implement, easy to use, modern, non-intrusive and well structured. In other words, the complexity of a chepurni solution is related to the problem's complexity, its use does not create extra dependencies, does not complicate the project maintenance, does not require changes to the existing designs, and every element of the solution addresses a single concern[Wikipedia-2] or bears a single responsibility [Wikipedia-3].

Problem overview

A dynamic, run-time dispatching requires knowledge about the class of the object. C++ facilitates Run-Time Type Information (RTTI), which is the number one choice for a project which already deploys it or is allowed to. However, the solution would not be chepurni if it did not offer an alternative for those projects where enabling RTTI would make them unviable.

Traditionally, multiple dispatching in C++ is implemented via a virtual method (the first dispatch) that performs next level dispatches with an `if` or `switch` statement, or with another virtual call to the other object. From the data modelling point of view, in many cases these methods do not look like natural parts of the classes implementing them, but rather as a workaround, caused by a missing language feature. They would look more organic if implemented as functions. However, in this study we will not impose a constraint to use only functions. Instead, we allow to use dispatchable methods along with functions.

```
namespace shapes {
  struct Shape {
    virtual ~Shape() {}
  };
  struct Rect : Shape {};
  struct Circle : Shape {};
  struct Square : Rect {};
}
namespace calculus {
  struct Expression {
    virtual ~Expression() {}
  };
  struct Constant : Expression {};
  struct Integer : Constant {};
  struct Float : Constant {};
}
```

Listing 1

Data models

In this study we will experiment with hierarchies of shapes and calculus expressions with simple inheritance (Listing 1).

Functions multidispatcher

Classes/templates, introduced in this section:

- `MULTIDISPATCHER`

  The main template, implementing a dispatcher over a list of functions
- `ENTRY`

  An auxiliary template for type identification and invocation
- `EXPECTED`

  An auxiliary template for argument type matching

Let's start with the calculus model and assume that it defines operations add, sub, implemented as template functions:

```
template<class A, class B>
Expression* add(const A&, const B&);
template<class A, class B>
Expression* sub(const A&, const B&);
```

Eugene Hutorny is a software engineer from Ukraine. He started his professional career in his last year in the Kyiv Polytechnic Institute back in 1994. Since then, Eugene has participated in many different projects using various technology stacks, keeping a passionate interest in C++ through the years and advocating its wider use in the software engineering in general, and particularly in embedded systems. Contact: eugene@hutorny.in.ua

the complexity of a chepurni solution is related to the problem's complexity: its use does not create extra dependencies, complicate project maintenance, or require changes to the existing design

```
template<auto Entry, auto ... Entries>
struct multidispatcher {
  template<class ... Arguments>
  static auto dispatch(Arguments& ... arguments) {
    if (entry<Entry>::matches(arguments...)) {
      return entry<Entry>::call(arguments...);
    }
    if constexpr (sizeof...(Entries)>0) {
      return multidispatcher<Entries...>
        ::dispatch(arguments...);
    }
  }
};
```

Listing 2

This assumption greatly simplifies the start, and we will make it more complicated as we advance.

As it was mentioned earlier, our multimethod should discover actual argument types, select a proper function from the list of available according to the discovered types, and invoke the selected function. Here we stated three concerns. Let's address them.

Function list

To define a list of function to dispatch we will use a variadic template with `auto` parameters, so it can be used as the following:

```
multidispatcher<
  add<Integer, Integer>,
  add<Float, Integer>,
  add<Integer, Float>,
  add<Float, Float>>::dispatch(a,b);
```

For this kind of usage, the template may look like Listing 2.

In the `dispatch` method we delegated type identification and invocation to another template entry, let's write it as in Listing 3.

Another template, `expected`, determines the actual argument's type and whether it matches the expected type. With this design decision, the complete implementation is in Listing 4, which is available for experiments on this link: godbolt.org/z/oz585K

This implementation is very simple, although, comparing to the open methods, it has certain constraints:

1. the length of the function list is limited by the compiler's recursion depth
2. the return type covariance is not supported
3. the parameter covariance is not supported
4. the first good candidate is selected instead of the best one
5. virtual parameters not distinguished form regular, all treated as virtual

The second constraint restricts the users from using a function that returns a covariant result, for example one like this `Float* sub(const Float&, const Float&)`. Also, this implementation does not do anything special about multiple inheritance. Such cases are handled the same way as simple inheritance – the first matching function is selected.

```
template<auto Method>
struct entry;
template<class Return, class ... Parameter,
    Return (*Function)(Parameter...)>
    struct entry<Function> {
  template<class ... Argument>
  static constexpr bool matches(const Argument&
      ... argument) noexcept {
    return (expected<Parameter>::matches(argument)
       and ...);
  }
  template<class ... Argument>
  static Return call(Argument& ... argument)
      noexcept(noexcept(Function)) {
    return (*Function)((Parameter)(argument)...);
  }
};
```

Listing 3

Third, fourth and fifth constraints limit the use cases. We will address them in the sections below. For now, we will focus on the first and second constraints.

