Live and Let Die
Martin Janzen reminds us how important destructors are and when to be mindful of their limitations.

Enodo, Divide, et Impera
Lucian Radu Teodorescu shows us how to divide and conquer difficult problems.

C# v12 Primary Constructors
Steve Love investigates how primary constructors fit into the language ecosystem.

Drive Musings on Linux
Ian Bruntlett muses on various approaches that can help with large files on Linux.

Afterwood
Chris Oldwood considers metaphors as inspiration for naming in code.
Run by programmers for programmers, join ACCU to improve your coding skills

- A worldwide non-profit organisation
- Journals published alternate months:
  - CVu in January, March, May, July, September and November
  - Overload in February, April, June, August, October and December
- Annual conference
- Local groups run by members

Join now!
Visit the website

professionalism in programming

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

4  Enodo, Divide et Impera
Lucian Radu Teodorescu shows us how to divide and conquer difficult problems.

10  Live and Let Die
Martin Janzen reminds us how important destructors are and when to be mindful of their limitations.

14  C# v12 Primary Constructors
Steve Love investigates how primary constructors fit into the language ecosystem.

17  Drive Musings on Linux
Ian Bruntlett muses on various approaches that can help with large files on Linux.

20  Afterwood
Chris Oldwood considers metaphors as inspiration for naming in code.
Frozen or Buffering?
Sometimes things grind to a halt.
Frances Buontempo reminds us we cannot be productive every minute of the day and that downtime is important

“...”

My contract has come to an end, and I haven’t lined anything else up. I’m in the privileged position of having some savings and my husband has a job, so the lack of income on my part hopefully won’t be a problem for now. This will give me a chance to catch up on several tasks, which will be useful. I might even get a chance to do something different once in a while, like go for a long walk. My head is spinning with all the incomplete jobs and half-baked ideas I’ve started on, but not finished. Of course, this means I haven’t got round to writing an editorial so, yet again, I apologise.

I had such plans for my first day off, but ended up spending hours watching a new phone trying to transfer everything from my old phone, so as usual I spent hours staring at a screen. I then failed to appear on a pod-cast, since a host couldn’t make it. By the end of the day, I felt as though I’d done nothing, which is an all too common state of affairs. In the time sitting around waiting, I did manage to start thinking about how to organise my time and what to prioritise. The day seemed like a buffering day, both as a space between the old and the new, and as a place to line plans up for the future. Sometimes, stopping and seemingly doing nothing is actually much more important than randomly doing a variety of things just because they spring to mind. Have you ever gone in one room to do one thing, got distracted and done something completely different? Almost certainly. Or opened a file in a code base to add a log line, and refactored some horror you found without adding what was needed? Then spent an hour or more waiting for the new log line to appear before realizing your oversight? Easily done.

Rather than running your brain at 100% CPU usage, running around doing 100s of things you didn’t mean to do and forgetting the important tasks, you might go into a room and freeze instead, having forgotten why you went there in the first place. Either way, the important work doesn’t get done, so the outcome is the same. One looks like frantic buffering, while the other appears frozen. Nothing happening and lots of things happening can have the same outcome. In fact, sometimes, they look very similar. How can you tell if a program is really doing something? It may show high CPU utilization, but that can happen if code is stuck in a loop, calculating the same thing over and over. In a previous role, I had to be on overnight support from time to time. Our team ran various finance simulations overnight which needed to be ready for 9 a.m. the following morning. It was often touch and go as to whether we’d be on time or not. One job in particular often took a long time, and I was called in the middle of the night and asked to bounce the job because it had got stuck. How could we tell it was stuck? It was hammering the CPU, but we couldn’t see any logs, so what, if anything, was it doing? I bowed to pressure, and restarted the job. It got to the same point and still didn’t appear to be doing anything. This time, when the inevitable call came, I refused to restart it, and it did finish with a couple of minutes to spare. The job had not frozen. It was lining up lots of calculations and they took a long time. Unfortunately, there was no way to tell from the outside whether it was doing anything or not. A spot of judicious logging in the right places helped in the long run, as well as optimizing the code where possible.

Many situations have no visible progress, not just an overnight job appearing to be stuck. The same can happen on software projects. I’ve picked up a few Jiras that have spilled over several sprints. Sometimes, the person who wrote the task did a code review and announced “One more thing” We called him Columbo, for reasons that are obvious if you’ve ever watched the show [Columbo]. Other times, far more foundational changes were required so every time you think you’re done, you have to update, merge, retest, fix, rinse and repeat. Like running on the spot for several, ahem, sprints. Often, abandoning the task and finding a way to make the change in smaller steps is better, but we tend to get determined or bullied into completing something once we’ve started. Making fundamental changes can take a long time, and there may be no visible changes for a while. That doesn’t mean no progress has been made: we just can’t see the internal improvements from the front end. I guess some kind of code metrics can help here, provided the non-coders on a team understand what they mean and why they are important. Recently, McKinsey produced a report about measuring developer productivity [McKinsey23] McKinsey are a large management consultancy who regularly publish reports on a variety of subjects, which tend to carry weight and influence many companies worldwide. The report starts by pointing out:

There is no denying that measuring developer productivity is difficult. Other functions can be measured reasonably well, some even with just a single metric; whereas in software development, the link between inputs and outputs is considerably less clear.

They mention Google’s DevOps Research and Assessment (DORA) metrics [Google], along with SPACE metrics (Satisfaction and well-being, Performance, Activity, Communication and collaboration, and Efficiency and flow – which is a bit of a mouthful!) [Forsgren21]. Their report builds and extends on these ideas, but doesn’t really say anything I find useful. I have seen several responses to the report. For example, Kent Beck told LinkedIn the report is naïve, but found McKinsey thinking their intended market want a report like this is interesting in and of itself [Beck]. Gergely Orosz and Kent Beck have written a more detailed analysis [Orosz23], questioning some of the measures such as effort. Now, I go to the gym, and have to put in a huge effort to curl 7kg dumb-bells. I watch other people using 10kg weights, and making it look effortless. Does than make me more productive? No, I’m just not as good
as them. Maybe as I keep practising, I’ll get better and be able to lift more.
In the meantime, there won’t be any visible progress.

Programming isn’t the only place where it’s hard to measure progress.
If you’ve ever had work done on your house, you will know this.
Recently, a small part of a boundary wall fell over into the neighbour’s
garden. We found a builder, and he was happy to reuse the bricks, after
cleaning them up. He hadn’t factored in how long that would take. It
turns out ivy can be very destructive and grow through almost anything.
It required a huge effort to untangle the mess, and then considerably more
digging than envisaged to get the roots out so a new foundation could
be laid. For many days, it looked as though nothing more had happened
than a pile of bricks had moved from one spot to another. The builder
couldn’t be precise about how much longer would be needed, which is
understandable. He’s never rebuilt this wall before, so couldn’t be sure.
Now he’s spent time getting the ground cleared for firm foundations,
he’s making visible progress. Writing code can be like that too. If you’ve
ever coded a specific algorithm or solved a particular problem before,
you can’t tell how long it will take. You can say what you’re up to at the
moment and what other tasks will need doing, but you won’t know the
unknown unknowns. They are, after all, unknown. Furthermore, progress
is often non-linear. If you break work down into, say, five chunks, and the
first takes all of Monday, that is no guarantee you’ll be done by the end of
Friday. As for clearing the ground to build firm foundations, how many
of us have had to justify “no visible progress” and explain “tech debt” on
more than one occasion?

The wall is nearly finished now, so our neighbour will be able to let
their dog down the end of the garden again. Without the barrier, he was
concerned the dog could stray into our garden, and I’m sure our cat might
have opinions about that. The dog could probably jump over the wall
if it wanted to, but the boundary seems to form a physiological barrier
too. For the dog. The cat does what he wants, including wandering into
neighbours’ gardens and sitting on my seat. When the wall is rebuilt, I
will try to clear up more of the ivy round the garden. Having a buffer
zone between the wall and the plants to avoid a repeat of the collapse
might be a good idea. Buffer zones give space to see what’s going on.
I’ve tracked buffer overruns and similar by adding variables to the stack
to pinpoint where my code was doing something daft. These “canary
variables” were a simple but effective approach. There are better tools
available nowadays, for example using The /GS flag in Visual Studio to
enable buffer security checks [Microsoft21], and OWASP gives details on
problems to watch out for and other tools that might help [OWASP].

The word ‘buffer’ means anything to reduce shock or damage due to
contact, something that cushions against shock of fluctuations in finance
or more generally a protective barrier, according to Merriam-Webster
[Merriam-Webster]. Adding a buffer to protect a buffer seems recursive,
which is a different problem. Of course, a software memory buffer is not
about cushioning or protection, but rather a space to put things. We use
a buffer to store user input or other temporary data. We also talk about
a webpage or network traffic buffering. Data is stacked up, so it can be
accessed quickly or stop lag on the receiving end. This buffering should
be a good thing, but we also complain if a video stream or similar is
buffering, meaning it has frozen waiting for the buffer to fill up. Sitting
watching spinning wheels or stalled progress bars is very annoying.
Whether we need to bounce a router, restart a job or just wait depends.
Some things take time and we need to learn to be patient.

