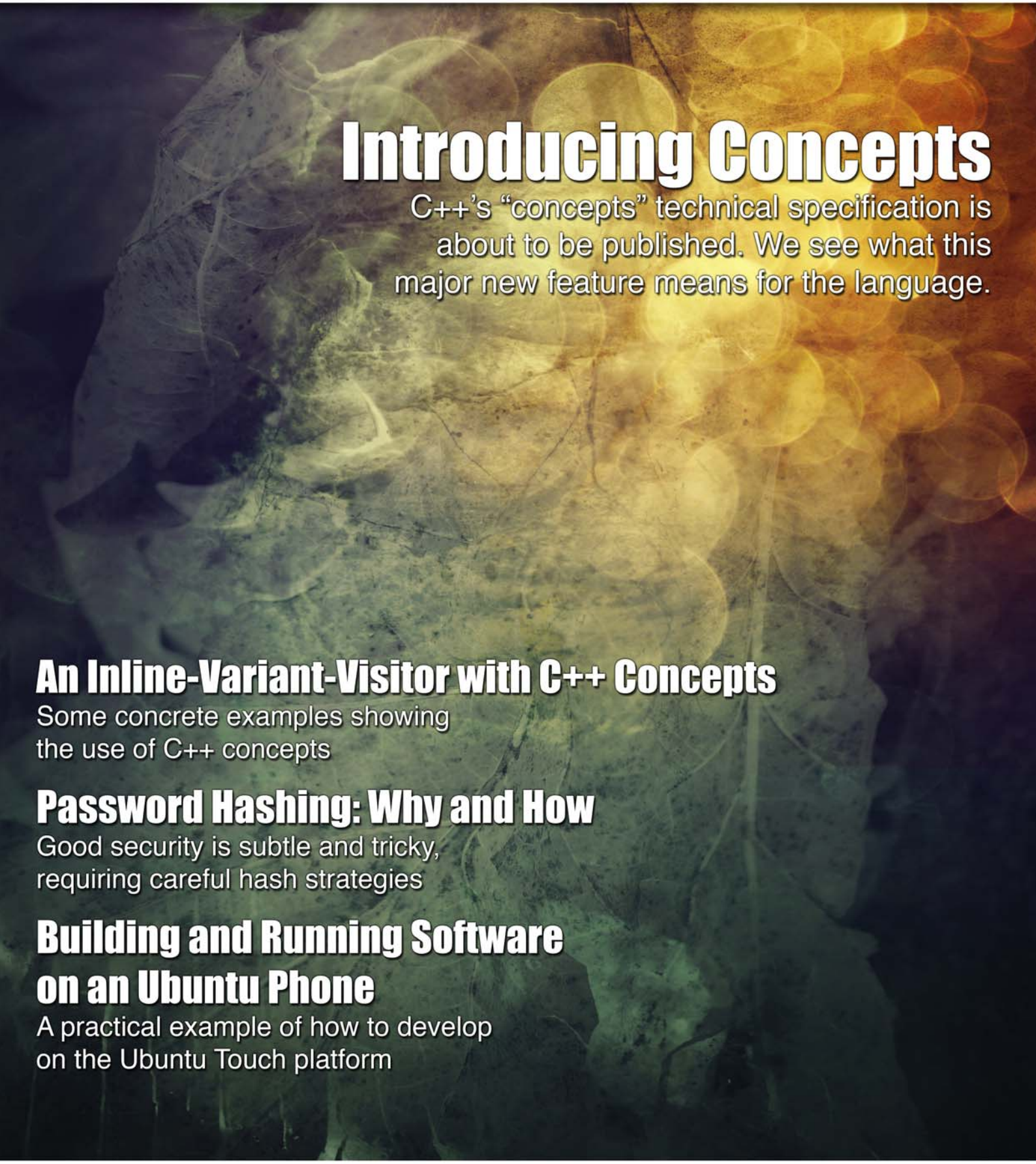
overload 129
OCTOBER 2015 £3



Introducing Concepts

C++'s "concepts" technical specification is about to be published. We see what this major new feature means for the language.

An Inline-Variant-Visitor with C++ Concepts

Some concrete examples showing the use of C++ concepts

Password Hashing: Why and How

Good security is subtle and tricky, requiring careful hash strategies

Building and Running Software on an Ubuntu Phone

A practical example of how to develop on the Ubuntu Touch platform

A magazine of ACCU
ISSN: 1354-3172

OVERLOAD 129

October 2015

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Matthew Jones
m@badcrumble.net

Mikael Kilpeläinen
mikael@accu.fi

Klitos Kyriacou
klitos.kyriacou@gmail.com

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Matthew Wilson
stlsoft@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 130 should be submitted by 1st November 2015 and those for Overload 131 by 1st January 2016.

The ACCU

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

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

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

Copyrights and Trade Marks

Failure is an option

Motivational speeches and aphorisms are clichés. Frances Buontempo wonders if they sometimes do more harm than good.

At the expense of sounding like a broken record, and sticking in a rut I have dug for myself, I still do not have a proper editorial for you. Given my persistent failure this should come as no surprise to you. Despite several of ways of trying to motivate myself, musing on how to do this has distracted me as ever. Unfortunately, no snappy one-liner managed to snap me out of it. Indeed, the first statement of Hippocrates' Aphorisms could, and did, leave one thinking for hours:

> Life is short, art long, opportunity fleeting, experience deceptive, judgement difficult. [Aphorisms]

A most pertinent adage, 'Failure is not an option' was impotent. As with many sayings in common parlance this stems from a film. Many proverbs originate from films, books or plays. In order to become well-known they need a vector to disseminate. Failure not being a possibility was used as a title of an autobiography by Gene Kantz, director of Mission Control for the Apollo 13 team. The phrase itself has seeds in an interview with people involved in the team. We are told:

> One of their questions was "Weren't there times when everybody, or at least a few people, just panicked?" My answer was "No, when bad things happened, we just calmly laid out all the options, and failure was not one of them. We never panicked, and we never gave up on finding a solution." [Failure]

Staying calm in the face of difficulties and trying to find a way to fix things is honourable, though I suspect many of us over our careers have been told "Failure is not an option" as a deadline attempts to go whooshing by, wherein we discover this is being misused as a code-phrase meaning you have to stay all night to make some software work. Though this has failed to motivate me to write an editorial, it did spark a train of thought about failure and if it is an option after all.

Consider for a moment how to get a grade A in an exam. Though a complete success, with a mark of 100% would be expected to gain an A, less than perfection will usually do. A measly 80% will often prove sufficient for a top grade. The extent to which an accomplishment must be a total triumph can vary with context. Watching a student heart-broken because they only got a B interact with a supportive parent who is delighted they passed with 'flying colours' is not uncommon. The different perspectives and hopes shade the result in different tones. Sometimes 80% sounds splendid, while at other times 4 out of 5 doesn't sound so good. If my mobile phone sends texts, allows internet access, has a camera and an alarm clock but will not make phone calls anymore, 80% is not good enough – this might indicate it's time for an upgrade. We can conclude "Failure is sometimes an option, or even considered success but it depends on context and the person involved." This is probably not pithy enough for a proverb or succinct enough for a saying, though.