Multimethods

Classes/templates/functions, introduced in this section

- `multimethod`

  The main template, implementing a dispatcher over a list of functions or methods

- `resolve`

  A helper function for resolving overloaded functions

- `class_hash()`

  A function for generating unique class ID

- `is_virtual_parameter`

  An auxiliary template for distinguishing between virtual and non-virtual parameters

Return type covariance

To support the return type covariance, we need to define the most generic return type for all used functions. To make it simple, we put this responsibility on the user and add this type as a parameter to our new template:

```
template<typename ReturnType, auto ... Entries>
struct multimethod;
```

As we are given the return type, we may use it for replacing the recursion with a folding expression (see Listing 5).

```
template<class Parameter>
struct expected {
  template<class Argument>
  static constexpr bool matches(const Argument&
      argument) noexcept {
    return typeid(Parameter) == typeid(argument);
  }
};
template<auto Method>
struct entry;
template<class Return, class ... Parameter,
    Return (*Function)(Parameter...)>
    struct entry<Function> {
  template<class ... Argument>
  static constexpr bool matches(const Argument& ...
      argument) noexcept {
    return (expected<Parameter>::matches(argument)
      and ...);
  }
  template<class ... Argument>
  static Return call(Argument& ... argument)
      noexcept(noexcept(Function)) {
    return (*Function)((Parameter)(argument)...);
  }
};
template<auto Entry, auto ... Entries>
struct multidispatcher {
  template<class ... Arguments>
  static auto dispatch(Arguments& ... arguments) {
    if (entry<Entry>::matches(arguments...)) {
      return entry<Entry>::call(arguments...);
    }
    if constexpr (sizeof...(Entries)>0) {
      return multidispatcher<Entries...>
        ::dispatch(arguments...);
    }
  }
};
```

Listing 4

```
template<typename ReturnType, auto ... Entries>
struct multimethod {
  template<class ... Arguments>
  static auto dispatch(Arguments& ... arguments) {
    ReturnType value;
    if (((entry<Entries>::matches(arguments...)
        and ((value = entry<Entries>::
        call(arguments...)), true)) or ...))
      return value;
    else
      throw std::logic_error
        ("Missing dispatch entry");
  }
};
```

Listing 5

This modified example is available on this link: godbolt.org/z/rj5vvY

We have to admit that this approach requires the return type to be default constructible and to support move semantics. For our examples it makes no difference. For the other use cases, this may need taking into account.

A curious reader perhaps noticed that the last example on godbolt.com is not using overloaded functions for **sub**:

```
Float* subf(const Float&, const Float&);
Integer* subi(const Integer&, const Integer&);
```

This is a workaround for a C++ feature for the overloaded functions. To get an address of an overloaded function, one has to specify its full type:

```
//Shape
virtual bool instance_of
  (const class_info& expected) const noexcept {
  return classinfo<decltype(*this)>() ==
    expected;
}
//Rect, Circle
bool instance_of(const class_info& expected)
  const noexcept override {
  return classinfo<decltype(*this)>() == expected
    or Shape::instance_of(expected);
}
```

Listing 6

(Float* (*)(const Float&, const Float&))&sub. This inconvenient syntax could be a bit sugared with a template resolve:

```
resolve<Float*, const Float&,
  const Float&>{}(&sub).
```

RTTI substitute

To work around the dependency on RTTI contributed with **typeid**, the model has to implement a custom type identification and introspection facilities (CTII). For instance, it could be a manually or automatically generated ID, assigned to every class. We may follow a simple autogenerating approach with a hash of **__PRETTY_FUNCTION__**:

```
template<class Class>
constexpr auto class_hash() noexcept {
  return
    hash(std::string_view(__PRETTY_FUNCTION__));
}
```