I’ve ground to a halt several times while trying to write this. My mind
keeps wandering to my ever growing to-do list, while also day-dreaming
about what I might be able to do with the spare time I now have. We all
need time out occasionally, to give ourselves time to allow things settle.
Downtime is important. It may look like inactivity or stagnation from the
outside, but buffering moments can lead to innovative sparks or changes
of direction. This has got to be an improvement on keeping digging or
being stuck in a rut. Let’s try to measure our ‘productivity’ in a positive
way, without ending up striving for 100% CPU usage but no constructive
outcomes. Failing that, certainly consider adding traces or logging to see
what is going on. And try to make them more informative than Terry Pratchett’s computer Hex
unhelpfully pronouncing “+++?????++ Out of Cheese
Error. Redo From Start.” [Discworld]

References


[Forsgren21] Nicole Forsgren, Margaret-Anne Storey, Chandra Maddilla, Thomas Zimmermann, Brian Houck and Jenna Butler ‘The SPACE of Developer Productivity’, acmqueue, Volume 19 Issue 1, 5 March 2021, available at: https://queue.acm.org/detail.cfm?id=3454124


October 2023 | Overload | 3
Enodo, Divide et Impera
How do you untie the knotty problem of complexity?
Lucian Radu Teodorescu shows us how to divide and conquer difficult problems.

“The life which is unexamined is not worth living” says Socrates, according to Plato [Apology]. This should apply both to personal life, but also to professional life. Thus, it needs to be our duty to analyse various aspects of Software Engineering. And, what is more important to analyse than the fundamentals of our discipline?

This article aims to analyse one of the most useful techniques in software engineering: the divide et impera (Divide and Conquer) technique. And maybe the most useful one.

We define the divide et impera method as a way of breaking up a problem into smaller parts and fixing those smaller parts. This applies to recursive functions (where the phrase divide et impera is most often used), but it will also apply to the decomposition of problems. At some point, we will also discuss using abstraction as a way of applying divide et impera. Finally, we will show how to use this technique in our daily engineering activities that are not strictly related to coding.

Definition, generalisations, and distinctions
In this article, we call divide et impera a method of approaching problems that has the following characteristics:

- breaking the problem into sub-problems
- solving the sub-problems independently of each other
- occasionally, an answer to a sub-problem may render solving the rest of the sub-problems unnecessary
- sporadically, a small amount of information passes one sub-problem to another
- combining the results of the sub-problem solutions to form the solution to the initial problem

We take a relaxed view on what a problem can be. In our exposition, it may mean a software algorithm, as expressed in code (e.g., quick-sort algorithm), or maybe the actions that a software engineer needs to do to complete a task (e.g., fixing a defect may imply analysis of the defect, discussions with other engineers, and perhaps closing the defect as ‘by design’).

Please note that this definition is more general than what’s typically understood by the Divide and conquer algorithm [Wikipedia-1], which requires the sub-problems to be of the same nature, similar to the original problem, but simpler. In our case, we allow the sub-problems to be widely different from each other (and different from the original problem, too).

Our definition overlaps in a greater measure with the decomposition technique [Wikipedia-2]. Decomposition techniques tend to be associated with breaking down software systems. In our definition, we incorporate that, but we will also apply the same technique to processes outside the code (debugging, programming methodologies, etc.).

Truth be told, the name decomposition fits better than divide et impera; I’m choosing the latter name because, besides decomposition (i.e., divide), it also highlights the idea of conquering (i.e., impera) the sub-problem. The main point of applying this method is to simplify the original problem, and we are doing this by making the sub-problems much more approachable, easy to conquer.

Simplifying complexity
The typical model for analysing the complexity of a system is to look at the interaction within its parts. If the system has \( n \) parts, then it may have up to \( n(n-1)/2 \) interactions between these parts. We may associate complexity with each of the parts, and with each interaction between these parts. If the complexity associated with a part is \( c_0 \) and the complexity associated with an interaction is \( c_i \), then the total complexity of the system is:

\[
C = c_0 \cdot n + c_i \cdot n \cdot (n-1) / 2
\]

Its magnitude would be \( O(C) = n^2 \).

One can easily remark here that the bulk of the complexity arises from the interaction between parts, not from the parts themselves. Thus, limiting the interaction between parts would be beneficial.

Let’s take an example. Let’s consider a system of \( n = 10 \) parts, and complexity values \( c_0 = c_i = 1 \). The total complexity of the system, if we had interactions between all parts, would be \( C = 10 \times 45 = 55 \). Now, let’s assume we could group the parts into two groups, such that the parts from one group and the parts from the other don’t interact with each other, just one interaction between the groups. If we do that, the complexity of each group would be \( C = 15 \), and the total complexity of our system \( C’ = 15 + 15 + 1 = 31 \). Thus, this grouping does a reduction in terms of complexity of 1.77 times.

In our method, we divide a problem into sub-problems not to create more parts that interact with the rest of the parts, but to isolate the interactions; this is why we require the problems to be able to be solved independently of each other.

The two main benefits of this method are that it:

- simplifies reasoning about the problem
- makes the solving of the problem easier (performance improvements)

Let’s start by analysing the second benefit, and let’s use examples to drive that analysis.

Algorithmic examples
Merge sort
The idea of the merge-sort algorithm is relatively simple:

- divide the initial range of values into two halves;
- sort the two halves independently,
The bulk of the complexity arises from the interaction between parts, not from the parts themselves. Thus, limiting the interaction between parts would be beneficial.

Listing 1

```cpp
template <typename BidirIt>
void mergeSort(BidirIt begin, BidirIt end) {
    if (begin < end) {
        // break down the problem
        BidirIt mid = begin + (end-begin)/2;
        // solve sub-problems independently
        mergeSort(begin, mid);
        mergeSort(mid, end);
        // combine the results
        std::inplace_merge(begin, mid, mid, end);
    }
}
```

Listing 2

```cpp
template <typename BidirIt>
void quickSort(BidirIt begin, BidirIt end) {
    if (begin < end) {
        // break down the problem
        BidirIt mid = pivot(begin, end);
        partitionElements(begin, mid, end);
        // solve sub-problems independently
        quickSort(begin, mid);
        quickSort(mid, end);
        // nothing to combine
    }
}
```

The sorting of the halves can be done recursively;
finally, combine the two sorted sub-ranges into the resulting sorted range.

This would be implemented similar to the code in Listing 1.

The complexity of the algorithm is $O(n \log n)$ (assuming we always have enough memory to perform the merging). The breaking down of the problem into sub-problems is trivial: just compute the middle of the input range; this will divide the initial range into two sub-ranges. The two sub-ranges can be sorted independently of each other: to sort the left sub-range, we don’t need any information from the right sub-range. Each sort will be done in $O(n \log n)$.

Merging is slightly more complex, but this is double in $O(n)$ (if enough memory is available). If we ignore the fact that the merging algorithm can work in-place, the fundamental idea can be expressed as:

- while both input arrays have remaining values:
  - compare current elements of both input arrays;
  - copy the smaller element into the output array;
  - advance the current pointer for the array containing the smallest element;
  - copy to the output array the remaining elements.

It is important for us to analyse the possible interactions between the elements of the arrays being merged. Because the arrays are sorted, an element is related to its adjacent elements in the same array, but it doesn’t need to be compared/considered with any other elements. When merging, we compare elements from one array to elements from the other array. Again, here we only look at a limited set of interactions; when placing the left-side element $l_{mn}$ into the output array, we only look at the elements from the right-side that have values within the range $l_{mn}, l_{mn}$, where $l_{mn}$ is the previous element in the left-side array. For the entire merge process, we have $O(n)$ in interactions, if the output array contains $n$ elements.

We need to contrast these complexities with a naive approach (e.g., insertion sort) to sorting, in which we compare each element with all the other elements, which has a complexity of $O(n^2)$. We can see how the merge sort algorithm has lower complexity than a naive sorting algorithm.

**Quicksort**

Another good example for showcasing the *divide et impera* method is the quicksort algorithm. See Listing 2 for a possible implementation of the core algorithm. It has the same complexity of $O(n \log n)$ as merge sort, but it achieves it differently. Instead of combining the sorted sub-ranges, in quicksort we ensure that all elements on the left side are smaller than the elements on the right side. Doing this, assures us that there is nothing we need to do after we’ve sorted the sub-ranges.

For this algorithm, the choice of the middle element (called pivot) is important, as it may change the complexity of the algorithm to be $O(n^2)$. For this article, we assume we are using a scheme that guarantees the $O(n \log n)$ complexity.

**Binary search**

We need to analyse another example to highlight an important point of our *divide et impera* method: binary search; for a given sorted sequence of value, check if a given value is in the sequence. Please see Listing 3 (overleaf) for a possible implementation.

Here, we are doing minimal work to break down the problem, similar to merge sort. We are doing nothing to combine the results of the sub-problems, similar to quicksort. What is remarkable in this example is that we are short-circuiting the solving of the sub-problems. If we determine that the solution must be in the first half of the sequence, it doesn’t make sense for us to do anything for the second half.

The complexity of the binary search algorithm is $O(\log n)$, better than the complexity of $O(n)$ of doing a linear search.