If we cannot fail, how do we practise test-driven development (TDD)? Writing a failing test first is an important part of this discipline, even if just to make sure you get a clear and precise failure message. I have seen many people, yours truly included, write a test which happens to pass first time and then discover they need to break open the debugger if the test fails at a much later date when the code gets changed rather than just seeing everything they need to know in the test fail message. People not used to TDD are often surprised by how frequently the practitioner might use the compiler – if the language is compiled – feeling their approach of coding for a couple of days before resorting to kicking off the compiler and hoping for the best is vastly superior. The compiler is a tool of last resort, and perhaps can't really be leant on? (Oh yes it can – see [Feathers04].) The initial failure is important, though transitional. Sometimes you discover a bug as you add tests to legacy code and manage to write a test that characterises the problem. If you don't have time to fix the bug – for example it may be long and involved, or you would rather do one thing at a time to avoid getting distracted – most testing frameworks will allow you to mark the failing test as ignored. Ignoring failure is a transient option. Mind you, all code goes away in the end, so perhaps all code failures are transient. Avoid saying this out loud to your managers or customer though.

If you consider the use of the word failure in software development, you could conclude we expect it. We design failover clusters, fault tolerant systems, checksums so we can detect something went wrong. We catch exceptions – sometimes just logging them and carrying on regardless, though not always. Most 'automatic' algorithmic trading systems have a kill-switch, just in case something goes awry. In fact I can't think of a machine without an off-switch in my house. Admittedly, the off switch on my phone no longer works; however, I digress. Nature appears to build in some degree of fail-over. Most organisms have two lungs, kidneys and so on. If one fails they can still survive. On a smaller scale, biochemical functions are often encoded in two or more genes. This means things can function normally in the face of some mutation. Now, evolution would tell us that mutations cause the new normal. Genetic changes that start out as perceived failures can end up becoming the new norm provided that the mutant individuals do not die out. Apparent failure can lead to greater success. This may loiter behind the famed Silicon Valley phrase "Fail fast, fail often", not to be confused with fail-fast and may be closer to the idea of fail-safe. At very least we can say failure is interesting.

Having been considering my pension recently I noticed I might manage to retire by 2038, which is disappointing because I was looking forward

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She works at Bloomberg, 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.

to seeing how the next Y2k-style apocalypse/bug pans out. Some of us will remember the hard work put in by programmers to avoid the so-called Y2K bug – the media reported potential Armageddon when the clocks rolled over from 1999 to the year 2000 since many dates were just recorded with two digits. Many of us will be aware of the upcoming problem with storing UTC in a signed 32-bit integer – this will roll-over on a Tuesday in Jan 2038. I observe there are a few web pages up and ready for this. For example feel free to fill in the surveys (e.g. http://2038bug.com/) to let them know how aware the mass populace is of the problem. I hope we don't skew the results. At the start of the new millennium, many people complained that nothing went wrong. If consultants have been paid to fix a potential problem and they appear to have done so, complaining seems churlish. Perhaps public relations would have been better had one or two catastrophic failures been left behind, or sneaked into the system. Perhaps deliberately failing in order to look good is taking it too far.

Along with the idea of evolution continuing due to constant seeming failures or mutations, many inventors have succeeded at the wrong thing. For example post-it notes are fabled to have come from an attempt to build something else. Penicillin is often attributed to Fleming's discovery of mould on a petri dish, though ancient Egyptians are purported to have used mouldy bread in poultices on wounds [Penicillin]. Sometimes history has lessons of people succeeding for the wrong reasons. A case in point is the plague mask, which doctors wore when 'treating' people suffering from the bubonic plague in Europe. The mask was designed to stuff the beak with various herbs to hide the smell, the thinking being that the smell carried the disease. It seems that actually the material of the whole outfit was heavily waxed, which provided a hard barrier to the infectious fleas. I suspect you can think of many other examples of either outright failure or success for the wrong reasons.

Failure is not something to be feared. This is not an endorsement of deliberately sabotaging things, but an encouragement to try new things. As children we tend to be excited by new things, but as some people get older they become more anxious about trying new things. However, not all childhood experience is completely fearless, and failure seems less harsh if a supportive adult is to hand to smooth things over. As Batman's father says to him in *Batman Begins* when he falls down deep into the bat cave, "And why do we fall, Bruce? So we can learn to pick ourselves up," [Batman]. I hope to remain excited about trying out new programming languages or trying new technologies but do sometimes experience a twinge of worry when reading the documents or trying out a new machine for the first time. I choose to interpret this as adrenaline and carry on regardless. It is nice to have a supportive adult to share the experience with. If, no longer crippled by fear, you continue to practise TDD, try out new technologies or allow yourself to make mistakes, how do you respond to people around you when you think they are 'doing it wrong', to coin yet another phrase?

Before Cassandra became widely known as a database [Apache] she was more commonly known in Greek mythology as a woman cursed by never being believed, after fighting off an attempt at seduction. There are other stories, such as the boy who cried wolf, where the main character is not believed, having lied on previous occasions. Cassandra's curse lay in her foretelling truthfully what would come to pass. If Batman's father had stopped him playing near the end of the garden in case he fell down, we would not have Batman. If my parents had warned me I would fall off my bicycle when I tried to learn, that may have stopped me trying. Being a constant voice of gloom saying how things will probably go wrong is likely to lead to being ignored, but saying nothing if a path is fraught with danger is unproductive and probably cruel. If you are pair programming or code reviewing you need to choose a balance between banning someone from doing things their way and warning about unusual or downright dangerous approaches. If you resort to check-in gates – various automatic ways of enforcing code standards – people might rebel and try to find workarounds. Sometimes there is a genuine need to do something unusual. In this case a conversation rather than a convoluted hack might be more productive. If you warn somebody that recording every function call, with the precise parameters used, as an expectation in a mock might lead to brittle unit tests which need to change each time the code changes you might be proved right and might be listened to in the future. Banning such tests may have a different outcome. It is important to allow people space to fail while they learn. We learn from our mistakes.

Bad Unit Tests (BUTs [Henney15]) are one matter; they will usually become clear in the long run and can easily be deleted just like any other code. Other potentially dangerous behaviour might need 'nipping in the bud' or stopping quickly. I recall a colleague pointing out I might like to run a couple of SQL commands in a transaction, and though I muttered under my breath, they had a point. The SQL wouldn't have done quite what I intended and it would have taken a long while to sort out, had I not been able to simply roll-back to the previous state. The suggestion to use a transaction allowed me to fail, but safely. The support team who didn't use a transaction and managed to delete all of the risk figures for a year were a different matter. They were subsequently banned from making any C, U or D crud commands – strictly Read-only SQL for them from then on, with more dangerous commands saved for those who used transactions properly. Of course, if you want to get banned from over-night support this might give you ideas.

Failure has a time and a place. Perhaps you would rather not 'die on your feet' in a public situation, for example at demo to an important, or even unimportant, customer. [Are any customers unimportant?] If you run through first, check your specialist hard work works at their site, have two working laptops and so on, unexpected errors can still occur. [Are any errors expected?] The mark of your professionalism might be how you deal with the unexpected. I am not suggesting deliberately writing buggy software in order to look heroic by fixing the bug within hours of its discovery. Such an approach is bound to be discovered quicker than an unneeded sleep hiding in a loop, just so you can speed things up easily. Don't be afraid of failure, but rather try to create a safe place to fail while you learn. To end with another aphorism,

> "Ever tried. Ever failed. Try again. Fail again. Fail better." [Beckett]

References

[Apache] http://cassandra.apache.org/

[Aphorisms] Hippocrates. 'Aphorismi' according to the internet – see (for example) http://www.perseus.tufts.edu/hopper/text?doc=Perseus%3Atext%3A1999.01.0248%3Atext%3DAph

[Batman] *Batman Begins* Film, 2005 http://www.imdb.com/character/ch0000246/quotes

[Beckett83] *Westward Ho* Samuel Beckett, 1983

[Feathers04] *Working Effectively with Legacy Code* Michael Feathers, 2004.

