# overload 147

OCTOBER 2018 £3

## Implementing the Spaceship Operator for Optional

operator<=> will appear in C++20. We see a practical example of its implementation, using std::optional

## P1063 vs Coroutines TS: Consensus on High-Level Semantics

How coroutines will improve the C++ language

## How to Write a Programming Language

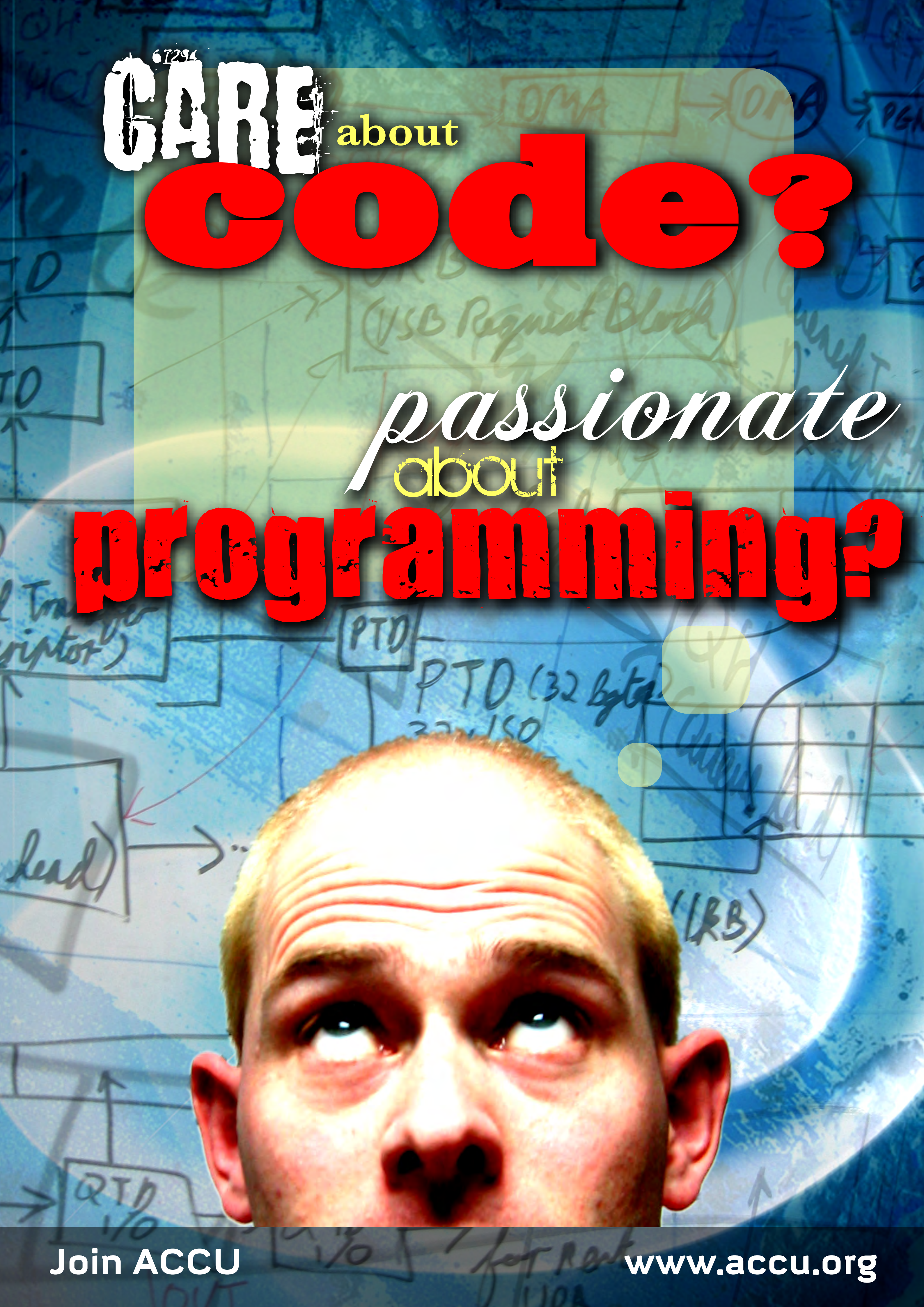
We continue writing a simple programming language, this time working on the evaluator

## Compile-time Data Structures in C++17 Part 2, Map of Types

Employing C++ templates to craft efficient code

A magazine of ACCU

ISSN: 1354-3172


CARE about
code?
passionate about
programming?
Join ACCU
www.accu.org

**OVERLOAD 147**

**October 2018**

ISSN 1354-3172

**Editor**

Frances Buontempo
overload@accu.org

**Advisors**

Andy Balaam
andybalaam@artificialworlds.net

Balog Pal
pasa@lib.h

Ben Curry
b.d.curry@gmail.com

Paul Johnson
paulf.johnson@gmail.com

Klitos Kyriacou
klitos.kyriacou@gmail.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Philipp Schwaha
<philipp@schwaha.net>

Anthony Williams
anthony@justsoftwaresolutions.co.uk

**Advertising enquiries**

ads@accu.org

**Printing and distribution**

Parchment (Oxford) Ltd

**Cover art and design**

Pete Goodliffe
pete@goodliffe.net

**Copy deadlines**

All articles intended for publication in Overload 148 should be submitted by 1st November 2018 and those for Overload 149 by 1st January 2019.

## Overload is a publication of the ACCU

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

## The ACCU

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

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

# Are we nearly there yet?

Deciding if you are making progress can be a challenge. Frances Buontempo considers various metrics and their effects.

Summer is a time for holidays including road trips for many, possibly with children screaming, “Are we nearly there yet?” from the back of the car. We didn’t get a chance to have a holiday but did go to a metal festival, so I’ve been too busy discovering new music to write an editorial. In fact, the deadline for this issue of *Overload* sneaked up on me, so I haven’t even tried thinking about it. Trying to keep a sense of how close targets or deadlines are is difficult. Even how long it might take to get to a location is not straight forward. While being physically close to your destination, the estimated time to arrival might be surprisingly large if there is a huge traffic queue. So near, and yet so far away.

‘Nearly’ is relative, of course. Topology, a branch of mathematics, defines relationships in a very abstract way. Lisa Lippincott gave a keynote at the ACCU conference this year, called ‘The shape of a program’ [Lippincott18]. She pulled on ideas from topology to talk about places and meaningful areas, particularly in a codebase. Now, topology provides a formal definition of points in a neighbourhood. Such points are close, in some sense, but topology doesn’t have a metric or distance measure, so ‘close’ is very abstract here. The concept of belonging to a neighbourhood depends on definition of an open set, which in turn has a complementary closed set, which has nothing to do with being close/nearby. Furthermore, sets can be both open and closed [Wikipedia-a]. This sounds counter-intuitive. Some sets are neither open nor closed. How can this possibly be useful? There is more to life than usefulness. I find topology mesmerising, but it is useful. It gives an abstract way to think about closeness and convergence. We could think more about convergence, but will have to save that for another time. For now, think tending towards somewhere. Where, exactly?

Defining ‘there’ is equally complicated. Some cultures or groups have an initiation ceremony to mark a transition to adulthood, or full membership status. Some professions require certified status. An event marks a boundary between before and after. A novice takes a vow and becomes an initiate. A trainee engineer passes exams and gains experience, allowing certification. Not everything has a clear definition or demarcation between in and out, open and closed, before and after. Even success or fail. If you are developing a product, how can you be sure what your customer needs? If you are learning a new subject, how do you know when you really understand it? What about learning a new programming language?

How do you prepare a talk, or proposal for a talk at a conference? How do you write an editorial for that matter? You need a topic or title. These might get chosen up front, or come into focus as you knock a few ideas around. Something similar happens when you try to learn a new programming paradigm or language. You need a topic, toy example or something to build. Once you decide, you start somewhere, possibly in the middle. Trying to measure your progress needs an idea of what ‘done’ means. For learning tasks or anything with a deadline, done often means time’s up. You may not have all the features you were dreaming of, but you have learnt or written something. A time limit provides a clear definition of ‘there’. Other tasks, such as tidying your bedroom, redecorating or documenting a system do not immediately give a way to define ‘done’. Again, a time limit may enforce this, or running out of money. Some tasks like this stop when you run out of stamina. If you go swimming in a pool, you may plan to swim for an hour, but discover you can only manage five lengths because you are out of practice. You may not achieve your target, but at least you turned up. Sometimes just getting started is good enough. Other tasks, like building a Lego rocket or a robot have a clearly defined endpoint. All the pieces are in the right place. In the case of the robot hiding behind my sofa, the failure was due to missing parts. Not something you want to discover when you’re right near the end of a twelve-hour build. That’s the reason my Dad used to lay out and count pieces before self-assembling furniture, or making jigsaws or models. He did try to fix things like clocks from time to time. The universal rule there appears to be that a spare screw or two is bound to materialise and can be safely stored in a tin, without affecting the performance of the machine. Something similar seems to happen when you refactor code, but you don’t keep the spare lines in a tin. Just delete them and jog on.

