overload 126

APRIL 2015 £3

Resource Management with Explicit Template Specialization

RAII is a useful C++ idiom. We present a powerful extension using explicit template specialization.

Alternatives to Singletons and Global Variables

Alternatives to these often-derided code idioms

Non-Superfluous People: UX Specialists

User experience specialists can be vital to the success of a project

Variadic and Variable Templates

Further investigation of useful C++ facilities

iterator_pair - a simple and useful iterator adapter

How to form a new container from two others

Seeing the Wood for the Trees

How code can reflect your environment

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 126

April 2015

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael@accu.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Jon Wakely
accu@kayari.org

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 127 should be submitted by 1st May 2015 and those for Overload 128 by 1st July 2015.

The ACCU

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

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

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

4 Non-Superfluous People: UX Specialists
Sergey Ignatchenko demonstrates why user experience specialists are not superfluous.

9 Alternatives to Singletons and Global Variables
Bob Schmidt summarises some alternatives to singletons.

14 Resource Management with Explicit Template Specializations
Pavel Frolov presents RAII with explicit template specialisation.

19 Variadic and Variable Templates
Peter Sommerlad showcases the compile-time possibilities Variadic and Variable templates offer.

23 iterator_pair – a simple and useful iterator adapter
Vladimir Grigoriev reminds us how to write an iterator.

32 Seeing the Wood for the Trees
Teedy Deigh takes an enterprising look at logs.

Where was I?

Space and time are relative. Frances Buontempo wonders whether this will wash as an excuse for another lack of editorial.

Previously we almost wandered into the religious realm, while considering fear, uncertainty and doubt [FUD]. If we were to bring things round to a more scientific perspective, we might find relativity leaves us doubting where, or when, we actually are, though I am getting ahead of myself. I realise I should bring things back on track and finally embark on an editorial but what with one thing and another I have been distracted yet again. Firstly, though perhaps less significantly, I have started a new job so am in the process of learning various new TLAs, eTLAs and TLs.1 Secondly, I have been drawn by a recurring theme of late around telepresence, perhaps the ultimate spooky action at a distance. Having just finished The peripheral by William Gibson [Gibson], which concerns people remotely operating various machines, from 'peripherals' which seem to be human sized dolls designed for such purposes, through homunculi to a 'wheelie-boy' which is a smart-phone on wheels. These allow people to interact across space, and being a sci-fi book, across time as well. For many years it has been possible to use a telephone to speak to someone a great distance away and given newer technologies like video-phones, a move towards a physical remote-presence seems like the next big thing.

I recently watched *Kraftwerk: Pop Art* [Kraftwerk], which fed into this train of thought. For quite a long time now, photo-shoots have used their robots instead of the band members. When performing, these robots frequently take up residence on stage playing the instruments instead of the people. This may seem odd, but clearly allows the music to continue, and to an extent, the band to continue performing long after the initial people are gone – a form of time travel. Other bands don't physically exist, for example The Gorillaz spring to mind [Gorillaz]. The audience is still at the gig even if the band physically are not. And flipping it around, I own DVDs of shows which I can then watch at any time, without being at the actual venue and this can almost feel like having engaged with the experience to an extent. Perhaps one day I will be able to operate a mini-drone to remotely experience a concert. It does not always matter exactly where or when you are.

It has been possible to operate a physical device remotely for some time now. More recent examples include surgery and surgical simulators with haptic feedback [LSRO], bomb disposal and fire-fighting robots [SAFFiR] and unmanned space-crafts. Some of these operate in real time, and others are more fire and forget. Simpler exemplars could be argued to include a telephone or a television, perhaps via a remote control. Again, spooky action at a distance. If haptics allow you to feel something that is very far away or even virtual, what other 'tele'-types are possible? Telesmell? Telesthesia? Telemetry? Teleportation? How far can this remote presence go? Would it be socially acceptable? It may be frowned upon to dial into a team's daily scrum meeting, but sometimes a team is distributed across the globe, so it is sensible to do this. What if I sent in a mini-me robot or wheelie-boy to a meeting instead of actually turning up? Is that different to sending a secretary? Could the whole team 'meet' in a virtual reality world to discuss things? Would this be easier than a phone meeting? Do you need to interview a candidate face to face? Could you get married via a phone conference? New technologies bring about new social norms, where the previously unthinkable becomes par for the course.

Many people in the industry work from home for a large percentage of time nowadays, while others, perhaps in a business facing role, do so almost never. Some people prefer to communicate directly, while others will prefer emails or chat rooms. It will always be context dependant. If someone is demonstrating a new API, I like to have some code snippets in an email to refer back to rather than trying to frantically scribble notes and listen at the same time. The method of communication can and must depend on the circumstances. Having wondered if everyone needs to be physically 'there' begs the question, where is 'there' anyway. How many times have you looked round a meeting to see people staring at their smart phones? If someone, say a politician, is physically present at a meeting, but seemingly engaged in a game, say Candy Crush Saga [Mills], at least in one sense they are not really at the meeting but elsewhere. If I am at my desk on my PC but remoted to another machine, where am I? If I log on as someone else, who am I? Of course, various machines will answer 'whoami' but where am I is clearly a harder question. If the machine I remote to is a virtual machine, am I in the 'Matrix' – some form of non-physical reality? And yet my body is still at my desk. I am in two places at once.

Almost everyone has bemoaned the impossibility of actually being in two places at once, even though we have all plainly touched on this possibility without taking the full Candy Crush leap. Suppose for a moment I could clone myself and genuinely be in two places at once. Then I would have had the time to write *Overload* a proper editorial. Whatever your reason for needing to be in two places at once, you might feel the need to rendezvous with yourself at some point in space and time to synchronise. This presupposes the clone is really a deep copy. A shallow copy would rather defeat the purpose. The confusion of two individuals being the same, identically, and in no way different, presumably being in the same place at the same time, goes beyond the horror of memory leaks or double deletes and breaks the laws of physics. Without harmonising or re-integrating between your many selves, at least one of you would in some sense cease to be you. Would the synchronisation require a lock of time

1. Three letter acronyms, extended three letter acronyms and two letters

Frances Buontempo works at Bloomberg, has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

or reality? That may prove tricky to implement. Even if it were possible, time could pass while catching up with yourself, so it isn't immediately apparent that cloning yourself would be the time-saver we hoped for. As with many applications that start life single-threaded, any attempt to save time by introducing some concurrency may actually slow things down, especially if you are using shared memory. A much simpler alternative is to delegate the editorial writing, or whatever tasks you are currently avoiding, to someone else and just hope for the best.

Even if we keep things simple and try to just be in one place at one time, things may not be straightforward. I mentioned relativity earlier. Though we may feel we are taking things slowly and methodically, going nowhere near the speed of light, precision regarding when and where we are often matters. I saw a recent plea on Twitter to retweet a post by 10pm in order to be in with a chance to win a book. First, I needed to know by 10pm on which day, and furthermore, I needed to know which time-zone. Not everyone is in the same place as you. Midday does not mean the same time to everyone. Neither does early in the morning, though questions of sleep-wake homeostasis and circadian biological clocks are beyond the scope of my current meanderings. How long I have before 10pm is another matter for discussion. Special relativity tells us about time dilation and length contraction, "A clock in a moving frame will be seen to be running slow, or 'dilated'" [Hyperphysics] Perhaps this is why deadlines don't seem so close until you are right on top of them. This might not be the best excuse to give your manager for being late with a project, so use judiciously.

Where was I? Without duplicating myself, even with an ersatz, phony, proxy other to do my dirty work for me, and attempting to slow down and just single task, I still might not achieve everything I set out to do. I can be self-reflective though. It is useful to keep notes to see how well I'm doing, or my team is doing. For those of a geek bent, there are various ways of automatically keeping track of things. If your code-base isn't terabad, then you might have it running on a continuous integration box allowing you to perform some software archaeology [TICOSA]. You can graph the build times, quickly spot churn in various code modules, notice early if tests slow down or speed up, glance at a burn-down chart, or see the team is giving 120%.2 Beyond the day-job, technology can be used to track all kinds of things. Various apps exist for tracking your phone in case you lose it. I personally need something which keeps track of where I put my notes, but that might just mean I need to be better organised. The phone tracking apps have recently moved up a notch, with 'spy apps' hitting the headlines [Spy apps]. The premise appears to be that the teens spend so much of the time on their smartphones, communicating with their friends and strangers on various forums and the like, that parents can take an interest by tracking exactly what they have been up to. Furthermore some of these applications claim to allow parents to track exactly where the children are. Without having delved into the details of the technology I suspect the apps will potentially tell parents exactly where the smartphone in question is, which may not be the same thing. This may require some form of tracking device implant, which brings to mind various stories regarding Kevin 'Captain Cyborg' Warwick:

> Warwick also surgically implanted a trivial chip in his arm, which allowed sensors to detect his presence and do things like turn on lights and open doors, then romped about in the media explaining gravely that he was now a cyborg: 'Being a human was OK,' he said. 'But being a cyborg has a lot more to offer.' Bravo. It was never clear why he couldn't just carry the chip in his pocket. [BadScience]

To me it is self-evident. The door would then open for anyone who borrowed his jacket. Alternatively, if he left his jacket at lunch he wouldn't be able to get back in again. If you instrument something or someone to see where it is and what it's up to, make sure you are measuring the right thing.

Has this diversion allowed me to clarify my thoughts and get myself on track? Almost certainly not. It has made me less concerned about figuring out where I am and what time it is. There's nothing like taking your watch off on holiday and just walking round an unknown town to see what happens. Getting lost can be a fruitful journey of discovery. We have all heard various myths and legends of people heading into the desert to mediate or find themselves. Being a bit vague is sometimes ok. Now if only I could remember where my smart-phone is. Let's ring it from the landline and see if that helps.

References

[BadScience] http://www.badscience.net/2004/04/the-return-of-captain-cyborg/

[FUD] 'FUD – Fear, uncertainty and doubt', Frances Buontempo *Overload* 125, Feb 2015

[Gibson] *The Peripheral* Sep 2014, Putnam.

[Gorillaz] http://en.wikipedia.org/wiki/Gorillaz

[Hyperphysics] http://hyperphysics.phy-astr.gsu.edu/hbase/relativ/tdil.html

[Kraftwerk] http://www.imdb.com/title/tt3262308/

[LSRO] http://lsro.epfl.ch/simulators 'Surgical simulators with haptic feedback – training of minimally invasive surgery'

[Mills] Nigel Mills, http://www.bbc.co.uk/news/uk-politics-30375609

[SAFFiR] Shipboard Autonomous Firefighting Robot http://www.livescience.com/49719-humanoid-robot-fights-fires.html

[Spy apps] for example http://www.bbc.co.uk/news/technology-30930512

[TICOSA] http://ticosa.org/

2. I suspect we set that graph up incorrectly.

Non-Superfluous People: UX Specialists

User experience specialists are sometimes regarded as superfluous people. Sergey Ignatchenko demonstrates why they can be vital.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with the opinions of the translators and *Overload* editors; also, please keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented an exact translation. In addition, the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

> *The superfluous man (Russian:* лишний человек*, lishniy chelovek) is an 1840s and 1850s Russian literary concept derived from the Byronic hero. It refers to an individual, perhaps talented and capable, who does not fit into social norms.*
>
> ~ Wikipedia

This article continues a mini-series on the people who're often seen as 'superfluous' either by management or by developers (and often by both); this includes, but is not limited to, such people as testers, UX (User eXperience) specialists, and BA (Business Analysts). However, in practice, these people are very useful – that is, if you can find a good person for the job (which admittedly can be difficult). The first article in the mini-series was about testers; this article tries to show why do you need to have user interface (or more generally – User eXperience) specialists on your team.

UI nightmares

As a user, I hate poorly designed UI. I really, really hate it. Poor UI takes away my time (and the time of thousands and millions of other users), and simply because of somebody not spending 5 minutes thinking about it. Decent UI might be not rocket science, but it certainly does require a view from the user's perspective – one thing developers (almost universally) and project stakeholders (sadly often) lack.

UIs designed by developers

Let's take a look at some of the UIs designed by developers.

LibreOffice Writer 'Find' – strike 1

While I use LibreOffice all the time and think overall it is a decent piece of software, the 'Find' feature of LibreOffice Writer is quite annoying to say the least. This is how it works in LibreOffice 4.0 under CentOS Linux (on other platforms details might be different):

- I press Ctrl+F and it opens a 'Search' bar at the bottom of the screen
- The focus is already in the search box, so I can start typing right away. Good. I enter search the term, and press Enter – it finds the first occurrence of the search term. So far so good.
- Jumping through further search term occurrences can be done just by pressing Enter, which is good too.
- However (LO Problem 1), if I'm already at the last occurrence of the search term, LibreOffice shows a dialog box, asking if I want to continue search from the beginning (which is fine). The problem here is that *the focus is not on this dialog, so to press 'Yes' I need to use the mouse*. Hey folks, it is a Writer application, where most of the work is done (surprise!) with a keyboard, and moving a hand from keyboard to mouse for such a routine task is a waste of time. It means that we have a bit of poor UI here (and no, it is not fatal – just as with many other UI flaws – but when we sum every bit of time wasted, it translates into hours of unproductive activity).
- Another problem (LO Problem 2) is that to move to the occurrence of the search term in the text from the 'Search' box, *I need to use the mouse again*. Which is not *that* bad, but I'd still prefer to have the 'Tab' key go straight there (rather than to move to the 'Find Next' button, which is pretty useless for the user).
- And yet another problem (LO Problem 3) is that when I've already moved focus from the search box to the text window, the only obvious way to continue my search without using the mouse is to press Ctrl+F (to move the focus to the search box), and then to press Enter (to move to the next occurrence). This is three key presses (two for Ctrl+F, and one for Enter), and I'd certainly prefer to have a single one (for example, the fairly standard F3). For those who'll say "Hey, there is already a hotkey XX" (which I wasn't able to find, but it might still exist) – my answer is that "If there is such a hotkey, it should be shown when I'm hovering the mouse over the 'Find Next' button in the search bar, so I can learn about it without Googling it". For those who says "Hey, you can configure all the hotkeys you want in the Tools > Customize menu" (I wasn't able to find this specific function there, but once again, it might still exist) – I will note that such an obviously necessary function should be pre-configured by default.
- However, all these problems are mere peanuts compared to the Big One (LO Problem 4). If I gave up staying with the keyboard and conceded to use the mouse (as a result of carefully crafted LO Problems 1–3), then there is one more surprise, and a very nasty one. If I click on the 'Find Next' button to find what I need, and the next occurrence of the search term is within a bulleted list, then all of a sudden, another bar (to manage the list) appears below the search bar, moving my search bar and 'Find Next' button up. Therefore, if I'm clicking 'Find Next' in a quick succession (and looking into the text window), another button (which is 'Move Up with Subpoints') appears right under my mouse, and I can click it without even realizing that I've just changed my document!

LO Problem 4 is illustrated in Figure 1 and Figure 2.

As you can see, position of the mouse on both screenshots is exactly the same, but the button underneath has changed...

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He is currently holding the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com

to fix the problem somebody should have spent time discovering that the problem exists in the first place

This kind of things (when a potentially dangerous button appears in place of a very harmless one) is a Big No-No in UI design (and the fact that it – as well as all the other poor UI manifestations – happens all the time, is not an excuse). In the case of LibreOffice, the fix for this particular problem is trivial – to get rid of it, it is sufficient to move all 'read-only' bars (such as the 'search' bar) below 'modifying bars' (such as the 'bullet list' bar). However, to fix the problem somebody should have spent time discovering that the problem exists in the first place, which apparently didn't happen here.

Now let's ask ourselves the question – why did these problems occur? I suggest that there are two reasons. The first is that it was technically simpler to implement it this way. While LO Problem One is about keyboard focus, which is always a headache to deal with properly, LO Problem 2 is about overriding the default behavior of the Tab key (and leaving it at default is always simpler), LO Problem 3 is about doing something instead of doing nothing, and LO Problem 4 is relying on one generic concept (that of 'stacked bars') without thinking about potential interactions between those bars. On the other hand, it is certainly not rocket science to fix these issues. OK, it might take a few hours to fix problems 1–4, but it is nothing compared to amount of work thrown into LibreOffice, and it would improve usability by a significant margin.

The second reason is that there is nobody on the development team who is responsible for making the software convenient for the user, and/or having enough influence to make developers do it. Without someone who advocates the needs of the user (against the natural need of developers to implement the function as simply as possible), all the good intentions to make good, usable software for the end-user won't materialize.