We have briefly covered, with examples, the important part of the *divide et impera* method, the way we defined it: there is a part in which we split the problem, there is a part in which we solve the sub-problems, and there is an optional part in which we combine the results. We’ve also looked

October 2023 | Overload | 5
Reasoning about programs, where one needs to keep track of all parts of the software, quickly grows to be impossible with the increase in the size of the program.

The results of the first one, but doesn’t require any understanding of the following instructions. And so on.

Dijkstra also considers blocks of code with one entry point and one exit point as a complex instruction. Thus, if, switch and while blocks can also be instructions. Furthermore, more importantly, we can make use of abstraction (i.e., making function calls) just like primitive instructions.

To the extent that the instructions are independent of each other, we apply divide et impera. While we have said that our method allows information to flow from one sub-problem to another, the amount of information passed around needs to be small for the payoffs of the methods to show.

Listing 4 shows a possible implementation of std::remove, trying to showcase the sequencing as a way to reduce cognitive load. In the first step in the algorithm, we find the first occurrence of the given value. The second step of the algorithm, if there is a match, is to shift left all subsequent elements by one position. One can reason about the first sub-problem completely independently from reasoning about the second sub-problem.

The mathematical induction mental aid that Dijkstra mentions is helping us cope with loops. While one can find similarities between this reasoning and divide et impera, the two are somehow distant, so we won’t cover their connection here.

The last mental aid that Dijkstra identified, perhaps the most important one, is abstraction. Abstraction stays at the core of what Dijkstra considers to be structured programming. The main idea is that abstraction allows the programmer to lose sight of (abstract out) unimportant details, while focusing only on important parts.

For example, for the std::remove function that we just discussed, the implementation details are irrelevant for someone that just wants to use the function according to the promised contract. There are many ways this function can be implemented, but it doesn’t matter that much if they all satisfy the same contract. Thus, our cognitive load for using this function is lowered.

```cpp
template <typename ForwardIt, typename T>
ForwardIt remove(ForwardIt first, ForwardIt last, const T& value) {
  if (first != last) {
    // subproblem 1: find the (first) occurrence of our value
    ForwardIt first = std::find(first, last, value);
    // subproblem 2: shift left one position // subsequent elements
    if (first != last) {
      for (auto it = first; ++it != last; )
        if (!(*it == value))
          *first++ = std::move(*it);
    }
  }
  return first;
}
```

Listing 4

at a trivial example in which doing work for sub-problems may be short-circuited.

It is important to mention that the splitting and the combining parts also have costs associated with them. However, it seems that a proper division of the problem into sub-problems would compensate for these costs.

We looked at how divide et impera can help improve the efficiency of software algorithms. We will look at applying the method to reasoning about software.

Reasoning with divide et impera

In the landmark Structured Programming book [Dahl72], Dijkstra argues that one of the major sources of difficulties in software engineering is our human inability to capture large programs in our head.

Reasoning about programs, where one needs to keep track of all parts of the software, quickly grows to be impossible with the increase in the size of the program. Like we mentioned above, for a program with n parts (e.g., instructions) the complexity that one needs to maintain in their mind is O(n²). And, studies show that, on average, our mind can keep track of 7 independent things at one time [Miller56].

To alleviate this problem, Dijkstra advocated that we must add structure into our programs, such as the interaction between different parts of the software is reduced. He argues that there are three main mental aids that allow us to better understand a problem: enumeration, mathematical induction, and abstraction. To some extent, they all overlap with divide et impera.

The idea behind enumerative reasoning is that it is (somehow) easy for humans to reason linearly on a sequence of instructions, especially if the sequence of instructions aligns with the execution of the program. Looking at the first instruction, one doesn’t need to care about the rest of them. Looking at the second instruction, it may require understanding of

```cpp
template <typename BidirIt, typename T>
bool binarySearch(BidirIt begin, BidirIt end, const T& value) {
  if (begin < end) {
    // break down the problem
    BidirIt mid = begin + (end-begin)/2;
    // solve the sub-problems independently,
    // with short-circuit
    if (value == *mid)
      return true;
    else if (value < *mid)
      return binarySearch(begin, mid, value);
    else
      return binarySearch(mid, end, value);
    // nothing to combine
  }
  else
    return false;
}
```

Listing 3

the sequence of instructions aligns with the execution of the program. But, we won’t cover their connection here.

The last mental aid that Dijkstra identified, perhaps the most important one, is abstraction. Abstraction stays at the core of what Dijkstra considers to be structured programming. The main idea is that abstraction allows the programmer to lose sight of (abstract out) unimportant details, while focusing only on important parts.

For example, for the std::remove function that we just discussed, the implementation details are irrelevant for someone that just wants to use the function according to the promised contract. There are many ways this function can be implemented, but it doesn’t matter that much if they all satisfy the same contract. Thus, our cognitive load for using this function is lowered.

```cpp
template <typename BidirIt, typename T>
bool binarySearch(BidirIt begin, BidirIt end, const T& value) {
  if (begin < end) {
    // break down the problem
    BidirIt mid = begin + (end-begin)/2;
    // solve the sub-problems independently,
    // with short-circuit
    if (value == *mid)
      return true;
    else if (value < *mid)
      return binarySearch(begin, mid, value);
    else
      return binarySearch(mid, end, value);
    // nothing to combine
  }
  else
    return false;
}
```

Listing 3

as, the two are somehow distant, so we won’t cover their connection here.
Good software needs to be **sustainable software**. Software that starts to rot from its inception, software that is hard to fit in your brain, is the opposite of what we want.

The reader is probably starting to realise where I’m going with this: the simple use of an abstraction is an application of divide et impera, as it separates the bigger problem into two sub-problems: implementing the abstraction and using the abstraction. We are applying the method not necessarily to the code itself, but to our reasoning about the code.

In our definition of the method, we identified a prerequisite step that is breaking down the problem into sub-problems. This is the definition of the function contract: the type of the function (parameters and return type), constraints around the function, complexity guarantees, etc.

It’s not directly obvious whether there is a step that ‘combines the results of the sub-problems’ in the case of using function abstractions. One may argue that the simple realisation that calling the function will actually execute the body of the function is such a step. Yes, this is so fundamental that we don’t consciously think about this every time we use a function, but that doesn’t mean that this thought process isn’t followed at some point. I do feel that we should consider this realisation as part of the process.

Thinking about the use of abstractions in terms of *divide et impera* allows us to realise the benefits of using the abstractions. To take an example, let’s assume that we write a function once, and we use it n times; assuming the function doesn’t have bugs, and the contract of the function is well understood, we only have to reason about the implementation of that function once. Thus, the total cost of using the function would be $C_{\text{total}} = C_{\text{impl}} + n \cdot C_{\text{contract}}$, where $C_{\text{impl}}$ is the mental cost spent while implementing the function and $C_{\text{contract}}$ is the cost of understanding the function contract in order to use it. If $n > 1$ and $C_{\text{contract}} < C_{\text{impl}} \cdot (n-1)/n$, which is probably true for most of the functions, then we are reducing the cognitive load to use this abstraction.

The reader should note that this applies to all types of abstractions, not just functions. We can have similar arguments for classes, concepts, etc.

Dijkstra uses the mental aids to build a model of how programs should be structured, describing a process for constructing programs. This process is essentially *decomposition*. In one of the lengthier examples he gives, printing a table of the first thousand prime numbers, he starts from the top, and recursively divides the problem at hand into multiple subproblems and solving those problems. The key point here is that, when a problem is divided into sub-problems, the sub-problems can be worked on in isolation. This is precisely what we discussed under the term *divide et impera*.

What is interesting to notice in Dijkstra’s exposition is that we can easily decompose programs without necessarily needing abstractions. He starts by representing the problem with placeholders (English sentences), and through repeated refinements he gradually transforms all the placeholders into actual code.

I would recommend everyone to go through the decomposition exercise that Dijkstra presented [Dahl72]. It’s a prime example of how to build a program, and how to reason about it.

---

**Practical tips**

**Techniques for writing sustainable code**

Good software needs to be sustainable software. Software that starts to rot from its inception, software that is hard to fit in your brain, is the opposite of what we want. But how do we write sustainable software?

The book *Code that Fits in Your Head* by Mark Seemann [Seemann21] tries to answer this question by providing numerous tips on writing sustainable code. Not surprisingly, many of these tips are related to our *divide et impera* method.

In one of the most important tips in the book, Mark offers a direct invitation to divide large problems into smaller ones:

> No more than seven things should be going on in a single piece of code.

There are many techniques that Mark explains in his book; it would be too long to analyse them here, but at least we can mention some of them:

- Fractal architecture
- The 80/24 rule (write code that fits into an 80/24 screen terminal)
- ‘Arrange, act, assert’ pattern for writing tests
- Command Query Separation
- Red Green Refactor (used in TDD)
- Slicing (work in small increments)
- Strangler
- Etc.

**Incremental development**

Some techniques mentioned above may seem a bit farther away from our definition of *divide et impera*. They would fit more into the *incremental development* paradigm. What would be the connection between incremental development and *divide et impera*?

We shall argue that incremental development is a form of *divide et impera*. Let us analyse.