Other tasks are longer running, and do need some kind of milestones to check the project is on target, or at least heading in the right direction. Some projects follow more traditional methods, forming detailed project plans in the form of Gantt charts [Wikipedia-b], think rows of tasks and columns of dates, showing who will do what, when. Other times, Kanban boards [Wikipedia-c] are used, usually starting with a to-do, doing, done column. In both cases, as time marches forward, things change. Relabelling the weeks or months or even years on a Gantt chart might help to provide new estimates of what’s left to be done before the final project is complete. The Kanban board might sprout a few extra columns. A project I worked on ended up with 17 columns. It was using a web interface, rather than being post-it notes on a wall, and I had to scroll right for quite a long way to get to the final column. And don’t get me started on scroll bars that move as you try to scroll, having cached the length of data then updating to discover more.

When tasks don’t form a neat linear progression, they often make something like a tree structure, which isn’t easy to shape into a list or table of rows and columns. I have never been a project manager so can’t claim expertise in this area, but I do usually try to track if a project is moving forward and keep a to-do list somewhere. I do this for personal projects too, which does mean I have several lists scattered around the house. I

**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.

need a list of lists to help me find the list, which suggests a tree structure yet again. As more items get added to the list, it becomes hard to estimate how long it will take to finish a project. If you add things at a faster rate than you complete things, you might start to feel as though you are going backwards. This can be dis-heartening. Frequently, what started as a straightforward collection of a few things to achieve over the next couple of days might end up as a few months' worth of research, experimentation and learning. The original estimate failed to take on board all the details that are required. We frequently under-estimate the effort required. George W Bush once claimed people had misunderestimated him [BBC09]. This suggests he thought you can make correct and incorrect underestimates. Logically ridiculous, however, many projects do leave a time slot towards the end for validation, deployment or testing, which are really code words for unknown unknowns [Wikipedia-d]. The misunderestimation comes from not allowing enough time to complete everything required, or possibly being too ambitious.

Project planning usually involves aiming towards a final goal, such as deploying some working software. Learning, whether via an academic institution, or attempting to teach yourself, doesn't always have a clear final goal. A timetable for a term, series of lectures or a book with exercises might give the impression of a linear progression from zero to master. A final exam sets a deadline and time boxes the learning into a term or period of time. Some employers may take people on as apprentices, perhaps offering a twelve month training programme. How does the neophyte or learner asses their progress? I was discussing this with Chris Oldwood recently. He has a mentee who asked how to tell how much she had improved. Chris suggested she noted how many of his jokes she laughed at. If you've not had to chance to listen to Chris' stand-up comedy routine, you are missing out, though traces of it are out there, somewhere [Oldwood15]. I suspect in a few cases, a groan rather than a laugh would disambiguate a yellow belt from a Dan master, but I'm no expert. This metric, which I shall refer to henceforth as 'the Oldwood quota', gives a straightforward way to measure something. It's not immediately clear what. If Chris tells ten jokes and his mentee laughs at all of them, she gets ten out of ten. Does this mean she's wiser? As children learn about jokes, they laugh, but you gain more insight when they try to re-tell the jokes. The punchlines sometimes get told first. The words come out wrong. Eventually the child starts to invent their own jokes, usually with varying success. I suspect Chris will know his mentee has surpassed him when she can make him laugh rather than vice versa. The conversation with Chris was very interesting. I love his jokes and am sure he's a great mentor. As we talked, it struck me that general knowledge quizzes are a thing. You can count how many answers people get correct. You can automate it, using a multiple choice format. Precise and straightforward, as long as the questions make sense and the answers are correct. Now, have you ever heard of a general wisdom quiz? No. It would be very difficult to award marks for this. As you are trying to learn a new subject, this is part of what you are trying to measure. Multiple choice questions are easy to mark, but do not show how much people understand. College exams are designed to test pupil's understanding, but this is very hard to do. An essay-based subject may tend to have shorter questions, which are relatively quick to think up, but the marking will take a long time. A more mathematical subject may have longer questions in several parts, which can take a long time to devise, trying to avoid questions from previous years where the answers can be memorised, though the marking might be slightly quicker. Assessing someone's understanding isn't easy. An exam result of 80% would usually be a pass, in fact, possibly distinction. A rocket launch that achieved 80% (I'm not sure how you would measure a percentage, perhaps a fifth of it went into space) is more likely to be a fail.

Trying to shoe-horn a metric onto a space leads to all kinds of anomalies. Sometimes introducing a metric to give league tables or performance metrics leads to problems. As we know, measuring car exhaust emissions caused a stir a while ago [Wikipedia-e]. This involved a claim of deliberately measuring the wrong thing to 'comply' with the Clean Air Act. Rather than deliberate misreporting, metrics can still have unintended consequences. If a team is required to achieve 100% test coverage, then deleting all the code is a sure-fire way to achieve this. Target hit, intention missed. When the league tables for GCSEs (exams for 16-year-olds in the UK) were introduced years ago, schools reported the percentage of pupils who achieved a C or above, aged 16. Many schools allowed their pupils to take exams a year early, allowing them to start on A levels or similar sooner. I remember reading about one school where the pupils took all their exams a year early, so despite being a popular school that was hard to get into, they would have ended at the bottom of this table. Solution? Make the pupils take the exam a year later. Ridiculous, in my opinion. However, trying to get the metric changed would have been difficult. Some programming teams using Agile measure the effort of a task in story points. We know this is an attempt to avoid a one-to-one correspondence with time estimates. I wonder whether these form a metric, in a mathematical sense. I'll leave that as an exercise for the reader.

Assessing progress is difficult, but important. Perhaps you have some metrics you want to share with us? Write in. Or perhaps you learn a smattering of new things by reading *Overload*. Don't forget to let authors know if you enjoyed reading their articles.

## References

[BBC09] 'The 'misunderestimated' president': http://news.bbc.co.uk/1/hi/7809160.stm

[Lippincott18] 'The shape of a program': https://www.youtube.com/watch?v=IP5akjPwqEA

[Oldwood15] http://chrisoldwood.blogspot.com/2015/04/the-daily-stand-up.html

[Wikipedia-a] https://en.wikipedia.org/wiki/Open_set#Open_and_closed_are_not_mutually_exclusive

[Wikipedia-b] https://en.wikipedia.org/wiki/Gantt_chart

[Wikipedia-c] https://en.wikipedia.org/wiki/Kanban_board

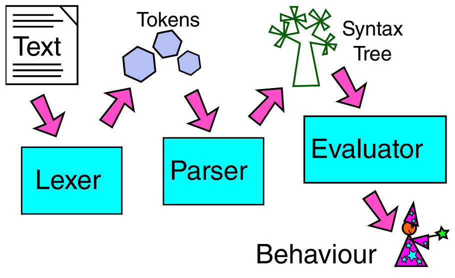
[Wikipedia-d] https://en.wikipedia.org/wiki/There_are_known_knowns

[Wikipedia-e] https://en.wikipedia.org/wiki/Volkswagen_emissions_scandal

# How to Write a Programming Language: Part 3, The Evaluator

We've parsed our tokens: now we need turn them into values. Andy Balaam continues writing a programming language with the evaluator.

This is the third part of our series on writing a programming language. In part one we broke up the code into chunks like numbers, strings and symbols (lexing), and in part two we assembled those parts into a tree structure (parsing). Now we are ready to start understanding and evaluating the parts of that tree structure to produce values and behaviour.



By the end of this article you will have seen all the most important parts of a programming language, and be ready to write your own!

## Recap – lexing and parsing

The lexer and parser take some text like this:

```
print( x + 2 );
```

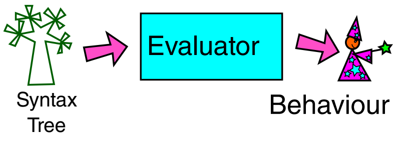
and break it into parts, and then assemble it into a tree structure like this:

```
("call",
  ("symbol", "print"),
  [
    ("operation",
      "+",
      ("symbol", "x"),
      ("number", "2")
    )
  ]
)
```

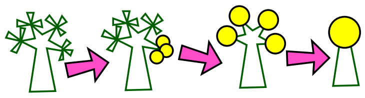
Cell is written in Python and uses Python tuples to hold all the structures it represents. Each tuple contains a string representing its type as the first element, and then other information in the other elements of the tuple.

So far, we have seen tuples representing tokens coming out of the lexer, and tuples representing syntax trees coming out of the parser. This time we will see more tuples, representing values that have been worked out by the evaluator.

## The evaluator calculates values



The evaluator starts at the leaves of the syntax tree and calculates the values of the leaves, then combines together leaves and branches until it ends up with a single value. On the way it may have produced some side effects such as printing something out.



## Scope

Before we look at how the evaluator works out values, we must look at the idea of scope. Scope describes what names can be seen where in our code. Cell, like almost all modern programming languages, uses 'lexical' scope, which means the values you can see are dictated by the position of the code in the text on the screen.