As a side note: a potential argument "hey, it is free software, so you cannot complain about it" doesn't really fly. If you folks want your software to be used, you should care about your end-user, whether the software is free or not. Of course, software being free as in 'free beer' does indeed help people to accept it, but doesn't guarantee acceptance at any rate; crappy free software will lose to good commercial software, whether we like it or not.

Figure 1

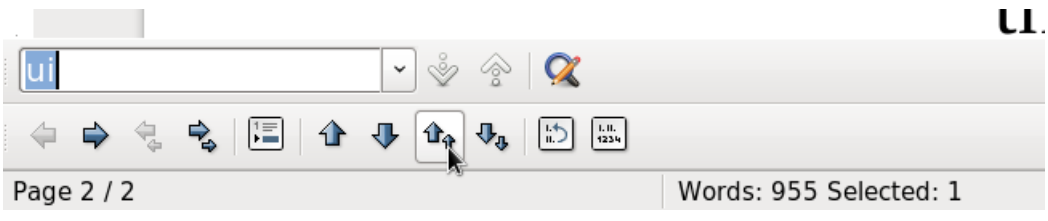


Figure 2

Windows MessageBox() – strike 2

The road to hell is paved with good intentions

~ proverb

Our next example of atrocious UI design touches an (in)famous Win32 `MessageBox()` function. For those few who don't know it, here is its prototype:

```
int WINAPI MessageBox(HWND hwnd, LPCTSTR lpText,
LPCTSTR lpCaption, UINT uType);
```

If you haven't seen it before, you'll ask yourself – hey, how does it know which buttons are to be shown? Apparently, buttons are 'conveniently' hidden behind that `uType` parameter, as something like `MB_YESNOCANCEL`, which specifies 3 buttons – 'Yes', 'No' and 'Cancel'– or as `MB_OK` with one 'Ok' button, or as `MB_ABORTRETRYIGNORE` etc.

Now let's see in which direction this API pushes developers. As the API doesn't allow you to specify exactly the buttons you need (and creating your own message box with your own buttons, while possible, is quite a lot of work), Windows software is full of message boxes with text like the following:

> If you want to save file before closing the window, press Yes. If you want to discard the changes you've made since last save, press No. If you want to keep editing, press Cancel.

It would be much more user-friendly to make it three buttons 'Save File', 'Discard Changes', and 'Keep Editing' – and avoid the potential for confusion and mistakes, but the `MessageBox()` API encourages developers to push complexity towards the end-user. No wonder developers are going down this road (obviously paved with good intentions by whoever designed the `MessageBox` API).

But I'm not done yet with presenting my evidence against `MessageBox()`. The real fun starts when your software is running on non-English Windows. In this case, Windows 'conveniently' replaces 'Yes', 'No', and 'Cancel' with their translated versions (while your software, unless you've spent quite

the difference between the two cases wasn't obvious at all to me as the user; one time it went one route, another time it went another way for no apparent reason

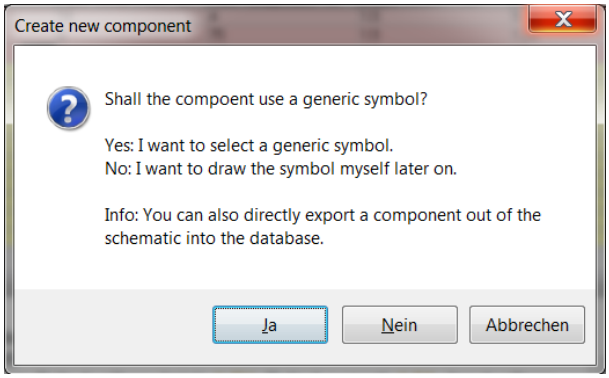


Figure 3

an effort translating it, remains in English). This often leads to such message boxes as the one in Figure 3.

I rest my case.

Fax machine UI – strike 3, developers out

Bad UI is certainly not restricted to PCs. One of the most ridiculous UIs I've seen was on a fax machine. If you ask yourself – what can be so bad about the UI of a fax machine – I will name just a few (mis)features of the machine. It was so bad that I don't want to name the company that made it, because the same company produces very usable printers, which I like a lot; I hope that they will learn from their mistakes. So, here goes the list of (mis)features:

When the fax I was sending didn't go through, two things could have happened (and the difference between the two cases wasn't obvious at all to me as the user; one time it went one route, another time it went another way for no apparent reason).

- In the first type of failure, the fax machine produced a sound which was enough to awake a nearby cemetery (and of course, none of the volume controls was able to affect it), and then it just considered the job done.

  To find out if the fax was successful or not, I needed to be near the machine. If I was away, it blinked three times with the error message, and then went to the idle state, leaving me, when I came back, wondering if the fax had gone through or failed. How it should be implemented (and actually is implemented on a competing fax machine from a different manufacturer) is that the status should blink, at least until the user interacts with the machine.

  To re-submit the fax, I needed to feed it through the machine once again. The machine was implemented as a scanner+printer, and by the point of failure, the fax had already been scanned. In the process of scanning it had already passed through the machine, but the machine in its infinite wisdom has decided to discard results of the scan in this particular case, so I needed to take the pile of paper and put it into the feeder again.

- In the second type of failure, there was no sound, and again the message blinked only three times. It appeared that in this second case, the fax machine has realized that the problem is transitory, and that it should retry the fax some minutes later. So far so good, but:
  - on the front panel of the machine there was no indication whatsoever that the machine has some fax in memory
  - in fact, the only place where you can find out what happened with your fax, was three levels deep into the fax machine menu, with one of the levels aptly named 'MEMORY SETTINGS' (this obviously was made to make sure that there is no chance to operate the machine without the manual).

Overall, the machine was such a nightmare, that when a lightning strike put the machine out of its misery, I was really grateful for this Act of God.

Once again, the reason for this UI was two-fold: first, it was technically simpler to do it like this, and second, there was nobody to represent the end-user and to advocate her interests.

In general (and as it has been observed in these three examples), developers are not good at designing UIs. Personally, I feel that this is because when designing the UI, a developer is inherently in a position of a severe conflict of interest: on the one hand, he needs to finish the job fast (and to move on to implementing other features), and on the other hand, user interests may require spending another few hours before moving ahead. In theory, this conflict of interest should always be resolved in favor of the end-user (for example, based on the logic from [NoBugs11]), but in practice, more often than not, developers ignore the end-user at least to some extent; in extreme cases, it results in really atrocious UIs (like our last two examples).

Act of God

An Act of God is a legal term for events outside human control, such as sudden natural disasters, for which no one can be held responsible. [Wikipedia1]

UIs designed by project stakeholders

Ok, developers are not good in writing UIs. But what about project stakeholders? They should know what is good for the user, right?

Unfortunately, the answer is "not necessarily". In many cases, it works well (especially if stakeholders are end-users themselves), but in many other cases, it doesn't. And if things go wrong with stakeholder design decisions (especially if stakeholders have had a Big Idea which overrides everything else, including common sense), the consequences can easily be on the much larger scale than that of developer-designed UIs.

QuickTime Player 4 – strike one

Back in 1999, with QuickTime Player 4.0 UI, Apple had a Big Idea to mimic a physical media player on-screen. And their developers have faithfully implemented this idea. Which, apparently, turned out to be

I've worked with a few UX specialists and was amazed by the things they're able to do

barely usable.[AskTog99] [HallOfShame99] As 'the interface hall of shame' has put it: "In an effort to achieve what some consider to be a more modern appearance, Apple has removed the very interface clues and subtleties that allowed us to learn how to use GUI in the first place. Window borders, title bars, window management controls, meaningful control labels, state indicators, focus indicators, default control indicators, and discernible keyboard access mechanisms are all gone." [HallOfShame99] Worse than that, at that point Apple has just repeated the same mistakes IBM has made with their RealThing software a few years earlier. Strike One.

20+-field forms – strike two

One thing which project stakeholders (especially in a commercial project) are notoriously bad with is requirements for more and more information. The Big Idea here is that you cannot possibly have too much information.

Let's see how a 'sign-up' or 'registration' form is usually designed in a medium-to-big company. First, business comes in and says, "We need to know a user name and e-mail". Then the marketing department adds, "Hey, we also need to know address, gender, and food preferences" (not specifying if it is 'gender' and 'food preferences', or 'gender preferences' and 'food preferences'). And last but not least, the legal department adds a dozen required fields such as 'legal name', 'domicile' (which nobody except them understands anyway), 'VAT number', and 'I hereby certify, under a penalty of perjury, that I am do not intend to perform terrorist acts using any of the web sites belonging to <insert organization name here>...'.

As a result, a simple sign-up form becomes a 20+-field monster, which literally scares the users away (in business terms, it is characterized by a 'drop-off rate'). It is amazing how many businesses don't realize how much harm can be done to their business by such a form. Just one example with A/B split testing is provided in [VWO12], and has shown that removing 3 fields from a sign-up form increased the number of customer registrations by 11% (!). It means that those 20+-field monster forms are effectively killing the very same departments which fight over the right to insert another field into sign-up form. At the very end, the approach described above will lead to a 50+-field form, and many-fold increase in number of people who're dealing with the statistics derived from this form; the only problem will be that there is no statistics, because there are no users.

Strike Two.

Windows 8 – Strike 3, stakeholders out

Windows 8 stakeholders have had their own Big Idea behind the new redesigned UI – to make the UI consistent between desktop and cellphone. And Windows 8 is actually not all bad – that is, if you have a laptop with a touchscreen. However, if you don't have a touchscreen (and 98+% people don't even now, over 2 years after the Windows 8 release) – it is an outright disaster. It is that bad that it is often compared to the 'New Coke' marketing disaster back in 1985, and that was a really bad one from a business perspective.

While the Windows 8 UI is a subject which is easy to write another five pages about, I feel that most of the readers have already formed their own opinion about it, so I won't go into floor-mopping with Windows 8 Metro UI once again (it has already been done on numerous occasions).

Strike Three, Stakeholders out.

New Coke

New Coke was the reformulation of Coca-Cola introduced in 1985 by The Coca-Cola Company to replace the original formula of its flagship soft drink, Coca-Cola (also called Coke).

The American public's reaction to the change was negative and the new cola was a major marketing failure.

What is to be done?

"What is to be done?"

~ The name of the novel by Nikolai Chernyshevsky

So, we've found that developers very rarely produce a good UI, and stakeholders, while having good ideas from time to time, are prone to certain very costly mistakes (which originally looked like The Next Big Thing).

At this point, our natural question is, "What can be done about it?" The answer is simple – you need to appoint somebody whose responsibility it is to advocate the end-user point of view.

Such a person needs information on "how usable our UI is" to do their job – and there are multiple sources for it. One such source of information can be the QA department (and they should be encouraged to file "usability defects"); another source of such information can be user forums (if any) and complaints (if you don't have them, your project is either very new, or is in deep trouble); and yet another source are the people using the software on a daily basis.

And as a last (but certainly not least) source of information about usability – you can (and I'd say, if you're targeting more than a 1000 customers, should) have an UX specialist on the team.

UX specialists – are they any good?

As usual, when you're faced with a suggestion to hire yet another specialist, there is a natural question – are they any good for the project? And as usual, it is not all black-and-white, and there are good UX specialists, and there are not so good ones.

I've worked with a few UX specialists and was amazed by the things they're able to do. A good UX specialist goes far beyond just trying to use software and saying, "Tsk-tsk, it is not good". And they go far beyond designing a usable UI based just on their own aesthetic perceptions.

Among the UX projects I've seen personally was a project optimizing software for a stock exchange. To do it, they took a control group (several traders in a real-world environment), and monitored them for several hours. This monitoring involved not only the distance of mouse travel (with relation to the operation being done), but also patterns of eye movements during the process. This information allowed not only optimal positioning of the buttons (which is related to mouse movements), but also

Funnel analysis

Funnel analysis involves using a series of events that lead towards a defined goal-from user engagement in a mobile app to a sale in an eCommerce platform. [Wikipedia2]

optimal positioning of the critical information (which is related to eye movements). While it wasn't immediately clear how much money this research has made for the company, it was clear that the software is an undisputed 'light years ahead' leader in terms of customer satisfaction.

Another project I've seen, was a much simpler one, aimed to optimize user sign-up process. As a result of the UX review, a funnel analysis, a few studies on a target group, some A/B split testing, and a few months of fighting with different departments, the number of fields on the form was reduced 3-fold, and the user drop off rate was reduced by 30% (ask your marketing department how much this is; for those who cannot ask them, a hint – it is HUGE).

Caveat emptor

I don't want to say that all those who claim they're UX specialists are good. In fact, there are many examples of their failures too. One thing to ask yourselves when hiring an UX specialist company is the following: do they perform any analysis of the target audience (with trials, split testing, etc.), or do they just have their own design ideas (without any objective justification for them)? Do they have a way to monitor user satisfaction (via trials, or surveys, or anything else to that effect), or they just make a design and then they're out of the picture? When you're dealing with the first type of folks, chances are they're good, but the second type can easily lead to an epic stakeholder-scale UI disaster.

Conclusions

In any project which has UI (and has more than 1 or 2 developers), you do need somebody to advocate end-user interests. It is very important to empower such a person to open bugs ('usability defects'), to assign reasonably high priority to these bugs, and to ensure they are fixed.

If you can afford a dedicated UX specialist, and can find a good one – they can make a Really Big Difference for your software (and to your bottom line too). ■

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

References

[AskTog99] 'Apple Quicktime Player 4.0 a Real Dud', Bruce Tongazzini, http://www.asktog.com/readerMail/1999-06ReaderMail.html

[HallOfShame99] 'quicktime 4.0 player', The Interface Hall Of Shame, 2011, http://hallofshame.gp.co.at/qtime.htm

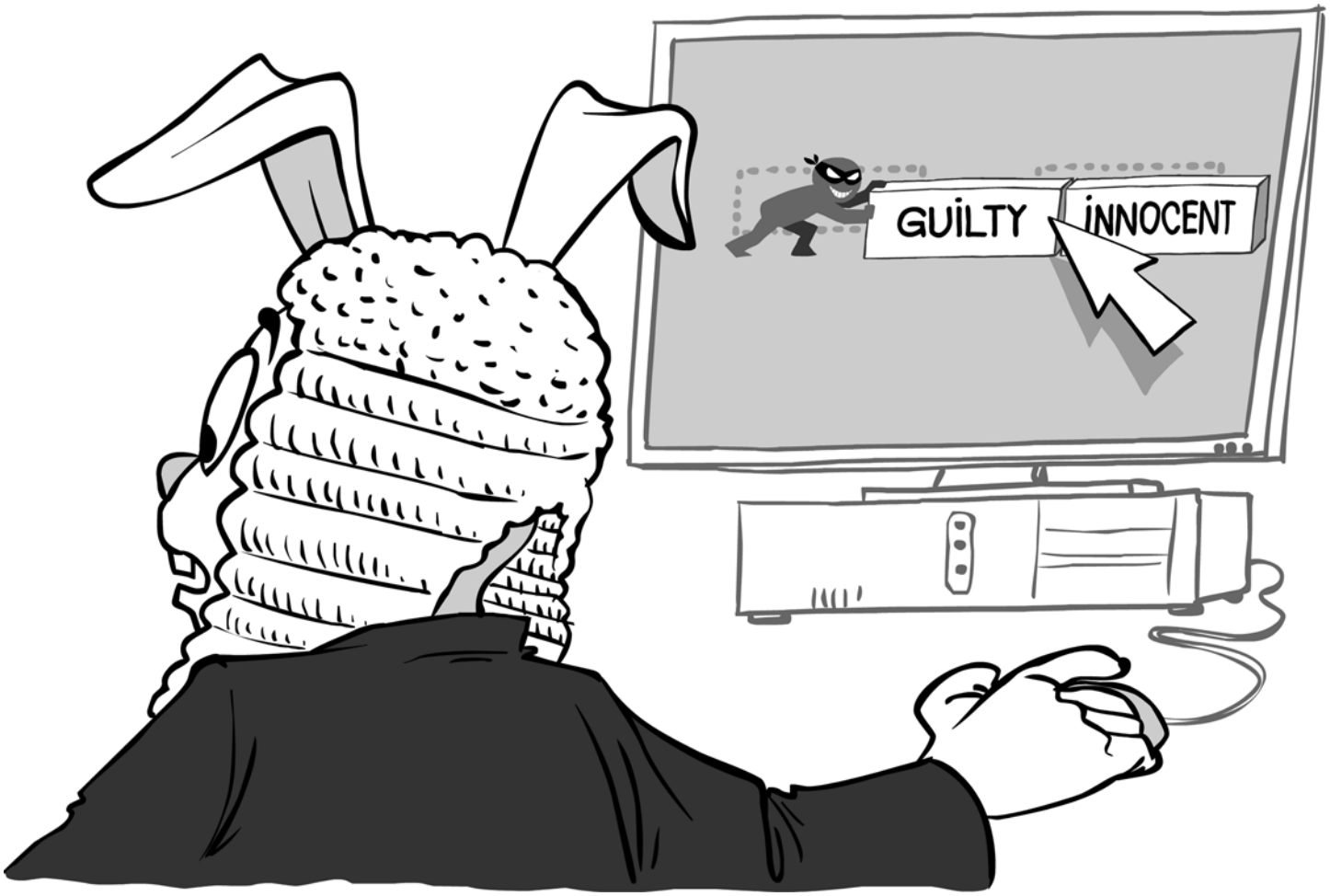
[Loganberry04] David 'Loganberry', Frithaes! – an Introduction to Colloquial Lapine!, http://bitsnbobstones.watershipdown.org/lapine/overview.html

[NoBugs11] Sergey Ignatchenko, 'The Guy We're All Working For', Overload #103.