Unfortunately, **std::hash** is not yet **constexpr**, thus we have to write out own hash function, (for example, one like given in [Hutorny]). Now we can easily assign unique IDs to our classes:

```
static constexpr auto classid =
  class_hash<Rect>();
```

To access classid we define a template function classinfo:

```
using class_info = size_t;
template<class Class>
class_info classinfo() noexcept {
  return Class::classid;
}
```

For the dynamic type introspection, we define a virtual method (Listing 6). To avoid a dependency from the custom class **class_info**, we hide it behind a façade:

```
//Shape
template<class Expected>
bool instanceof() const noexcept {
  return instance_of(classinfo<Expected>());
}
```

Note, these methods are not overloaded. This will simplify our feature detecting template **has_instanceof**.

While we were using RTTI, we could ignore differences between virtual and regular parameters – **typeid** works for all types, and optimizer removes extraneous type checking for static types from the generated binary code. With CTII, however, we need to decide how to distinguish virtual parameters from non-virtual and how to compare the types for the latter. We may set the following rule: lvalue reference parameters to polymorphic types are virtual, the others are not:

```
template<class Class>
struct is_virtual_parameter {
  static constexpr bool value =
    std::is_polymorphic_v<std::remove_reference_t
      <Class>> and
    std::is_reference_v<Class>;
};
```

```
template<class Parameter>
struct expected {
  using parameter_type =
    std::remove_reference_t<std
    ::remove_cv_t<Parameter>>;
  template<class Argument>
  static constexpr bool matches(Argument&&
      argument) noexcept {
    if constexpr(has_instanceof<Argument>(0)) {
      return argument.template
        instanceof<parameter_type>();
    } else {
#if __cpp_rtti >= 199711
      return typeid(Parameter) == typeid(argument);
#else
      static_assert(
          not is_virtual_parameter_v<Parameter>,
          "No class info available");
      return is_assignable_parameter_v<Parameter,
          Argument>;
#endif
    }
  }
};
```

Listing 7

```
// multimethod
template<class Target, class ... Arguments>
static auto call(Target& target,Arguments& ...
arguments) {
  ReturnType value;
  if (((entry<Entries>::matches(target,
      arguments...)
    and ((value = entry<Entries>::call(target,
      arguments...)),true)) or ...))
    return value;
  else
    throw std::logic_error("Dispatcher failure");
}
```

Listing 8

For the type check of not-virtual parameters we may use, for instance, **is_same**, or **is_assignable**. With this design of CTII, the adjusted template **expected** may look like Listing 7.

Supporting the methods

So far, our **multimethod** template only supports functions. Let's extend it to accept methods as well. First what we need is to separate an object from the remaining arguments. We may, for example, define a new method in **multimethod** that accepts the objects as its first argument (Listing 8) and specialize template **entry** for methods (Listing 9).

Dealing with methods are somewhat more complicated than with functions – their signature may have specifier **const**. Thus, we will need two specializations of **entry**, and two methods **call**. With such implementation our **multimethod** accepts functions intermixed with methods. The sources for this design are available for experiments on this link: godbolt.org/z/ehEnnc

Parameter covariance

The CTII design with **instance_of** calling the base class also enabled parameter covariance. However, to get it working properly, the list should be ordered in a specific way: functions with more specific parameters should precede ones with more generic. It does not seem feasible to sort the list at compile time. Instead, one could add an order check which ensures that there are no functions below the current, accepting classes derived from the current ones. Computational complexity of such checking is estimated as $O(kn^2)$, where k is number of parameters, and n – number

```
template<class Target, class Return,
    class ... Parameter,
    Return (Target::*Method)(Parameter...)>
    struct entry<Method> {
  template<class Object, class ... Argument>
  static constexpr bool matches(Object& obj,
      Argument& ... argument) noexcept {
    return expected<Target>::matches(obj)
      and (expected<Parameter>::matches(argument)
        and ...);
  }
  template<class Object, class ... Argument>
  static Return call(Object& target,
      Argument& ... argument) {
    return ((Target&)(target).*Method)
      ((Parameter&)(argument)...);
  }
};
```

Listing 9

of functions in the list. It worth to note that this checking may significantly slowdown the compilation.

Multiple inheritance

Our CTII may also help with multiple inheritance – a class, inheriting multiple base classes should simply call **instance_of** of all its base classes. However, there might be cases, when the list ordering would not be sufficient for selecting the best match for certain combinations of parameter types.