[Henney15] 'What we talk about when we talk about testing' ACCU Conference 2015, see http://www.infoq.com/presentations/unit-testing-tips-tricks

[Failure] https://en.wikipedia.org/wiki/Failure_Is_Not_an_Option

[Penicillin] http://www.acs.org/content/acs/en/education/whatischemistry/landmarks/flemingpenicillin.html

Introducing Concepts

Concepts in C++11 had many false starts. Andrew Sutton show why they are a big deal now they are with us.

August is turning out to be a big month for C++. The Technical Specification for Concepts has been submitted to the ISO for publication [N4549], and its implementation has landed in GCC [GCC]. I've heard the phrase "This is a big deal" a number of times, but I think I like Eric Niebler's July 20th tweet the best: he wrote, "This will change everything" [Niebler15].

It may not be obvious why a new language feature would be such a big deal, or why it would have any significant impact on the way you write code. Think about the way that your C++ programming has changed since C++11 was published. How often do you use `auto`? What about initializer lists? Lambda functions? `constexpr`? Move semantics? Variadic templates? I've often heard it said that C++11 feels like a totally different language. It certainly does to me.

C++11 feels different because we now have language support for ideas that had previously been supported by obscure programming idioms, copious amounts of template metaprogramming, or extensive macro libraries (yuck!). The improvements introduced in C++11 obviate the need for many of those approaches, allowing C++11 programs to much more clearly reflect their designers' intents. These features allow ideas to be represented directly in the language and not hidden behind impenetrable walls of clever code.

The Concepts TS includes a number of improvements to better support generic programming by:

- allowing the explicit specification of constraints on template arguments as a part of a template's declaration,
- supporting the ability to overload function templates and partially specialize class and variable templates based on those constraints,
- providing a syntax for defining concepts and the requirements they impose on template arguments,
- unifying auto and concepts to provide uniform and accessible notation for programming generically,
- dramatically improving the quality of error messages resulting from the misuse of templates, and
- doing all of this without imposing any runtime overhead or significant increases in compile times, and also
- without restricting what can be expressed using templates.

This is the first article in a series introducing the language features included in the Concepts TS. This one focuses on the use of concepts to declare and constrain generic algorithms. Future articles will focus on concept definition and design, another on overloading and specialization, and potentially an article on generic data structure design with concepts.

Andrew Sutton is an assistant professor at the University of Akron in Ohio where he teaches and researches programming software, programming languages, and computer networking. He is also project editor for the ISO Technical Specification, 'C++ Extensions for Concepts'. You can contact Andrew at asutton@uakron.edu.

For those interested in getting started with concepts, information is available at http://asutton.github.io/origin/. This includes instructions for downloading and installing GCC from source (the implementation was merged into GCC 6.0, which has not yet been released). Note that the Concepts TS does not include library extensions for concepts. Predefined concepts and wrappers around standard facilities are provided by the Origin C++ Libraries. An abbreviated list of those concepts is given in the appendix.

Background

The idea of constraining template arguments is just as old as templates themselves [Stroustrup94]. However, it wasn't until the early 2000s that work started in earnest on a language design to provide those capabilities. That work culminated in what eventually became known as C++0x concepts [Gregor06]. The development of those features, and their application to the C++ Standard Library were a major focus of the C++ Standards Committee for C++11 [N2914]. However, those features were ultimately removed due to some significant unanswered questions and a looming publication deadline [Stroustrup09].

Work resumed (outside of the C++ Standards Committee) on concepts in 2010. Bjarne Stroustrup and I published a paper discussing how to minimize the number of concepts needed to specify parts of the Standard Library [Sutton11], and a group from Indiana University initiated work on a new C++0x concepts implementation in Clang [Voufo11]. As a result of the rekindled work, we (all of those actively working on concepts), were invited by Alex Stepanov (father of the STL) to take part in a week-long workshop with the goal of specifying all of the concepts needed to fully constrain the STL algorithms, and to suggest a language design that could express those constraints. The result of that work is published in the *Palo Alto* report [N3351].

When I presented this paper at the C++ Kona 2012 meeting, the committee was cautious about engaging in another large-scale experiment involving concepts. When I left that meeting, I did so with the goal of designing the minimum set of language features that would allow users to constrain templates like we did in N3351. Working with Bjarne Stroustrup and Gabriel Dos Reis, that initial goal evolved into a much more complete language extension called Concepts Lite [N3701].

Around that time, the C++ Standards Committee began using Technical Specifications (TS) to publish extensions to the language and library. This gives the committee the opportunity to gain implementation experience and user feedback before committing to a particular design.At the C++ Bristol 2013 meeting (the week after the ACCU 2013 conference), a work item for a TS on concepts was voted into existence. The Concepts TS [N4549] is the result of that work.

Constraining templates

Let's start with a generic algorithm that would be typical of something you can find in production today.

This allows the algorithm designer to clearly state what is expected, and it allows the compiler to check the use of the algorithm against those expectations

```
template<typename R, typename T>
bool in(R const& range, T const& value) {
  for (auto const& x : range)
    if (x == value)
      return true;
  return false;
}
```

Ostensibly, this function returns **true** if **value** can be found within **range**. We infer this from the definition because we have some expectations about the behaviour of **for** loops and equality comparisons. However, it would be better if we had some kind of annotation that expresses kinds of types that we can accept for **range** and **value**. Today, we do this using comments, external documentation, naming conventions, or some combination thereof. While that might be helpful for a programmer reading the definition of the function or its reference documents, that documentation is not enforceable by the compiler.

What happens when a developer inevitably misuses the template? Maybe he or she writes something like this:

```
vector<string> v { ... };
in(v, 0);
```

I tested with both GCC-4.9 and Clang-3.7. They both agree that this use of those arguments with **in** is incorrect, and they both agree that the reason is because **std::string**s cannot be compared for equality with integer values. Who knew?

Of course, I had to do a little digging to figure that out. GCC produced 162 lines of diagnostic messages, while Clang produced 57. Errors resulting from the incorrect use of templates are notoriously verbose and can often be quite difficult to parse. This example isn't especially bad: there's only one level of instantiation. Getting errors within a stack of deeply nested or even recursive template definitions is not at all uncommon. These kinds of errors tend to discourage the use of templates, especially among students and novice users.

These are two of the problems that concepts are designed to solve. We want to explicitly state requirements on template arguments as part of a template's declaration, and we want to check those requirements at the point of that template's use. This allows the algorithm designer to clearly state what is expected, and it allows the compiler to check the use of the algorithm against those expectations. If an error occurs, it can be diagnosed immediately.

Using the concepts TS, we could write the constrained version of the algorithm like this:

```
template<typename R, typename T>
  requires Range<R>()
        && Equality_comparable<T, Value_type<R>>()
bool in(R const& range, T const& value);
```

We can use a **requires** clause to specify constraints. The **requires** keyword is followed by a constant expression that, when instantiated, determines whether or not the template declaration can be used, or in this case, called.