[VWO12] 'Removing 3 form fields increases customer registrations by 11%', vwo.com, https://vwo.com/blog/ab-testing-form-fields-increase-conversions/

[Wikipedia1] https://en.wikipedia.org/wiki/Act_of_God

[Wikipedia2] https://en.wikipedia.org/wiki/Funnel_analysis



Alternatives to Singletons and Global Variables

Global variables and Singletons are usually considered bad. Bob Schmidt summarises some alternatives.

Recently, I posted what I thought was a simple question to the ACCU general forum:

> My current project has several objects that are instantiated by main, and then need to be available for use by other objects and non-member, non-friend functions [NMNF] at all levels of the call tree. They really do represent global functionality, determined at program startup, for which there will be only one instance per executable.
>
> Current best practices hold that global variables are bad, as are singletons. C++ does not support aspect-oriented programming. I don't want to have to pass these objects around in every constructor and non-member function call so that they will be available when and where they are needed.
>
> In the past I have used a global pointer to an abstract base class, and assigned the pointer to the instantiated derived class to this global pointer right after the object was instantiated in main. A similar approach would be to have a class that owns the pointer to the derived class, which can be retrieved through a static function. Having the globals be pointers to abstract base classes makes the classes that use the globals easy to test, because a test or mock object can be used in place of the real derived object. (One problem I have with Singleton and Monostate objects in this context is the direct tie to the concrete class.)
>
> My Google-fu has failed me in my search for alternatives. There are plenty of people out in the blogosphere willing to say that using globals and singletons is bad, but few offer any practical alternatives. I'm not against bending or breaking the rules – perhaps one of those 'bad' options really is the best way. Anyone have any suggestions? Am I over thinking this?

Motivation

My current customer has a legacy system whose real-time data acquisition components are mostly in C code. I have been advocating migrating to C++, and over the past several years I have written two large, stand-alone subsystems in C++. Interfaces to new field hardware provided an ideal opportunity to start using C++ more extensively, with the long-term goal of re-writing some, if not all, of the existing code.

The short term goals were more realistic: develop a basic framework for processes that conforms to the system's current overall architecture, and reuse existing components, wrapping the C code in C++ interfaces where required or desirable. A lot of the existing C functions will remain NMNF functions with C linkage.

The particular case that prompted my question is the system's existing inter-process communications (IPC) subsystem. The details aren't important (and are proprietary); the important fact for this discussion is that each process has only one interface to the IPC subsystem, and that interface is always initialized in main. Reads from the IPC subsystem are isolated to the main processing thread. The send functions are called from wherever they are needed, in the main processing thread (in response to received data), or in one or more threads that process data from the field device. I wanted to wrap the existing C code in a C++ interface. There have been discussions about replacing the existing IPC scheme with something else (such as WCF), and my goal was to create an interface that would allow any IPC scheme to be used via dependency injection.

The forum discussion

It turns out the question was not so simple after all.

I should point out that I had already done a lot of research on the subject. There are a lot of similar questions out there, and a seemingly unlimited number of people with an opinion. The vast majority of responses to these questions contained very similar answers, which boil down to 'global variables are bad, singletons are bad, don't do that'. OK, I knew that already. I was trying to find an idiomatic C++ alternative, one that didn't require that I pass one or more parameters to every constructor and NMNF function just because it might be needed by the next object constructed or the next NMNF function called.

The first several answers to my question seemed reasonable enough. They can be summarized as, yes, global variables and singletons are bad when abused, but there are some times and places when they solve a problem, and can be used in very specific and limited circumstances. (I'll call these people the pragmatists.) One respondent mentioned the SERVICE LOCATOR pattern, with which I was not familiar [Fowler]. It doesn't really solve my problem, but it is another tool in the kit.

It wasn't too long before the strict 'never use' opinion showed up, and was bolstered by several concurring opinions. (I'll call these people the purists.) Then the pragmatists returned, and a spirited debate was held. I was content to read the points and counter-points as they arrived; I had asked the question in order to be educated on the subject, and figured the best way to learn was to keep my eyes open and my mouth closed. As with most of these things, responses trailed off, and I was left with the email trail.

Points and counter-points

I'm not going to spend a lot of ink trying to summarize the points made by the two sides in this debate. The good people that participated in the debate made their cases, and I doubt I can do them justice in a short summary. If you are interested in all of the details you can read the entire thread online [ACCU].

What follows is a summary of the arguments presented in the thread, along with some commentary of my own. Fair warning – for the most part, I find myself in the pragmatic camp.

Bob Schmidt is president of Sandia Control Systems, Inc. in Albuquerque, New Mexico. In the software business for 33 years, he specializes in software for the process control and access control industries, and dabbles in the hardware side of the business whenever he has the chance. He can be contacted at bob@sandiacontrolsystems.com.

it's still repetitive, and a source of mental noise for those functions that don't need the information themselves but are simply passing it on down the call tree

Parameterize from Above (PfA)

The PFA pattern, introduced by Kevlin Henney in 'The PfA Papers' [Henney07a], was the only alternative presented as an answer to my question. Unfortunately, its implementation is the one thing I didn't really want to have to do – pass the object to every class and every NMNF function that needs it, or calls something that needs it. Currently I'm working in a code base where in some subsystems almost every routine has the same list of multiple parameters, and have experienced the joy of having to add a parameter to multiple function signatures in order to get a piece of data where I needed it, because the previous maintainer didn't think it would be required any further down the call tree.

Context Object Pattern

A context object aggregates multiple, repeated parameters into one structure, so there needs to be only one common PfA parameter passed amongst functions and classes [Henney07b]. I've used this pattern when refactoring code without knowing it was a named pattern (recall the comment above about my current project); passing one parameter is certainly easier to manage than passing many, but it's still repetitive, and a source of mental noise for those functions that don't need the information themselves but are simply passing it on down the call tree.

Defined dependencies

One reason given for using the PFA pattern is that it defines the dependencies of a class or NMNF function. I don't find this reason all that compelling. Not all objects used by a class or a function can or will be defined as a parameter. There are other ways we declare that module A (and its classes and/or functions) is dependent on something from module B – include files and forward declarations are two that immediately come to mind.

Testing

The SINGLETON and MONOSTATE patterns both deal with concrete objects, not abstract classes. The purists rightly point out that this makes code that use these patterns hard to test, because the functionality provided by the objects cannot be mocked. Using PFA, the argument goes, allows the concrete object to be passed from function to function as a pointer to the abstract class, allowing for the substitution of mock objects during testing. I agree with the goal, if not necessarily with the implementation.

Exceptions for logging

Logging is one of those cross-cutting concerns that aspect oriented program [IEEE] was designed to address. Unfortunately, C++ does not support the aspect oriented paradigm. Several purists said that they will sometimes make an exception for logging objects, and use a Singleton or a global, while others were adamant about never going down that path.

Order of initialization

Order of initialization issues can be tricky, particularly with `static` and `extern` variables spread over multiple compilation units. This hasn't been a problem for me in past situations where I have used global variables or SINGLETONS. The limited types of functionality provided by those objects made it possible to initialize or instantiate the objects in `main`, before any other code needed to be executed.

Multi-threading

SINGLETONS are problematic in multi-threaded environments when lazy initialization is used. The possibility that the SINGLETON's instance method can be called for the 'first' time by two threads simultaneously leads to a nasty race condition. The race condition is easily removed by calling the instance method in `main` prior to spawning any of the threads that use it. Instantiating an object in `main` and then passing it around using PFA eliminates the race condition in a similar manner.

Other than that one case, I can't see where PFA makes multi-threading any easier or less prone to error. An object shared by multiple threads still has to be thread-aware, regardless of the method used to get the object into the thread.

Use of cin, cout, cerr, and clog

The use of the standard C++ I/O streams was offered up by the pragmatists as an example of objects that represent global state and are not handled using PFA. One respondent replied that "Code using these is untestable" and in his classes he teaches his students to "only include `<iostream>` in the .cpp module where `main()` is defined and only use the globals there to pass them down the call chain. […] In all other modules only use `<istream>` and `<ostream>` which define the API but not the global objects (or `sstream` or `fstream`)."

Instance() method

Having to call an instance method to return a pointer or a reference to an object, instead of just instantiating the object, is an awkward part of the SINGLETON pattern. I don't think this, by itself, is a sufficient reason to reject the use of the pattern, but it does add to the negative pile.

Introducing the Monostate NVI Proto-Pattern

Listings 1 through 5 contain my initial solution to the problem. In my one additional contribution to the forum thread I called it (with tongue firmly in cheek) "… a cross between the Monostate pattern and the template method pattern and/or Herb Sutter's Non-Virtual Interface Idiom, with a little pimpl added for good measure." [Sutter01] The version presented here is refined from that initial attempt, and (hopefully) fixes the typos.

Listing 1 is a simple abstract base class, and Listing 2 shows a class derived from the base class. This is all standard stuff. The examples are extremely simple, since complexity here wouldn't add anything to the discussion.

Listing 3 shows the new class. Its primary characteristic is a pair of constructors – one default, and one that takes a shared pointer to the abstract base class. The constructor that takes the parameter is used to tie the concrete derived object to the MONOSTATE NVI container. The default constructor is used to gain access to the derived object. The class contains

Interfaces are not supposed to change often. When they do change, you have to modify all of the derived classes to match the new interface

```
class abstract_base
{
public:
 abstract_base ()
 {
 }
 virtual ~ abstract_base ()
 {
 }
 virtual void foo () = 0;
};
```

Listing 1

inline, non-virtual functions that call the virtual functions defined by the abstract base class interface. Because the non-virtual functions are inline, the function call is removed by the compiler, with just the virtual function call remaining.

Listing 4 illustrates how the concrete object is created and tied to the Monostate NVI object. Listing 5 is an example of a function that uses the combined objects. The call to `nvi.foo()` in listing 5 calls `foo()` against the object `p` instantiated in `main`.

This new class is not a SINGLETON; it does not create the object, and does not have an `instance()` method. It looks a little like a MONOSTATE; it maintains a shared pointer to the resource being managed.

I see several advantages to this solution. First, unlike PFA, I don't have to worry about passing this information around. Second, like the MONOSTATE pattern, the object is accessed through a standard constructor. Third, the proto-pattern accesses objects through their abstract base class interface, making it easy to mock the object for testing.

One disadvantage of this solution is having to maintain the extra class. I don't consider this a big disadvantage. Interfaces are not supposed to change often. When they do change, you have to modify all of the derived classes to match the new interface. Under normal circumstances this requires *N* changes; this proto-pattern bumps that up to *N*+1. A bigger disadvantage is the lack of compiler support that indicates that the extra class needs to be changed in response to a change in the interface. Presumably, if the interface is changing, some code somewhere is going

```
class concrete_derived : public abstract_base
{
public:
 concrete_derived ()
   : abstract_base ()
 {
 }
 ~concrete_derived ()
 {
 }
 virtual void foo () override
 {
  // DO SOMETHING USEFUL
 }
};
```

Listing 2

```
class mono_nvi
{
public:
  explicit mono_nvi
    ( std::shared_ptr< abstract_base > p )
  {
    if ( p == nullptr )
      throw ( something );
    if ( mp != nullptr )
      throw ( something_else );
    mp = p;
  }
  mono_nvi ()
  {
    if ( mp == nullptr )
      throw ( something );
  }
  inline void foo ()
  {
    mp->foo ();
  }
private:
  static std::shared_ptr< abstract_base > mp;
};
std::shared_ptr< abstract_base > mono_nvi::mp;
```

Listing 3

```
int main ( int argc, char* argv[] )
{
  std::shared_ptr< abstract_base > p
    ( new concrete_derived );
  mono_nvi nvi ( p );
  top_layer_function ();
}
```

Listing 4

```
void nested_function ( void )
{
  mono_nvi nvi;
  nvi.foo ();
}
```

Listing 5

```
template< class T >
class mono_nvi_template
{
public:
  explicit mono_nvi_template
    ( std::shared_ptr< T > p )
  {
    // SAME AS IN LISTING 3.
  }
  // DEFAULT CONSTRUCTOR AND FUNCTIONS DEFINED
  // THE SAME AS IN LISTING 3.

private:
  static std::shared_ptr< T > mp;
};
template< class T > std::shared_ptr< T >
  mono_nvi_template< T >::mp;
```

Listing 6

to call the new or modified function, prompting a change or addition to the extra class.

But what about extensibility?

At one point during the project that prompted this whole discussion, a new requirement was discussed: the program would use one derived class object to communicate with X, and another, different derived class object to communicate with Y. Coincidentally, during this article's early review process one of the reviewers wrote: "One question it might be worth adding in, if Bob hasn't already got it listed, is whether the design allows for future change; for example you start with a requirement for one 'X' and then later on you need two of them..." It was if someone was reading my mind. Spooky.

That requirement didn't survive, but the question of how this might be accomplished remained. My first thought was to use templates, which presents a problem of its own. I'm not a strong template programmer. Most of what I do doesn't require that level of generality, so the templates I have created tend to be very straightforward. So, full disclosure – it is likely that the templates presented here are not idiomatically fully formed. (There are no associated traits classes, for example.)

My first attempt at a template solution is shown in Listing 6. This version allows multiple instances of proto-pattern objects to exist, as long as the types used in the template specialization are different. That is one weakness – the types need to be different.

```
template< int T >
class mono_nvi_template
{
public:
  explicit mono_nvi_template
    ( std::shared_ptr< abstract_base > p )
  {
    // SAME AS IN LISTING 3.
  }
  // DEFAULT CONSTRUCTOR AND FUNCTIONS DEFINED
  // THE SAME AS IN LISTING 3.

private:
  static std::shared_ptr< abstract_base > mp;
};

template< int T > std::shared_ptr< abstract_base >
  mono_nvi_template< T >::mp;

typedef  mono_nvi_template < 1 >  mono_nvi_one;
typedef  mono_nvi_template < 2 >  mono_nvi_two;
typedef  mono_nvi_template < 3 >  mono_nvi_three;
```

Listing 7

```
int main ( int argc, char* argv[] )
{
  std::shared_ptr< abstract_base > p1
    ( new concrete_derived_1 );          // NOTE
  std::shared_ptr< abstract_base > p2
    ( new concrete_derived_2 );          // THE
  std::shared_ptr< abstract_base > p3
    ( new concrete_derived_2 );          // TYPES

  mono_nvi_one   mnvi1 ( p1 ); // THESE ALL REFER
  mono_nvi_two   mnvi2 ( p2 ); // TO DIFFERENT
  mono_nvi_three mnvi3 ( p3 ); // OBJECTS OF TWO
                               // DIFFERENT TYPES
  top_layer_function ();
}
```

Listing 8

```
void nested_function ( void )
{
  mono_nvi_one mnvi1; // REFERS TO p1 IN LISTING 8
  mono_nvi_two mnvi2; // REFERS TO p2 IN LISTING 8
  mono_nvi_three mnvi3; // REFERS TO p3
                        // IN LISTING 8
  // ETC.
}
```

Listing 9

This led to the code in Listing 7. I had no idea if this was idiomatic or not, but it worked. It looks ugly, but I find most template code to be at least mildly unattractive. The **typedef**s at the end of Listing 7 make the usage of the template easier. (In real life I would have used enumerations instead of the magic numbers.) Listing 8 illustrates how we can now create multiple objects of the same or differing types. Listing 9 shows how the new objects are used.

At this point the article was submitted for another round of reviews. The reviewers pointed out that the way I was using the integer to specialize the template was not, in fact, idiomatic. I was pointed in the direction of *tag dispatching*, the use of empty structs whose purpose is to provide a type-safe name as a template parameter. The reviewers also recommended using **std::make_shared** to create the object and a shared pointer to it in one step [Meyers].