Possible improvements

Maintaining a long list of functions may become too difficult for long function lists. To address this issue, one may group the list by the first parameter, like in the following example:

```
groupdispatcher<
  group<Expression*,
    add<Integer, Float>,
    add<Integer, Integer>>,
  group<Expression*,
    add<Float, Integer>,
    add<Float, Float>>>::dispatch(a,b);
```

Each group then can be maintained in a separate header file. Also, one can use virtual methods for the first dispatch and **multimethod** – for the next dispatches.

Performance

This implementation of **multimethod** sequentially examines the functions till it finds a suitable one. Experiments shows 700 instructions in average for a list of 40 functions (please refer to perf.cpp on the github or to godbolt.org/z/1caGTs). With this implementation, a test run on x64 completes 10,000,000 dispatches in 6.4 sec. A table dispatching, expectingly, would show a better performance.

Table dispatching

Classes/templates/functions, introduced in this section

- **matrixdispatcher**

  The main template, implementing a matrix dispatcher over a list of functions or methods

- **jumpvector**

  An auxiliary template defining a row in the matrix

- **jumpmatrix**

  An auxiliary template defining the dispatcher's matrix

- **compute_score()**

  A helper function, computing the score for a given pair of actual argument, formal parameter

- `make_score()`

  A helper function, computing the score for a given function/method
- `find_best_match()`

  A helper function, selecting the best scored function/method from the list
- `function_traits`

  An auxiliary template, revealing characteristics of a function
- `func`

  An internal template, generating wrapper functions

Assumptions

For a table dispatching, which is a matrix dispatch for two parameters, we need to fulfil some preconditions and solve some challenges:

1. An effective table dispatching requires sequential class IDs, preferably with no gaps.
   - A manual ID assignment is error prone and, for some projects, may be too difficult in maintaining.
2. A type-safe matrix may contain only functions of the same type, e.g. with identical signature.
   - A manual matrix filling is error prone, difficult even for small set of classes and practically impossible for large hierarchies.

For now, we just assume that sequential identification is made by the user, and address this concern in a chapter below. Also, as in previous designs, we assume that the functions are defined with the parameter types they actually operate on.

Matrix design

We have outlined the challenges, let's find a design to solve them. As in earlier chapters, we start with a hierarchy of N classes that has K dispatchable functions defined on it. These functions we may list in our variadic template:

```
matrixdispatcher<
  add<Integer, Integer>,
  add<Float, Integer>,
  add<Integer, Float>,
  add<Float, Float>>::dispatch(a,b);
```

As we assumed, class identification is made by the user:

```
static constexpr auto classid = 1;
virtual size_t classID() const
{ return classid; }
```

If we require the same signature for all K functions, our template would not be able to distinguish them and pick the best one. Also, requiring the same signature we would transfer responsibility of down casting on the user. This seems to be too intrusive. So, we instead assume that the functions have different signatures with type of parameters they actually operate. To close the gap between different signatures on input and identical in the matrix we need some intermediate functions. With help of templates, we may delegate generating needed quantity of functions to the compiler and fill the matrix with pointers to them. They all should have the same signature, with the most common parameters of all dispatchable functions. Deriving such signature does not seem feasible, so we let the user to specify it as an input parameter `FunctionType` of our template. Also, we are not able to determine the actual number of classes, that we may get in our `dispatch` method. So, we will require this information to be a part of the input as well.

With this design, our matrix may look like in this example:

```
FunctionType matrix[sizeA][sizeB];
```

However, filling this matrix in a constexpr manner is not handy, so we make it a bit more complex:

```
std::array<std::array<FunctionType, sizeA>,
  sizeB> matrix;
```

Now, let's find a way to fill it with function pointer. We need a `constexpr` filling to ensure that it will happen during compilation. C++

```
template<template<size_t> class Template,
  size_t N>
class jumpvector : public
  std::array<typename Template<0>::value_type,N>
{
public:
  using value_type =
    typename Template<0>::value_type;
  constexpr jumpvector() : jumpvector<Template,N>
    (std::make_index_sequence<N>()) {}
private:
  template<size_t ... I>
  constexpr jumpvector
    (std::index_sequence<I...>)
    : std::array<value_type,N> {
      &Template<I>::function ... } {}
};
```