Range and **Equality_comparable** are *concepts*. A concept is a constant expression involving template arguments that we can evaluate at compile time. Here, we use two predicates (functions returning **bool**s) to say that **R** must be a **Range** and that we must be able to compare values of **T** with the value type of **R** using **==**. The definitions of those predicates can be found in the Origin libraries and will discussed in depth in a future article.

Value_type is an alias that names the type of the contained objects. You can think of **Value_type<C>** as shorthand for **typename C::value_type**, although in practice, its definition is more complex.

When the **in** function is used, the compiler checks the **requires** clause against the deduced template arguments. Compiling the ill-formed example above with the concepts-enabled version of GCC gives the following (note type names have been simplified):

```
In function 'int main()':
error: cannot call function 'bool in(const R&,
const T&) [with R = vector<string>; T = int]'
   in(v, 0);
          ^
note:   constraints not satisfied
   in(R const& range, T const& value)
   ^
note:   concept
'Equality_comparable<Value_type<vector<string>>,
int>()' was not satisfied
```

The error is diagnosed at the point of use – where **in** is called – and not within its instantiation. The diagnostics clearly indicate the specific failure in the use of the template. Your experience with compiler diagnostics may vary; designing good diagnostics is hard, and the implementation of concepts is still a work in progress. However, any conforming implementation should be able to tell you that you cannot call **in** with those arguments because the constraints are not satisfied.

The Concepts TS also allows you to constrain class templates, variable templates, alias templates, and template template parameters (of course!). You can attach constraints to member functions and even non-member functions (seriously). Constraints can include predicates with non-type template arguments (**Prime<N>**, anybody?) and even template template arguments. However, these topics are outside the scope of this article, and many of them deserve a more thorough treatment than could possibly be given here.

Declaration style

In the example above we first said that we needed two type arguments and then, separately, what we required from those types. However, concepts allow us to state both at the same time. We could have defined **in** like this:

```
template<Range R,
  Equality_comparable<Value_type<R>> T>
bool in(R const& range, T const& value);
```

Here, both **Range** and **Equality_comparable<Value_type<R>>** declare type parameters. The concept named by each declaration

the syntax allows a variety of expressions, allowing a developer to judge for themselves what is appropriate in their work

determines the kind (and type) of the declared template parameter. Note that you can use the same notation to declare non-type template parameters, and even variadic templates.

When concepts are used to declare template parameters, the compiler internally transforms these into a **requires** clause. In fact, this declaration is equivalent to the previous one; the constraints are the same, so they both declare the same function. You can also combine the two notations. Here is another way of declaring the function **in**.

```
template<Range R, typename T>
  requires Equality_comparable<T, Value_type<R>>
bool in(R const& range, T const& value);
```

The Concepts TS defines other ways that you can declare this function, which will be introduced in subsequent articles.

The Concepts TS supports multiple notations for the declaration of constrained templates in order to provide flexibility in their presentation, and to improve readability and writability. Not all templates need a **requires** clause, but many do. If we had only added a **requires** clause, declarations would necessarily become more verbose. If we had only added a terse syntax, we would not be able to fully express the constraints for every template. As a result, the syntax allows a variety of expressions, allowing a developer to judge for themselves what is appropriate in their work.

Programming with placeholders

The overarching goal of the Concepts TS is to improve support for generic programming, and 'generic' is starting to show up everywhere in our source code. How many times have you written something like this?

```
for (auto iter = c.begin(); iter != c.end();
  ++iter)
  // Do stuff...
```

Or like this?

```
for (auto const& x : c)
  // Do stuff
```

These examples don't appear to be generic (where's the template?), and yet they are. The types of **iter** and **x** depend entirely on the type of **c**; **auto** simply acts as a placeholder for the deduced type of those variables. When you're trying to read the code, the resolution of that placeholder doesn't actually matter. On the surface, the **iter** and **x** have generic type.

One of the criticisms that I often hear about **auto** is that it hides information in the source text, and that the deduction rules can sometimes give unexpected errors or behaviours [Orr13]. In order to know what **iter** and **x** are, and how to use them correctly, we need to look at the declaration of **c** and understand the rules of type deduction.

One of the arguments for using **auto** is that it improves encapsulation by hiding type details [Sutter12]. Essentially, it helps you to write against the interface of the **c** such that changes to the definition of **c**'s type (or associated types) may not require changes to these **for** loops.

The Concepts TS provides two related features. First, it allows the use of a concept in place of **auto**. This is important. It allows us to gain the benefits of type deduction without the total loss of information. Contrast these examples with the ones above:

```
for (Pointer iter = c.begin(); iter != c.end();
  ++iter)
  // Do stuff

for (String const& x : c)
  // Do stuff
```

Here, **Pointer** and **String** are concepts, and they introduce placeholder types (just like **auto**). The deduced type is required to satisfy that constraint. That means, for example, that **iter**'s type is required to have the 'shape' **T***, for some **T**.

The second thing that the Concepts TS does is to allow placeholders nearly everywhere. For example, we can write declarations like this.

```
tuple<Number, String> t = make_tuple(0, "abc"s);
```

Number and **String** are concept names and therefore introduce placeholders. Here, the type of **t** is ultimately deduced as **tuple<int, string>**. The reason that we included this feature is that it more precisely allows us to specify the shape of a type. Here, we might not want a specific **tuple**, and we don't want any old **tuple**. We want one whose first element is numeric and whose second is a type of string.

The concepts TS also allows the use of placeholders in function parameters. We could declare **sort** like this:

```
void sort(Sortable_container& c);
```

This declares **sort** to be a function template with a single template parameter, and an associated constraint. It is equivalent to writing this:

```
template<Sortable_container C>
void sort(C& c);
```

Which is of course equivalent to this:

```
template<typename C>
  requires Sortable_container<C>()
void sort(C& c);
```

This particular extension of the language has been somewhat controversial. Some say, "it isn't easy to tell if the declaration is a template or not". But I don't think that matters too much. Developers tend not to look at actual declarations. If you've ever tried to read a Standard Library implementation, you will immediately understand why. Declarations often turn out to be macros, templates, or overloaded sets of functions. What's ultimately important about a declaration (or set of declarations) is how they can be used.

Supporting multiple styles of declaration of constrained templates allows flexibility in presentation. For simple generic functions, this use of placeholders is ideal. I can think of no more effective or elegant way to declare **sort** than the first declaration above.

However, this notation is not suitable for every template declaration. Many templates involve multiple template parameters and need more complex

A concept is not simply the minimal set of requirements for a particular implementation of a particular algorithm but a fundamental building block for our thinking and our code

requirements to fully describe their interactions. Bjarne Stroustrup refers to this as the *onion principle*. As he puts it, "the more layers you peel off, the more complicated things you can do, and the more you cry" [Stroustrup15].

The use of concept names as placeholders allows us to write our programs almost entirely in terms of abstract data types. I admit that I am excited to see the impact these features will have on the way C++ is written and taught.

Defining templates

While library developers have always had to consider a template's constraints, the language has not provided a means of enforcing them. Concepts gives us the means to enforce those constraints and a framework for thinking about them in a more concrete way. That said, adding constraints to a template declaration does not change the way that you write its definition. The concepts TS makes no changes to lookup rules (for better or worse), and there are no extra language rules or restrictions that apply to the definition of constrained templates.