Listing 10 shows the class template modified to use tag dispatching. It features two template parameters. The first typically will be the abstract base class. The second, when the default is not used, is the tag that allows two objects of the same type **T**. Listing 11 contains examples of creating three distinct objects, similar to those created in listing 8.

```
template< class T, class TAG = void >
class mono_nvi_template
{
public:
  explicit mono_nvi_template
    ( std::shared_ptr< T > p )
  {
    // SAME AS IN LISTING 3.
  }
  // DEFAULT CONSTRUCTOR AND FUNCTIONS
  // DEFINED THE SAME AS IN LISTING 3.
private:
  static std::shared_ptr< T > mp;
};
template< class T, class TAG >
  std::shared_ptr< T > mono_nvi_template
    < T, TAG >::mp;
```

Listing 10

```
struct mono_nvi_two {};   // THESE ARE THE TAGS
struct mono_nvi_three {};

mono_nvi_template < abstract_base > mnvi1
  ( std::make_shared< concrete_derived_1 > () );
mono_nvi_template
  < abstract_base, mono_nvi_two > mnvi2
  ( std::make_shared< concrete_derived_2 > () );
mono_nvi_template
  < abstract_base, mono_nvi_three > mnvi3
  ( std::make_shared< concrete_derived_2 > () );
```

Listing 11

Wrap-up

My original solution was satisfactory; it provided the ease-of-use and testability I was looking for. (This is the format of the solution used in the first iteration of my client's production code.) The final template version, prompted by an abandoned requirement and an astute reviewer (thank you), with further refinements provided by several reviewers, provides a more flexible solution. ■

Acknowledgements

I would like to thank all of you who participated in the thread. In the order in which you made your first comments, you are: Fernando Cacciola, Anna-Jayne Metcalfe, Alison Lloyd, Balog Pal, Pete Barber, Daire Stockdale, Aaron Ridout, Jonathan Wakely, Russel Winder, Thomas Hawtin, Giovanni Asproni, Martin Moene, Andrew Sutton, Kris, Paul Smith, Peter Sommerlad, and Hubert Mathews. Collectively you deserve credit for anything I got right this month. Any mistakes I made are my own.

As always, thanks also to Fran and the reviewers. This is my first attempt at writing about a technical subject, with real code that needed to compile correctly, and their encouragement and input were invaluable. As Fran stated in her article last month, "(the reviewers) might be able to give a few pointers […] or other ways of doing things." [Buontempo15] I certainly learned several new ways of doing things, and for that I am grateful.

Thanks also to Michael Chiew and Larry Jankiewicz, who provided feedback during this idea's early development.

An opposing opinion

One reviewer pointed out that this solution is still a global in disguise, with the usual downsides (I agree). He or she asked the rhetorical question, is it that much better than a simple global with **get/set** to do the checking?

```
unique_ptr< abstract_base > global_base;

void set_base
  ( unique_ptr< abstract_base >  new_base )
{
  global_base = new_base;
  if ( global_base == nullptr )
      throw ( something );
}

abstract_base& get_base ( void )
{
  if ( global_base == nullptr )
    throw ( something );
  return *global_base;
}

void  using_function ( … )
{
  get_base ().foo ();
}
```

On the plus side the reviewer noted that my solution allows for substitutability and better controlled access than a global, and gets closer to having a template generate a lot of the boiler-plate.

One issue I see with this approach is one that the SINGLETON has – a non-standard way of getting the object. In this case, it's a call to **get_base()**, as opposed to the **instance()** static member function common to SINGLETONS.

Corrections

There is an error in the print edition of my article in *Overload* 125, 'I Like Whitespace'. The error was discovered by Martin Moene while he was preparing the article for the online edition. I'll let him describe what he found (from his email to me):

> "As web editor, I already have seen Overload 125 with your article 'I Like Whitespace'. In it you have the [example at the bottom of the right-hand-column on page 14] featuring a 'dangling else'. To me there's a cognitive disconnect in the corrected version between function name **process_x_is_0** and value of **x** for which it is invoked **(!0)**. I.e. the non-braced version does what it says, whereas the second does not. (In C and C++, **else** is associated with the nearest **if**.)"

Martin is, of course, correct. My example was in error. The name of the function called in the dangling **else** should have been **process_x_is_not_0**. The online version of the code has been corrected. My thanks to Martin for discovering the error and publishing the corrected version online, and Alison Peck for the extra work she performed supplying the corrected version to Martin.

There also is a typo (yeah, I'm going with typo) in the complex Boolean expression at the top of the left-hand column on page 14 – an open parenthesis is missing before the subexpression **z == 6**. This was pointed out to me by astute reader Jim Dailey, who also shared his preferred style for messy tests:

```
if (     (     ( x == 0 )
           && ( x == 1 )
         )
      || (     ( y == 3 )
           && (     ( z == 5 )
                || ( z == 6 )
              )
         )
   )
```

I thank Jim for pointing out my error, and sharing his style.

I regret the errors and any confusion they might have caused.

Bob

References

[ACCU] accu-general Mailing List, http://lists.accu.org/mailman/private/accu-general/2015-January/046003.html

[Buontempo15] Buontempo, Frances, 'How to Write an Article', *Overload* 125, February 2015

[Fowler] Fowler, Martin, 'Inversion of Control Containers and the Dependency Injection Pattern', http://martinfowler.com/articles/injection.html#UsingAServiceLocator

[Henney07a] Henney, Kevlin, 'The PfA Papers: From the Top', *Overload* 80, August 2007

[Henney07b] Henney, Kevlin, 'The PfA Papers: Context Matters', *Overload* 82, December 2007

[IEEE] Various authors, *IEEE Software*, January/February 2006, vol. 23

[Meyers] Meyers, Scott, *Effective Modern C++*, O'Reilly, Item 21, p. 139

[Sutter01] Sutter, Herb, 'Virtuality', *C/C++ Users Journal*, 19(9), September 2001

Variadic and Variable Templates

C++11 and 14 offer new features for Variadic and Variable templates. Peter Sommerlad showcases the compile-time possibilities they offer.

C++11 introduced Variadic Templates and **constexpr** that ease and allow type-safe computations at compile time. In combination with the C++14 mechanism of Variable Templates, which actually define constants, there are unprecedented possibilities for compile-time computations.

This article not only shows the mechanisms available but also demonstrates a non-trivial example, how they can be used to compute interesting data at compile time to be put into ROM on an embedded device, for example.

Introduction

C++ templates have allowed compile-time meta-programming for some time now. However, with C++03 many interesting applications require herculean efforts to achieve results using class-template specializations and function template overloads with variable number of template arguments. Getting such code using variable number of template arguments right is very tedious in the C++03 landscape and even a tiny mistake can produce horrific compiler error messages which are hard to trace back to the origin of the error. Any user of some of the Boost libraries that make heavy use of template meta-programming, such as **boost::spirit** or **boost::mpl** can sing that song. [Boost]

However, the variadic templates introduced with C++11 make things much more comfortable at the language level. **<type_traits>** for meta programming were even further improved in C++14. In addition to many more traits, C++14 introduced template aliases for each trait with a suffix **_t** that allow us to rid the template code of many uses of the **typename** keyword when working with traits. Also new with C++14 come variadic lambdas, that allow us to use **auto** as the type for a lambda's parameters, so that their type can be deduced from the calling context of the lambda. Another recent change are the relaxed rules for type deduction, so that lambdas and **auto** return type functions can be specified without a trailing return type, even in the case of multiple return statements. It is only when multiple returned expressions differ in their type that one needs to specify a return type explicitly.

In addition to increased possibilities with lambdas and return type deduction, many of the limitations on C++11 **constexpr** functions have also been relaxed. In the future, you might see many uses of '**constexpr auto**' functions that do interesting compile-time computations. Some are shown later.

Finally, variable templates, which are actually parameterized compile-time constants, make the concept of templates more symmetric across the language.

Prof. Peter Sommerlad is head of IFS Institute for Software at FHO/HSR Rapperswil where he inspired the Cevelop C++ IDE (www.cevelop.com). Peter is co-author of the books *POSA Vol.1* and *Security Patterns*. His goal is to make software simpler by Decremental Development: Refactoring software down to 10% its size with better architecture, testability and quality and functionality.

```
#ifndef PRINTLN_H_
#define PRINTLN_H_
#include <ostream>
// base case overload
void println(std::ostream &out){
  out <<'\n';
}
// variadic template
template <typename HEAD, typename ... T>
void println(std::ostream & out,HEAD const &h, T
const & ... tail){
  out << h; // cut off head
  if (sizeof...(tail)){
    out <<", ";
  }
  println(out,tail...); // recurse on tail...
}

#endif /* PRINTLN_H_ */
```

Listing 1

As a library component, **std::tuple** extends the idea of **std::pair** to arbitrary collection of values of arbitrary types and **std::integer_sequence** eases the writing of code employing such lists of values.

With so much stuff, you might ask, how does that help a 'normal' programmer and how should I employ these. The rest of this article will show you some applications that are useful in day-to-day work or for embedded code employing modern compilers.

Variadic templates with typename parameters (C++11)

Whoever has been bitten by the lack of type-safety of **printf()** might employ a variadic template solution to create a type-safe **println** function. Recursion is the key to defining successful variadic template functions and makes using classical **...** varargs parameters in C++ mostly obsolete. (See Listing 1.)

A variadic template introduces a so-called 'template parameter pack' by placing three dots (ellipsis) after the typename parameter introduction. Using the template parameter pack's name (**T**) to define a function parameter creates a parameter pack (**tail**). The name of the parameter pack (**tail**) can later be used within the template to denote a so-called pack-expansion, where the three dots are placed after an expression using the parameter pack name. The corresponding expression is then repeated for each concrete argument. In our **println** example, even while the base case is not really called, an empty **tail(sizeof...(tail)==0)** would not call **println()**, it is necessary to make the code compile. As you might have guessed the **sizeof...** operator gives the number of elements in a parameter pack. It is also applicable on a template parameter pack name.

lambdas and auto return type functions can be specified without a trailing return type

Variable templates basics (C++14)

In C++, it has always been possible to define constants that were dependent on template arguments using static data members of a class template. To make the language with respect to templates more symmetric and for constants depending on template arguments, C++14 introduced variable templates, which can even be variadic, by the way.

The canonical example from the C++ standard [ISO/IEC] is the definition of pi for any kind of numerical type that looks like the following:

```
template<typename T> constexpr T pi
  = T(3.14159265358979323846264338327L);
```

This allows `pi<float>` or `pi<double>` to be computed at compile time and used as such without casting the value. Note that the number of digits given as a long double value are sufficient up to the precision long double allows on today's platforms. You can even write `pi<complex<double>>` to obtain the complex constant with pi as the real part.

If you ever need to calculate with a full circle `two_pi` might also be defined accordingly:

```
template<typename T> constexpr T two_pi
  =pi<T>+pi<T>;
```

While the example of Pi might not be very impressive, take a look at the examples given later, where whole table computations are hidden behind the initialization of a template variable.

As a more interesting helper, we implement the conversion of degrees to radian values at compile time, using our `pi<T>` constant. Because degrees, minutes and seconds can be given as integral values, we can implement that using a variable template as well:

```
template <short degrees, short minutes=0,
  short seconds=0>
constexpr long double
rad{(degrees+minutes/60.0L+seconds/3600.0L)
  *pi<long double>/180.0L};
static_assert(pi<long double>/2 == rad<90>,
  "90 degrees are pi half"); // test it
```

Variadic templates with non-type parameters and std::integer_sequence (C++11/14)

In addition to typename parameter packs, C++11 and later also allow parameter packs of non-type template parameters. The usual restrictions on non-type template parameters apply and all arguments of a non-type template parameter pack have to have the same type.

C++14 introduced `std::integer_sequence<typename T,T ... elts>` to represent such sequences of integers or indices with `std::index_sequence<size_t ...>` as different types at compile time. A companion factory function `make_index_sequence<size_t n>()` creates an `index_sequence` with the numbers up to `n`.

The following example shows how such an `index_sequence` can be used to create a `std::array` with `n` elements of type `size_t` is initialized-potentially at compile time-with multiples of parameter `row` (1 to n times):

```
template <size_t...I>
constexpr auto
make_compile_time_sequence(size_t const row,
   std::index_sequence<I...>){
  return std::array<size_t,sizeof...(I)>{
    {row*(1+I)...}};
}
constexpr auto
   array_1_20=make_compile_time_sequence(1,
   std::make_index_sequence<20>{});
```

Please excuse the complication of the additional parameter row, but you will see later that we will use that to construct different rows of a multiplication table. For example, `make_compile_time_sequence 10,std::make_index_sequence<10>{})` will create an array with the values 10, 20, 30,... 100. That will be the last row in a multiplication table from 1 times 1 up to 10 times 10.

While it is quite easy to convert the parameter pack to values, using pack-expansion, it is impossible to use a function parameter as a template argument within a `constexpr` function. This hurdle makes some applications a bit of a burden. However, as the rules for `constexpr` functions are also relaxed, there is less need for such variadic template machinery to ensure compile-time computation of tables.

As a-slightly unnecessary-complicated example the following code shows how to compute a multiplication table at compile time.

```
template <size_t...I>
constexpr
auto make_compile_time_square
  (std::index_sequence<I...> ){
    return std::array<std::array<size_t,
      sizeof...(I)>,sizeof...(I)>
           {{make_compile_time_sequence(1+I,
             std::make_index_sequence
             <sizeof...(I)>{})...}};
}
```

The pack expansion actually will generate a row in the table for each value the parameter pack I takes. With that, we can create a complete multiplication table from 1*1 to 20*20 with just a single declaration in the 2-dimensional array constant `multab_20` at compile time:

```
constexpr auto multab_20 =
   make_compile_time_square(
   std::make_index_sequence<20>{});
```

The corresponding test code will output the multiplication table from the constant `multab_20` (see Listing 2). I even implemented a version that uses `std::integer_sequence<char,char ...>` to create the multiplication table as a string constant at compile time. But the code is not as nice as I would like it to be. There is work on the way to ease compile-time string processing in a similar way and a means (already implemented by g++ and clang) to create a `char_sequence<char ...>` from a

it is impossible to use a function parameter as a template argument within a constexpr function

```
void testCompileTimeArray(std::ostream &out){
  using namespace std;
  for_each(begin(multab_20),end(multab_20),
           [&out](auto row){
    out << '\n';
    for_each(begin(row),end(row),[&out](auto elt){
      out << setw(4) << elt;
    });
  });
  out << '\n';
}
```

Listing 2

regular string literal using a user-defined literal template operator that might be standardized for C++17.

More 'ROMable' data

Let us conclude with an example of a compile-time computed table of sine values to enable a quick lookup-and-interpolation-based implementation of a sine function for an embedded system.

To build such a table, we first need a compile-time constexpr version of `std::sin(double)`. This can be implemented using a Tailor-series that converges quickly [Wikipedia]. It can be used independently from the table to create individual sine values at compile time. A run-time use is not recommended, because it will definitely be inferior to `std::sin(x)`.

The code starts first with some scaffolding to implement the tailor series development of the sine value of x. (See Listing 3.)

With that quite slow `sin()` function in place we can start implementing more. Using the tricks we learned from our multiplication table we can now implement a compile-time lookup table for the sine values for each degree from 0..360 as in Listing 4.

Listing 5 contains some compile-time tests of our sine table to show that the table is really ROMable using only 5 values.

And Listing 6 is our compile-time table from 0 to 360 degrees of a circle.

What is still missing from the standard

As of C++14 many standard library functions and some types are not yet as compile-time usage friendly. For example, `std::array` is a literal type, but it can not be incrementally constructed in a `constexpr` function. A replacement for the time being is cloning `std::array` and adding `constexpr` to all member functions. The keyword `constexpr` was only added to the `const`-member functions, because these were the only useful positions with C++11's restrictions and nobody recognized the usefulness for C++14 of also having the non-`const` member functions declared as `constexpr`.