Listing 10

```
template<template<size_t, size_t> class Template,
  size_t N, size_t M>
class jumpmatrix : public
  std::array<std::array<typename
  Template<0,0>::value_type, M>, N> {
public:
  using value_type = std::array
    <typename Template<0,0>::value_type, M>;
  constexpr jumpmatrix() : jumpmatrix<Template,
    N, M>(std::make_index_sequence<N>()) {}
private:
  template<size_t ... I>
  constexpr jumpmatrix(std::index_sequence<I...>)
    : std::array<value_type, N> {
        jumpvector<reduce<Template,I>
          ::template type,M>{} ... } {}
};
```

Listing 11

provides some means for filling `constexpr` arrays, in this design we use `index_sequence`. For instance, we may fill one row in our matrix with this line of code:

```
std::array<FunctionType,N> { &Template<I> ... };
```

Where `Template` is a template function with class ID as a parameter, and `I` is an index sequence. Elaborating this design to some degree of completeness we get Listing 10.

Please note, in the code above we had to shift from a template function `Template`, to a template class `Template` with a static method `function`. This is because a template function would require signature, defined at this time. While with a static method we may defer the signature definition to a late stage.

Applying this design to all rows, we may fill the entire matrix (Listing 11).

This implementation sets up the requirements for the template class `Template`:

```
template<size_t A, size_t B>
struct func {
  static result_type function(parama_type& a,
    paramb_type& b);
};
```

where `result_type`, `parama_type`, and `paramb_type` are parts, deduced from the `FunctionType` signature.

Scoring and selecting

For each specialization of our template `func`, we know exact class of each parameter – they are set by `A` and `B`. Thus, we can select the best candidate yet at compilation. To make this selection simple, we split it on two steps – scoring all functions with some criteria and selecting one with the highest

```
template<auto Entry>
static constexpr function_score make_score(size_t
index) noexcept {
  using entry = function_traits<decltype(Entry)>;
  using paramA = typename entry::template
nth_arg_type<0>;
  using paramB = typename entry::template
nth_arg_type<1>;
  return function_score {
    index, compute_score<paraьA,argA>() *
compute_score<paramB,argB>() };
}
```

Listing 12

```
template<typename Function>
struct function_traits;

template<class Return, class ... Parameter>
struct function_traits<Return (*)(Parameter...)> {
  using result_type = Return;
  static constexpr size_t parameter_count =
    sizeof...(Parameter);
  template<size_t N>
  using nth_arg_type = std::tuple_element_t<N,
    std::tuple<Parameter...>>;
};
```

Listing 13

score. The criteria should operate on actual and expected types. In C++17 we have standard templates **is_same** and **is_base_of**. For example, a class scoring template may be implemented as this:

```
template<class Parameter, class Argument>
constexpr ssize_t compute_score() noexcept {
  if( std::is_same_v< Argument, Parameter> )
    return 2;
  if( std::is_base_of_v<Parameter, Argument> )
    return 1;
  return 0;
}
```

The function score then can be computed as a product of its parameter scores. This would work for simple hierarchies without multiple inheritance. More complex hierarchies may require a more advanced scoring design, based on a custom genesis inspection.

Now, we fill an array with the scores for every function from the list (Listing 12).

And select an element with the highest score:

```
template<class ClassA, class ClassB,
  auto ... Entries>
constexpr ssize_t find_best_match() noexcept {
  constexpr auto sc = function_scores<ClassA,
    ClassB, Entries...>{};
  return sc.highest();
}
```

In a snippet of code above, **function_traits** is an auxiliary template for determining the function's characteristics (see Listing 13).

Once we have a **constexpr** function index, we can get a function pointer from the list of input functions. To pass parameters to that function, we need a **down cast**, but since we already proved the proper relationship between the types, we may safely use **static cast**. Thereby, our function template becomes as in Listing 14.