When considering a template's constraints, we must select concepts that reflect coherent abstractions used within the algorithm. A concept is not simply the minimal set of requirements for a particular implementation of a particular algorithm, but a fundamental building block for our thinking and our code. The `in` algorithm requires two such building blocks: ranges and equality comparison.

A range type is any type that can be used in a range-based `for` loop. That is, every range `r` must provide the operations `begin(r)` and `end(r)`. Equality comparison requires the use of `==` and `!=`. Even though `!=` is not used by the `in` algorithm, it is nonetheless required. The building block of the algorithm is equality, not the set of expressions used in its implementation.

An implementation should use only the operations and types that are required of its arguments. Otherwise, we will get the same kinds of template errors that we have always gotten. However, this rule is not enforced by the Concepts TS. What if, at some point, we added some ad hoc debugging code to our algorithm?

```
template<Range R,
Equality_comparable<Value_type<R>> T>
bool in(Range const& range, T const& value) {
  for (auto const& x : range) {
    cout << x << '\n';
    if (x == value)
      return true;
  }
  return false;
}
```

Should we update the requirements? Doing so would almost certainly break existing code.

In the Concepts TS, it doesn't matter that streaming values of `T` is not part of the building blocks of the algorithm. However, if you supply a range of some non-streamable type, your program will fail to compile, and you will get the usual, lengthy template error messages.

Checking template definitions against their requirements would be useful, but we shouldn't do so at the expense of useful facilities such as debugging output, logging, timing, and statistics gathering. We are currently thinking about how best to support template definition checking.

Conclusions

The Concepts TS offers a number of improvements to better support generic programming. The ability to clearly and concisely state the constraints on template arguments and the abstractions of placeholders greatly improve the language's readability and usability. However, the features presented here only just scratch the surface. Concepts will fundamentally change the way that generic libraries are designed, implemented, and used. They will change how generic programming is taught. Concepts are a big deal, and this does change everything.

Acknowledgments

The design of the features in the Concepts TS was the result of collaboration with Bjarne Stroustrup and Gabriel Dos Reis. That material is based upon work supported by the National Science Foundation under Grant No. ACI-1148461. Bjarne Stroustrup also provided valuable feedback on an early draft of this paper.

Jason Merrill is responsible for merging the Concepts implementation into GCC, and has improved its usability significantly. Ville Voutilainen and Braden Obrzut have also contributed to the implementation.

The WG21 Core Working group spent many, many hours over several meetings and teleconferences reviewing the Concepts TS design and wording. This work would not have been possible without their patience and attention to detail. Many people have submitted pull requests to the TS or emailed me separately to describe issues or suggest solutions. I am grateful for their contributions. ■

References

[GCC] http://www.gcc.org

[Gregor06] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. 'Concepts: Linguistic support for generic programming in C++'. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA'06), pages 291–310, Portland, Oregon, 22-26 Oct 2006.

[Niebler15] Niebler, E (ericniebler). 'The Concepts TS was voted out today! Concepts are (almost) an ISO standard. Congrats, A. Sutton. This will change everything.' 20 Jul 2015. Tweet.

[N2914] Becker, P. (ed). Working Draft, Standard for Programming Language C++. ISO/IEC WG21 N2914, Jun 2009.

[N3351] Stroustrup, B., Sutton, A. (eds). A Concept Design for the STL. ISO/IEC WG21 N3351, Feb 2012.

An implementation should use only the operations and types that are required of its arguments. Otherwise, we will get the same kinds of template errors that we have always gotten.

[N3701] Sutton, A., Stroustrup, B., Dos Reis, G., Concepts Lite. ISO/IEC WG21 N3701, Jun 2013.

[N4549] Sutton, Andrew (ed). ISO/IEC Technical Specification 19217. Programming Languages – C++ Extensions for Concepts, Aug 2015.

[Orr13] Orr, R., 'Auto – a necessary evil (Part 2)', *Overload*, vol. 116, Aug 2013.

[Sutton11] Sutton, A. and Stroustrup, B. 'Design of concept libraries for C++. In *Proceedings of Software Language Engineering* (SLE'11). 3–4 Jul, 2011.

[Sutter12] Sutter, H. 'GotW #94 Solution: AAA Style (Almost Always Auto)'. Blog post. 12 Aug 2013.

[Stroustrup94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Stroustrup09] B. Stroustrup. 'No 'Concepts' in C++0x'. ACCU *Overload* vol. 92. Aug 2009.

[Stroustrup15] Personal communication, 30 August 2015.

[Voufo11] Larisse Voufo, Marcin Zalewski, and Andrew Lumsdaine. 'ConceptClang: an implementation of C++ concepts in Clang'. In *Proc. 7th ACM SIGPLAN Workshop on Generic Programming* (WGP'11), pages 71–82, 2011.

Appendix: Origin concepts

The following is a brief list of concepts provided by the Origin C++ Libraries and their requirements. Note that this is not intended to be an exhaustive list. These are based on the concepts defined in the Palo Alto TR [N3351].

Foundational concepts

These are the base concepts that reflect the value-semantic style of modern C++. They establish the basic requirements for construction, comparison, and the semantic foundation needed to support logical reasoning about programs.

- **`Movable<T>`** – Requires a type that can be both move constructed and assigned from an rvalue of type **`T`**.
- **`Copyable<T>`** – Requires a type that can be both copy constructed and assigned from an rvalue of type **`T`**. All copyable types must also be movable.
- **`Equality_comparable<T>`** – requires a type that can be compared using **`==`** and **`!=`**.
- **`Totally_ordered<T>`** – Requires a type that can be compared using **`<`**, **`>`**, **`<=`**, and **`>=`**. Totally ordered types must also be equality comparable.
- **`Equality_comparable<T, U>`** – Requires the equality comparison of values of type **`T`** and type **`U`**.
- **`Totally_ordered<T, U>`** – Requires the total ordering of values of type **`T`** and type **`U`**.
- **`Semiregular<T>`** – Requires a copyable type that can be default constructed.
- **`Regular<T>`** – Requires a semiregular type that is also equality comparable.
- **`Ordered<T>`** – Requires a regular type that is also totally ordered.

Functions

The function concepts in Origin are variadic concepts. The first template argument is the type of the callable object: either a function pointer, function object type, or a lambda closure type. The remaining template arguments are the types of the function arguments that the function type must accept.

- **`Function<F, Args...>`** – Requires a type **`F`** that can be called with the argument types in **`Args`**.... The return type is unspecified.
- **`Predicate<P, Args...>`** – Requires a function **`P`** that can be called with the argument types in **`Args`**... and returns **`bool`**.
- **`Relation<P, T>`** – Requires a binary predicate **`P`** that takes arguments of type **`T`**.
- **`Unary_operation<F, T>`** – Requires a function **`F`** that takes an argument of type **`T`** and has a return type of **`T`**.
- **`Binary_operation<F, T>`** – Requires a function **`F`** that takes two arguments of type **`T`** and has a return type of **`T`**.

Iterators and ranges

The iterator category is essentially the same as that in C++. All iterator types can be dereferenced and incremented.