Let’s say we have a complex project $P$ and we approach it incrementally, and we will end up solving this problem by performing the increments $I_1, I_2, \ldots, I_n$. Assuming that the project was successfully completed, we have $P = I_1, I_2, \ldots, I_n$. A naive application of *divide et impera* would suggest that we should break down $P$ into $P_1$ and $P_2$, where $P_1 = I_1, I_2, \ldots, I_{\text{mid}}$ and $P_2 = I_{\text{mid}+1}, I_{\text{mid}+2}, \ldots, I_n$ (assuming the increments are roughly the same size).

The problem is that, in practice, we don’t properly know the increments upfront. To perform this division, we need to precisely know their number. And, for most software engineering problems, we simply don’t.

If we intuitively try to divide the big problem into two halves, and jump ahead to solving the first half, we probably end up with a mess. The first half is also very complex, so it accumulates delays, technical debt and
If we have a complex problem, attacking it directly tends to lead to failure, or at least consumes a large amount of energy/time.

poor quality. I’ve seen many projects go down this route. (That does not imply that we shouldn’t be combining this type of division with incremental development; I can argue that this combination would be the best approach.)

Using incremental development, we perform a division that is very uneven. We divide $P$ into two parts: $I_1$ and everything else. We need to have clean boundaries for what $I_1$ means so that we know that we solve it correctly. After solving $I_1$ we apply the same process and divide the rest into $I_2$ and what comes after. We apply this repeatedly until we finish all the iterations and solve the initial problem.

This is equivalent to an unbalanced binary tree, as shown in Figure 1.

Let’s look at the elements on our definition of divide et impera and see how they fit incremental development. Breaking down the problem into sub-problems consists of identifying the first increment that can be done. It is important at this step to have a clear definition of ‘done’, and ensure that we don’t redo the effort for $I_1$ in the next sub-problem; that is, the two sub-problems must be independent. The sub-problems are the first increment and the rest of the problem. The combining step is typically non-existent.

Bisection

Git bisect is a great example of divide et impera. If we have a bug that appeared between two different releases, and there are multiple commits in between, we can use this method to narrow down the search. It is essentially an implementation of the binary search algorithm discussed above.

The important aspect of this method is finding a good way to expose the bug (i.e., have a definite test). This test needs to be accurate. If the test indicates that the bug is present, it means that it was introduced in a prior commit; if the test indicates that the bug is not present, it might have been introduced in a later commit.

Debugging tips

Let’s say we have a bug in a large code that is hard to understand, and we need to find the source of this bug.

One method of approaching this problem is looking at different pieces of the codebase and trying to understand/debug whether the bug comes from there. This technique can be useful for experts as they probably have good intuition of what could be the possible causes for the bug. But, in general, this can be an expensive method of searching for the bug. Especially if we don’t have a good test for the bug, the exploration space needed to detect the bug is massive (considering that we often look at the same part of the code multiple times). The method is related to the bogosort algorithm that tries different permutations until it finds one that is good.

Another method of analysing is to linearly look at the entire flow. That is typically expensive for complicated flows with many steps. We need to clearly understand all the expected outputs for all the steps, typically for every instruction in the codebase.

A better approach is to apply divide et impera (if possible). The idea is to find a point, ideally around the middle of the flow, where we can relatively easily check to see whether we are behaving correctly or not. This divides the search space in two parts: what happens before that point, and what happens after. In other words, it’s a manual bisection.

The test points are best to be chosen to be easily testable. For example, at the end of a quick-sort, it’s easy to see whether the sorting contract is met or not; on the other hand, if one adds a breakpoint somewhere in the partition function, sometime during the execution of the algorithm, it may be harder to understand whether the algorithm is working as expected.

If we have a client-server application, then checking the messages sent between the two would be a good start. If the requests don’t follow the agreed protocol, it’s probably a client problem. On the other hand, if the responses don’t follow the agreed protocol, it’s most likely a server issue.

For complex flows, I often used a form of printf debugging to divide the search space and narrow down on the problem. I carefully choose the test points and dump the information available at those points, and then I can reason whether the outputs are expected or not.

To be honest, I rarely use a proper debugger. It tends to create a linear flow, and it doesn’t necessarily show the actual values that are important for the entire flow. In the end, I find using a debugger slows me down when investigating issues.

Solving complex problems: first breakdown

If we have a complex problem, attacking it directly tends to lead to failure, or at least consumes a large amount of energy/time. One of the chief difficulties of a complex problem is actually understanding the
problem in its entirely and its boundaries. If we clearly understand the problem, it’s much easier to provide a solution.

Thus, solving a complex problem should have 2 steps:

- Understand the problem and its limits
- Actually solve the problem

I often say that understanding the problem is the harder part of the two. This may sound commonsense, but unfortunately, I’ve seen too many cases in which we start implementing a solution before clearly understanding the problem we are trying to solve.

### Solving complex problems: prototypes and mathematical models

After the problem is clearly defined, we can actually start solving it. If the problem is complex, there may be multiple possible alternatives to solving it. Choosing the right alternative to use is hard. If we select a bad alternative, we either don’t solve the problem (e.g., the solution doesn’t have all the desired quality attributes), or we solve it with large costs. Thus, it makes sense to spend some more time at the beginning to figure out the shape of the solution.

We can spend this investigation time doing prototypes or mathematical models for the problem. Oftentimes, just performing some back-of-the-envelope calculations proves extremely valuable; they can determine whether a particular solution fits the problem or not. If the initial effort is small compared to the overall costs of implementing a solution, then that effort is well spent.

What we are arguing is that we divide the implementation part into two sub-problems: figure out the right approach and actually implement the solution.

To be honest, this idea prompted me to write this article. At the beginning of the day on which I started to write this article, I had another topic in mind to write about. But then, during the day, I had a discussion with a fellow engineer who wanted to implement a better version of a task scheduler. He tried to explain to me the model that he wanted to implement and expressed the desire to start coding soon; I started to suggest that he should first build a mathematical model of the solution, to analyse what would be the consequences of implementing that algorithm; more specifically, we wanted to understand if the new algorithm can be better than the existing algorithm. Going meta, like I often do, I started to argue that we should use divide et impera to attack these types of programs. And this was the seed for the article.

Coming back to complex problems, if there are multiple solutions that may or may not work, we should spend time upfront screening for possible solutions. If we put this together with what we argued in the previous section, solving complex problems often requires solving three different sub-problems:

- Properly defining the problem
- Do a screening of potential solutions, and choose the right solution
- Implement the chosen solution

### Conclusion

We argued, in this article, that one of the most useful methods in software engineering is *divide et impera*. As there are multiple possible meanings for this, we tried at the beginning of the article to formulate a definition of the method that would be general enough to fit other definitions. We positioned *divide et impera* as a general method that can be applied both to organising code, and to how we are working.

We argued that this method is one of the widely used methods for solving problems in our field. It overlaps with the recursive method used in algorithms, it overlaps with decomposition, it’s a method that stands behind several core ideas of structured programming, and it also stands behind several practices in software engineering.

During the course of the article, we provided a series of examples of how this method can be used. The examples cover things like algorithms, reasoning about code as long as examples on applying *divide et impera* in day-to-day work, from development methods to more concrete tips.

I’m always of the opinion that we should be constantly analysing important things around us. That is why a method like *divide et impera* is worth analysing, and that’s precisely what this article sets out to do. At the end, if I should name two tips for software engineers, those would be: constantly analyse and apply *divide et impera*. In other words, *Enodo, divide et impera.* □

### References


---

1 Analyse, divide and conquer
Live and Let Die
Resource lifetime management can be problematic. Martin Janzen reminds us how important destructors are and when to be mindful of their limitations.

Most experienced C++ programmers will agree that one of the best properties of our language is the ability to manage object lifecycles using constructors and destructors.

Bjarne Stroustrup [Stroustrup19] has described dtor/dtor pairs as one of C++’s most elegant features, giving us the ability to create clean types which tidy up after themselves, with predictable performance, minimal overhead, and no need for garbage collection.

In this year’s ACCU Conference Lightning Talks, Nico Josuttis singled out destructors as (spoiler alert!) “the most important C++ feature” [Josuttis23]; and Wiktor Klonowski told a sad tale of time wasted debugging a .NET program that kept running out of ports, a fate which could have been avoided by the use of dtors [Klonowski23].

At the same conference, as well as at the recent C++ On Sea, numerous speakers talked about C++ and safety, a subject that’s been very much in the news recently [NSA22], with C++ predictably receiving a lot of flak for the ease with which one can write code containing buffer overflows, memory leaks, and of course a rich and varied choice of ways to introduce undefined behaviour (UB).

License to Kill
In its favour, though, C++ also provides at least one way in which we can improve safety, and reliability, greatly, by use of the powerful RAII (Resource Acquisition is Initialisation) idiom: taking ownership of a resource in the ctor, then releasing it in the dtor.

If we ensure that all of our program’s resources are managed via RAII-based classes, it becomes fairly straightforward to avoid resource leaks [Core23]. Memory is freed automatically, mutexes unlocked, threads joined, database connections released, files and sockets closed, and so on.

Furthermore, this approach makes it much easier to write code which is exception-safe, because RAII-based resource management classes can ensure that every newly-acquired resource is released if a scope is exited because of a thrown exception.