Here we used another template **type<B>**, returning a type by its ID, e.g., implementing the reverse class-ID mapping. Since we have custom class IDs, this mapping has to be custom as well. The mapping and supply of the maximal class ID are related responsibilities, so we may delegate them

```
template<size_t A, size_t B>
struct func {
  static result_type function(parama_type& a,
      paramb_type& b) {
    constexpr auto best = find_best_match<type<A>,
      type<B>, Entries...>();
    if constexpr(best >= 0) {
      constexpr auto f = get_entry<best,
        Entries...>();
      return (*f)( static_cast<type<A>>(a),
        static_cast<type<B>>(b));
    } else {
      LOGIC_ERROR("Dispatcher failure");
    }
  }
};
```

Listing 14

```
template<typename FunctionType, class DomainA,
class DomainB, auto ... Entries>
class matrixdispatch {
public:
  using entry = function_traits<FunctionType>;
  using result_type = typename entry::result_type;
  using parama_type =
    typename entry::template nth_arg_type<0>;
  using paramb_type =
    typename entry::template nth_arg_type<1>;
  constexpr matrixdispatch() noexcept {}
private:
  template<size_t A, size_t B>
  struct func {
    static result_type function(parama_type& a,
      paramb_type& b) {
      //See code snippet above
    }
  };
  static constexpr jumpmatrix<func, DomainA::size,
    DomainB::size> matrix {};
public:
  static result_type dispatch(parama_type arg1,
      paramb_type arg2) {
    return matrix[arg1.classID()][arg2.classID()]
      (arg1, arg2);
  }
};
```

Listing 15

to a single concept, further named domain. We have two input parameters and they may belong to different domains, thus we need two custom domains, which we will get as the parameters of our **matrixdispatch**. When both parameters share the same domain, the same domain class will be used in place of both domain parameters. All these design decisions can be implemented with the code in Listing 15.

With some more work, we may improve this template to support methods, functions and combinations of them. Live example for this template is available on the following link: godbolt.org/z/Y89ThK

Performance

The matrix dispatch shows much better performance – 40 instructions (vs 700 in linear) for a list of 40 functions. However, it is also much more resource consuming for the compiler: $O(pnk^2)$ where p is a number of parameters, n is the number of functions, and k is the number of classes. Compilation of an example with 25 classes and 40 functions takes 20 sec more, when the **matrixdispatch** template is actually used. Also, the

```
template<class ... Classes>
struct domain {
  static constexpr auto size = sizeof...(Classes);
  template<size_t ID>
  using type = std::tuple_element_t<ID,
    std::tuple<Classes...>>;
  template<class Class>
  static constexpr size_t id_of() noexcept {
    constexpr auto value = index_of<Class,
      Classes...>();
    return value;
  }
};
```

Listing 16

15-times decrease of the instruction count per dispatch does not lead to the same decrease of the dispatch time. An experiment on x64 for 10,000,000 dispatches complete in 1.2 sec vs 6.4 sec for the linear dispatch.

Automatic identifier assignment

To make the class identifier assignment simple, we may craft a domain template (see Listing 16).

This template accepts a list of classes, and implements forward (`id_of<MyClass>()`) and reverse (`type<ID>`) mapping. This template does not require the listed classes to be completely defined, just forward declarations of them is sufficient.

To reduce the compilation complexity, we may exclude abstract classes from the matrix – instances of such classes cannot be created, and thus, they will never participate in the dispatch. However, to make this exclusion efficient, we need to assign abstract classes IDs from a lower diapason [0..M]. To address this challenge in a simple way we may assume that all abstract classes listed prior the concrete classes and provide a validation facility to check, whether this assumption is true.

Performance comparison

The results of performance tests are summarized in the table below. In this table, test `std::visit` denotes a reference implementation with `std::visit`, dispatching over variant-of-object arguments, `std::visit*` dispatching over variant-of-pointers, obtained via virtual calls to the objects. Wall time column lists timing for 10,000,000 dispatches over 40 functions/methods. ■

	Instructions per call		Wall time (sec)	
	G++-10	CLANG++-11	G++-10	CLANG++-11
std::visit	38	25	0.64	1.0
std::visit*	58	50	0.93	1.3
multimethod	750	702	5.8	6.4
matrixdispatch	54	47	0.88	1.2

Final notes

1. gcc compiler for some examples, published on godbolt.org, generates code with static dispatching instead of dynamic. To make it truly dynamic, the test functions should be compiled in different compilatons units, as in examples on github.
2. Examples from this study are also available on gist.github.com/hutorny:

   calculus multidispatcher

   calculus multifunction

   shapes multimethod

   matrix dispatch
3. Ready to use templates are available as include-only library: github.com/hutorny/multimethods
4. The sources of performance test are available in the same repository.