- **`Input_iterator<I>`** – Requires an iterator type that can be used in single pass algorithms, and dereferencing can be used to read the iterator's value type.
- **`Output_iterator<I, T>`** – Requires an iterator type that can be used in single pass algorithms and values of type **`T`** can be assigned through its dereferenced result.
- **`Forward_iterator<I>`** – Requires an input iterator that can be used in algorithms requiring multiple passes over a range of values, or that require multiple simultaneous references to values in a range.
- **`Bidirectional_iterator<I, T>`** – Requires a forward iterator that can also be decremented.
- **`Random_access_iterator<I, T>`** – Requires a bidirectional iterator that supports constant time random access.
- **`Range<R>`** – Requires a type that can work with a range-based **`for`** loop.

Building and Running Software on an Ubuntu Phone

Many of us has a smartphone. Alan Griffiths shows us how to hack an Ubuntu Touch phone.

It has often been said that nowadays we carry around more computing power in our pocket than NASA used to put men on the moon. It is likely true but, so far, that computing power has not been readily available for general purpose computing. All that is changing. I've been working with Canonical on a project that is included in their 'Ubuntu Touch' phone operating system.

We've seen Linux on phones before (Google's Android has been very successful) but one thing different about Ubuntu Touch is that it uses a large part of the same software stack as you'll find on desktops and servers. Canonical are working on 'convergence' where the same software can provide either a phone or desktop experience depending on the connected hardware. Full 'convergence' isn't here yet but you can already accomplish serious computing tasks like building and running software entirely on the phone.

This is quite a cool concept and I thought I should share the experience.

The first thing you need is a suitable phone. I've a growing collection owned by Canonical (plus one I own myself – I got sucked into the coolness of carrying a general purpose computer around). You can find a list of the supported hardware at https://wiki.ubuntu.com/Touch/Devices – there you'll see that there are phones supported by Canonical as well as some community ports. The following works on Nexus 4 (mako), Nexus 10 (manta), BQ Aquaris E4.5 Ubuntu Edition (krillin) and Meizu MX4 Ubuntu Edition (arale) and I expect only small adjustments to be needed for other phones (like specifying an image server if the phone is not supported on the Canonical server).

I work on Ubuntu Desktop which means the desktop tools I mention below are readily available. On Ubuntu you can install these with:

```
sudo apt-get install ubuntu-device-flash
```

On other systems you'll have to work out the equivalent incantations. (They are in the packages android-tools-adb and ubuntu-device-flash.)

One final warning: if you simply want to develop Apps for the Ubuntu phone, this isn't the best approach (and there is plenty of guidance at http://www.ubuntu.com/phone/developers). What follows will instead help you with setting up a general purpose development environment on the phone.

Installing an OS image

Depending on what you want to work on you can, of course, skip this section and use the production image that came with a commercial Ubuntu Touch phone. If you're re-purposing a phone that has Android on it, or you want access to one of the pre-release versions, then you need to write or update the OS image. (I am working on parts of the software that runs the phone and fall into the latter category.)

Note that the tools and commands that follow can completely replace the existing OS and user data. Use backup appropriately!

In this section, and the following one, we assume there's a USB connection to your phone and that the phone is in 'developer mode'. Essentially, you need to go into **Settings** > **About this phone** > **Developer mode**. For security reasons connecting to the phone like this requires you to have a lock code set and for the phone to be unlocked with the screen on.

The channels providing OS images for various uses are listed here: https://developer.ubuntu.com/en/start/ubuntu-for-devices/image-channels/.

The tool used to manage installing the OS is ubuntu-device-flash. For example, to change the OS on a phone with an existing Ubuntu Touch installation:

```
ubuntu-device-flash touch   –channel=ubuntu-
touch/devel/ubuntu
```

If you want a clean installation (or to overwrite Andrioid) you need to get the phone into recovery mode (typically, power on with both volume buttons pressed) and add **–bootstrap** to the above command. NB you will lose any user data.

The ubuntu-device-flash touch tool does have decent **–help** options. For example, **ubuntu-device-flash query –help**.

After installation, start the phone up, go through the setup and tutorial setting a security code and connecting to your wifi.

Getting ssh working

In this section, like the previous one, we assume there's a USB connection to your phone and that the phone is in 'developer mode'.

The instructions here are copied from http://askubuntu.com/questions/348714/how-can-i-access-my-ubuntu-phone-over-ssh for convenience. These enable ssh and push your public key to the default 'phablet' account.

```
adb shell android-gadget-service enable ssh
adb shell mkdir /home/phablet/.ssh
adb push ~/.ssh/id_rsa.pub
  /home/phablet/.ssh/authorized_keys
adb shell chown -R phablet.phablet
  /home/phablet/.ssh
adb shell chmod 700 /home/phablet/.ssh
adb shell chmod 600
  /home/phablet/.ssh/authorized_keys
```

Next, assuming your phone is connected to your LAN find the phone's IP address:

```
adb shell ip addr show wlan0|grep inet
```

(You can also find the IP address through **System Settings** > **Wi-Fi** > '**>**'next to network name.)

Alan Griffiths has been developing software through many fashions in development processes, technologies and programming languages. During that time, he's delivered working software and development processes to a range of organizations, written for a number of magazines, spoken at several conferences, and made many friends. He can be contacted at alan@octopull.co.uk

Ubuntu Touch uses a large part of the same software stack as you'll find on desktops and servers...the same software can provide either a phone or desktop experience depending on the connected hardware

From now on you don't need the USB connection and can connect over your network:

```
ssh phablet@<IP from above command>
```

Note that the ssh service isn't started automatically so, if you restart the phone, it may need starting again. You don't need a USB connection to do this, just type `android-gadget-service enable ssh` in the terminal app.

Creating a chroot to work in

There are a couple of reasons to do development work in a chroot. Firstly the system partition is almost full, so installing the tools you want would likely overfill it. Secondly, the system image isn't managed using debian packages, but by downloading deltas. If you make changes to the system file system, things will break. Also, If you later re-flash the OS image while preserving user data you don't lose the chroot.

The following is pulled together from email discussions and steals ideas from a number of developers. I especially mention Michal Sawicz who offered the original of the script in Listing 1 (he deserves all the credit for the good bits but no blame for the bad bits as I've tweaked it since).

I've just explained that using apt on the phone makes it incompatible with the OTA updates – but I'm going to take a shortcut and install debootstrap temporarily. I hope this returns the system to its original state – it hasn't broken on me yet. It is definitely safe if you don't allow OTA updates.

Now in the ssh session you started at the end of the last section:

```
sudo mount -oremount,rw /
sudo apt-get install debootstrap
sudo mount -o remount,dev /home
sudo debootstrap vivid /home/phablet/chroots/vivid
sudo apt-get purge debootstrap
sudo mount -o remount,ro /
```

For convenience, set up a matching user account in the chroot. Listing 1 has a useful "ch" script that makes setting up the and using the chroot simpler.

To add the "phablet" user to sudoers in the chroot. In the ssh session:

```
sudo ~/bin/ch --root chroots/vivid/
```

Now, you're root in the chroot and type:

```
sudo adduser phablet sudo
passwd phablet
```

This set up the chroot "phablet" account to be able to use sudo. Now end the chroot root session and return to the ssh shell:

```
^D
```

Building some software

From the ssh session you get into the "phablet" account inside the chroot:

```
sudo ~/bin/ch chroots/vivid/
```

You'll use this command every time you want to switch into the chroot for development.