So, for example, this snippet of Cell code:

```
x = "World!";
myfn = {
  x = "Hello,";
  print( x );
};
myfn();
print( x );
```

prints:

```
Hello,
World
```

because the value of **x** inside the function **myfn** is set to **"Hello,"** within the function definition, but it reverts to **"World!"** in code that is outside that block.

This more complicated example:

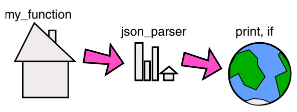
```
outerfn = {
  x = 12;
  innerfn = {
    print(x);
  };
  innerfn;
};

thing = outerfn();
thing();
```

prints **"12"** because the function **innerfn** carries the values it knows about with it, meaning that when we call the function returned by **outerfn()**, which is actually **innerfn** because that is what is returned by **outerfn** when we call it, it runs the **print(x)** line and it still knows what **x** is. Functions that are carrying their values with them are called Closures, and the set of values that is passed around is called an Environment. Environments are key to the way the evaluator works.

## Environments

An environment is a namespace that holds all the symbols that are defined in your program. As illustrated here, each piece of code operates inside a local environment (such as the current function) but can also



**Andy Balaam** Andy is happy as long as he has a programming language and a problem. He finds over time he has more and more of each. You can find his open source projects at artificialworlds.net or contact him on andybalaam@artificialworlds.net

**Without an environment we can't do anything since we don't know where to look up the symbols that are being used in the code**

access symbols from outer environments (such as an outer function) and the global environment, that contains important symbols such as the **if** function.

This structure is provided in Cell by a class called **Env**. It takes a parent environment as a constructor argument, which it holds in **self.parent**. To look up a name we call the **get()** method:

```
class Env:
  # ...
  def get(self, name):
    if name in self.items:
      return self.items[name]
    elif self.parent is not None:
      return self.parent.get(name)
    else:
      return None
```

This method checks whether a symbol is defined locally, and if not, it asks the parent environment. It gives up when it gets to the global environment, which has **None** for its **self.parent** value.

Defining a symbol means calling **set()**, which is simpler:

```
class Env:
  # ...
  def set(self, name, value):
    self.items[name] = value
```

So newly defined symbols are always defined in the local environment, and don't leak out into wider scopes.

## The evaluator

Listing 1 shows the main logic of the evaluator. You can see the full code at https://github.com/andybalaam/cell/blob/master/pycell/eval_.py, but here we can see the main structure is very similar to the code we saw in the previous two parts – a large **if-elif** block responding differently to the various possible structures.

In the evaluator, the **eval_expr()** function takes in an expression to evaluate, and the environment in which to work. Without an environment we can't do anything since we don't know where to look up the symbols that are being used in the code. The environment is an instance of the **Env** class we saw earlier, and the expression is a Python tuple representing part of a syntax tree.

The first part of the tuple tells us the type of syntax tree section we have. We place this into a variable called **typ**, and use it in the **if** block.

## Ordinary values

If we are evaluating a number, we use Python's **float()** function to convert the string form that was captured in the lexer token (as **expr[1]**, the second value in the tuple) into a Python number. This means we can do arithmetic with it later if we need to, and illustrates the fact that the evaluator is where the textual and structural forms of the code are converted into 'meaning' such as finding actual numbers and looking up the values of symbols.

```
def eval_expr(expr, env):
  typ = expr[0]
  if typ == "number":
    return ("number", float(expr[1]))
  elif typ == "string":
    return ("string", expr[1])
  elif typ == "none":
    return ("none",)
  elif typ == "operation":
    return _operation(expr, env)
  elif typ == "symbol":
    name = expr[1]
    ret = env.get(name)
    if ret is None:
      raise Exception(
        "Unknown symbol '%s'." % name)
    else:
      return ret
  elif typ == "assignment":
    var_name = expr[1][1]
    val = eval_expr(expr[2], env)
    env.set(var_name, val)
    return val
  elif typ == "call":
    return _function_call(expr, env)
  elif typ == "function":
    return ("function", expr[1], expr[2],
      Env(env))
  else:
    raise Exception(
      "Unknown expression type: " + str(expr))
```

Listing 1

If we find a string, we have very little to do, since the value of a string looks identical to its form as a lexer token – i.e. it is a tuple of two values, the first of which is **"string"** and the second is the contents of that string.

Next we deal with a special case – there is a special type of value in Cell that is called **None** – we inject this value into the global environment with the name **None**, and the Python tuple to represent its value is **("none",)** i.e. a tuple with just one value in it to represent a special none type. In Cell, **None** is used to describe a missing or empty value. If we find a **None** value like this, we simply return a similar **None** value from **eval_expr**.

The next type of syntax tree we handle is **"operation"** – this represents an arithmetic operation like **"+"** or **"*"**. We call a dedicated function **_operation()** to deal with this, which is shown in listing 2.

The **_operation()** function takes in an expression and environment just like **eval_expr**, and it looks at **expr[1]** to find out what kind of operation is being asked for. As we saw in the previous article, this was populated by the parser and can be **"+"**, **"-"**, **"*"** or **"/"**. In each case, we evaluate the expressions on the left and right of the operator and place them into variables **arg1** and **arg2**, and then combine them together using

```
def _operation(expr, env):
  arg1 = eval_expr(expr[2], env)
  arg2 = eval_expr(expr[3], env)
  if expr[1] == "+":
    return ("number", arg1[1] + arg2[1])
  elif expr[1] == "-":
    return ("number", arg1[1] - arg2[1])
  elif expr[1] == "*":
    return ("number", arg1[1] * arg2[1])
  elif expr[1] == "/":
    return ("number", arg1[1] / arg2[1])
  else:
    raise Exception(
      "Unknown operation: " + expr[1])
```

Listing 2

the appropriate Python arithmetic operator. If the rules of arithmetic in Cell were different from those in Python, this is where we would see the difference. Similarly, if we chose to use some other class to represent numbers, we would have seen that being used when we found a **`"number"`** type, instead of the **`float()`** function. In fact, because Cell is designed to be simple to implement, we choose to use Python's float type and built-in arithmetic for all the numeric operations.

Back in the main code (listing 1) we see the next type is **`"symbol"`**. This means we found a symbol like **`my_function`** in the code, or a built-in symbol like **`print`** or **`if`**. To evaluate it, we simply look it up in the environment using **`Env`**'s **`get()`** method that we saw earlier. If we can't find it, we throw an exception, producing a crude error message for the user.

Similarly, if the type is **`"assignment"`**, we have found code like **`"x = 3"`**, and we use the environment's **`set()`** method to store the value inside the symbol we were given. We make sure to evaluate the value before storing it, by calling **`eval_expr()`** again. In this case, and in other cases, we call **`eval_expr`** from inside **`eval_expr`**. This is called recursion, and makes perfect sense if you don't think about it too much, or, alternatively, if you think about it a lot.

## Functions

The last two types to deal with both concern functions. First, **`"call"`** means we are looking at a function call, something like **`"my_fn(3)"`**. We deal with this in a separate function called **`_function_call`**, shown in listing 3.

The **`_function_call`** function (did I mention that writing a programming language can get confusing when you start writing functions about functions, or variables containing variables?) evaluates the function object that is being called and checks its type.

The type will be either **`"function"`** or **`"native"`**. The **`"function"`** type means that this is a normal function written in Cell. In order to run it, we check we have been given the right number of arguments, and then create a temporary environment based on the environment carried around by the function itself (recall the discussion of scope and closures earlier). Next it puts the argument values into that environment using the names provided in the function definition, and then calls **`eval_list`**. It is not shown here, but **`eval_list`** just evaluates each line of the function one by one. It actually ignores the values of all those lines except the last one, which it uses as the function's return value.

A **`"native"`** type is a function that is not written in Cell, but instead is provided as part of Cell's implementation. This means the function is written in Python (because Cell is written in Python). In this case, **`fn[1]`** is a Python function. We check the number of arguments again, and call the function, passing in the environment and the arguments. Python functions that provide native Cell functions actually take one more argument in Python than you see in Cell, because the first argument is the environment in which to run. We don't create a sub-environment in which to run in this case, because native functions can do all kinds of magic, like modifying the environment in which they are running. Writing these

```
def _function_call(expr, env):
  fn = eval_expr(expr[1], env)
  args =
    list((eval_expr(a, env) for a in expr[2]))
  if fn[0] == "function":
    params = fn[1]
    fail_if_wrong_number_of_args(expr[1],
      params, args)
    body = fn[2]
    fn_env = fn[3]
    new_env = Env(fn_env)
    for p, a in zip(params, args):
      new_env.set(p[1], a)
    return eval_list(body, new_env)
  elif fn[0] == "native":
    py_fn = fn[1]
    params = inspect.getargspec(py_fn).args
    fail_if_wrong_number_of_args(expr[1],
      params[1:], args)
    return fn[1](env, *args)
  else:
    raise Exception(
      "Attempted to call something that is not a
function: %s" % str(fn)
    )
```

Listing 3

functions is slightly odd because the arguments passed in are tuples representing Cell values, rather than simple Python types, and any symbols etc. need to be looked up in the environment provided.