In many cases we don’t even need to write the RAII code ourselves:

- Memory can be owned via a std::shared_ptr or std::unique_ptr.
- Mutex locks can be managed by std::lock_guard and its variants.
- A std::packaged_task or C++20 std::jthread can often eliminate the need to write a custom thread guard class, as in [Williams19].

Of course, all of this works because the C++ language promises us that the destructor will be called exactly once, when the lifetime of an object ends.

To review, this happens:

- at the end of a full expression, for temporary objects,
- at the end of a scope, for automatic (stack-based) objects, either normally or when the stack is unwound due to an exception,
- on thread exit, for thread-local objects,
- on program exit, for objects with static storage duration[1], or
- when the dtor is called directly, by using a delete expression or via a direct call when using placement new, or via an allocator’s destroy() function. (In most cases, though, direct calls should be reserved for RAII classes and library code.)

So, job done; our resource management headaches are solved. What can possibly go wrong?

No Time to Die
Unfortunately, destructors are not always called exactly once.

First, let’s look at some situations in which an object’s dtor may not be called at all.

Sometimes this may be due to factors which are entirely beyond our control, causing our program to terminate without any warning or recourse:

- Power failures and hardware faults can put a stop to things.
- Finite resources such as memory can become exhausted, even if we are managing them correctly.
- In a POSIX-like environment, an uncaught signal may terminate our process immediately. SIGKILL(-9), in particular, cannot be caught.
- The last two may occur together – as when Linux decides that the system is dangerously low on memory and its out-of-memory killer starts getting rid of particularly greedy processes.

In other cases, it may be due to a software bug:

- When not using RAII, it’s easy to forget to delete an object.
- Even if a resource manager such as std::shared_ptr is used, it is possible to create two or more objects which hold shared pointers to each other, creating a cyclic graph which prevents any of the objects from being destroyed automatically.

An uncaught exception will cause a call to std::terminate() and, by default, to std::abort(). (More on that later.)

1 For this article I’ve assumed a POSIX-like environment, simply because that is what I know. Windows developers should have little difficulty finding equivalents in their own world.
most of us will have run into shutdown errors, in which a program works perfectly well until it is time to stop, but then comes to an undignified end

UB. As the name suggests, pretty much anything can happen next. Then, we have the halting problem.

No, not that one [Turing37]. I’m concerned here with the way in which we exit from a C++ program.

For many programs, such as command-line utilities, this is obvious: simply exit from the main() function when finished, either by returning an exit code or just falling off the end.

However, other programs are meant to run for indeterminate periods of time. Software with a graphical user interface is normally started by its user, and runs until asked to quit. Server-based software, from system daemons to web servers to trading systems, is usually started and stopped by a controller such as init or systemd, or by some sort of task manager or framework. For these cases, the C++ Standard Library provides a number of functions that will stop the current process, with varying degrees of speed and grace.

The first one which comes to mind will likely be std::exit(). It sounds like just the thing, doesn’t it? But any C++ programmer should not be surprised to find that it’s not that simple.

This came to my attention in a recent conversation with a colleague [McGuinness23] who was unhappy about the presence of std::exit() call in a code base he was reviewing. When asked why, he explained that while this would call the destructors for static and thread-local objects, it would not call the dtors for automatic variables. This sounded surprising to me, but after a bit of digging on cppreference.com and in the C++ standard, I found that this is in fact the case.

But why would std::exit() ignore the dtors for automatic variables? It turns out that, for a normal exit in which the program returns from main(), there won’t be any. Returning from the main() function has the effect of ending its scope, causing objects with automatic storage duration to be destroyed. This is followed by an implicit call to std::exit(), which destroys the remaining static objects and terminates the program.

So, what happens if std::exit() is called elsewhere in the program? Does it matter that some automatic objects’ dtors may not be called on exit?

Often it does not. If the program is running under any of the usual operating systems, the OS will reclaim memory used by the process, close files and sockets automatically, and so on. If functions higher up in the call stack have existing automatic variables which own these resources, the fact that their dtors are not called may not make any difference at all.

However, it is dangerous to assume that this is the case – or that it will remain so in the future. If the program in question is large enough and complex enough, and if it has even a small team of developers all making changes to it, we are leaving ourselves open to some very subtle and intermittent bugs.

Most obviously, if the program has acquired resources which are not cleaned up automatically by the operating system – think of temporary files, System V IPC structures, GUI objects, database sessions, hardware devices, open orders, or worse – then this can cause a resource leak which is extremely hard to track down, especially if we believe that we have cleverly ruled out this possibility by wrapping our resource with a nice RAII manager.

Also, most of us will have run into shutdown errors, in which a program works perfectly well until it is time to stop, but then comes to an undignified end, perhaps leaving behind a corefile or a set of disturbing log messages. Often this is caused by code which expects that objects will be destroyed in a particular order, and thus it is safe for one object’s dtor to refer to another object that is still presumed to exist. (Data structures representing complex graphs are good candidates for this; ordering of data members also can be a culprit.) If we have a dependency graph containing a mixture of objects with automatic and static storage duration, then std::exit() may alter the usual destruction order, with unfortunate results.

What can we do about all this? The C++ standard library does offer a number of other exit functions; Table 1 (overleaf) is a summary of information from cppreference.com.

Looking at the ‘auto’ column, it’s clear that none of these functions will cause the stack to be unwound and dtors for objects with automatic storage duration to be executed.

Therefore, the only way to ensure a clean exit is to not call std::exit() and its friends at all, but to ensure that the program always returns from main().

This may seem impractical in a complex program in which the decision to exit is made far down the call stack. However, if the program is single-threaded then a simple solution is to throw an exception that propagates all the way back to main(), where it is caught and converted to a return. We might choose to throw an exception type that is not derived from std::exception in order to avoid inadvertent catches on the way up, but to still allow catch/rethrow by objects which must do something specific during shutdown.

In a multi-threaded program this is trickier. A thread which throws an uncaught exception will terminate the entire program – and the default std::terminate_handler() calls std::abort(), which doesn’t call any dtors at all. A different approach is required, possibly using std::packaged_task and std::future to return the exception to the main thread, as well as some means of stopping and joining other running threads before returning from main(). The details are well outside the scope of this article, but see [Williams19], as well as later C++20/23 changes such as std::jthread and std::stop_token.

Last but not least, consider that where external resources are involved, when the program is restarted it may be wise to ensure that those resources are in fact in a known and useable state; that they have not been left in a bad state by an earlier unclean exit. This is highly application-dependent, and may be much easier said than done.
A duplicate delete used to be a very hard problem to track down. If you were lucky, the program would crash immediately and leave a nice corefile to help with debugging.

### Table 1

<table>
<thead>
<tr>
<th>Exit via:</th>
<th>Dtor called for object with storage duration</th>
<th>Registered functions called</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>auto</td>
<td>thread_local</td>
</tr>
<tr>
<td>return from main()</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>std::exit()</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>std::_Exit()</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>std::quick_exit()</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>std::terminate()</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>std::abort()</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>gcc__builtin_exit()</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>gcc__builtin_trap()</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Note 1. The last two columns refer to the lists of functions registered using `std::atexit()` and `std::at_quick_exit()`, which will be executed during processing of `std::exit()` and `std::quick_exit()`, respectively.

Note 2. The `std::abort()` function (which is called by the default handler for `std::terminate()`) may write a corefile if the environment allows it.

Note 3. On POSIX-like systems it is also possible to terminate a C++ program by calling C library functions such as `exit()` and `abort()`. In some implementations, it turns out that these exhibit the same behaviour as `std::exit()` and `std::abort()` — including dtor calls, somewhat surprisingly. However, since they are not part of the C++ Standard Library it would be unwise to count on any of this.

Note 4. For sake of comparison, the last two rows show the behaviour of two gcc compiler intrinsics.

---

### Die Another Day

As if all of that wasn’t bad enough, let’s consider a number of situations in which a program can attempt to destroy an object more than once:

- Calling `delete` with a pointer to an object that was not created by an earlier call to new (generally caused by calling delete twice with the same pointer value, with no intervening new) is UB; but in practice it’s likely to take the form of a second call to an already-destroyed object’s dtor.

- A similar duplicate call to delete can occur if two or more instances of a `std::shared_ptr` are created, all of which point to the same object, because the separate `std::shared_ptr` instances have distinct reference counts.

- A duplicate dtor call may also occur due to an error in the move `ctor` or move assignment operator of a resource manager class, if the pointer (or other resource handle) in the moved-from instance isn’t set to null, or otherwise made to give up ownership.

- The same error can occur in a copy `ctor` or copy assignment operator — though [Müller19] points out that a resource manager class should be move-only, and so these functions arguably should have been deleted in the first place.

- A dtor may be called explicitly, with `p->~T()`, which is fine when destroying an object created via placement new — but not if a bug causes this to happen twice for the same pointer value.

As usual, any UB could conceivably manifest itself as a duplicate dtor call.

Fortunately, these are all software errors, and should therefore be preventable, or at least debuggable if they do occur.

A duplicate delete used to be a very hard problem to track down. If you were lucky, the program would crash immediately and leave a nice corefile to help with debugging. If not, the symptoms might not appear until much later, when there would be almost no chance of spotting the original error.

Today, though, we are fortunate to have lots of help. Most C++ compilers now provide sanitizers such as ASAN and UBSAN which detect most such errors, and produce very detailed reports showing where the duplicate dtor call occurred, where the object was initially created, and where it was first deleted. There’s no excuse for not taking advantage of these wonderful tools.