Also, the standard library's non-modifying algorithms and may be even some of the modifying algorithms could be used in more elaborate `constexpr` functions, if they would be declared as `constexpr`.

```
// sin(x) = sum (-1)^n*(x^(2*n+1))/(2n+1)!
namespace tailor {
template<typename T>
constexpr T pi = T(3.14159265358979323846264433L);
namespace detail{
constexpr long double fak(size_t n) {
  long double res = 1;
  for (size_t i = 2; i <= n; ++i) {
    res *= i;
  }
  return res;
}

constexpr long double sin_denominator
   (long double x, size_t n) {
  long double res{ x }; // 1 + 2n
  for (size_t i = 0; i < n + n; ++i) {
    // naive, could be log(n), but n<20
    res *= x;
  }
  return res;
}

template<typename T>
constexpr T two_pi =2.0l*pi<T>;

constexpr
long double adjust_to_two_pi(long double x) {
  while (x > two_pi<long double> ) {
    x -= two_pi<long double>;
  } // very naive... not for run-time use
  while (x < -two_pi<long double> ) {
    x += two_pi<long double>;
  }
  return x;
}
} // detail
constexpr long double sin(long double x) {
  long double res = 0;
  x = detail::adjust_to_two_pi(x); // ensures
                                   // convergence
  for (size_t n = 0; n <= 16; ++n) {
    long double const summand
    {detail::sin_denominator(x, n)
     / detail::fak(2 * n + 1)};
    res += n % 2 ? -summand : summand;
  }
  return res;
}
}
```

Listing 3

learn how to use variadic templates, since these are reasonable and can simplify template code significantly

```
namespace tables {
template <typename T, size_t ...indices>
constexpr auto
make_sine_table_impl
  (std::index_sequence<indices...>){
  static_assert(sizeof...(indices)>1,
      "must have 2 values to interpolate");
  return std::array<T,sizeof...(indices)>{{
    sin(indices*two_pi<T>
    /(sizeof...(indices)-1))...
  }};
}
template <size_t n, typename T=long double>
constexpr auto make_sine_table=
    make_sine_table_impl<T>
    (std::make_index_sequence<n>{});
```

Listing 4

```
constexpr auto testsinetab=tables::make_sine_table<5,long double>;
static_assert(testsinetab[0]==0.0, "sine 0 is 0");
static_assert(abs(testsinetab[2])<1e-10, "sine pi is 0");
static_assert(abs(testsinetab.back()) <1e-10, "sine two pi is 0");
static_assert(abs(testsinetab[1]-1.0)<1e-10, "sine pi/2 is 1");
static_assert(abs(testsinetab[3]+1.0)<1e-10, "sine pi+pi/2 is -1");
```

Listing 5

```
constexpr auto largesinetab
  =tables::make_sine_table<360+1,double>;
// limited to 1 entry per degree,
// if not giving compiler argument:
//   -fconstexpr-steps=larger

// check it:
static_assert(largesinetab.front()==0,
  "sine 0 is 0");
static_assert(abs(largesinetab.back())
  <1e-12,"sine 2 pi is 0");
```

Listing 6

However, there is some tension, since some algorithms might be more efficiently implemented as run-time versions where the limitations of `constexpr` don't apply.

A final missing piece are string literals and compile time computation of string values. Work has started on these things and you should expect corresponding compiler and library support for the next standard C++17 making compile time computation still more powerful, allowing even more ROMable data being computed in C++ at compile time.

However, interpreting C++ at compile time is slowing your compiles, and current compilers (clang, g++) will give a strict limit to the number of computations allowed, so to be able to detect endless recursion or endless loops. These limits usually allow for a compile time of single file to be within a minute or a couple of minutes and it can be easily reached. For example, I can create my sine table for 360 degrees, but not per minute or a quarter of a degree, because of the default compiler limits, and even then the compile time is clearly recognizable. You don't want to include such a header in more than one compilation unit, otherwise we get compile times in days rather than minutes.

So compile-time `constexpr` computation is a powerful tool in modern C++ to create ROMable data and relieve small processors from the burden of some computation at run time. But it is also a potentially expensive thing that might slow your development, if you try too complicated things at compile time giving people again a reason to complain how slow C++ compiles. But as of today, that won't be only the fault of the compiler, but of the developer pushing it to its limits. So use this powerful feature wisely.

Nevertheless, learn how to use variadic templates, since these are reasonable and can simplify template code significantly, especially in a cases where you'd like to use template template parameters, but that might be a story for a future article. ■

References

[Boost] Boost Libraries, http://www.boost.org;

Boost Spirit, http://www.boost.org/doc/libs/1_57_0/libs/spirit/doc/html/index.html;

Boost MPL, http://www.boost.org/doc/libs/1_57_0/libs/mpl/doc/index.html;

both accessed April 5th 2015

[ISO/IEC] *ISO/IEC International Standard 14882*, Fourth edition 2014-12-15, Information technology – Programming languages – C++

[Wikipedia] Sine Tailor Series definition; Wikipedia, http://en.wikipedia.org/wiki/Sine#Series_definition, accessed December 1st 2014

Example source code

The example source code is available on Github: https://github.com/PeterSommerlad/Publications/tree/master/ACCU/variadic_variable_templates

Resource Management with Explicit Template Specializations

RAII is a useful idiom. Pavel Frolov presents a powerful extension using explicit template specialization.

RAII is one of the most important and useful C++ idioms. RAII efficiently relieves the programmer of manual resource management and is a must for writing exception-safe code. Perhaps, the most ubiquitous usage of RAII is dynamic memory management with smart pointers, but there are a plenty of other resources for which it can be applied, notably in the world of low-level libraries. Examples are Windows API handles, POSIX file descriptors, OpenGL primitives, and so on.

Applying RAII: available options

There are several choices we could make when deciding to implement an RAII wrapper for a resource of some kind:

- write a specific wrapper for that particular resource type;
- use a standard library smart pointer with custom deleter (e.g., `std::unique_ptr<Handle, HandleDeleter>`);
- implement a generic one ourselves.

The first option, writing a specific wrapper, may seem reasonable at the beginning, and in fact, is a good starting point. The simplest RAII wrapper may look something like Listing 1.

However, as your code base grows in size, so does the number of resources. Eventually you'll notice that most of resource wrappers are quite similar, usually the only difference between them is the clean-up routine. This causes error-prone copy/paste-style code reuse. On the other hand, this is a great opportunity for *generalization*, which leads us to the second option: smart pointers.

The smart pointer class template is a generic solution to resource management. Even so, it has its own drawbacks, which we will discuss shortly. As their name suggests, smart pointers were designed mainly for memory management and their usage with other kinds of resources is often at least inconvenient. Let's look at the smart pointer option in more detail.

Why smart pointers are not smart enough

Consider the code in Listing 2.

Why is the `ScopedHandle` constructor expecting an argument of type `void**`? Recall, that smart pointers were designed primarily for pointer types: `std::unique_ptr<int>` actually manages `int*`. Similarly `std::unique_ptr<Handle>` manages `Handle*` which is an alias for `void**` in our example. How can we work around this? First, we could use the `std::remove_pointer` metafunction:

```
using ScopedHandle =
  std::unique_ptr<std::remove_pointer_t<Handle>,
  HandleDeleter>;
```

Pavel Frolov is a software developer from Moscow. His experience includes SCADA software development for space ground-based infrastructure, namely, Angara Space Rocket Complex and Land Launch projects. He is currently working at Positive Technologies on an innovative automated malware analysis system. He can be contacted at ark.fps@gmail.com

```
class ScopedResource {
public:
  ScopedResource() = default;
  explicit ScopedResource(Resource resource)
    : resource_{ resource } {}
  ScopedResource(const ScopedResource&) = delete;
  ScopedResource& operator=
    (const ScopedResource&)
    = delete;
  ~ScopedResource() { DestroyResource(resource_);
}
operator const Resource&() const {
  return resource_; }

private:
  Resource resource_{};
};
```

Listing 1

Second, we could use an obscure feature of the smart pointer deleter: if there exists a nested type named `pointer`, then this type is used by `unique_ptr` as a managed pointer type:

```
struct HandleDeleter {
  using pointer = Handle;
  void operator()(Handle h) { CloseHandle(h); }
};
using ScopedHandle = std::unique_ptr<Handle,
                       HandleDeleter>;
```

As you can see, neither of these solutions is as user-friendly as we want them to be, but the main problem is another. Smart pointer forces you to make assumptions about `Handle` type. But handle is meant to be an

```
#include <memory>

// From low-level API.
using Handle = void*;
Handle CreateHandle() { Handle h{ nullptr };
  /*...*/ return h; }
void CloseHandle(Handle h) { /* ... */ }

struct HandleDeleter {
  void operator()(Handle h) { CloseHandle(h); }
};
using ScopedHandle = std::unique_ptr<Handle,
                       HandleDeleter>;
int main() {
  // error: expected argument of type void**
  ScopedHandle h{ CreateHandle() };
}
```

Listing 2

```
#include <memory>
using Handle = int;
Handle CreateHandle() {
  Handle h{ -1 }; /*...*/ return h; }
void CloseHandle(Handle h) { /* ... */ }

struct HandleDeleter {
  using pointer = Handle;
  void operator()(Handle h) { CloseHandle(h); }
};
using ScopedHandle = std::unique_ptr<Handle,
                     HandleDeleter>;
int main() {
  // Error: type mismatch: "int" and
  // "std::nullptr_t".
  ScopedHandle h{ CreateHandle() };
}
```

Listing 3

opaque descriptor, the actual definition of handle is an implementation detail of which the user is not required to be aware.

There is another, more serious problem with smart pointer approach (see Listing 3).

In practice, the code above may work without problems with some of **`std::unique_ptr`** implementations, but in general this is not guaranteed and definitely is not portable.

The reason for an error in this case is a violation of the **`NullablePointer`** *concept* [NullablePointer] by the managed type. In a nutshell, the *model* of the **`NullablePointer`** concept must be pointer-like type, comparable to **`nullptr`**. Our **`Handle`**, defined as an alias to **`int`**, is no such thing. As a consequence, we can't use **`unique_ptr`** for something like POSIX file descriptors or OpenGL GLuint handles.

There is a workaround, though. We can define an adaptor for **`Handle`** which fulfils the requirements of **`NullablePointer`**, but writing *a wrapper for a wrapper* is way too much.

Yet another smart pointer issue is related to convenience of use. Consider idiomatic usage of a hypothetical **`Bitmap`** resource (Listing 4).

Now compare this with the usage of **`std::unique_ptr`** for managing **`Bitmap`** (Listing 5).

```
// Graphics API.
bool CreateBitmap(Bitmap* bmp) {
  /*...*/
  return true;
}

bool DestroyBitmap(Bitmap bmp) {
  /* ... */
  return true;
}

bool DrawBitmap(DeviceContext ctx, Bitmap bmp) {
  /* ... */
  return true;
}
...

// User code.
DeviceContext ctx{};
Bitmap bmp{};
CreateBitmap(&bmp);
DrawBitmap(ctx, bmp);
```

Listing 4

```
struct BitmapDeleter {
  using pointer = Bitmap;
  void operator()(Bitmap bmp) {
    DestroyBitmap(bmp); } };
  using ScopedBitmap = std::unique_ptr<Bitmap,
                       BitmapDeleter>;
...
DeviceContext ctx{};
Bitmap tmp;
CreateBitmap(&tmp);
ScopedBitmap bmp{ tmp };
DrawBitmap(ctx, bmp.get());
```

Listing 5

As you can see, the **`ScopedBitmap`** is more awkward to use. In particular, it can't be passed directly to functions designed for **`Bitmap`**.

Considering the above, let's move to the third option: implementing an RAII wrapper ourselves.

Implementation

The implementation presented below is using a different approach to clean-up routine than standard library smart pointers. It takes advantage of an ability to selectively specialize non-template members of class template [Template Specialization]. (See Listing 6.)

```
#include <cassert>
#include <memory> // std::addressof

template<typename ResourceTag,
  typename ResourceType>
class Resource {

public:
  Resource() noexcept = default;
  explicit Resource(ResourceType resource)
    noexcept : resource_{ resource } {}
  Resource(const Resource&) = delete;
  Resource& operator=(const Resource&) = delete;
  Resource(Resource&& other) noexcept
    : resource_{ other.resource_ }
    { other.resource_ = {}; }

  Resource& operator=(Resource&& other) noexcept {
    assert(this != std::addressof(other));
    Cleanup();
    resource_ = other.resource_;
    other.resource_ = {};
    return *this;
  }

  ~Resource() { Cleanup(); }
  operator const ResourceType&() const noexcept {
    return resource_;
  }

  ResourceType* operator&() noexcept {
    Cleanup();
    return &resource_;
  }

private:
  // Intentionally undefined - must be
  // explicitly specialized.
  void Cleanup() noexcept;
  ResourceType resource_{};
};
```

Listing 6

First, some minor design points.

- The class is *noncopyable*, but *movable*, thus, it provides *sole ownership semantic* (just like `std::unique-ptr`). One can provide *shared ownership* counterpart (akin to `std::shared_ptr`) if needed.
- Taking into account that most `ResourceType` arguments are simple resource handles (like `void*` or `int`), the class methods are defined `noexcept` .
- Overloading `operator&` is a questionable (if not bad) design decision. Nevertheless, I decided to do it in order to facilitate the usage of the class with factory functions of the form `void CreateHandle(Handle* handle)`.

Now to the core. As you can see, the `Cleanup` method which is the cornerstone of our RAII wrapper is left undefined. As a result, an attempt to instantiate such a class will lead to an error. The trick is to define an explicit specialization of `Cleanup` for each particular resource type. For example:

```
// Here "FileId" is some OS-specific file
// descriptor Type which must be closed with
// CloseFile function.
using File = Resource<struct FileIdTag, FileId>;
template<> void File::Cleanup() noexcept {
  if (resource_)
    CloseFile(resource_);
}
```

Now we can use our class to wrap `FileId` objects:

```
{
  File file{ CreateFile(file_path) };
  ...
} // "file" will be destroyed here
```

You can think of the `Cleanup` declaration inside `Resource` as a 'compile-time pure virtual function'. Similarly, explicit specialization of `Cleanup` for `FileId` is a concrete implementation of such a function.

What's the deal with ResourceTag?

You may wonder, why do we need a `ResourceTag` template parameter which is used nowhere? It solves two purposes.

First is *type-safety*. Imagine two different resource types, say `Bitmap` and `Texture`, both of which are defined as type aliases for `void*`. Without the tag parameter, the compiler simply couldn't detect the nasty bug in Listing 7.

With the help of the tag, however, the compiler can detect the error (Listing 8).

The second purpose of the tag: it allows us to define `Cleanup` specializations for conceptually different resources having the same C++ type. Once again, imagine that our `Bitmap` resource requires a `DestroyBitmap` function while `Texture` requires `DestroyTexture`.

```
using ScopedBitmap = Resource<Bitmap>;
using ScopedTexture = Resource<Texture>;
void DrawBitmap(DeviceContext& ctx,
                ScopedBitmap& bmp){
  /* ... */
}

int main() {
  DeviceContext ctx;
  ScopedBitmap bmp;
  ScopedTexture t;
  // Passing texture to function expecting bitmap.
  // Compiles OK.
  DrawBitmap(ctx, t);
}
```

Listing 7

```
using ScopedBitmap = Resource<struct BitmapTag,
                              Bitmap>;
using ScopedTexture = Resource<struct TextureTag,
                               Texture>;

int main() {
  DeviceContext ctx;
  ScopedBitmap bmp;
  ScopedTexture t;
  DrawBitmap(ctx, t);  // error: type mismatch
}
```

Listing 8

Without tag parameters, `ScopedBitmap` and `ScopedTexture` would be the same type (recall that both `Bitmap` and `Texture` are in fact `void*` in our example), preventing us from defining specialized clean-up routines for each of them.

Speaking about the tag, the following expression may seem odd-looking to some:

```
using File = Resource<struct FileIdTag, FileId>;
```

In particular, I'm talking about the usage of `struct FileIdTag` as a template argument. Let's see the equivalent expression, the meaning of which I bet is clear to those familiar with tag dispatching [Tag Dispatching]:

```
struct FileIdTag{};
using File = Resource<FileIdTag, FileId>;
```

Conventional tag dispatching makes use of function overloading with the argument of tag type being an overload selector. The tag is passed to the overloaded function by value, hence, tag type must be a complete type. In our case however, no function overloading is taking place. The tag is used only as a template argument to facilitate explicit specialization. Taking into account that C++ permits *incomplete types* as template arguments, we can replace tag type definition with a declaration:

```
struct FileIdTag;
using File = Resource<FileIdTag, FileId>;
```

Now, considering that `FileIdTag` is needed only inside the type alias declaration, we can move it directly into the place of usage:

```
using File = Resource<struct FileIdTag, FileId>;
```

Making an explicit specialization requirement a little more explicit

If the user fails to provide an explicit specialization for the `Cleanup` method, he/she will not be able to build the program. This is by design. However, there are two usability issues involved:

- the error is reported at link-time, while it is preferable (and possible) to detect it much earlier, at compile-time;
- the error message gives the user no clue about the actual problem and the way solve it.