References

[Bettini] L. Bettini 'Doublecpp – double dispatch in C++' – http://doublecpp.sourceforge.net/

[Filipek18] B. Filipek, 'How To Use std visit With Multiple Variants' – https://www.bfilipek.com/2018/09/visit-variants.html

[Hutorny] E. Hutorny, 'FNV-1b hash function with extra rotation' – https://gist.github.com/hutorny/249c2c67255f842fae08e542f00131b5

[LeGoc14] Y. Le Goc and A. Donzé (2014) 'EVL: A framework for multi-methods in C++' – https://www.sciencedirect.com/science/article/pii/S0167642314003360

[Loki] Loki Library, Multiple dispatcher – https://sourceforge.net/projects/loki-lib/ Reference/MutiMethod.h

[Mertz18] A. Mertz, 'Modern C++ Features – std::variant and std::visit' – https://arne-mertz.de/2018/05/modern-c-features-stdvariant-and-stdvisit/

[Pirkelbauer07] P. Pirkelbauer, Y. Solodkyy, B. Stroustrup (2007) 'Open Multi-Methods for C++' – https://www.stroustrup.com/multimethods.pdf

[Shopyrin06a] D. Shopyrin, 'MultiMethods in C++: Finding a complete solution' – https://www.codeproject.com/Articles/7360/MultiMethods-in-C-Finding-a-complete-solution

[Shopyrin06b] D. Shopyrin, 'Multimethods in C++ Using Recursive Deferred Dispatching' – https://www.computer.org/csdl/magazine/so/2006/03/s3062/13rRUxBa5ve

[Smith03a] J. Smith 'Draft proposal for adding Multimethods to C++' – http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1529.html

[Smith03b] J. Smith 'Multimethods' – http://www.op59.net/accu-2003-multimethods.html

[stfairy11] stfairy 'Multiple Dispatch and Double Dispatch' – https://www.codeproject.com/Articles/242749/Multiple-Dispatch-and-Double-Dispatch

[Wikipedia-1] 'Multiple dispatch' – https://en.wikipedia.org/wiki/Multiple_dispatch

[Wikipedia-2] Separation of Concerns – https://en.wikipedia.org/wiki/Separation_of_concerns

[Wikipedia-3] Single-responsibility principle – https://en.wikipedia.org/wiki/Single-responsibility_principle

<script>

"Now I will believe that there are unicorns." Teedy Deigh loses the plot a little.

Verona is in lockdown, but the business and troubles of software development continue. Romeo wanders the streets alone, pondering a production problem he hasn't been able to figure out. Before the plague hit, he was struck by feelings for his co-worker, Julia. He wonders, does she feel the same? He finds himself on the street where she lives.

Julia appears on the balcony of her apartment, laptop in hand, unaware of Romeo on the street below.

Romeo: But soft! What loop through yonder window handler breaks?
It is the east *const* and Julia is the fractal set.
See how she codes and uses lean techniques to improve her flow.
O, that I could pair program with her.

Julia: Ay me!

Romeo: She speaks! O, speak again, bright engineer!
For your code craft being o'er my head
Is wisdom and inspiration to my soul.

Julia: O Romeo, Romeo! Wherefore art thou Romeo?
Denial of service is the cause of our ills.
Refuse the connection after so many retries.
Once again, I must fix the oversight in your code.

Romeo: Shall I hear more, or shall I speak at this?

Julia: 'Tis thy naming of variables that is my enemy;
I come to fix your code, not to bury it,
You obsess over your *x* and I know not *y*.
Brevity is the soul of wit,
But here is an obstacle to understanding.
Lo, in this block you name a variable *b* and another *b2*.
b2 or not *b2* is not the question,
Whether 'tis nobler in the code to state meaning clearly,
Or to use comments against a C of troubles.
Were I to comment, I would say much,
But 'tis better to rename.
I shall reveal intent with words filled with sense.
I shall extract functions and write new tests and then,
Before the night is out,
I shall put this bug to bed and check it into the repo. Man,
The debt of your code weighs heavy against my schedule.
But hark, what did Romeo name the retry counter?

Romeo: Julia, it is *i*!

Julia: What man art thou that thus bescreen'd in night
So stumblest on my code rant?

Romeo: Julia, you ask "Wherefore art thou Romeo?"
I am here!

Julia: O Romeo, Romeo, let *dictionary.com* be thy companion.
Wherefore means not *where* but *why* or f*or what*.
And I ask this of you and your code
As it keeps me from sleep each night.

Romeo: I would suggest other ways that could happen,
But would I be too bold?