Back in listing 1, the last type we deal with is **`"function"`**. So far we've only dealt with calling functions, but this is about the definition of a function – in Cell that means code inside curly braces. In fact, most of the hard work of defining the function has been done by the parser, which made us a list of argument names and expressions that make up the body of the function. All we need to do in the evaluator is wrap all that up with a new **`Env`** object that is the environment passed around with the function. This means if we return a function definition from another function, it can still access the variables it could see when it was defined because its environment (and the parent environments) are held with it. We rely on Python's object references to make sure the values we are interested in are still available when we use them.

The **`else`** part of listing 1 throws an exception because we have found a syntax tree that we don't recognise (in the famous last words of all programmers, this should never happen) and we are done.

## Side effects

With all this discussion of finding values, it seems strange to say that most programming languages, including Cell, actually do nothing with the values they find. In order to make a useful program, the programmer must use the values to produce 'side effects' – things that make something happen in the world outside the program. In Cell, we have a native function called **`"print"`** that prints out values we have calculated. Most other languages have lots of available side effects such as creating and modifying files, displaying windows, and making sounds.

## Summary

We've completed our journey: this time we saw how to take a syntax tree and turn it into meaningful values. When we combine this with being able to break code into separate chunks (lexing) and building those chunks into a syntax tree (parsing), we've covered all the basic building blocks needed to write an interpreter. Are you ready to design your own language?

You can find all the code for Cell at https://github.com/andybalaam/cell, and I would to hear from you if you have made your own language – let me know through GitHub or Twitter on @andybalaam or in the fediverse on andybalaam@mastodon.social, and check out a video series about Cell on my YouTube page at https://www.youtube.com/user/ajbalaam. ■

# P1063 vs Coroutines TS: Consensus on High-Level Semantics

Dmytro Ivanchykhin, Sergey Ignatchenko and Maxim Blashchuk argue that we need coroutines TS now to improve-based-on-experience later.

**Disclaimer:** This article takes for granted that readers understand what coroutines are about. If this concept is unfamiliar to you (hey, we're speaking about standard proposals here!) make sure to take a look at [Nishanov15] and [McNellis16].

**Disclaimer #2:** Just to avoid any doubt, this article is not written with the help of some magical oracle or other source of infinite wisdom; rather, this article (just as any other article for that matter) merely represents an opinion of its authors (which may or may not coincide with the opinion of the *Overload* editor). In addition, this article is neither sanctioned nor sponsored by any government, WG21, or other official body.

Quite recently, we have learned that newly appeared [P1063R0][1] and its 'Core Coroutines' proposal has led to controversy, which got in the way of voting Coroutines TS [N4760] (a.k.a. Gor-routines ☺) into C++20. As big fans of coroutines in general and asynchronous processing in particular, we became worried about this development, so we took a look at this situation from the point of view of an app-level developer (and occasional architect). In other words, we do *not* really care about implementation details and compiler complexities – instead, we care about stuff such as readability, performance, backward compatibility and code maintenance costs; and of course, another extremely important consideration is when we'll be able to start using those exciting new C++ features (without standardization we're not really able to use any feature on a massive scale as the associated risks are just too high).

## App-level point of view

From our app-level point of view we can say that all code-using coroutines we can think of, in most of real-world projects will fall into two separate categories:

- code which uses `co_await`[2]. Let's call this code *end-programmer* code (mimicking *end-user*-related terminology). This code will be interspersed with business logic. It will change very frequently, and will be spread all over the code base; as a result, any change to the semantics of `co_await` will be crazily expensive at app-level, and most likely such a change won't be feasible.
- code which enables using `co_await` (for Coroutines TS, it is all the `await_*()` stuff; for Core Coroutines, it is overloaded `operator [<-]` etc.). Let's call this code *infrastructure app-level code*. For all the use cases we can think of for serious projects, this code is going to be confined to some kind of framework/glue/... layer. Moreover, this layer usually doesn't contain business logic and tends to be quite limited in size, with changes to this layer being quite rare. In fact, a similar point of view is articulated in [P1063R0]: "authors of wrapper libraries... we expect those to be relatively rare".

## Coroutines TS vs P1063: end-programmer example

Let's take the very same piece of code and see how it can be written under both proposals.

1. Based on [P0973R0], with two of the three authors being the same.
2. Or `operator [<-]`, it doesn't really matter.

### Coroutines TS a.k.a. Gor-routines

```
future<int> count_bytes(Connection& connection) {
  int bytes_read = 0;
  vector<char> buffer(1024);

  while(!connection.done()) {
    bytes_read +=
      co_await connection.Read(buffer.data(),
      buffer.size());
  }
  co_return bytes_read;
}
```

### P1063R0 a.k.a. "Core Coroutines"

```
 auto count_bytes(Connection& connection) =>
      make_future<int>([&connection] do {
  int bytes_read = 0;
  vector<char> buffer(1024);

  while(!connection.done()) {
    bytes_read +=
      [<-]connection.Read(buffer.data(),
      buffer.size());
  }
  return bytes_read;
});
```

## End-programmer semantics: exactly the same for Coroutines TS and Core Coroutines

Following from the 'App-level point of view' section above, the most important (and utterly unchangeable later) portion of any coroutines proposal is the semantics of `co_await` (or whatever other syntax it may have). Historically, there have been several significantly different semantics of await (for example, in a relatively recent [P0114R0], it was argued not to require a marker for a suspend point – which, BTW, was argued later to be a Bad Thing™ for app-level [NoBugs17]).

**Dmytro Ivanchykhin** has 10+ years of development experience, and has a strong mathematical background (in the past, he taught maths at NDSU in the United States). Dmytro can be contacted at d_ivanchykhin@yahoo.com

**Sergey Ignatchenko** has 15+ years of industry experience, including being a co-architect of a stock exchange, and the sole architect of a game with 400K simultaneous players. He currently holds the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com

**Maxim Blashchuk** Maxim Blashchuk has substantial development experience, most of it with embedded programming. Recently he joined a team performing research on low-level C++ libraries providing properties such as determinism and memory safety.

However, if we take a look at currently competing proposals (Coroutines TS and P1063), we'll see (to the best of our understanding) that

> *the semantics of* `co_await` *and the proposed* `operator [<-]`*, at least at the point where* `co_await`/`[<-]` *is used by end-programmer code, is* ***exactly the same****.*

Not only is the flow interrupted (with the possibility of being resumed) in the very same manner for both proposals, but also all properties that are observable from the business-logic level (such as enforcing calls around async call to be asynchronous) are the same too.

As noted above, such consensus on high-level semantics (compared to `co_await`) wasn't the case for earlier proposals such as [P0114R0], but is the case for [P1063R0].

## On end-programmer syntax

While the *semantics* of the proposals are exactly the same, there are a few high-level *syntactic* differences between P1063 and 'Coroutines TS':

- Replacement of `co_await` with an identically used but differently named `operator [<-]`. Not that it really matters for our current discussion, but we have to mention that we have our doubts about an argument from [P1063R0] that the "`co_await` keyword is an overt manifestation of the TS's preference for the asynchronous use case".

  We feel that, even when we're writing generators, we can consider what is happening at that point as 'awaiting' something external to our code flow to happen (even if it is another generator). Indeed, with `co_await` (or `[<-]`) we're interrupting the program flow – but why? To await something external to our program flow to happen (whether it is an async event, another generator, or whatever else). In addition, the concept of unwrapping is guaranteed to be alien to the vast majority of app-level developers (even more so for existing C++ app-level developers). That being said, we are quite indifferent to the choice between `co_await` and `[<-]`.
- Explicit designation of coroutines (vs implicit one in Coroutines TS, where being coroutine is derived from `co_await` or `co_return` being used). In general, there are arguments to have app-level code explicitly documented, but this is still a very minor issue. OTOH, the way it is done in P1063 is very verbose (that's even after they're relying on a yet another pending proposal – and modifying it further (!) – to make syntax more palatable) and we feel that it is at odds with the all-important "direct expression of ideas" principle which was laid out in [Stroustrup04].
- Lambda-like syntax in P1063 vs traditional function syntax in Coroutines TS. Again, it doesn't matter too much for the purposes of our current discussion, but we have to say that lambda-like syntax (a) is more error-prone (keeping all those brackets matching is yet another thing to care about while programming), and (b) as lambda syntax differs significantly from usual function syntax, we feel that it undermines the time-honoured understanding of subroutines being "special cases of more general program components, called coroutines" [Knuth].
- Replacement of `co_return` with `return`. TBH, this is the least of our syntactic concerns (not that other syntactic concerns are significant); we explicitly do *not* care about it. Either way is perfectly fine with us and we have no idea why it is so important for the authors of [P1063R0].

However, the most important property of all the syntactic differences is

> *As the differences are purely syntactical, nothing prevents us from either (a) choosing whatever syntax is preferred right now, without delaying the whole thing for N years, and/or (b) adding syntactic alternatives later*

## Customization points: mostly an implementation detail that can be changed later

In fact, what we have already discussed above is only a minor part of the differences between Coroutines TS and P1063; however, all the remaining differences we're aware of are either (a) about optimizations (which we'll discuss a bit later), or (b) about so-called 'customization points' in P1063-speak, or, from our current perspective, are about what we decided to call 'app-level infrastructure code'. Let's take a closer look at those customization points and app-level infrastructure code.