Static code analyzers are becoming very smart as well, though I’m not yet aware of one which can spot this sort of error at compile time. (I’d be delighted to be corrected on that.)

### You Only Live Twice

Finally, I’d like to point out one other scenario in which our dtors can surprise us by being called more than once.
C++ destructors are a powerful tool; one which we tend to take for granted, both for better and for worse. We should try to make the most of them, using RAII wherever possible.

In a POSIX-like environment, a process can call `fork()` to create a new copy of itself, resulting in a running parent and child process. The child process inherits a copy of the parent’s memory space, as well as a number of operating system structures, such as the list of open file descriptors.

This can be used to implement concurrent processing – though today other mechanisms such as threads (or, preferably, higher-level structures based on threads) and parallel algorithms offer better ways to achieve concurrency.

More commonly, the `fork()` call is used in conjunction with `exec()` in order to launch an entirely separate program; perhaps an existing utility program whose services our parent process wants to use, or an interactive program which the parent can control via stdin/stdout or another IPC mechanism.

In this case it’s good practice for the child process to close all file descriptors (fds) it inherited from the parent, then call `exec()`. If successful, `exec()` overlays the child process image with that of the new executable, and starts the new program running.

But what if `exec()` is not successful?

This might happen if the executable which was meant to replace the child process cannot be found, or is unavailable due to file permissions or other restrictions. In this case, both the original parent and child processes continue to execute.

So, what can the child process do now? It is an exact copy of the parent, which means that its memory contains copies of the instances of each of the parent’s objects. If the child process exits normally, via `main()`, all dtors for existing objects will be invoked!

If these dtors simply free up allocated memory, that may be all right, as the memory being freed is a copy of the parent’s memory. If they are associated with buffered streams, such as `stdout`, there may be some confusion as anything in the buffer at the time of the `fork()` call will be displayed twice.

However, if any child dtors free resources that exist outside of the program, this will not end well. Those resources will suddenly become unavailable to the parent – and when the parent process exits, it will attempt to free the resources again, with unpredictable consequences. The only safe thing for the child process to do at this point is to exit, and to exit in a way which guarantees that absolutely no dtors are invoked (and no registered dtors at all, nor any of the registered functions – exactly what we need. The child process simply disappears, leaving the parent process’s objects intact.

You can try this for yourself, at [Janzen23].

Conclusion

C++ destructors are a powerful tool; one which we tend to take for granted, both for better and for worse. We should try to make the most of them, using RAII wherever possible to protect against the sorts of resource management problems which befriend so many other languages.

At the same time, we need to be mindful of their limitations – to understand how they work and how they can fail.

I’ll close with a few recommendations:

- Always try to exit your programs gracefully; that is, by returning from `main()`.
- Avoid trying to exit a process from within application code, and certainly from within library code.
- When implementing a fork/exec, always have the child process call `std::_Exit()` if `exec()` should fail.
- Avoid `std::exit()` entirely.

References


[Janzen23] Compiler Explorer demo, [https://godbolt.org/z/YPovWq7a](https://godbolt.org/z/YPovWq7a)


[Klonowski23] Wiktor Klonowski, ACCU 2023 Lightning Talk, “‘Huzzah!’ for destructors in C++”, [https://www.youtube.com/watch?v=0WmriNuQu60](https://www.youtube.com/watch?v=0WmriNuQu60)


[Müller19] Jonathan Müller’s blog, 2019-02-26, [https://www.foonathan.net/2019/02/special-member-functions](https://www.foonathan.net/2019/02/special-member-functions)


[Stroustrup19] Lex Fridman Podcast, 2019-11-07, [https://www.youtube.com/watch?v=0WmriNuQu60](https://www.youtube.com/watch?v=0WmriNuQu60)


Credits / Apologies

All movie titles are trademarks of EON Productions Limited, and are used for educational purposes only, under the ‘fair dealing’ exceptions to UK copyright law.

October 2023 | Overload | 13
C# v12 Primary Constructors

C# v12.0, part of .NET 8, introduces a feature called Primary Constructors. Steve Love investigates how they fit into the language ecosystem.

Primary constructors are one of those features introduced to C# whose purpose is to simplify class and struct definitions by reducing the amount of code that needs to be written. However, they also have some potential for introducing confusion. In this article we’ll look at the essentials of using primary constructors, but also investigate how they compare with other established C# features, and some recommendations for introducing them in C# code.

The primary constructor

The idea behind primary constructors is simple: many, perhaps most constructors written for classes and structs have one purpose, which is to initialize fields or properties of the class with values copied from constructor parameters. Primary constructors remove much of the syntax of doing so. Listing 1 shows a bare-bones example of a simple Address class defined using the primary constructor syntax.

```csharp
public sealed class Address(string property, string postCode) {
    public string Property => property;
    public string PostCode => postCode;
}
```

Listing 1

The Address class doesn’t have a normal constructor method – a method with the same name as the type that optionally takes parameters. Instead, the Address type declaration has parameters named property and postCode. Beginning with C# v12.0, classes and structs can use the primary constructor syntax and the underlying mechanics are identical.

The Address class has two expression-bodied [MS-1] properties which use the parameter variables introduced in the primary constructor. The class in Listing 1 is almost equivalent to the class definition in Listing 2.

```csharp
public sealed class Address(string property, string postCode) {
    public string Property = property;
    public string PostCode = postCode;
}
```

Listing 3

If we add any of our own constructors, they must invoke the implicitly defined primary constructor using the this(...) syntax, showing in Listing 3. This ensures the parameter variables are always definitely assigned; failing to invoke the primary constructor results in a compiler error. Listing 3 shows an example where a user-defined constructor taking a value tuple [MS-2] forwards to the primary constructor.

The other thing to note about a primary constructor is that it’s always public. The user-defined constructor in Listing 2 can be made private, but a primary constructor cannot.

The primary constructor syntax is superficially similar to positional records [MS-3], introduced in C# v9.0, but records (as well as record structs in C# v10.0) differ from classes that have primary constructors in a number of important ways.

Positional records and record structs

As it stands, the Address class could easily be implemented as a positional record:

```csharp
public sealed record Address(string Property, string PostCode);
```

The most obvious difference between the Address record and the class with a primary constructor is that the record version has no explicit methods or properties, resulting in a type definition that doesn’t have a body. The compiler generates read only properties for the record type, using the parameter names given in the positional parameters of the primary constructor. Those properties are initialized by the parameter values according to the arguments passed to the constructor when an Address record instance is created.

Listing 4 (overleaf) demonstrates how the parameters in a record’s primary constructor translate to both the property names and the constructor parameter names, emphasized here by using named arguments in the constructor call.

Properties are not generated by the compiler for class or struct types using a primary constructor – we have to define them ourselves.
Classes, by default, have reference semantics so two variables compare equal only when they refer to the same instance in memory.

Equality semantics
A much more important but less visible difference between a record and class with a primary constructor is that for the purposes of equality comparisons, a record type has value-like semantics as demonstrated in Listing 5.

```csharp
var sweeney = new Address("186", "EC4A 2HR");
Assert.That(sweeney.Property, Is.EqualTo("186");
Assert.That(sweeney.PostCode, Is.EqualTo("EC4A 2HR");
```

```
var newCafe = new Address("186", "EC4A 2HR");
Assert.That(sweeney.Equals(newCafe), Is.True);
Assert.That(sweeney == newCafe, Is.True);
```

Where `Address` is a record type, this test passes. Where `Address` is a class – with or without a primary constructor – this test fails unless the class overrides the `Equals` method and `operator==` to give the desired behaviour. Classes, by default, have reference semantics so two variables compare equal only when they refer to the same instance in memory. The compiler generates the implementation required for value equality in a record type, but not in a class.

In keeping with the good practices for defining value-like equality behaviour for a type, the compiler also generates an implementation of `GetHashCode` for a record to ensure that two instances that compare equal according to `Equals` will always have the same hash code. Record types can usually be safely used as keys in collections like `Dictionary` and `HashSet` provided the instances are immutable, and caveats regarding floating-point equality are carefully considered where they’re appropriate.

The compiler doesn’t generate a custom implementation for either of the `Equals` or `GetHashCode` methods for a class. In the absence of an explicit override for these methods, classes inherit the default reference-based behaviour from the `object` base class. This doesn’t preclude instances being used as keys in hashing collections, but does require a bit of extra care.

Primary constructors and structs
Struct types can have a primary constructor in exactly the same way as classes, as shown in Listing 6.

```csharp
public readonly struct Address(string property,
    string postCode)
{
    public string Property => property;
    public string PostCode => postCode;
}
```

The differences between structs and classes do play a part here because all struct types inherit the `Equals` and `GetHashCode` methods from the `System.ValueType` class, giving them value semantics for equality. However, that inherited implementation may not be optimal, relying as it does on reflection in most cases (certainly for `Address`).

As with a class, the presence of a primary constructor on a struct definition doesn’t mean the compiler provides any special implementation of equality comparisons or property definitions as it does for a record or record struct. In particular, the compiler provides both `operator==` and `operator!=` for records and record structs, but does not do so for a struct, whether or not it has a primary constructor. Therefore, the test shown in Listing 5 using the `Address` struct in Listing 6 won’t compile, owing to the use of `==` to compare the `sweeney` and `newCafe` variables.