Let's try to fix it with the help of `static_assert`:

```
void Cleanup() noexcept {
  static_assert(false,
  "This function must be explicitly "
  "specialized.");
}
```

Unfortunately, it won't work as expected: the assertion may produce an error even though the *primary* `Cleanup` method is never *instantiated*. The reason is the following: the condition inside `static_assert` does not depend in any way on our class template parameters, therefore, the compiler can evaluate the condition even before attempting to instantiate the template.

Knowing that, the fix is simple: make the condition dependent on template parameter(s) of the class template. We could do this by writing a *compile-*

```
class Bitmap {
public:
  Bitmap(int width, int height);
  ~Bitmap(){};
  int Width() const;
  int Height() const;
  Colour PixelColour(int x, int y) const;
  void PixelColour(int x, int y, Colour colour);
  DC DeviceContext() const;
  /* Other methods... */

private:
  int width_{};
  int height_{};
  // Raw resources.
  BITMAP bitmap_{};
  DC device_context_{};
};
```

Listing 9

time member function which unconditionally produces a constant of the value **false**:

```
  static constexpr bool False() noexcept {
    return false; }

  void Cleanup() noexcept {
    static_assert(False(),
    "This function must be explicitly "
    "specialized.");
  }
```

Thin wrappers vs. full-fledged abstractions

The RAII-wrapper template presented provides a thin abstraction dealing strictly with resource management. One may argue, why bother using such a wrapper instead of implementing a proper high-level abstraction in the first place? As an example, consider writing a bitmap class from scratch (see Listing 9).

To see why such a design is a bad idea in general, let's write a constructor for the **Bitmap** class (Listing 10).

As you can see our class is actually managing two resources: the bitmap itself and the corresponding device context (this example is inspired by the Windows GDI, where a bitmap must be backed up by an in-memory device context for most of the drawing operations and for the sake of interoperability with modern graphics APIs). And here goes the problem: if the **device_context_** initialization fails, the **bitmap_** will be leaked!

```
Bitmap::Bitmap(int width, int height)
  : width_{ width }, height_{ height } {

  // Create bitmap.
  bitmap_ = CreateBitmap(width, height);
  if (!bitmap_)
    throw std::runtime_error{
      "Failed to create bitmap." };

  // Create device context.
  device_context_ = CreateCompatibleDc();
  if (!device_context_)
    // bitmap_ will be leaked here!
    throw std::runtime_error{
      "Failed to create bitmap DC." };

  // Select bitmap into device context.
  // ...
}
```

Listing 10

Gotchas

A couple of gotchas to watch for when defining explicit template specializations:

- explicit specialization must be defined in the same **namespace** as the primary template (in our case, the **Resource** class template);
- an explicit specialization function definition residing in a header file must be **inline**: remember, the explicit specialization is a regular function, not a template anymore.

On the other hand, consider the equivalent code with the usage of scoped resources (Listing 11).

This example leads us to the following guideline: *do not keep more than one unmanaged resource as a class member*. Better consider applying RAII to each of the resources, and then use them as building blocks for a more high-level abstractions. This approach both ensures exception safety and code reuse (you can recombine those building block as you wish in the future without the fear of introducing resource leaks).

More examples

In Listing 12, you can see some real-world examples of useful specializations for Windows API objects. Windows API is chosen, because it provides many opportunities for RAII application. The examples are self-explanatory enough; no Windows API knowledge is required.

Comparing with `unique_resource` from N3949

The limitations of smart pointers as a generic resource management tool discussed earlier have led to development of standard proposal N3949 [N3949]. N3949 suggests a **unique_resource_t** class template similar to the one presented in the article but with a more conventional approach to the clean-up routine (i.e., in the vein of **std::unique_ptr**) – see Listing 13.

As you can see, **unique_resource_t** uses a clean-up routine *per resource instance*, while the **Resource** class utilizes a clean-up routine *per resource type* approach. Conceptually, a clean-up routine is more a property of a resource type rather than instance (this is obvious from most of the real-world usage of RAII wrappers). Consequently, it becomes

```
using ScopedBitmap = Resource<struct BitmapTag,
                              BITMAP>;
using ScopedDc = Resource<struct DcTag, DC>;
...
Bitmap::Bitmap(int width, int height)
  : width_{ width }, height_{ height } {

  // Create bitmap.
  bitmap_ = ScopedBitmap{
    CreateBitmap(width, height) };
  if (!bitmap_)
    throw std::runtime_error
      { "Failed to create bitmap." };

  // Create device context.
  device_context_ = ScopedDc
    { CreateCompatibleDc() };
  if (!device_context_)
    // Safe: bitmap_ will be destroyed in case of
    // exception.
    throw std::runtime_error
      { "Failed to create bitmap DC." };

  // Select bitmap into device context.
  // ...
}
```

Listing 11

```
// Windows handle.
using Handle = Resource<struct HandleTag, HANDLE>;
template<> void Handle::Cleanup() noexcept {
  if (resource_ &&
      resource_ != INVALID_HANDLE_VALUE)
    CloseHandle(resource_);
}

// WinInet handle.
using InetHandle
  = Resource<struct InetHandleTag, HINTERNET>;
template<> void InetHandle::Cleanup() noexcept {
  if (resource_)
    InternetCloseHandle(resource_);
}

// WinHttp handle.
using HttpHandle
  = Resource<struct HttpHandleTag, HINTERNET>;
template<> void HttpHandle::Cleanup() noexcept {
  if (resource_)
    WinHttpCloseHandle(resource_);
}

// Pointer to SID.
using Psid = Resource<struct PsidTag, PSID>;
template<> void Psid::Cleanup() noexcept {
  if (resource_)
    FreeSid(resource_);
}

// Network Management API string buffer.
using NetApiString
  = Resource<struct NetApiStringTag, wchar_t*>;
template<> void NetApiString::Cleanup()
  noexcept {
  if (resource_ && NetApiBufferFree(resource_)
    != NERR_Success) {
    // Log diagnostic message in case of error.
  }
}

// Certificate store handle.
using CertStore
  = Resource<struct CertStoreTag, HCERTSTORE>;
template<> void CertStore::Cleanup() noexcept {
  if (resource_)
    CertCloseStore(resource_,
                   CERT_CLOSE_STORE_FORCE_FLAG);
}
```

Listing 12

tedious to specify clean-up routine during each and every resource creation. On rare occasions, however, such a flexibility can be useful. As an example, consider the clean-up function which takes a policy flag to control the deletion of resource, such as the **`CertCloseStore`** Windows API function presented earlier in the examples section.

Speaking about the amount of code needed to define a resource wrapper, there is not much difference between **`Resource`** and **`unique_resource_t`**. Personally, I find function specialization definition to be more elegant than *functor* definition (i.e., **`struct`** with **`operator()`**). For **`unique_resource_t`** we could also use in-place lambda instead, as shown above, but this quickly becomes inconvenient as we need to create resources in more than one place in the code (the lambda definition must be repeated then). On the other hand, passing *callable* objects in constructors to provide custom logic is widely used in C++, while defining explicit specializations may seem more exotic to most programmers.

```
template<typename Resource, typename Deleter>
class unique_resource_t {
  /* … */
};

// Factory.
template<typename Resource, typename Deleter>
unique_resource_t<Resource, Deleter>
unique_resource(Resource&& r, Deleter d) noexcept
{
  /* … */
}
...

// Usage (predefined deleter).
struct ResourceDeleter {
  void operator()(Resource resource)
     const noexcept {
    if (resource)
      DestroyResource(resource);
  }
};

using ScopedResource =
  unique_resource_t<Resource, ResourceDeleter>;
ScopedResource r{ CreateResource(),
  ResourceDeleter{} };

// Alternative usage (in-place deleter definition).
auto r2{ unique_resource(
  CreateResource(),
  [](Resource r){ if (r) DestroyResource(r); })
};
```

Listing 13

Conclusion

The RAII wrapper presented in the article resolves most of the shortcomings of standard library smart pointers for managing resources of types other than memory. To be specific:

- non-obvious declaration syntax for pointer type aliases;
- limited support for non-pointer types;
- awkward usage of managed resources with low-level APIs in comparison to unmanaged ones.

We have also become acquainted with a simple but interesting static polymorphism technique based on the usage of explicit template specialization. Historically, explicit template specialization has had the fame of an advanced language feature aimed mainly towards library implementers and experienced users. As you can see however, it can play a much more prominent role of a core abstraction mechanism on par with virtual functions, rather than being merely a helpful utility in a library implementer's toolbox. I am convinced that the full potential of this feature has yet to be unlocked. ■

Code available at https://goo.gl/cK46xF

References

[N3949] http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2014/n3949.pdf

[NullablePointer] http://en.cppreference.com/w/cpp/concept/NullablePointer

[Tag Dispatching] http://www.boost.org/community/generic_programming.html#tag_dispatching

[Template Specialization] http://en.cppreference.com/w/cpp/language/template_specialization

iterator_pair – A Simple and Useful Iterator Adapter

Can you form a new contain from two others? Vladimir Grigoriev reminds us how to write an iterator.

The article describes a simple and useful iterator adapter **iterator_pair**, provides its implementation and shows some use cases of the iterator. It argues that because standard iterator adapters hide the property value type of the underlying containers and objects, they make it difficult to write safe and generic code. The article points to a potention surprise with the standard algorithms **std::partial_sum** and **std::adjacent_difference**, and offers a way to remedy it.

Readers may also be interested to know that there is an iterator named **zip_iterator** in the boost libraries that is also based on the idea of combining several iterators as one iterator. It models Readable iterator as described in the documentation on the iterator [Boost]. Nevertheless **zip_iterator** and **iterator_pair** are different iterator adapters, with different implementations and their own usages.

Let's help a student

In a forum for beginners I encountered the following assignment.

> Let's assume that there are two arrays of integers, A and B, with equal sizes. Form third array C with the same size elements of which will be set to the minimum of values of corresponding elements of the first two arrays.

It is obvious that the task of a beginner is to demonstrate his skill in managing loops.

Nevertheless the assignment can be easy done by means of the standard algorithm **std::transform**. Listing 1 is a possible solution of the assignment using **std::transform**.

Note: If you are using MS Visual C++ to compile the code then specify **=** as the capture-default option in the lambda expression used in the calls of the algorithm **std::generate**. Otherwise you can get a compilation error.

The program might have the following output:

```
A: 2 0 3 7 2 4 0 4 2 2 6 5 3 6 7 0 6 9 0 2
B: 6 2 1 4 0 5 5 4 6 0 2 8 2 7 7 7 1 7 3 5
C: 2 0 1 4 0 4 0 4 2 0 2 5 2 6 7 0 1 7 0 2
```

This is nothing unusual or difficult. The only detail you should take into account is that you may not write simply **std::min<int>** instead of the lambda expression in the call of the algorithm because the compiler will report an error saying that there is an ambiguity for **std::min<int>**.

Indeed, when the C++ 2011 Standard was adopted a new special type appeared. It is **std::initializer_list**. Consequently several standard algorithms, including **std::min**, were overloaded to accept **std::initializer_list** as a parameter type. So **std::min<int>** can be either a specialization of the function with two parameters of type **int** (more precisely of type **const int &**) or a specialization of the function that has parameter of type **std::initializer_list<int>**. Thus you need a means of helping the compiler to select the right function. And using lambda expressions as wrappers around overloaded functions in similar situations is such a means. Of course you may omit the template argument of the function within the lambda expression because the compiler can deduce it itself in this case.

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <cstdlib>
#include <ctime>
int main()
{
  const size_t N = 20;
  int a[N], b[N], c[N];
  std::srand( ( unsigned int )
    std::time( nullptr ) );
  std::generate( std::begin( a ), std::end( a ),
    [] { return ( std::rand() % ( N / 2 ) ); } );
  std::cout << "A: ";
  for ( int x : a ) std::cout << x << ' ';
  std::cout << std::endl;
  std::generate( std::begin( b ), std::end( b ),
    [] { return ( std::rand() % ( N / 2 ) ); } );
  std::cout << "B: ";
  for ( int x : b ) std::cout << x << ' ';
  std::cout << std::endl;
  std::transform( std::begin( a ), std::end( a ),
    std::begin( b ), std::begin( c ),
    [] ( int x, int y )
    {
      return ( std::min( x, y ) );
    } );
  std::cout << "C: ";
  for ( int x : c ) std::cout << x << ' ';
  std::cout << std::endl;
}
```

Listing 1

From the simple to the complex

Whether a student will use the standard algorithm or write an appropriate loop himself is not important for us now: as usual, there is just a step between trivial and non-trivial tasks.

Indeed, let's make the assignment a bit more complicated. Assume that now we need to fill elements of one array, array C, with the minimum values of corresponding elements of the original arrays and to fill elements

Vladimir Grigoriev is active in user groups devoted to the C++ Standard at isocpp.org. He submitted several proposals to the C++ Standards Committee, some of which were adopted. He is a highly visible participant of C and C++ sections at Stack Overflow. His nickname there is Vlad from Moscow.

what I am sure of is that if such solutions using other standard algorithms exist, they will look artificial and will not be easily readable

of another array, say, array D, with the maximum values of corresponding elements of arrays A and B.

What should we do in this case? We could use the previous call of **std::transform** twice, first to form array C with the minimum values and then, in the second call, to form array D with the maximum values.

However, it is obvious that such an approach is inefficient. Moreover, if we make the assignment even more complicated and require that instead of using two additional arrays, C and D, we have to overwrite the original arrays A and B with the minimum and maximum values of their elements respectively – that is, to do the task 'in place' – it is clear that this approach simply will not work.

Is it a dead end and will we be forced to use an ordinary loop as a student would do?

It is possible that these complicated assignments could be done with some other standard algorithms. I think that maybe **std::inner_product** will cope with the tasks. I am not sure, I did not try it. It is simply my supposition.

But what I am sure of is that if such solutions using other standard algorithms exist, they will look artificial and will not be easily readable.

It seems that there are no satisfactory solutions. There are indeed no satisfactory solutions if we must act within the frames of available standard constructions (algorithms, iterators and so on) provided by the C++ Standard (if you have such solutions please let me know).

However, let's not abandon hope but return to the previous code example and consider the call of **std::transform** more closely.

```
std::transform( std::begin( a ), std::end( a ),
  std::begin( b ), std::begin( c ),
  [] ( int x, int y )
  {
    return ( std::min( x, y ) );
  } );
```

For the new, more complicated, assignments we need to get the minimum and maximum values of each pair of elements of arrays A and B simultaneously. There is a standard algorithm that can do this job. It is algorithm **std::minmax**. Not so bad! Let's replace **std::min** in the lambda expression with **std::minmax**.

```
std::transform( std::begin( a ), std::end( a ),
  std::begin( b ), std::begin( c ),
  [] ( int x, int y )
  {
    return ( std::minmax( x, y ) );
  } );
```

So the lambda expression will now return an object of type **std::pair<const int &, const int &>**. The problem is that the iterator for array C cannot accept objects of that type and our purpose is to deal with two arrays simultaneously instead of one array.

Hmm...What will happen if we substitute the iterator of array C for a pair of iterators (the same way as we did with the substitution of **std::min** that returns a single object for **std::minmax**) that returns a pair of objects (actually one object of type **std::pair<const int &, const int &>**)?

It is an idea! Let's write the renewed call of **std::transform** first and then discuss it.