As for app-level infrastructure code, the most important properties are:

1. it is hidden from the view of the end-user programmer
2. it is rarely changed
3. and it is small.

(BTW, as it was already noted above, P1063 itself has indications which agree with this point of view.)

As a direct result of item #1 above, from the end-user programmer point of view,

> *customization points/app-level infrastructure code are nothing but implementation details*

Moreover, from #2 and #3 it follows that costs of rewriting such code – if such a need will ever arise – will be small; this opens us a door to change them later *if*/when it is demonstrated that such a change is necessary.

## Performance and allocations

Another set of objections to Coroutines TS laid out in P1063 is about performance and lack of normative control over allocations. This one is simple – P1063 itself acknowledges that all their performance/allocation concerns can be addressed by extending Coroutines TS later: "These all appear to be pure extensions, so they could be done post-C++20 if need be." As a result, we don't really care about performance issues now, as optimizations (most of them already existing) can be made normative later.

This is without mentioning that the whole argument along the lines of "we don't want allocations" becomes more and more moot as soon as we take into account that modern single-threaded allocators can perform `malloc()`+`free()` pairs in as little as 15 CPU cycles [Ignatchenko18]; with this cost being comparable to the cost of a single branch mis-prediction(!), efforts related to eliminating allocations become more and more of a 'yet another optimization' rather than 'a thing we should care about *a lot*'.

## Analysis: coroutines TS CAN be voted in, even if P1063 is right on every point

Now, we're done with the preliminaries and can proceed to the point of this article. Let's assume for the moment that ISO committee and the industry follow this path:

- WG21 has a short discussion on syntax for Coroutines TS (or makes a joint proposal in this regard). Our own preferences in this regard were outlined above, but TBH we will accept any kind of syntax to get coroutines into C++20 (that is, as long as end-programmer semantics remains the same).
- WG21 votes Coroutines TS into C++20.
- In a few years, everybody and their dog are using Coroutines TS.

Now, let's consider all the possible scenarios with regards to the merits of P1063 in this context (keeping in mind its claims about being more generic than Gor-routines):

- **If** by the end of the day (and as Gor currently argues), P1063 won't be able to provide any significant improvements (that is, over an improved-over-time Coroutines TS), accepting Coroutines TS was the right thing; end of discussion.
- **If** P1063 happens to be perfect as promised, it should be possible to rewrite the current implementation of Coroutines TS (including the code providing for `await_suspend()` etc.) in P1063 style. This means that: (a) at end-programmer level, there will be *exactly zero* changes; (b) at the level of the app-level infrastructure code: (b1) for the time being, we'll have Coroutines TS (good enough for us), and (b2) when P1063 is standard-ready (in the very best case C++26(!)), we'll have both ways of describing things (NB: unless demonstrated to be superior in performance, we're sure that lots of developers – ourselves included – will still prefer the Coroutines TS way).

- **If** P1063 happens to be not as perfect as promised but still better than Coroutines TS, it might be impossible to rewrite the current implementation of Coroutines TS in the P1063 style. This will mean that: (a) at end-programmer level, there are still *exactly zero* changes; (b) at the level of the app-level infrastructure code: (b1) for the time being, we'll have Coroutines TS, and (b2) when P1063 is standard-ready, we'll have two separate ways of describing things. This *might* mean – when the project benefits from it – that a very small portion of the project code (from experience, 2–5%) *may* need to be rewritten; taking into account that for the vast majority of projects (90+% being a conservative estimate) Coroutines TS are expected to be 'good enough', we're speaking about 0.2–0.5% of all the code using Coroutines-TS being rewritten. We are confident that it is not too much of a price for having Coroutines TS at least 6 years earlier (and note that this 0.2–0.5% rewrite happens only IF P1063 is better than Coroutines TS *but* is not as perfect as promised).
- **If** some other way to implement customization (even better than P1063) arises meanwhile: (a) at end-programmer level, there are still *exactly zero* changes; (b) at the level of the app-level infrastructure code: (b1) for the time being, we'll have Coroutines TS, and (b2) when some-other-way is standard-ready, we'll have one or two separate ways of describing things. However, along the lines above, our estimate is that – even in the worst case – only 0.2–0.5% of the code using the Coroutines TS will have to be rewritten.

In other words:

> *In each and every conceivable scenario, including the one where P1063 is right with each and every significant claim they're making, voting in Coroutines TS is The Right Thing To Do™.*

Voting Coroutines TS into C++20 will provide two all-important benefits:

- in the industry, we'll be able to use goodies of coroutines right now (and not 6+ years later)
- even more importantly, while we're using it – **we'll see more real-world use cases, and will be able to criticize current implementation not from purely abstract point of view, but based on the needs of the real world**.

In a sense, what we have is a situation similar to *prima facie* hearing in the criminal law of some countries; in such hearings, even if all the evidence presented by the prosecution, is taken at face value, but the defendant is still not guilty, there is no need to argue about the merits of the evidence, and the decision can be made in favour of the defendant without conducting a full hearing. Such cases are admittedly rare, but in our case of P1063-vs-Coroutines-TS, it is possible because of two major observations:

- when considering 99+% of the relevant code, the semantics of the Coroutines TS and P1063 is *exactly the same*. In other words, **we have consensus on end-programmer semantics**.
- And from the point of view of the all-important end-programmer, anything else can be seen as an implementation detail, and Coroutines TS sets the abstraction boundary for *customization points* to be very close to the end-user programmer, preventing app-level programmers from implementing it themselves. This, in turn, allows specifying this layer later (which is essentially what P1063 tries to do). In other words, we're going in the direction from being under-specified to over-specified (which, unlike the other way around, is perfectly feasible).

Or, trying to approach the same thing from a different perspective: we clearly feel that current Coroutines TS does represent 'gradual expansion' without degenerating into 'opportunistic hacking' as defined in [P0976] by Bjarne Stroustrup.

> Gradual expansion, relying on feedback, is my ideal. Better an incomplete design than a poor/clumsy/bloated 'complete solution'.

And FWIW, 'relying on feedback' is *not* really possible until `co_await` makes it into the standard one way or another; it means that the **merits of voting in Coroutine TS right now go far beyond our simple desire to start using it ASAP: it is also important to ensure that the end-product (the C++ standard) is the best one possible**. Indeed, if some over-specified stuff makes it into the standard, it will be next to impossible to replace it later – *and right now we just don't have sufficient information to say which way is the best one*; in this sense, the approach taken by Coroutines TS (to hide as much as possible beyond the implementation boundary, or – in other words – 'to under-specify rather than over-specify') is a Good Thing™; combined with an as-early-as-possible acceptance of Coroutines TS into the standard, this allow to get that all-important feedback Bjarne refers to in [P0976].

## Conclusion

We hope that we have made a case for 'voting for Coroutines TS right now regardless of the merits of the finer points of P1063' (that is, points going beyond two major observations listed above):

- we'll be able to use coroutines at end-programmer level (where consensus already exists) right away
- as for customization points, even if P1063 is The Way To Go(tm) – it can be added later when (**if**) this becomes apparent. In addition, while we're using coroutines in the wild, **we'll become much more knowledgeable about real-world use cases** – and the ways that Coroutines TS needs to be improved (who knows, maybe a more-straightforward model to express 'customization points' arises as we learn more about coroutines from deploying Coroutines TS – and current Coroutines TS has abstraction boundaries which leave room for different ways of specifying 'customization points').

In other words, we hope we have demonstrated that **voting in Coroutines TS is The Right Thing To Do™** without criticizing P1063 itself.

Phew. We rest our case. ■

## References

[Ignatchenko18] (Re)Actor Allocation At 15 CPU Cycles, Sergey Ignatchenko, Dmytro Ivanchykhin, Marcos Bracco, *Overload* #142, https://accu.org/index.php/journals/2533

[Knuth] *The Art of Computer Programming*, Donald Knuth, Vol. I

[McNellis16] Introduction to C++ Coroutines, James McNellis, *CppCon2016*, https://www.youtube.com/watch?v=ZTqHjjm86Bw

[Nishanov15] C++ Coroutines – a negative overhead abstraction, Gor Nishanov, CppCon2015, https://www.youtube.com/watch?v=_fu0gx-xseY

[N4760] Working Draft, C++ Extensions for Coroutines, Gor Nishanov, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4760.pdf

[NoBugs17] Eight Ways to Handle Non-Blocking Returns in Message-Passing Programs, 'No Bugs' Hare, http://ithare.com/eight-ways-to-handle-non-blocking-returns-in-message-passing-programs-with-script/3/, CppCon17

[P0114R0] Resumable Expressions, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0114r0.pdf

[P0973R0] Coroutines TS Use Cases and Design Issues, Geoff Romer, James Dennett, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0973r0.pdf

[P0976] The Evils of Paradigms Or Beware of one-solution-fits-all thinking, Bjarne Stroustrup, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0976r0.pdf

[P1063R0] Core Coroutines, Geoff Romer, James Dennett, Chandler Carruth, http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1063r0.pdf

[Stroustrup04] Speaking C++ as Native (Multi-paradigm Programming in Standard C++), Bjarne Stroustrup, http://ewh.ieee.org/r5/central_texas/austin_cs/presentations/2004.02.25.pdf

# Implementing the Spaceship Operator for Optional

Comparison operators can get complicated. Barry Revzin explores how the new operator <=> helps.