In any case, since C# v10.0, in most cases where a value type is needed in a program, a record struct is a better choice than a plain struct.

Properties vs. parameters
One final difference between class or struct primary constructors and the positional type arguments for records or record structs is how the identifiers are used within the body of the type. To demonstrate, consider the record in Listing 7, which has an instance method that uses the positional parameters as arguments to call an imaginary `AddressLookupService.Resolve` method.

Recall that the compiler uses the positional parameters of a record type to generate properties with the names of the parameters; when the `Resolve` method is called, the `get`-accessor for each property is invoked to obtain the value.

The parameters of primary constructors in classes and structs are different: the compiler does not generate properties, but we can still use the parameters in a similar member method, as in Listing 8 (overleaf).

Because the `property` and `postCode` parameter variables are used in the body of the class, the compiler stores them in hidden fields, and
Mistaking a primary constructor on a class for a positional record could easily lead to code being misunderstood, or even the introduction of hard-to-find errors

public sealed class Address(string property, string postCode)
{
    public string GetFullAddress()
    {
        return AddressLookupService.Resolve(property, postCode);
    }
}

Listing 8

directly accesses the field values to call the Resolve method. This may represent a very small performance gain over the record equivalent: accessing a property involves a method call. Accessing a field value directly is always optimally efficient.

This might matter if you’re especially sensitive to performance. In practice, at run time the method call will very likely be inlined, but there’s no guarantee that it will be.

The observant reader will have noticed the difference in casing between the record positional parameters and the class primary constructor parameters. The compiler generates property names from the positional parameters of a record, and C# property names – by convention [MS-4] – use PascalCase. The identifiers in a primary constructor are parameter variables which by convention use camelCase.

Conclusion

In short, primary constructors for classes and structs offer a concise and convenient way to define how instances of those types will usually be created. Although the benefits are hardly dramatic, a primary constructor is undoubtedly more compact than the equivalent full constructor definition. However, the primary constructor syntax is similar enough to positional records that some care is probably needed, especially when reading code using either syntax. Mistaking a primary constructor on a class for a positional record could easily lead to code being misunderstood, or even the introduction of hard-to-find errors. Records are semantically different from classes, whether or not those classes have primary constructors.

When a class does not require value-like behaviour for equality, and needs a custom constructor only to initialize fields or properties using the parameter variables, then a primary constructor may provide some, albeit minor, benefit. If the constructor needs to do more than simply assigning values to fields and properties, then a fully-defined constructor is required in any case.

If a class needs an overridden Equals method to compare fields or properties for value-like equality behaviour, then a record or record struct is almost always preferable, although using the positional syntax for them isn’t always the best approach.

References


Write for us!

CVU and Overload rely on article contributions from our members. That’s you! Without articles, there are no magazines.

What do you have to contribute?

- What are you doing right now?
- What technology are you using?
- What did you just explain to someone?
- What techniques and idioms are you using?

For further information, contact the editors: cvu@accu.org or overload@accu.org
Drive Musings on Linux
Dealing with large files can be hard.
Ian Bruntlett muses on various approaches that can help on Linux.

Motivation for this article

Files are important, which is why we entrust them to computer systems. For backups, I use a personal shell script which creates .tar.gz files which then get backed up to external drive [Bruntlett16, Bruntlett21]. For a long time that worked. Until one of my .tar.gz files exceeded the 4GB barrier. Attempting to copy big files eventually resulted in an error message complaining the file was too big for the file system. So, I started using ext4 instead.

That works except, if you distro-hop (I don’t) – ext4 drives store user and group metadata on the filesystem, leading to complications if you save data as one user, with, say a UID (User ID) of 1000 and try to access it later on a different user account, say with a UID of 1001.

As of the 5.4 release (November 2019) of the Linux Kernel, native exFAT support is built in. This filesystem does not store user or group metadata so UID and permission conflicts don’t arise. After much experimentation, I figure that extfat is the way to go with USB Flash or hard drives.

More about storage using ext4

I use Linux and my files are currently all stored on drives formatted to the ‘ext4’ format. What is ext4? Well, to quote Wikipedia, “ext4 (fourth extended filesystem) is a journaling file system for Linux”. It works. That was good enough for me.

Until you try to access the saved data from a different user account. From what I have seen on my systems, the conflict tends to be that on the originating system, you have full read/write/execute permissions whereas if you try to access the data using the ‘other’ user permissions, you only have read/execute permissions. As always, with Linux, ‘it depends’ on how your system is set up and what your current default access permissions have been set to using the umask command – a shell built-in command, so you’ll need to use the command help umask for more information.

Well, ext4 on Linux systems is usually configured to reserve a certain percentage of the drive for privileged processes – typically 5%. This is usually helpful if you are mounting your root filesystem on the partition. However, if you are strange enough to use ext4 for things like external drives (such as a USB hard drives), you invariably find that you run out of space quicker. This is because an ext4 filesystem has various parameters stored away. The one we are interested in is ‘Reserved block count’. If it is non-zero, it indicates the number of sectors exclusively available to privileged processes. So if you are using an ext4 formatted external drive, you have three options:
1. Just ignore the fact you aren’t using the full drive.
2. Use sudo to copy files to the drive.
3. Set the Reserved Block Count to 0 using the tune2fs command.

Linux provides a filesystem that is stored across one or more drives. That filesystem provides a way for programs to access stored data – such as

$ df -Th

<table>
<thead>
<tr>
<th>Filesystem</th>
<th>Type</th>
<th>Size</th>
<th>Used</th>
<th>Avail</th>
<th>Use% Mounted on</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmpfs</td>
<td>tmpfs</td>
<td>784M</td>
<td>2.1M</td>
<td>782M</td>
<td>1% /run</td>
</tr>
<tr>
<td>/dev/sda5</td>
<td>ext4</td>
<td>909G</td>
<td>370G</td>
<td>493G</td>
<td>43% /</td>
</tr>
<tr>
<td>tmpfs</td>
<td>tmpfs</td>
<td>3.9G</td>
<td>0</td>
<td>3.9G</td>
<td>0% /dev/shm</td>
</tr>
<tr>
<td>/dev/sdb1</td>
<td>ext4</td>
<td>148K</td>
<td>784M</td>
<td>1% /run/user/1000</td>
<td></td>
</tr>
<tr>
<td>/dev/shm</td>
<td>tmpfs</td>
<td>3.9G</td>
<td>0</td>
<td>3.9G</td>
<td>13% /media/ian/HERMES</td>
</tr>
</tbody>
</table>

Figure 1

/home without having to think about which device a file or directory is on – or even which partition that a file or directory is on.

How do you find out where a file or directory is physically stored? You use the df command which ‘reports file system disk space usage’. Cryptic, no? To get an overview of your filesystem storage, you just type in df at the command line and you might see something like Figure 1.

The above output is partially interesting. I ignore the entries for tmpfs, which are ram disks used by Linux itself.

The interesting stuff begins with /dev – an abbreviation of ‘device’. The critical stuff is the (/ root) partition. Interestingly enough, you can place (‘mount’) parts of the Linux filesystem over multiple drives and partitions. So, if you wanted to, you could mount the root (/) filesystem on one drive and the home files of all users (/home) on a different drive. Note that the ‘Type’ column is important – this article is only relevant to filesystems with a type of ext4 (and presumably other earlier versions of ext as well – this hasn’t been tested on them).

To view the Reserved Block Count using the tune2fs command (default is 5%), assuming that the filesystem is mounted to /dev/sdb1

$ sudo tune2fs -l /dev/sdb1 | grep Reserved

Reserved block count: 97766
Reserved GDT blocks: 477
Reserved blocks uid: 0 (user root)
Reserved blocks gid: 0 (group root)

To set the Reserved Block Count to 0, use the tune2fs command again

generate@lucas:$ sudo tune2fs -m 0 /dev/sdb1

tune2fs 1.46.5 (30-Dec-2021)

Setting reserved blocks percentage to 0% (0 blocks)

So, now we can use all of an ext4 filesystem’s space for our files. How do we keep them safe? By backing up, to a memory stick. How do we keep our memory sticks synchronised with the contents of our main drive(s)?

Keeping backup drives up to date

What if you want to quickly check if a main drive directory is reasonably similar with a copy on an external drive? I wrote a shell script, irb-dirstat (in Listing 1, at the end of the article) which, given a

Ian Bruntlett
Ian is a keen reader of software development books. He has promised himself a long stint at dealing with C++, once he has got to grips with Git.

October 2023 | Overload | 17
directory will count the number of bytes, files, and directories in a
directory (including its children directories).