Phone development uses an 'overlay ppa' on top of vivid (a.k.a. 15.04) so you will want to add this to the chroot. This will continue into next year when it will switch to a 16.04 base. The following steps add the overlay:

```
sudo apt-get install software-properties-common
sudo add-apt-repository  ppa:ci-train-ppa-
service/stable-phone-overlay
sudo apt-get update
```

Still as phablet within the chroot you should install your favourite build tools. Here are the ones I use for Mir:

```
sudo apt-get install cmake g++ make bzr
```

```
#!/bin/bash

usage() {
  [ -n "$2" ] && ( echo $2; echo )
  echo "Usage:"
  echo "  sudo $0 [OPTIONS] CHROOT_PATH"
  echo
  echo "  Options:"
  echo "    -h|--help  displays this help message"
  echo "    -r|--root  chroot as root, not current
user"
  exit $1
}

ARGS=$(getopt -o r,h --long "root,help" -n "$0" --
"$@");

if [ $? -ne 0 ];
then
  usage 3
fi

eval set -- "$ARGS";

while true; do
  case "$1" in
    -h|--help)
      usage
      ;;
    -r|--root)
      shift;
      USERSPEC=0:0
      TARGET_UID=0
      ;;
    --)
      shift;
      break;
      ;;
  esac
done
```

Listing 1

```
setup_chroot() {
  # $1 is chroot path
  # $2 is user id

  # add current user in the chroot if doesn't exist
  chroot $1 id $2 &> /dev/null || useradd -R $1 -u
$2 -m phablet

  # make the prompt hint when chrooted
  echo `id -un ${TARGET_UID:-$2}`@`basename $1` >
$1/etc/debian_chroot

  mount -t proc none $1/proc
  mount -t sysfs none $1/sys
  mount --bind /dev $1/dev
  mount --bind /usr/lib/locale/ $1/usr/lib/locale/
  mount --bind $HOME $1/$HOME

  cleanup() {
    umount $1/proc $1/sys $1/dev $1/$HOME
      $1/usr/lib/locale/
  }
  trap "cleanup $1" EXIT
}

[ `id -u` == 0 ] || usage 2 "This script must be
ran as root."
[ -d  "$1" ] || usage 1 "You need to pass a valid
chroot path as argument."

CHROOT=`readlink -f $1` || exit 2
shift
BASE_UID=`id -u $( logname )`
BASE_GID=`id -g $( logname )`

setup_chroot $CHROOT $BASE_UID

chroot --userspec ${USERSPEC:-$BASE_UID:$BASE_GID}
$CHROOT $@
```

Listing 1 (cont'd)

Now, one ought to be able to set up building a project as normal. I use the Mir project on which I work as an example but you should vary as appropriate for your own needs. The relevant instructions for Mir are at http://unity.ubuntu.com/mir/building_source_for_pc.html. Sill as phablet in the chroot:

```
cd
bzr branch lp:mir mir
cd mir/
```

Now the next step tells you to `sudo apt-get install devscripts equivs cmake` but **`equivs`** isn't available in the phone archive so the incantation doesn't work. So instead of that command (and the one that follows), use what is in Listing 2.

Then resume the steps to build the project:

```
mkdir build
cd build
cmake ..
make -j2 all ptest
```

Depending on the phone you're using you may be able to start more processes (e.g. -j4), but at this stage you should have built Mir and run the basic test suites. A few of the tests may fail as a side effect of running in the chroot: don't worry about them. You can also install Mir in the chroot as normal:

```
sudo make install
```

Then exit the chroot for the top-level ssh shell:

```
^D
```

Running the software

Of course, installing software in a chroot is pretty limited (see the test failures mentioned above) but the following instructions for running the Mir "performance test" is an example of what you can do to work around this.

The performance test uses glmark and that is hard coded to find its data at /usr/share/glmark2. So that can work we once again make the file system writeable. I think adding an empty directory that can used for a "mount --bind" is unlikely to cause problems (but it is a risk you may wish to avoid by removing it after use). Once again from the top-level ssh shell:

```
sudo mount -oremount,rw /
sudo mkdir /usr/share/glmark2
sudo mount -o remount,ro /
sudo mount --bind ~/chroots/vivid/usr/share/
glmark2//usr/share/glmark2
```

Now, stop the graphics stack running the normal phone interface so that Mir can have sole access:

```
sudo stop lightdm
```

Now run the "performance test":

```
export MIR_CLIENT_PLATFORM_PATH=~/chroots/vivid/
usr/local/lib/mir/client-platform/
export MIR_SERVER_PLATFORM_PATH=~/chroots/vivid/
usr/local/lib/mir/server-platform/
export PATH=~/chroots/vivid/usr/bin:~/chroots/
vivid/usr/local/bin:$PATH
export LD_LIBRARY_PATH=~/chroots/vivid/usr/local/
lib/
mirbacklight
mir_performance_tests
```

Finally, start the graphics stack running the normal phone interface again.

```
sudo start lightdm
```

Conclusion

The above example shows how to go about working on Mir. Like most open source projects, that's entirely possible for you to do. If you're not interested in doing that I hope you will use these notes as a starting point for your own ideas. The possibilities will become ever more interesting as the Ubuntu Touch platform matures. ■

Acknowledgement

Thanks to Kevin Gunn, Alexandros Frantzis and the *Overload* team for their feedback, which improved the article.

```
sudo apt-get install cmake-data pkg-config debhelper doxygen xsltproc graphviz libboost-dev libboost-
date-time-dev libboost-program-options-dev libboost-system-dev libboost-filesystem-dev protobuf-
compiler libdrm-dev libegl1-mesa-dev libgles2-mesa-dev libgbm-dev libglm-dev libprotobuf-dev pkg-
config android-headers libhardware-dev libandroid-properties-dev libgoogle-glog-dev liblttng-ust-dev
libxkbcommon-dev libumockdev-dev umockdev libudev-dev libgtest-dev google-mock valgrind libglib2.0-dev
libfreetype6-dev libevdev-dev uuid-dev python3 dh-python glmark2-es2-mir
```

Listing 2

Password Hashing: Why and How

Password hashing is important. Sergey Ignatchenko explains how to do it properly.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Hare, 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.

Password hashing is a non-trivial topic, which has recently become quite popular. While it is certainly not the only thing which you need to do make your network app secure, it is one of those security measures every security-conscious developer should implement. In this article, we'll discuss what it is all about, why hash functions need to be slow, and how password hashing needs to be implemented in your applications.

What is it all about?

Whenever we're speaking about security, there is always the question: what exactly is the threat we're trying to protect ourselves from? For password hashing, the answer is very unpleasant: we're trying to mitigate the consequences arising from stealing the whole of your site's password database. This is usually accompanied by the potential for stealing pretty much any other data in your database, and represents the Ultimate Nightmare of any real-world security person.

Some (including myself) will argue that such mitigation is akin to locking the stable door after the horse has bolted, and that security efforts should be directed towards preventing the database-stealing from happening in the first place. While I certainly agree with this line of argument, on the other hand implementing password hashing is so simple and takes so little time (that is, if you designed for it from the very beginning) that it is simply imprudent not to implement it. Not to mention that if you're not doing password hashing, everybody (your boss and any code reviewers/auditors included) will say, "Oh, you don't do password hashing, which is The Second Most Important Security Feature In The Universe (after encryption, of course)."

The most important thing, however, is not to forget about a dozen other security-related features which also need to be implemented (such as TLS encryption, not allowing passwords which are listed in well-known password dictionaries, limits on login rate, etc. etc. – see 'Bottom Line' section below for some of these).