In November 2017, the C++ Standards Committee added **`operator<=>`**, known as the spaceship operator [P0515], to the working draft for what will eventually become C++20. This is an exciting new language feature for two reasons: it allows you to write *one* function to do all your comparisons where you used to have to write *six*, and it also allows you to write *zero* functions – just declare the operator as defaulted and the compiler will do all the work for you! Exciting times.

The paper itself presents many examples of how to implement the spaceship operator in various situations, but it left me with an unanswered question about a particular case – so I set out trying to figure out. This post is about the journey of how to implement **`operator<=>`** for **`optional<T>`**. First, thanks to John Shaw for helping work through all the issues with me. And second, the resulting solution may not be correct. After all, I don't even have a compiler to test it on. So if you think it's wrong, please let me know (and please post the correct answer in this self-answered StackOverflow question [StackOverflow]).

First, the specs. **`optional<T>`** has three categories of comparisons, all conditionally present based on the facilities of the relevant types:

- **`optional<T>`** compares to **`optional<U>`**, where valid (6 functions).
- **`optional<T>`** compares to **`U`**, where valid (12 functions). I'm sceptical of this particular use-case, but this post is all about implementing the spec.
- **`optional<T>`** compares to **`nullopt_t`** (12 functions). This case is trivial to implement, since several of the operations are simply constants (e.g. **`operator>=(optional<T>, nullopt_t)`** is **`true`**). But, that's still 12 trivial-to-implement functions.

In all cases, the semantics are that a disengaged optional is less than any value, but all disengaged values are equal. The goal is to take advantage of the new facilities that the spaceship operator provides us and reduce the current load of 30 functions to just 3.

We'll start with the **`optional`** on **`optional`** comparison. There are four cases to consider: both on, left on only, right on only, and both off. That leads us to our first approach (Listing 1).

The spaceship operator returns one of five different comparison categories:

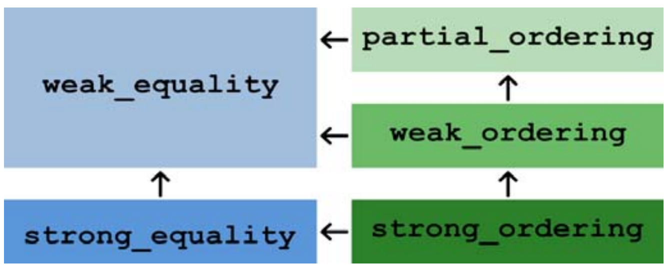
- **`strong_ordering`**
- **`weak_ordering`**
- **`partial_ordering`**
- **`strong_equality`**
- **`weak_equality`**

**Barry Revzin** I'm a C++ developer for Jump Trading, member of the C++ Standards Committee, also 'Barry' on StackOverflow. On the side, I'm also an avid swimmer and do data analysis for SwimSwam magazine. And take care of my adorable Westie and life mascot, Hannah. You can reach me at barry.revzin@gmail.com

```
template <typename T>
class optional {
public:
  // from here on out, assuming that heading
  // exists ...

  template <typename U>
  constexpr auto operator<=>(
    optional<U> const& rhs) const
    -> decltype(**this <=> *rhs)
  {
    using R = decltype(**this <=> *rhs);
    if (has_value() && rhs.has_value()) {
      return **this <=> *rhs;
    } else if (has_value()) {
      return R::greater;
    } else if (rhs.has_value()) {
      return R::less;
    } else {
      return R::equal;
    }
  }
};
```

Listing 1



The comparison categories for the spaceship operator

Figure 1

Each of these categories has defined named numeric values. In the paper, the categories are presented in a way that indicates the direction in which they implicitly convert in a really nice way, so I'm just going to copy that image as Figure 1 (all credit to Herb Sutter).

Likewise, their table of values is shown in Table 1 on the next page.

Just carefully perusing this table, it's obvious that our first implementation is totally wrong. **`strong_ordering`** has numeric values for less, equal, and greater… but the rest don't! In fact, there is no single name that is common to all 5. By implementing it the way we did, we've reduced ourselves to only supporting strong orderings.

**The shapeship operator for bools gives us a strong_ordering, which is convertible to everything**

| Category | Numeric values | | | Non-numeric values |
|---|---|---|---|---|
| | -1 | 0 | 1 | |
| strong_ordering | less | equal | greater | |
| weak_ordering | less | equivalent | greater | |
| partial_ordering | less | equivalent | greater | unordered |
| strong_equality | | equal | non-equal | |
| weak_equality | | equivalent | non-equivalent | |

Table 1

So if we can't actually name the numeric values, what do we do? How can we possibly do the right thing?

Here, we can take advantage of a really important aspect of the comparison categories: convertibility. Each type is convertible to all of its less strict versions, and each value is convertible to its less strict equivalents. `strong_ordering::greater` can become:

- `weak_ordering::greater` or
- `partial_ordering::greater` or
- `strong_equality::nonequal` or
- `weak_equality::nonequivalent`

And the way we can take advantage of this is to realize that we don't really have *four* cases, we have two: both on, and not that. Once we're in the 'not' case, we don't care about the values anymore, we only care about the bools. And we already have a way to do a proper 3-way comparison: `<=>`! (See Listing 2.)

The shapeship operator for `bool`s gives us a `strong_ordering`, which is convertible to everything. So that part is guaranteed to work and do the right thing (I encourage you to work through the cases and verify that this is indeed the case).

But this still isn't quite right. The problem is actually `<=>` (thanks, Captain Obvious?). You see, while `a < b` is allowed to fallback to `a <=> b < 0`, the reverse is not true. `a <=> b` is not allowed to call anything else (besides `b <=> a`). It either works, or it fails. By using the spaceship operator directly on our values, we're actually reducing ourselves to only those modern types that support 3-way comparison. Which, so far, is no user-defined types. Moreover, `<=>` doesn't support mixed-integer comparisons, so even for those types that come with built-in spaceship support (that's a fantastic phrase), we would effectively disallow comparing an `optional<int>` to an `optional<long>`. So, this operator in this particular context isn't very useful.

```
template <typename U>
constexpr auto operator<=>(optional<U>
  const& rhs) const
  -> decltype(**this <=> *rhs)
{
  if (has_value() && rhs.has_value()) {
    return **this <=> *rhs;
  } else {
    return has_value() <=> rhs.has_value();
  }
}
```

Listing 2

```
template <typename U>
constexpr auto operator<=>(optional<U>
  const& rhs) const
  -> decltype(compare_3way(**this, *rhs))
{
  if (has_value() && rhs.has_value()) {
    return compare_3way(**this, *rhs);
  } else {
    return has_value() <=> rhs.has_value();
  }
}
```

Listing 3

So what are we to do? Re-implement 3-way comparison ourselves manually? Nope, that's what the *library* is for! Along with language support for the spaceship operator, C++20 will also come with several handy library functions and the relevant one for us is `std::compare_3way()`. This one will do the fallback: it prefers `<=>`, but if not will try the normal operators and is smart enough to know whether to return `strong_ordering` or `strong_equality`. *And* it's SFINAE-friendly. Which means for our purposes, we can just drop-in replace our too-constrained version with it (see Listing 3).

And I think we're done.

Now that we've figured out how to do the optional-vs-optional comparison, comparing against a value is straightforward. We follow the same pattern for the value-comparison case, we just need to know what to return in the case where the optional is disengaged. Semantically, we need to indicate that the optional is less than the value. Again, we can just take advantage that all the comparison category conversions just Do The Right Thing and use `strong_ordering::less` (see Listing 4).

```
template <typename U>
constexpr auto operator<=>(U const& rhs) const
  -> decltype(compare_3way(**this, rhs))
{
  if (has_value()) {
    return compare_3way(**this, rhs);
  } else {
    return strong_ordering::less;
  }
}
```

Listing 4

```
constexpr strong_ordering operator<=>(nullopt_t )
const
{
  return has_value() ? strong_ordering::greater
                     : strong_ordering::equal;
}
```

Listing 5

We just replaced 12 *functions* (that, while simple, are certainly non-trivial to get right) with 10 *lines of code*. Mic drop.

All that's left is the **`nullopt_t`** comparison, which is just a simple comparison (Listing 5).

Putting it all together, and Listing 6 is what we end up with to cover all 30 **`std::optional<T>`** comparisons.

Not bad for 25 lines of code?

Let me just reiterate that I'm not sure if this is the right way to implement these operators. But that's the answer [StackOverflow] I'm sticking with until somebody tells me I'm wrong (which, if I am, please do! We're all here to learn).

Needless to say, I'm very much looking forward to throwing out all my other comparison operators. Just… gotta wait a few more years.