Here is the help/usage message for the script:

```
$ ./irb-dirstat --help
irb-dirstat: usage irb-dirstat directory1
                         [directory2 etc]
Used to check actual number of bytes, files, and
directories in a directory

Formatting output options for bytes used in files
   -b or -B output number of bytes (this is the
default)
   -k or -K output number of KiB
   -m or -M output number of MiB
   -g or -G output number of GiB
   -t or -T output number of TiB
   -e or -E output number of EiB
   --commas output byte count with commas
   --no-commas output byte count without commas

--help display this message
```

The above information should be fairly obvious. It is useful when doing
rough comparisons of a couple of sub-directories. Here is a sample of its
output:

```
~$ ./irb-dirstat --commas ~/isos
/home/ian/isos
Byte count 76,884,580,097
Dir count 71
File count 148
```

To take advantage of globbing, `irb-dirstat` can handle one or more
directory names as arguments. This is particularly useful when dealing
with multiple directories or to compare a source directory with a
destination directory.

However, `irb-dirstat` is best used for rough but quick comparison
of directory trees. GNOME’s `meld` command will do a thorough check
of two subdirectories (it does other things as well). Unfortunately, it is
necessarily slow and sometimes crashes with an error message so it isn’t
something I rely on alone.

The `diff` command can check two directories recursively, if you pass it
two directories and the `-r` flag. I haven’t managed to crash this command
and its output is very helpful. I tend to use `irb-dirstat` to quickly
ensure the drives are reasonably synchronised and finally use the `diff`
command for a more thorough, byte by byte comparison.

References

[Bruntlett16] Ian Bruntlett ‘Stufftar’ in Overload 132, April
bruntlett_2226/

[Bruntlett21] Ian Bruntlett ‘Stufftar Revisited’ in Overload 165, October
2021, available at https://accu.org/journals/overload/29/165/
bruntlett/

```bash
#!/bin/bash
# default values, to be overridden by command
# line options
divisor_scaling="Byte"
divisor=1
use_commas=0

function report_bytes_used_in_files()
{
    echo -n "$divisor_scaling count "
    byte_count=$(count_bytes_used_in_files "$1")
    if [ $use_commas -eq "1" ] ; then
        printf %'f
"byte_count"
    else
        echo "$byte_count"
    fi
}

function count_bytes_used_in_files()
{
    # this command inspired by
    # https://stackoverflow.com
    find "$1"/! -type d | awk -v divisor="$divisor" '{s+=$1} END {print s/divisor}'
}

function report_no_of_directories()
{
    echo "Dir count "$count_no_of_directories "$1"
}

function count_no_of_directories()
{
    find "$1"/* -type d | wc -l
}

function report_no_of_files()
{
    echo "File count "$count_no_of_files "$1"
}

function count_no_of_files()
{
    find "$1"/* -type f | wc -l
}

# return value 0=files present,
# 1=error or no files present
function are_there_any_files()
{
    find "$1"/* -maxdepth 1 -type f
    -o -type d -iname "*" 1> /dev/null
}
```

Listing 1

ext4 on Linux systems is usually configured
to reserve a certain percentage of the drive
for privileged processes – typically 5%
function do_dirstat() {
    if [ $# -ne 1  ]; then
        echo "do_dirstat insufficient no of " \ "parameters ($#)." >&2;
        return 1;
    fi;

    if ! are_there_any_files "$1" ; then
        # echo NO FILES
        echo "$1"
        echo "$divisor_scaling" count 0
        echo "Dir count 0"
        echo "File count 0"
        echo return 1 # is a useful value?
    fi

    if [ ! -d "$1"  ]; then
        echo "Parameter $1 is not a directory" >&2;
        return 1;
    fi;

    echo "$1"
    report_bytes_used_in_files "$1"
    report_no_of_directories "$1"
    report_no_of_files "$1"
    echo
}

function display_help() {
    cat <<END_OF_HELP
irb-dirstat: usage irb-dirstat directory1 [directory2 etc]
Used to check actual number of bytes, files, and directories in a directory

Listing 1 (cont’d)
The whole of irb-dirstat can be found online at https://github.com/ian-bruntlett/studies

Listing 1 (cont’d)

The whole of irb-dirstat can be found online at https://github.com/ian-bruntlett/studies

Calling all photographers...

If you would like to see your work on a cover of CVu or Overload, get in touch.

You don’t need a fancy camera – using your phone is fine. But photos do need to print at A4 size, so need to be at a high resolution.

CVu tends to use images of the sea (‘sea view’) and Overload is more likely to have something with a technical focus, or featuring a structure or pattern.

Contact alison.accu@virginmedia.com for more information.
Afterword
What’s in a name? Chris Oldwood considers metaphors as inspiration for naming in code.

The second most popular joke in programming reminds us that naming is hard. (In a Programming Jokes Top 10, this would be the other one.) That hasn’t stopped some people apparently attempting to overcome this problem by applying a ‘formula’, which generally involves concatenating various Computer Science and business terms together and finally appending one of Manager, Service, or Provider to give it extra gravitas. The problem appears to be particularly acute in the enterprise world of Java and C#, where Design Pattern Bingo is a popular pastime.

What triggered this latest Afterwood was working in a C# codebase where there had been a need to write tests for code that invoked DateTime.Now, which is a static method that returns the current date and time. Doing a spot of software archaeology, I noticed there had been an initial attempt to work around the non-deterministic nature of this method by doing an assertion in the tests with a small tolerance, but the woefully underpowered CI server put paid to that as the tolerance was widened every time the tests took longer than usual to run.

The solution had been to mock out the method call entirely and return a fixed value in the tests, and the real date and time in production, by introducing a suitable interface and pair of concrete implementations. Naturally, if you’re looking to name an interface for such an abstraction you’re probably thinking, ‘What do I call an interface that provides a date and time value?’ and so you go with the first thing that pops into your head – IDateTimeProvider. Wait, that’s not right…

I don’t know for sure how this name really came about but the seemingly robotic approach to so many names in the codebase suggested they were driven by terms from the solution domain instead of the problem domain or, say, The Real World™. I posited on the team chat that the IDateTimeProvider abstraction was basically just a ‘clock’, and that was largely met with approval, so the refactoring went straight in. It also opened the door for a further discussion about naming, metaphors, and typing less. (Strunk famously tells us to ‘omit needless words’ which has the marginal added benefit of less wear and tear on the keyboard.)

The world of software is entirely virtual in nature and therefore we must rely very heavily on metaphors as a source of inspiration for how we name stuff. The great thing about the world of horology is that it provides us with a whole host of physical devices to draw from. ‘Clocks’ and ‘calendars’ allow us to discover the current time and date, ‘stopwatches’ allow us to measure time, and ‘timers’ can be used to notify us when a period has elapsed. Clocks come in many different shapes and sizes, and degrees of precision, so if you want to capture that in your naming scheme you could use ‘wall clock’ for the low-end and a highfalutin’ name like ‘chronometer’ for the high-end.

While the name IDateTimeProvider might on the surface appear to be sufficient for the task, I argued that it’s too abstract. This also gave the perfect opportunity to play one of my Programming Quote Top Trump cards [Oldwood23] from Edsger Dijkstra:

Being abstract is something profoundly different from being vague … The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

The point of using a metaphor is to allow us to be more precise about the kind of thing we’re talking about by borrowing from a recognisable domain. When talking to the business we would expect to use a ‘ubiquitous language’ and expect the problem domain to provide many of the terms we use in our code. When it comes to the technical domain, something which the business will have almost no input on, we are left to our own devices. This does not absolve us of the responsibility to be clear about what we mean in the code. If anything, we have to work harder because it’s not handed to us on a plate. (Pro Tip: always keep a thesaurus on hand for inspiration [Oldwood15].)

One comment to my suggested renaming was that it didn’t really matter because although the code said IDateTimeProvider, in their head they mentally translated that into ‘clock’ anyway. This misses one of the key points about why we refactor code – to ensure that it always reflects the best understanding we have of the domain at any given point in time. If your best mental model is currently a clock, then that’s what it should be called (for now), don’t make people waste brain cycles second-guessing what might have been meant.

And it is just a model, and an imperfect one at that. Playing my second Quote Top Trump card – George Box – we are reminded that “all models are wrong, some are useful”. In some scenarios, it could be a poor metaphor because a clock may conjure up a different kind of device, such as in electronics where the clock is an oscillating signal, more like a metronome ticking left and right, than a pair of hands slowly turning around a circle. Fortunately, the same stable that brought us the notion of a ubiquitous language also helps us resolve our conflict here by applying a ‘bounded context’ around our codebase so that the interpretation is the most fitting one for our part of the business instead of encompassing every potential definition covered by Wikipedia.

Hopefully, the use of the clock metaphor will be timeless, but as we get older we do need to be aware of anachronisms, such as the nautical terms I wrote about back in June [Oldwood23]. Once upon a time, Hi-Fi Separates (where the turntable, tape deck, CD player, amp, etc. were all distinct devices connected by standard RCA cables) was a common metaphor for a component-based architecture but that seems to have died out as headphones (except on public transport) are the only accessory which the business will have almost no input on, we are left to our own devices. This does not absolve us of the responsibility to be clear about what we mean in the code. If anything, we have to work harder because it’s not handed to us on a plate. (Pro Tip: always keep a thesaurus on hand for inspiration [Oldwood15].)

References

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it’s enterprise grade technology from plush corporate offices to the comfort of his breakfast bar. He has resumed commentating on the Godmanchester duck race but continues to be easily distracted by messages to gort@cix.co.uk or @chrisoldwood
“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 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 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.
Professional development
World-class conference
Printed journals
Email discussion groups
Individual membership
Corporate membership
Visit accu.org for details