```
std::transform( std::begin( a ), std::end( a ),
  std::begin( b ), make_iterator_pair
    ( std::begin( c ),std::begin( d ) ),
  [] ( int x, int y )
{
  return ( std::minmax( x, y ) );
} );
```

So how does it look?

The functional object returns an object of type **std::pair<const int &, const int &>** and it is met by an iterator of type **std::pair<int *, int *>** that is by the pair of iterators. Each iterator will get its own value. Thus arrays C and D will be filled as required.

Of course there is no such a function as **make_iterator_pair** at present in the C++ Standard, in the same way as there is no iterator adapter **iterator_pair** itself. It is only my proposal. However, as you can see if there were such constructions our complicated assignments could be done very simply and elegantly.

Now all that we need to enjoy the luxury of using this iterator adapter to run programs for the assignments is to implement it.

Time to build the iterator adapter

The iterator adapter **iterator_pair** will have the iterator category **std::output_iterator_tag**. This allows us to combine any two iterators that satisfy the requirements of output iterators. Its value type will be a pair of value types of the underlying iterators. For convenience the definition of the iterator adapter is placed in a separate header file with name **"iterator_pair.h"** inside the name space **usr**.

Listing 2 is the iterator adapter definition, with boilerplate **include** guards removed for brevity.

All is ready. It is time to enjoy the fruits of our labor. Below a program is presented that performs both assignments. First, it fills the two arrays C and D with the minimum and maximum values of each pair of elements of arrays A and B, and then overwrites arrays A and B themselves with the same minimum and maximum values. See Listing 3.

The program might have the following output:

```
A: 3 1 2 2 9 3 4 9 8 8 2 5 7 2 3 5 3 0 8 4
B: 6 8 7 2 5 7 5 2 1 2 4 7 3 7 1 2 2 5 3 2
C: 3 1 2 2 5 3 4 2 1 2 2 5 3 2 1 2 2 0 3 2
D: 6 8 7 2 9 7 5 9 8 8 4 7 7 7 3 5 3 5 8 4

A: 3 1 2 2 9 3 4 9 8 8 2 5 7 2 3 5 3 0 8 4
B: 6 8 7 2 5 7 5 2 1 2 4 7 3 7 1 2 2 5 3 2
A: 3 1 2 2 5 3 4 2 1 2 2 5 3 2 1 2 2 0 3 2
B: 6 8 7 2 9 7 5 9 8 8 4 7 7 7 3 5 3 5 8 4
```

two separate operations – the default construction of objects and assigning actual values to them – can be substituted for one operation of copy construction

```
#include <iterator>
#include <utility>
namespace usr
{
  using namespace std;
  template <class Iterator1, class Iterator2>
  class iterator_pair
    : public iterator<output_iterator_tag,
    pair<typename iterator_traits<Iterator1>
    ::value_type,
    typename iterator_traits<Iterator2>
    ::value_type>, void, void, void>
{
public:
  typedef pair<Iterator1, Iterator2>
    iterator_type;
  iterator_pair( Iterator1, Iterator2 );
  explicit iterator_pair( const pair<Iterator1,
    Iterator2> & );
  explicit iterator_pair( pair<Iterator1,
    Iterator2> && );
  iterator_type base() const;
  iterator_pair<Iterator1, Iterator2> &
  operator =( const
    pair<typename iterator_traits<Iterator1>
      ::value_type,
    typename iterator_traits<Iterator2>
      ::value_type> & );
  iterator_pair<Iterator1, Iterator2> &
  operator =
    ( pair<typename iterator_traits<Iterator1>
      ::value_type,
    typename iterator_traits<Iterator2>
      ::value_type> && );
  iterator_pair<Iterator1, Iterator2>
    & operator *();
  iterator_pair<Iterator1, Iterator2>
    & operator ++();
  iterator_pair<Iterator1, Iterator2>
    operator ++( int );
protected:
  iterator_type it;
};
template <class Iterator1, class Iterator2>
iterator_pair<Iterator1, Iterator2>
make_iterator_pair( Iterator1, Iterator2 );
template <class Iterator1, class Iterator2>
iterator_pair<Iterator1, Iterator2>
make_iterator_pair( const pair<Iterator1,
  Iterator2> & );
}
```

Listing 2

```
namespace usr
{
template <class Iterator1, class Iterator2>
iterator_pair<Iterator1,
Iterator2>::iterator_pair
( Iterator1 it1, Iterator2 it2 )
  : it( it1, it2 ) {}
template <class Iterator1, class Iterator2>
iterator_pair<Iterator1,
Iterator2>::iterator_pair
( const pair<Iterator1, Iterator2> &it_pair )
  : it( it_pair ) {}
template <class Iterator1, class Iterator2>
typename iterator_pair<Iterator1,
Iterator2>::iterator_type
iterator_pair<Iterator1, Iterator2>::base() const
{
  return ( it );
}

template <class Iterator1, class Iterator2>
iterator_pair<Iterator1, Iterator2> &
iterator_pair<Iterator1, Iterator2>::operator =
  ( const pair<typename iterator_traits<Iterator1>
      ::value_type,
  typename iterator_traits<Iterator2>
      ::value_type> &value )
{
  *( it.first )  = value.first;
  *( it.second ) = value.second;
  return ( *this );
}

template <class Iterator1, class Iterator2>
iterator_pair<Iterator1, Iterator2> &
iterator_pair<Iterator1, Iterator2>::operator =
( pair<typename
iterator_traits<Iterator1>::value_type,
       typename
iterator_traits<Iterator2>::value_type> &&value )
{
  *( it.first )  = value.first;
  *( it.second ) = value.second;
  return ( *this );
}
template <class Iterator1, class Iterator2>
iterator_pair<Iterator1, Iterator2> &
iterator_pair<Iterator1, Iterator2>::operator *()
{
        return ( *this );
}
```

Listing 2 (cont'd)

```
template <class Iterator1, class Iterator2>
iterator_pair<Iterator1, Iterator2> &
iterator_pair<Iterator1, Iterator2>::operator
++()
{
  ++it.first;
  ++it.second;

  return ( *this );
}
template <class Iterator1, class Iterator2>
iterator_pair<Iterator1, Iterator2>
iterator_pair<Iterator1, Iterator2>
  ::operator ++( int )
{
  iterator_pair<Iterator1, Iterator2> tmp( it );
  it.first++;
  it.second++;
  return ( tmp );
}
template <class Iterator1, class Iterator2>
iterator_pair<Iterator1, Iterator2>
make_iterator_pair( pair<Iterator1, Iterator2>
   &&it_pair )
{
  return ( iterator_pair<Iterator1,
    Iterator2>( it_pair ) );
}
template <class Iterator1, class Iterator2>
iterator_pair<Iterator1, Iterator2>
make_iterator_pair( Iterator1 it1, Iterator2 it2 )
{
  return ( iterator_pair<Iterator1,
    Iterator2>( it1, it2 ) );
}
template <class Iterator1, class Iterator2>
iterator_pair<Iterator1, Iterator2>
make_iterator_pair( const pair<Iterator1,
  Iterator2> &it_pair )
{
  return ( iterator_pair<Iterator1,
    Iterator2>( it_pair ) );
}
}
```

Listing 2 (cont'd)

You can see that the program has done all that is required in the assignments.

To gain a more complete insight about the possibilities of the iterator adapter, let's consider one more use case that occurs in practice where the iterator adapter would come in handy.

Sometimes it is required to copy the key values and mapped values of some associative container having, for example, type **`std::map`** in two other separate sequential containers. So let's assume that there is a container of type **`std::map<int, std::string>`** and your task, for example, is to copy the key values of the map in a container of type **`std::vector<int>`** and mapped values of the map in a container of type **`std::forward_list<std::string>`**.

Before you continue to read the article further, it would be useful for you to try to do the assignment yourself using some standard algorithms and then compare your solution with the solution based on applying iterator adapter **`iterator_pair`**.

Have you written your solution yet? How much time did it take to write it?

Now compare it with what is being suggested in Listing 4.

The program has the following output:

```
0 1 2 3 4
```

Hello new iterator adapter **`iterator_pair`**!

```
#include "iterator_pair.h"
int main()
{
  const size_t N = 20;
  int a[N], b[N], c[N], d[N];
  std::srand
    ( ( unsigned int )std::time( nullptr ) );
  std::generate( std::begin( a ), std::end( a ),
  [] { return ( std::rand() % ( N / 2 ) ); } );
  std::cout << "A: ";
  for ( int x : a ) std::cout << x << ' ';
  std::cout << std::endl;
  std::generate( std::begin( b ), std::end( b ),
    [] { return ( std::rand() % ( N / 2 ) ); } );
  std::cout << "B: ";
  for ( int x : b ) std::cout << x << ' ';
  std::cout << std::endl;
  std::transform( std::begin( a ), std::end( a ),
    std::begin( b ),
    usr::make_iterator_pair( std::begin( c ),
    std::begin( d ) ), [] ( int x, int y )
    {
      return ( std::minmax( x, y ) );
    } );
  std::cout << "C: ";
  for ( int x : c ) std::cout << x << ' ';
  std::cout << std::endl;
  std::cout << "D: ";
  for ( int x : d ) std::cout << x << ' ';
  std::cout << std::endl;
  std::cout << std::endl;
  std::cout << "A: ";
  for ( int x : a ) std::cout << x << ' ';
  std::cout << std::endl;
  std::cout << "B: ";
  for ( int x : b ) std::cout << x << ' ';
    std::cout << std::endl;
  std::transform( std::begin( a ), std::end( a ),
    std::begin( b ),
    usr::make_iterator_pair( std::begin( a ),
      std::begin( b ) ), [] ( int x, int y )
    {
      return ( std::minmax( x, y ) );
    } );
  std::cout << "A: ";
  for ( int x : a ) std::cout << x << ' ';
  std::cout << std::endl;
  std::cout << "B: ";
  for ( int x : b ) std::cout << x << ' ';
  std::cout << std::endl;
}
```

Listing 3

The central point of the program is the statement

```
std::copy( m.begin(), m.end(),
usr::make_iterator_pair( v.begin(), l.begin() ) );
```

that does all the work. I think that you will agree with me that the statement looks very clear and does not require much time to understand what is being done here.

It seems that we could end the article here. We have gotten a remarkably simple and useful iterator adapter. However, it is the C++ Standard that does not allow us to do this.

In the previous listings, the given number of elements of containers **`std::vector<int>`** and **`std::forward_list<std::string>`** were created beforehand. So at first the elements were created and initialized with the default values and only then the actual values were assigned to them in the call of algorithm **`std::copy`**.

```
#include "iterator_pair.h"
int main()
{
  std::map<int, std::string> m;
  std::istringstream is
   ( "Hello new iterator adapter
     iterator_pair!" );
  int i = 0;
  std::transform(
    std::istream_iterator<std::string>( is ),
    std::istream_iterator<std::string>(),
    std::inserter( m, m.begin() ),
    [&i]( const std::string &s )
    {
      return ( std::make_pair( i++, s ) );
    } );
  std::vector<int> v( m.size() );
  std::forward_list<std::string> l( m.size() );
  std::copy( m.begin(), m.end(),
    usr::make_iterator_pair( v.begin(),
      l.begin() ) );
  for ( int x : v ) std::cout << x << ' ';
  std::cout << std::endl;
  for ( const std::string &s : l )
    std::cout << s << ' ';
    std::cout << std::endl;
}
```

Listing 4

The two separate operations – the default construction of objects and assigning actual values to them – can be substituted for one operation of copy construction. Calls of the copy assignment operator can be eliminated. For simple types – such as fundamental types – it is not as important. However, in general for objects of complex types, two calls of two special functions instead of one call of one special function can be wasteful.

Moreover, if we want new elements to be added to container `std::forward_list<std::string>` in the reverse order relative to the order of the elements in container `std::map<int, std::string>` then it makes no sense to create the elements of `std::forward_list<std::string>` beforehand, because the class `std::forward_list` does not have a reverse iterator.

Therefore let's make some minor changes to the program. We will not create elements of containers `std::vector<int>` and `std::forward_list<std::string>` beforehand. Instead only reserve enough memory for the container `std::vector<int>`'s future elements and then respectively `std::back_insert_iterator` and `std::front_insert_iterator` will be used in the call of `std::copy`.

Now the program will look like Listing 5.

If you have typed the program correctly without any typos (or if I did this myself correctly because you are going simply to copy and paste the program) then you may bravely compile the program and ... you will get compilation errors!

There is no visible cause why the code might not be compiled. Therefore all questions should be addressed to the compiler or, to be more correct, to the C++ Standard: why it does not allow the compiler to compile the program.

What is your name?

If you are going to contact someone you should use either their real name or a common form of address. Otherwise the result can be unexpected: you might get ignored entirely or there might be even more unpleasant consequences. (Imagine what might happen if you call your spouse by the wrong name.)

```
#include "iterator_pair.h"
int main()
{
  std::map<int, std::string> m;
  std::istringstream is
   ( "Hello new iterator adapter
      iterator_pair!" );
  int i = 0;
  std::transform
    ( std::istream_iterator<std::string>( is ),
      std::istream_iterator<std::string>(),
      std::inserter( m, m.begin() ),
      [&i]( const std::string &s )
  {
      return ( std::make_pair( i++, s ) );
  } );
  std::vector<int> v;
  v.reserve( m.size() );
  std::forward_list<std::string> l;
  std::copy( m.begin(), m.end(),
    usr::make_iterator_pair
      ( std::back_inserter( v ),
        std::front_inserter( l ) ) );
  for ( int x : v ) std::cout << x << ' ';
  std::cout << std::endl;
  for ( const std::string &s : l )
    std::cout << s << ' ';
    std::cout << std::endl;
}
```

Listing 5

The same is true for the world of classes and objects.

If you want your programs to be safe, flexible, and portable, you should use the following general convention:

- Your classes have to provide common forms of address for their properties.
- Code that uses your classes has to use these common forms of address when it tries to access properties of your classes (or use their real names, but that can be tricky).
- In any case, it is better to use a common form of address because the real name of a property can vary between implementations.

As you already know, 'names' in the given context means the type names of class properties.

Here are two simple examples that demonstrate what can occur if you do not comply with the convention.

The first example. Let's assume that you have a project where there is a set of flags. You chose to use the standard class `std::vector<bool>` as the container for the flags. Throughout the project a few methods do some processing based on the number of a flag in the set and its value passed together to the methods as arguments. Of course you tried to make your code flexible and independent of the details of the underlying container. The code could look something like Listing 6.

After a time, you conclude that it would be better to replace `std::vector<bool>` with `std::bitset` because the set of flags has a small fixed size. You might think that it will be enough to substitute only the alias declaration (and it would be indeed great if it was enough) because, after all you tried to write the code in such a way that it would be independent of the details of the underlying container.

However, if you make the substitution and instead of

```
using special_flags_t = std::vector<bool>;
```

write

```
using special_flags_t = std::bitset<N>;
```

(where `N` is some predefined constant), the project will not compile and the compiler will issue numerous errors!

```
using special_flags_t = std::vector<bool>;
void method1( special_flags_t::size_type
flag_number, bool flag_value )
{
  // some processing using the flag
}
//...
void methodn( special_flags_t::size_type
flag_number, bool flag_value )
{
  // some processing using the flag
}
//...
special_flags_t flag_values;
//...
for ( special_flags_t::size_type flag_number = 0;
  flag_number < flag_values.size();
  flag_number++ )
{
  method1( flag_number,
    flag_values[flag_number] );
}
//...
for ( special_flags_t::size_type flag_number = 0;
  flag_number < flag_values.size();
  flag_number++ )
{
  methodn( flag_number, flag_values[flag_number]
);
}
```

Listing 6

The problem is that the standard class **std::bitset** does not provide the common form of address **size_type** for its property size. Thus your good intentions were completely subverted.

Note: the author has submitted a proposal to add a typedef declaration **size_type** for standard class **std::bitset**.

The second example. A programmer wrote the following snippet of code being sure that nothing extraordinary can occur within it.

```
std::string s;
std::string source;
std::string dest;
//...
unsigned int pos = s.find( source );
if ( pos != std::string::npos )
{
  s.replace( pos, source.size(), dest );
}
```

He was very suprised when this code snippet generated an exception **std::out_of_range**!

The reason for the exception is that the size of the unsigned integral type used by the container to represent its own property size happened to be greater than the size of type **unsigned int** in the environment where the program was compiled and run. So when string source had not been found in string **s** in statement

```
unsigned int pos = s.find( source );
```

the value returned by the method find of **std::string** was reduced using the arithmetic modulo 2 operation to fit **pos**. And then in the statement

```
if ( pos != std::string::npos )
```

it was again enlarged to the size of the type of **std::string::npos** according to the rules of the usual arithmetic conversions by setting the most significant bits to zeroes. As a result the condition in the **if** statement was evaluated to **true** and the incorrect value of **pos** was used further in method **replace**.

The exception could be avoided and the program would be portable if the programmer were to use the common form of address **std::string::size_type** in the declaration of variable **pos** instead of the type **unsigned int**.