## Bonus level

Here's what I think a comparison operator would look like for **`std::expected<T, E>`**. The semantics here are that the values and errors compare against each other, if they're the same. If they're different types, the value is considered greater than the error. Although, for the purposes of this exercise, the specific semantics are less important than the fact that we get consistent semantics. And I think the right way to implement consistent semantics is as shown in Listing 7.

```
template <typename T>
class optional {
public:
  // ...

  template <typename U>
  constexpr auto operator<=>(optional<U>
    const& rhs) const
    -> decltype(compare_3way(**this, *rhs))
  {
    if (has_value() && rhs.has_value()) {
      return compare_3way(**this, *rhs);
    } else {
      return has_value() <=> rhs.has_value();
    }
  }

  template <typename U>
  constexpr auto operator<=>(U const& rhs) const
    -> decltype(compare_3way(**this, rhs))
  {
    if (has_value()) {
      return compare_3way(**this, rhs);
    } else {
      return strong_ordering::less;
    }
  }

  constexpr strong_ordering
    operator<=>(nullopt_t ) const
  {
    return has_value() ? strong_ordering::greater
    : strong_ordering::equal;
  }
};
```

Listing 6

```
template <typename T, typename E>
class expected {
public:
  // ...
  template <typename T2, typename E2>
  constexpr auto operator<=>(expected<T2, E2>
    const& rhs) const
    -> common_comparison_category_t<
        decltype(compare_3way(value(),
          rhs.value())),
          decltype(compare_3way(error(),
          rhs.error()))>
  {
    if (auto cmp = has_value() <=>
        rhs.has_value(); cmp != 0) {
      return cmp;
    }
    if (has_value()) {
      return compare_3way(value(), rhs.value());
    } else {
      return compare_3way(error(), rhs.error());
    }
  }
};
```

Listing 7

**`common_comparison_category`** is a library metafunction that gives you the lowest common denominator between multiple comparison categories (which hopefully is SFINAE-friendly, but I'm not sure). The first if check handles the case where the value-ness differs between the two **`expected`** objects. Once we get that out of the way, we know we're in a situation where either both are values (so, compare the values) or both are errors (so, compare the errors). Just thinking of how much code you have to write today to accomplish the same thing makes me sweat… ■

## References

[P0515] 'Consistent comparison' (2017) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0515r3.pdf

[StackOverflow] 'Implementing operator<=> for optional<T>' https://stackoverflow.com/questions/47315539/implementing-operator-for-optionalt

# Compile-time Data Structures in C++17: Part 2, Map of Types

Compile time type selection allows static polymorphsim. Bronek Kozicki details an implementation of a compile time map.

In part one of the series, we were introduced to a 'set of types' – a compile-time data structure where both insertions and lookups are performed during the compilation, rather than in runtime, which gave them 'O(0)' complexity. The useful functionality this enabled was a **`test`** meta-function (technically, a template variable), returning **`true`** during the compilation time if a given tag type was present in the set (and **`false`** otherwise). The use case for such a data structure was perhaps not the most convincing **`if constexpr`** or a parameter to **`std::enable_if`** and **`std::conditional`**.

This part of the series starts with a more sophisticated scenario shown in Listing 1.

As we can see, the class **`Foo`** is decoupled from any implementation of **`IFuzzer`**, and yet it can instantiate it with the help of **`Map`**, which performs the necessary mapping with the selector type **`Fuz`**, to find the implementation type **`Fuzzer`**. The mapping of **`Fuz`** to **`Fuzzer`** is defined in one place only, inside **`int main()`**, which means that we have achieved 'parametrisation from the above' with static polymorphism.

The implementation of such a 'map of types' is the subject of this article.

```
struct IFuzzer {
  virtual ~IFuzzer() = default;
  virtual std::string fuzz(std::string) const = 0;
};

struct Fuz {};

struct Foo {
  template <typename Map> Foo(Map) {
    if constexpr ((bool)Map::template test<Fuz>)
      fuzz = std::make_unique
             <typename Map::template type<Fuz>>();
  }
  std::unique_ptr<IFuzzer> fuzz;
};

struct Fuzzer : IFuzzer {
  std::string fuzz(std::string n) const override {
    return n + "-fuzz!"; }
};

int main() {
  constexpr static auto m1 = map<>{}
    .insert<Fuz, Fuzzer>();

  const Foo foo {m1};
  std::cout << foo.fuzz->fuzz("Hello")
            << std::endl;
}
```

Listing 1

## Overview of part 1

There is no such thing as 'O(0)' time complexity of a function (hence the quotes) because time complexity implies that some action will actually be performed during program execution. Here we are asking the compiler to perform all the actions required by the function (or more accurately, a meta-function) during compilation itself, which allows us to use the result as an input for further compilation.

A meta-function might be a function with a **`constexpr`** specifier, but typically we will use either a template type (wrapped in a **`using`** type alias if nested inside a template type) or a **`constexpr static`** variable template (also nested inside a template type). In the former case, a result of a meta-function is a type, and in the latter, it is a value.

A tag type is a type which is meant to be used as a name – it has no data members and no useful member functions. The only purpose of objects of such types is the deduction of their type. Examples in the C++ standard library include **`std::nothrow_t`** or types defined for various overloads of the **`std::unique_lock`** constructor.

A pure function is a function that has no side effects and, for any valid and identical set of inputs, always produces the same output. For example, any deterministic mathematical function is also a pure function. A function which takes no input, but always produces the same result and has no side-effect, is also a pure function. Sadly, mathematical functions in the C++ standard library are not pure functions: to be compatible with C, they are saddled with a side effect (manipulating the **`errno`** variable). We can view many meta-functions as pure functions.

A limitation of meta-functions is that they do not have a stack in any traditional sense (they have template instantiation depth instead), and cannot manipulate variables. They can produce (immutable) variables or types, which means that they can be used to implement recursive algorithms. Such an implementation will be typically a template type, where at least one specialisation implements the general algorithm, while another specialisation implements the recursion terminating condition. The compiler selects the required 'execution path' of the recursive algorithm by means of template specialisation matching.

A higher order function is a function which consumes (or produces, or both) a function. Since, in our case, a (meta)function is a template, we can implement a higher order (meta)function consuming a (meta)function, as a template consuming template parameter (or in other words, a 'template template parameter'). Since template types can naturally output a template type, any meta-function which is a type can trivially produce a meta-function.

What is a 'selector type'? In a C++ standard library, a selector in **`std::map<int, std::string>`** is some value of type **`int`** used to select value in a map. In our case, a selector type is just a tag type but used

**Bronek Kozicki** developed his lifelong programming habit at the age of 13, teaching himself Z80 assembler to create special effects on his dad's ZX Spectrum. BASIC, Pascal, Lisp and a few other languages followed, before he settled on C++ as his career choice. In 2005, he moved from Poland to London, and promptly joined the BSI C++ panel with a secret agenda: to make C++ more like Lisp, but with more angle brackets. Contact him at brok@incorrekt.com

What should we do with an element which is not there, but something is mapped to it?

to select something (usually other than itself). Any type can be employed as a selector with the help of overloading, but there are few points to note:

- The result of such performed selection will be the return type of the selected overload. Keyword **`decltype`** can be used to deduce this type.
- If we want to do something useful when a match is not found (rather than fail the compilation), a template parameter can be used.
- The functions in the overload set do not need to be defined, as we only care about the return type of the function found by overloading (the actual function never gets called).
- The overload selection should be strengthened against accidental overloading matches, e.g. by implicit type conversion.
- We should avoid instantiating objects of arbitrary types for overloading parameters or return value.

The two last points may seem to contradict the use of overloading until we realise that any type can be made a parameter of a simple ‘wrapper’ template type, and such a template can then be used instead, both for function parameters and their result. An example of such a selection based on type is presented below, in Listing 2.

The example in Listing 2 has an obvious limitation – selector and return types are both tightly coupled, inside the declarations of overload set **`pair`** functions. It does, however, demonstrate the principles. The **`type`** template alias instantiates a wrapper class appropriate for the selector type (for example, **`wrap<Fuz>`**) and then passes it to the overload set of **`pair`** functions. The result of overloading is deduced by the **`decltype`** keyword. Since that return type is also a wrapper template, we refer to its **`type`** nested alias to unwrap the result type. In this example, we do not do much with the selected type, using the compile-time **`static_assert`** only to verify that it is, indeed, the type expected.

```
#include <type_traits>

template <typename T> struct wrap {
  constexpr explicit wrap() = default;
  using type = T;
};

struct Fuz {};
struct Baz {};
struct Bar {};

wrap<double> pair(wrap<Fuz>);
wrap<bool> pair(wrap<Baz>);
template<typename T> wrap<void> pair(wrap<T>);

template <typename T>
using type = typename
decltype(pair(wrap<T>{}))::type;

int main() {
  static_assert(std::is_same_v<type<Fuz>,
    double>);
  static_assert(std::is_same_v<type<Baz>, bool>);
  static_assert(std::is_void_v<type<Bar>>);
}
```

Listing 2

The limitation mentioned above can be overcome using the same means as demonstrated in the previous article in the series – building the type of the collection with an **`insert`** function, which will create a new type containing both the newly inserted types, and (by means of inheritance) the types of the original collection. The **`using`** keyword is used to inject the **`pair`** and **`test`** overloads of the base types into the scope of the newly created type. See Listing 3.