Julia: You follow me on Twitter and Instagram.
Is that not enough? Must you also do that IRL?
Your tags are unmatched and braces askew.
Your data structuring leaves much to desire,
And there is not desire left in me for its author.
That which you call rows
By another name would be a code smell.
Besides, we are in lockdown.
Our *std::distance* must be social
And my apartment model has but a single thread.

Romeo: It is true, there is a plague on both our houses.

Julia: And the rest of the world.
It's but a stage, and we are merely players,
But our parts we must play until we are rid of this pandemic.

Romeo: O that this virus would not spread so!
That I could firewall against it.
That we could work together
And I could learn from thy counsel.

Julia: Alas, poor Romeo! I know you, and your ratio:
The quantity of learning to quantity of feedback is poor.
Alack, there lies more peril in thy code
Than Dunning–Kruger could e'er have known.

Romeo: Thy words are harsh but true.
The fool doth think he is wise,
But the wise man knows himself to be a fool.
My code ambition is such stuff as dreams are made on.
I would cast all this away to know what you know,
To code as you do.

Julia: Cast it all away?
Like that time you compared *this* to a summer's day?
That was a monstrosity.
The loops were infinite, and the execution confined.
The allocations were boundless, and the act a slave to limit.
It took the team ages to find that bug.

Romeo: O Julia, what would you have me do?
When shall we tweet again?

Julia: When the hurlyburly's done.
When this work item's fixed and done.
Give the code leave awhile.
'Tis one thing to be tempted,
Another thing to fall.

Romeo: Thou desirest me to stop?

Julia: Spend time on thyself, good Romeo.
Improve thy skills and sharpen thy knowledge.
Watch videos from *$SponsorName*.
They have many a course on code and its practice.

Romeo: You are fair and wise.
I thank thee and *$SponsorName* for your help.

Julia: Laters.

Romeo: Laters.

The rest is silence.

Teedy Deigh has heard that a varied diet is important to lockdown living. She is currently living off a mixed diet of home-baked sourdough bread, delivered takeaways, coffee, homemade cocktails and screen radiation. In lieu of travel, she has been experimenting with varying her time zones, sometimes on a daily basis. At least, that is how she has been reframing her unstable sleep patterns.

JOIN THE ACCU!

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

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

With full ACCU membership you get:

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

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

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

How to join

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

Also available

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

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

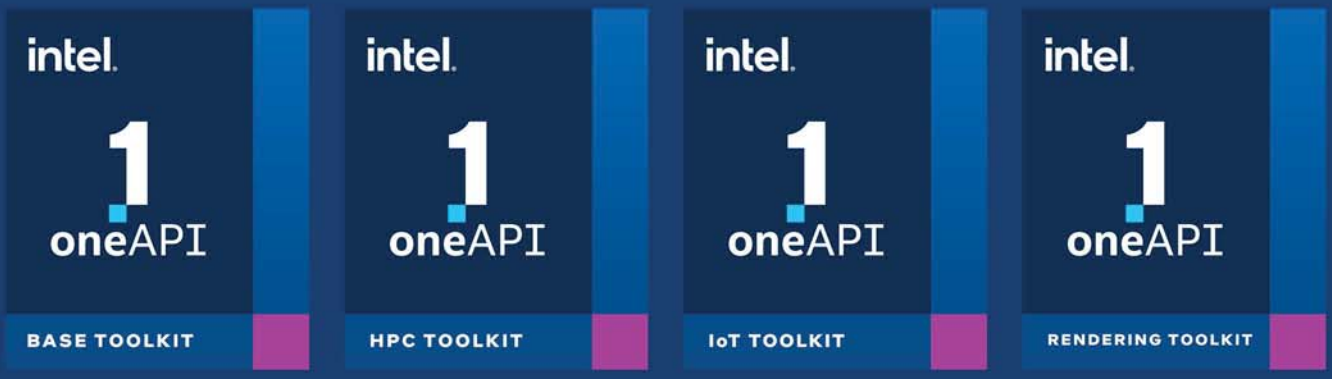
PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

intel software

oneAPI: New Era of Accelerated Computing

oneAPI

Take the open, productive path to accelerate cross-architecture computing using Intel® oneAPI Toolkits.



Developers, take advantage of oneAPI's unified, standards-based, cross-architecture programming model that sets you free to develop applications for your choice of architectures. Get full hardware performance using a complete set of proven tools without the limits of proprietary language lock-in.

Learn more: www.qbssoftware.com