```
std::string::size_type pos = s.find( source );
```

Or it would be even better to write simply

```
auto pos = s.find( source );
```

When you deal with containers or sequences of data directly or through iterators, one of the most important and useful pieces of information is about the value type of elements of the container or sequence. Without having such information, it is difficult to write generic and safe code. You should provide the common form of address **value_type** so that code can access the elements. Otherwise you will be helpless and will be unable to write generic template code.

That's exactly what happened for our **iterator_pair**. Both of the iterator adapters (**std::back_insert_iterator** and **std::front_insert_iterator**) hide the actual value of the common form of address **value_type** of the underlaying containers from the user, making the property value type itself inaccessible.

If you look at how the iterators are defined [ISO/IEC] you will see that the second template argument of the inherited base class **std::iterator** that corresponds to the common form of address **value_type** is set to **void**. (See Listing 7.)

Thus when the property value type of iterator adapter **iterator_pair** that in turn is defined like

```
pair
 <typename iterator_traits<Iterator1>::value_type,
  typename iterator_traits<Iterator2>::value_type>
```

was instantiated then the compiler issued an error because it cannot instantiate **std::pair** with data members of type **void**.

These two iterators look like black holes. If a container finds itself in a constructor of the iterators then it instantly loses without a trace its main property, the value type.

On the other hand, if you look at how assignment operators are defined [ISO/IEC] for these iterators, you will see that they use the property value type of underlaying containers. For example

```
front_insert_iterator<Container>&
 operator=(const typename
   Container::value_type& value);
```

But they use the property bypassing their own common form of address **value_type**.

The same problem exists with **std::ostream_iterator**. Ask any programmer, for example, what type of objects the iterator **std::ostream_iterator<std::string>**, can output and he will answer without delay: "Objects of type **std::string** or at least those objects that can be implicitly converted to type **std::string**."

```
template <class Container>
class back_insert_iterator :
public
  iterator<output_iterator_tag,void,void,
    void,void>
{
  //...
};
template <class Container>
class front_insert_iterator :
public
  iterator<output_iterator_tag,void,void,
    void,void>
{
  //...
};
```

Listing 7

And he will be right. But if you look at how the iterator is defined [ISO/IEC] you will see that its property value type is defined the same way as this property is defined for iterators **back_insert_iterator** and **front_insert_iterator**; that is, it is set to **void** and thus the real value type is hidden and inaccessible for the user of the iterator.

```
template <class T, class charT = char,
  class traits = char_traits<charT> >
  class ostream_iterator:
    public iterator<output_iterator_tag, void,
      void, void, void>
{
  //...
};
```

The very notion of an iterator adapter implies that it does not modify the properties of the underlying containers or objects. Instead it gives them new opportunities based on their own functionality.

It will not be difficult to define these iterator adapters, appending the iterator **std::insert_iterator** to them in such a way that they do not hide the main property, the value type, of the underlaying containers or objects. Their definitions could look like Listing 8.

And it should be done because as you will soon see, the problem is not limited only to the definition of the **iterator_pair**.

May an unsafe algorithm be called an algorithm in programming?

At the very beginning of the article, we helped a student. Now let the student help us.

We will ask the student to write a function that will store partial sums of elements of an array of type **std::uint8_t[N]** filled with random values in some other integer array.

Because the student does not know standard algorithms yet, he has written the function in C-style.

Listing 9 is his function.

```
template <class Container>
class back_insert_iterator :
public iterator<output_iterator_tag,
  typename Container::value_type,void,void,void>
{
  //...
};

template <class Container>
class front_insert_iterator :
public iterator<output_iterator_tag,
  typename Container::value_type,void,void,void>
{
  //...
};

template <class Container>
class insert_iterator :
public iterator<output_iterator_tag,
  typename Container::value_type,void,void,void>
{
  //...
};

template <class T, class charT = char,
  class traits = char_traits<charT> >
class ostream_iterator:
  public iterator<output_iterator_tag, T, void,
    void, void>
{
  //...
};
```

Listing 8

```
int * partial_sum( const std::uint8_t a[],
  size_t n, int b[] )
{
  if ( n )
  {
    auto acc = int( *a++ );
    *b++ = acc;
    while ( --n )
    {
      acc = acc + *a++;
      *b++ = acc;
    }
  }
  return b;
}
```

Listing 9

To be sure that the student's function is correct we need to test it. Because each of us is a qualified programmer (aren't you?) and, in contrast to the student, we know that the standard algorithm **std::partial_sum** already exists and how to use it, we can conclude that it will be reasonable simply to compare the results of using the student's function and standard algorithm **std::partial_sum** applied to the same array. Otherwise what is the use of the algorithm?

The test program can look like Listing 10.

```
int * partial_sum( const std::uint8_t a[],
  size_t n, int b[] )
{
  if ( n )
  {
    auto acc = int( *a++ );
    *b++ = acc;
    while ( --n )
    {
      acc = acc + *a++;
      *b++ = acc;
    }
  }
  return b;
}
int main()
{
  const size_t N = 10;
  std::uint8_t  a[N];
  int b[N];
  std::srand( ( unsigned int )
    std::time( nullptr ) );
  std::generate( std::begin( a ), std::end( a ),
    [] ()
    {
      return std::rand() %
        std::numeric_limits<std::uint8_t>::max();
    } );
  for ( int x : a ) std::cout << std::setw( 4 )
    << x << ' ';
  std::cout << std::endl << std::endl;
  ::partial_sum( a, N, b );
  for ( int x : b ) std::cout << std::setw( 4 )
    << x << ' ';
  std::cout << std::endl;
  std::partial_sum( std::begin( a ),
    std::end( a ), std::begin( b ) );
  for ( int x : b ) std::cout << std::setw( 4 )
    << x << ' ';
  std::cout << std::endl;
}
```

Listing 10

Well, let's run the test program, shall we?

The program output might look like:

```
110  152  109  192  160  180   82  212   74    6

110  262  371  563  723  903  985 1197 1271 1277
110    6  115   51  211  135  217  173  247  253
```

Oops! What are we seeing? Even for the second partial sum the values do not match and it seems that it is not the student's function that has not passed the test but the standard algorithm.

There is no need to go far to find the reason for the incorrect result yielded by the algorithm. The answer is staring you in the face.

It is evident that if you are going to use some accumulator for a sequence of data then the accumulator should be defined with a type that has a larger size than the size of the type of the source data so that it would be able to accomodate all accumulated values correctly without overflowing.

This is exactly how an accumulator is defined in the student's function. It has the type of the elements of the output array that stores accumulated values.

On the other hand if you have a dip into the description of algorithm **std::partial_sum** in the C++ Standard [ISO/IEC] you will see that according to the C++ Standard the algorithm creates an accumulator **acc** whose type is the value type of the input iterator.

Thus as the output of the test program has shown, in general you can ensure the safe and correct work of the algorithm only for sequences that contain a single element. You cannot control the type of the accumulator.

The same problem exists for another standard algorithm **std::adjacent_difference**.

Ask yourself what is the use of such algorithms?

Fixing this defect of the algorithms will not require a lot of effort. It is enough within the algorithms to define an accumulator as having type of the value type of the output iterator. Below is an updated version of the algorithm **std::partial_sum** without the template parameter of **operation**.

```
template <class InputIterator,
  class OutputIterator>
OutputIterator partial_sum( InputIterator first,
  InputIterator last, OutputIterator result )
{
  if ( first != last )
  {
    typename std::iterator_traits<OutputIterator>
      ::value_type acc = *first++;
    *result++ = acc;
    for ( ; first != last; ++first, ++result )
    {
      acc = acc + *first;
      *result = acc;
    }
  }
  return result;
}
```

In the same way, algorithm **std::partial_sum** could be defined with the template parameter of **operation** and the corresponding versions of the algorithm **std::adjacent_difference**.

Now if we substitute the call of standard algorithm **std::partial_sum** for a call of its new implementation as it is shown in Listing 11, then both outputs of partial sums will match each other.

```
140  138   70   20  134  191  181   45   56   37

140  278  348  368  502  693  874  919  975 1012
140  278  348  368  502  693  874  919  975 1012
```

However, this is only half the story. To get the fully functional algorithms, the standard iterator adapters **std::back_insert_iterator**, **std::front_insert_iterator**, **std::insert_iterator**, and **std::ostream_iterator** should be modified in the way described in the previous section. Only then will separate parts of the mosaic develop into a coherent picture.

```
int * partial_sum( const std::uint8_t a[],
  size_t n, int b[] )
{
  if ( n )
  {
    auto acc = int( *a++ );
    *b++ = acc;
    while ( --n )
    {
      acc = acc + *a++;
      *b++ = acc;
    }
  }
  return b;
}
namespace usr
{
  template <class InputIterator,
    class OutputIterator>
  OutputIterator partial_sum( InputIterator first,
    InputIterator last, OutputIterator result )
{
  if ( first != last )
  {
    typename std::iterator_traits<OutputIterator>
      ::value_type acc = *first++;
    *result++ = acc;
    for ( ; first != last; ++first, ++result )
    {
      acc = acc + *first;
      *result = acc;
    }
  }
  return result;
}
}   //  end of namespace usr
int main()
{
  const size_t N = 10;
  std::uint8_t  a[N];
  int b[N];
  std::srand(
    ( unsigned int )std::time( nullptr ) );
  std::generate( std::begin( a ), std::end( a ),
    [] ()
  {
    return std::rand()
      % std::numeric_limits<std::uint8_t>::max();
  } );
  for ( int x : a ) std::cout << std::setw( 4 )
    << x << ' ';
  std::cout << std::endl << std::endl;
  ::partial_sum( a, N, b );
  for ( int x : b ) std::cout << std::setw( 4 )
    << x << ' ';
  std::cout << std::endl;
  usr::partial_sum( std::begin( a ), std::end( a
),
    std::begin( b ) );
  for ( int x : b ) std::cout << std::setw( 4 )
    << x << ' ';
  std::cout << std::endl;
}
```

Listing 11

Let's consider two algorithms **std::partial_sum** and **std::accumulate** that supplement each other. It is natural to expect

that partial sums of elements of a container or data sequence produced by algorithm **std::partial_sum** would be the partial sums calculated inside algorithm **std::accumulate** for the same container or data sequence and that the last partial sum produced by the **std::partial_sum** would be equal to the final value returned by the **std::accumulate**.

In other words if, for example, there is an integer array

```
int a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

you expect that these two programs

```
int main()
{
  int a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  for ( auto last = std::begin( a );
    last != std::end( a ); )
  {
    std::cout << std::accumulate
      ( std::begin ( a ), ++last, 0 ) << " ";
  }
  std::cout << std::endl;
}
```

and

```
int main()
{
  int a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  std::partial_sum( std::begin( a ),
    std::end( a ), std::ostream_iterator<int>
    ( std::cout, " " ) );
  std::cout << std::endl;
}
```

will yield the same output

```
0 1 3 6 10 15 21 28 36 45
```

And what about for example an array of pointers to string literals instead of the array of integers? Let the array will be defined the following way

```
const char * s[] =
{
  "Hello ", "new ", "iterator ", "adapter ",
  "iterator_pair!"
};
```

In this case the first program can look like

```
int main()
{
  const char * s[] =
  {
    "Hello ", "new ", "iterator ", "adapter ",
      "iterator_pair!"
  };
  for ( auto last = std::begin( s );
    last != std::end( s ); )
  {
    std::cout << std::accumulate
      ( std::begin( s ), ++last, std::string() )
      << std::endl;
  }
}
```

And its output will be

```
Hello
Hello new
Hello new iterator
Hello new iterator adapter
Hello new iterator adapter iterator_pair!
```

So the 'partial sums' of the pointers to string literals produced by algorithm **std::partial_sum** have to look the same.

```
namespace usr
{
  template <class InputIterator,
    class OutputIterator>
  OutputIterator partial_sum( InputIterator first,
    InputIterator last, OutputIterator result )
  {
    if ( first != last )
    {
      typename
        std::iterator_traits<OutputIterator>
        ::value_type acc = *first++;
      *result++ = acc;
      for ( ; first != last; ++first, ++result )
      {
        acc = acc + *first;
        *result = acc;
      }
    }
    return result;
}
}   //  end of namespace usr
int main()
{
  const char * s[] =
  {
    "Hello ", "new ", "iterator ", "adapter ",
      "iterator_pair!"
  };
  usr::partial_sum( std::begin( s ),
                    std::end( s ),
  std::ostream_iterator<std::string>
  ( std::cout, "\n" ) );
}
```

Listing 12

All you need to do to get this result is:

- Substitute the standard algorithm **std::partial_sum** for the updated version presented in this section of the article.
- In the definition of std::ostream_iterator, change the second template argument of its base class std::iterator from void to T as shown earlier. You can do this temporarily in the implementation of the iterator in the corresponding standard header provided by the compiler.

Thus the second program will look like Listing 12.

And if you have changed **std::ostream_iterator** as it is described then you indeed will see the 'partial sums' of the pointers to string literals as it is shown above. On the other hand, if you will try the program using the original standard algorithm **std::partial_sum** then it will not compile!

Thus standard algorithms **std::partial_sum** and **std::adjacent_difference** are not only able to guarantee a correct and predictable result, but sometimes they might not even compile. ■

Acknowledgement

The author would like to thank Jonathan Leffler for his help.

References

[Boost] http://www.boost.org/doc/libs/1_57_0/libs/iterator/doc/zip_iterator.html

[ISO/IEC] ISO/IEC 14882:2011 Programming Language C++

The full code is available at: http://cpp.forum24.ru/?1-10-0-00000000-000-0-0-1428232189

Seeing the Wood for the Trees

The outdoors is fabled to be great. Teedy Deigh suggests your code reflects your environment without ever having to look out of the window.

It's good to go outside and enjoy nature every now and then. Apparently.

For screen-hardened programmers this can present something of a challenge. Outside is a domain normally negotiated in order to get to and from the office, or for stealthy forays to the nearest supermarket to stock up on drinks and snacks to fuel a night of hacking on some open source or simply correcting someone in an online forum. If you want to actually admire it, well, that's what pictures on the web are for.

But it's possible to have your cake and eat it without even going to the supermarket! Look at your code. What do you see? The chances are, if you work in a proper enterprisey system, your code will reflect your environment: managers, proxies and singletons everywhere, with work avoidance, vague responsibilities and unclear methods characterising how tasks are partitioned, resources are allocated and goals are met.

Lots of bureaucracy and not a lot of action – and that's just the identifiers.

Names and titles matter, as any interim managing senior principal vice president of idempotent operations will tell you, but what we usually see in enterprice code is dominated by MBA thinking and tired industrial metaphors. It's not just the manager and controller and factory and service objects, but also the need, for instance, to clarify to readers that an object that is thrown as an exception and caught as an exception is an exception by including `Exception` in its name, or that a class defined as abstract needs to be named `Abstract` just in case the memo didn't get through. Droolproof paper is in increasingly short supply.

We see the division of labour in conventions that stretch from naming into the architecture. The traditional class struggle – where there are those who talk about the work and those who actually do it – runs through codebases like a naked Marxist through an overdressed stock exchange. It is not enough for objects to be manufactured, managed and stripped of behavioural responsibilities so they deliver little more than data: they must also conform to interfaces and contracts. Depending on convention, there is the me-centred `I` prefix, which is popular with millennial programmers, or there is the contrapuntal summoning of supernatural creatures that hold the promise of behaviour, the `Impl` suffix – a contraction of ImpWill.

The problem with these homeopathic naming conventions, where affixes are continually added to a name to dilute its meaning, is not simply that they reduce the informational content of information technology: they project, coerce and reinforce an industrial and post-industrial view of the world onto the semiotic space of our noosphere.

But it doesn't have to be like this.

Where is nature? Where are the rural styles of coding? Instead of software architecture, what of software agrarianism? It is possible to reclaim a more rustic and ecologically balanced approach to code, whilst still ensuring sufficient verbosity to win the enterprize. And all this can be done without setting foot outdoors!

Consider logging. This practice is rife in both tropical rainforests and enterpricey systems. It is generally arbitrary and unsustainable and few people are ever clear what the exact requirements are, so it spreads like a cat meme. It would be simple enough to pass a log in only when it is needed, or to define a need during construction rather than introducing a global trade dependency, but it is clear that to be taken seriously by the enterprison we must work closely with the existing architecture and start by rebranding.

For familiarity's sake we can keep the `Log` class, but where do logs come from? The conventional answer is a `LogFactory`. But no, work the metaphor: a `Forest`! And how should they be managed? By a `LogManager` – or by a `Lumberjack`? These should be defined in a `timber` package, with a specialisation for `SustainableForest` – which throws an `IllegalLogging` exception when overused – and a specialisation for `PoorlyManagedForest` – which is deprecated with immediate effect. Rather than write to a log, we can carve, and rather than dispose of a log, we can fell it.

Such subtle shifts in style allow the preservation of an enterpies growth model, but give developers the illusion of doing something worthwhile, offering them a simulated engagement with nature, but without all the nastiness of having to actually go outdoors. ■

Teedy Deigh believes in sustainable development, which she generally takes to mean a steady and sustainable flow of coffee, energy drinks and ersatz potato snacks in exchange for lines of code and a promise of no client contact (although it is not entirely clear from which side this promise is extracted...).