While the map of types presented here will suffice for the simple use case presented at the start, it does not fulfil some implied requirements. These are:

- Uniqueness of the selector elements
- Prohibit **`void`** as selector type
- Constraints on the type of selector or mapped types.

The first requirement sets the behaviour of our map when a duplicate selector is being used to insert a new element. As opposed to a set of types, we are not going to ignore duplicate elements quietly, but will fail the compilation instead – this is to protect against accidentally hiding existing elements in a map. Reusing a set of types discussed in the first part of the series will help with this requirement, as we can **`static_assert`** that the newly inserted selector type is not already present in the set of selectors.

The second requirement needs a short explanation. In the previous part, we defined **`void`** as a placeholder for an element when no element is available (an empty set, in other words). However, what should we do with an element which is not there, but something is mapped to it? That makes no sense, hence prohibition. Interestingly, if we employ **`test`** from the set of types to enforce the first constraint, it will automatically apply this one as well, because **`void`** is considered to be an element of every set (including an empty one). Do we also want to prohibit a **`void`** mapped element? Surprisingly, we do not have to – the map of types is perfectly capable of returning a mapped **`void`** type, although specific user semantics of a map instance might prohibit it.

This is where the last requirement comes in – it provides us with the semantics of ‘a map of something mapped to something else’ (as opposed to a ‘map of anything’). We are also going to extend the meaning of the constraint to optionally perform transformation of types – this enables the use case where, rather than prohibit e.g. reference types as selectors, the user would rather apply **`std::decay_t`** on them. In the previous part, we have already defined a similar constraint for a set of types. We could reuse such a check for a map of types, but we need two (for the selector type **`T`** and for the mapped type **`V`**). For example, see Listing 4.

Note, the **`PlainTypes`** constraint does not enforce a non-void selector type – as explained above, the **`test`** to prohibit duplicate selector types the constraints are not independent types – they have no other purpose than to allow the customisation of the semantics of our data structure

```
namespace map_impl {
  template<typename T> struct wrap {
    constexpr explicit wrap() = default;
    using type = T;
  };
  template<typename ...L> struct impl;
  template<> struct impl<> {
    constexpr explicit impl() = default;
    template <typename U>
    constexpr static void pair(wrap<U>) noexcept
    {}

    template <typename U>
  constexpr static bool test(wrap<U>) noexcept {
    return false; }
  };
  template<typename T, typename V, typename ...L>
      struct impl<T, V, L...> : impl<L...> {
    constexpr explicit impl() = default;

    using impl<L...>::pair;
    constexpr static auto pair(wrap<T>) noexcept {
      return wrap<V>(); }

    using impl<L...>::test;
    constexpr static bool test(wrap<T>) noexcept {
      return true; }
  };
}
template <typename ...L> class map {
  using impl = map_impl::impl<L...>;

public:
  constexpr map() = default;

  template <typename T, typename V> constexpr
      auto insert() const noexcept {
    using result = map<std::decay_t<T>,
        std::decay_t<V>, L...>;
      return result();
  }

  template <typename U>
  constexpr static bool test =
    impl::test(map_impl::wrap<U>());

  template <typename U>
  using type = typename
    decltype(impl::pair(map_impl::wrap<U>()))
    ::type;
};
```

Listing 3

```
template <typename T> struct PlainTypes {
  static_assert(std::is_same_v<T,
std::decay_t<T>>);
  static_assert(not std::is_pointer_v<T>);
  using type = T;
};

template <typename T> struct PlainNotVoid {
  static_assert(std::is_same_v<T,
    std::decay_t<T>>);
  static_assert(not std::is_void_v<T>);
  using type = T;
};

int main() {
  constexpr static auto m1 = map<PlainTypes,
     PlainNotVoid> {}
  .insert<Fuz, Fuzzer>();
```

Listing 4

inside the implementation of the map will perform this role. On the other hand, the check for non-void mapped type is implemented in the **`PlainNotVoid`** constraint. This is because (as discussed above) this constraint belongs to the domain where the map is used, rather than its inherent limitation. We are passing two parameters to our map, just like **`std::map`** requires two parameters. However, in our case the constraints are not independent types – they have no other purpose than to allow the customisation of the semantics of our data structure. This could be a good reason to consider passing a set of both constraints as a single parameter, but we are not going to pursue this path.

Since we are going to reuse 'a set of types', we might as well expose it in place of the **`test`** meta-function. This avoids the duplication of the functionality of both data structures. Note that because of the dependency on set, Listing 5 requires the reader to copy a large part of Listing 6 from the Part 1 [Kozicki18].

The techniques presented here can be also applied to create a heterogeneous collection of values (as opposed to types), with compile time insertion and extraction of data, for elements supporting such operations (and runtime otherwise). Such collections will be the subject of the next article in the series. ■

## Reference

[Kozicki18] See Listing 6 from https://accu.org/index.php/journals/2531, from the top until the declaration of 'struct Baz'.

```
// copy the definition of set in Listing 6 from
// https://accu.org/index.php/journals/2531
// to here
namespace map_impl {
  template<typename T> struct wrap {
    constexpr explicit wrap() = default;
    using type = T;
  };

  template<template <typename> typename CheckT,
    template <typename> typename CheckV,
    typename ...L> struct impl;
  template<template <typename> typename CheckT,
      template <typename> typename CheckV> struct
      impl<CheckT, CheckV> {
    using selectors = set<CheckT>;
    constexpr explicit impl() = default;

    constexpr static void pair() noexcept;
  };

  template<template <typename> typename CheckT,
    template <typename> typename CheckV,
    typename T, typename V, typename ...L>
  struct impl<CheckT, CheckV, T, V, L...>
  : impl<CheckT, CheckV, L...> {
    using check = typename CheckV<V>::type;
    using base = impl<CheckT, CheckV, L...>;
    using selectors =
      typename base::selectors::template
insert<T>;
    static_assert(not base::selectors
      ::template test<T>);
    constexpr explicit impl() = default;

    using base::pair;
    constexpr static auto pair(wrap<T>) noexcept {
       return wrap<check>(); }
  };
}

template <template <typename> typename CheckT,
    template <typename> typename CheckV,
    typename ...L> class map {
  using impl = map_impl::impl<CheckT, CheckV,
    L...>;

public:
  constexpr map() = default;

  template <typename T, typename V> constexpr auto
      insert() const noexcept {
    using result = map<CheckT, CheckV, T, V,
      L...>;
    return result();
  }
```

Listing 5

```
  using set = typename impl::selectors;
  template <typename U>
  using type = typename
     decltype(impl::pair(map_impl::wrap<U>()))
     ::type;
};

struct IFuzzer {
  virtual ~IFuzzer() = default;
  virtual std::string fuzz(std::string) const = 0;
};

struct Fuz {};

struct Foo {
  template <typename Map> Foo(Map) {
    if constexpr ((bool)Map::set::template
        test<Fuz>) {
      fuzz = std::make_unique<typename
        Map::template type<Fuz>>();
    }
  }

  std::unique_ptr<IFuzzer> fuzz;
};

struct Fuzzer : IFuzzer {
  std::string fuzz(std::string n) const override {
    return n + "-fuzz!"; }
};

template <typename T> struct PlainTypes {
  static_assert(std::is_same_v<T,
    std::decay_t<T>>);
  static_assert(not std::is_pointer_v<T>);
  using type = T;
};

template <typename T> struct PlainNotVoid {
  static_assert(std::is_same_v<T,
    std::decay_t<T>>);
  static_assert(not std::is_void_v<T>);
  using type = T;
};

int main() {
  constexpr static auto m1 = map<PlainTypes,
    PlainNotVoid> {}
  .insert<Fuz, Fuzzer>();

  const Foo foo {m1};
  if (foo.fuzz)
    std::cout << foo.fuzz->fuzz("Hello")
       << std::endl;
  else
    std::cout << "Sad Panda" << std::endl;
}
```

Listing 5 (cont'd)

## "The magazines"

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

## "The conferences"

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

## "The community"

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

## "The online forums"

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

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

**ACCU | JOIN: IN**

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

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

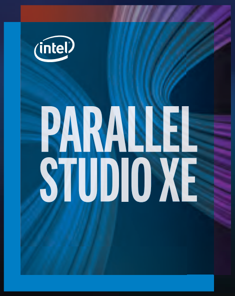
Join a community of people who care about code. Join the ACCU.

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

Design: Pete Goodliffe

Intel Software

# CODE MAXIMIZED



from £510

#HighPerformance

Develop high performance parallel applications from enterprise to cloud, and HPC to AI using Intel® Parallel Studio XE. Deliver fast, scalable and reliable, parallel code.

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

To find out more about Intel products please contact us:
020 8733 7101 | sales@qbs.co.uk | www.qbssoftware.com/parallelstudio