'No Bugs' Hare 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 currently holds the position of Security Researcher and writes for a software blog (http://ithare.com). Sergey can be contacted at sergey@ignatchenko.com

Attack on non-hashed passwords

So, we're considering the scenario where the attacker has got your password database (DB). What can he do with it? In fact, all the relevant attacks (at least those I know about) are related to recovering a user's password, which allows impersonation of the user. Subtle variations of the attack include such things as being able to recover *any* password (phishing for passwords), or to being able to recover *one specific* password (for example, an admin's password).

If your DB stores your passwords in plain text, then the game is over – all the passwords are already available to the attacker, so he can impersonate each and every user. Pretty bad.

Attempt #1: Simple hashing

You may say, "Hey, let's hash it with SHA256 (SHA-3, whatever-else secure hash algorithm), and the problem is gone!", and you will be on the way to solving the problem. However, it is more complicated than that.

Let's consider in more detail the scenario when you're storing passwords in a form of

P'=SHA256(P) (*)

where P is user-password, and P' is password-stored-in-the-database.

Dictionary attacks

So, an attacker has got your password DB, where all the passwords are simply hashed with SHA256 (or any other fast hash function), as described above in formula (*). What can he do with the database?

First of all, he can try to get a dictionary of popular passwords, and – for each such dictionary password – to hash it with SHA256 and to try matching it with all the records in your database (this is known as 'dictionary attack'). Note that using simple P'=SHA256(P) means that *the same P will have the same P'*, i.e. that *the same passwords will stay the same after hashing*.

This observation allows the attacker to pre-calculate SHA256 hashes for all the popular passwords once, and then compare them to all the records in your DB (or any other DB which uses the same simple hashing). While this kind of attack is certainly much more difficult than just taking an unhashed password, and therefore simple hashing is clearly better than not hashing passwords at all, there is still a lot of room for improvement.

Attempt #2: Salted hash

To deal with the issue when the hash is the same for the same password (which allows the pre-calculation for a dictionary attack, and also allows some other pre-calculation attacks, briefly mentioned below), so-called 'salted hashes' are commonly used.

The idea is the following: in addition to P' (user password – password-stored-in-the-database), for each user we're storing S – so-called 'salt'. Whenever we need to store the password, we calculate

P'=SHA256(S||P)

where || denotes concatenation (as string/data block concatenation).

none of the C++11 random number engines is considered good enough for cryptographic purposes ... they're way too predictable

As long as S is different for each user, the hashes will be different for different users, even if their passwords are exactly the same. This, in turn, means that pre-calculation for a dictionary attack won't work, making the life of an attacker significantly more complicated (at virtually no cost to us). In fact, 'salted hashes' as described above, defeat quite a few other flavours of pre-calculation attacks, including so-called 'rainbow table' attacks.

The next practical question is what to use as salt. First of all, S *must* be unique for each user. Being statistically unique (i.e. having collisions between salts for different users very unlikely) is also acceptable; it means that if you're using random salts of sufficient length, you're not required to check the salt for uniqueness.

Traditionally (see, for example, [CrackStation]), it is recommended that S should be a crypto-quality random number of at least 128 bit length, usually stored alongside the password in the user DB. And crypto-quality random numbers can be easily obtained from `/dev/urandom/` on most Linux systems, and by using something like `CryptGenRandom()` on Windows.

Note that, despite a very confusing wording on this subject in [CppReference], none of the C++11 random number engines (LCG, Mersenne-Twister, or Lagged Fibonacci) is considered good enough for cryptographic purposes – in short, they're way too predictable and can be broken by a determined attacker, given enough output has leaked. Overall, random number generation for security purposes is a very complicated subject, and goes far beyond the scope of this article, but currently the safest bet in this regard is to use Schneier and Ferguson's [Fortuna] generator with OS events fed to it as entropy input (this is what is implemented in most Linuxes for `/dev/urandom/`), or (if you do not have the luxury of having entropy on the go, but can get a cryptographic-quality seed), the so-called Blum-Blum-Shub generator.

However, some people (such as [Contini]) argue that using a concatenation of a supposedly unique site-ID with a unique user-ID (which is already available within the database) as S, is about as good as using crypto-random S. While I tend to agree with Contini's arguments in this regard, I still prefer to play it safe and use crypto-random S as long as it is easy to do so. At the very least, the 'playing it safe' approach will save you time if/when you need to go through a security review/audit, because you won't need to argue about using not-so-standard stuff (which is always a pain in the neck).

So, the suggested solution with regard to server-side salting is the following:

- to store S for each user in the same DB (there is no need to encrypt S, it can be stored as a plain text)
- whenever a password needs to be stored for user P, S (of at least 128-bit length) is taken from `/dev/urandom`[1] or from `CryptGenRandom()`
- store a (S,P') pair for each user, calculated as P'=SHA256(S||P), where || denotes concatenation. P must never be stored in the DB.

As discussed above, this approach is much more solid than simple hashing, but... there is still a caveat.

Prohibition on passwords in known dictionaries

Some may ask: "Hey, why bother with salting if we can simply prohibit users from using passwords from known dictionaries?" The answer to this question is the following:

> You ***do*** need ***both*** to prohibit passwords from known dictionaries ***and*** to use salt as described above.

Prohibiting dictionary-passwords is necessary even if passwords are 'salted', because dictionary attack is still possible; if the attacker looks for *one single* password, he can still run the whole dictionary against this specific password, whether it is salted or not (what the salt does is increase many-fold the cost of 'phishing' of *any* password out of DB).

Salt is necessary even if passwords from dictionaries are prohibited, because besides a dictionary pre-computation attack, there is a whole class of pre-computation attacks, including 'rainbow table'-based attacks. The idea behind pre-computed 'rainbow tables' is not trivial, and is beyond the scope of this article (those interested may look at [WikiRainbow]), but it is prevented by 'salting' in a pretty much the same way as a pre-computed dictionary attack is.

Offline brute-force attacks on fast hash functions

Even after we have added 'salt' to our P' as described above, and prohibited dictionary passwords, there is still a possible attack on our password DB ☹.

This attack is known as an offline brute-force attack, wherein an attacker has the whole password DB; it is different from a online brute-force attack, when an attacker simply attempts to login repeatedly (and which can and should be addressed by enforcing a login rate limit).

To mount an offline brute-force attack, the attacker needs to have the password DB (or at least one entry out of it, including the password and salt). Then the attacker may simply take this password and salt, and run all possible password variations through our SHA256(S||P) construct; as soon as SHA256(S||attempted-P) matches P' – bingo! attempted-P *is* the real password P for this user.[2]

Brute-force attacks, such as the one described above, are practical only if the number of possible passwords is quite small. If we had 2^{256} (or even a measly 2^{128}) different passwords for the attacker to analyze, a brute-force

1. Strictly speaking, you need to double-check the documentation of your target distribution to be sure that `/dev/urandom` generates crypto-quality numbers (or uses [Fortuna]), which is common, but not guaranteed. However, I would argue that for the purposes of generating salt S such double-checking is not 100% required.
2. Strictly speaking, a matching attempted-P may represent a hash collision, if there is more than one attempted-P which corresponds to (S,P') pair. However, for all intents and purposes attempted-P found in this way will be indistinguishable from the real password P; most importantly, it can be used for impersonation. Also after going through the full search space, the real P will be found too.

Prohibiting dictionary-passwords is necessary even if passwords are 'salted', because dictionary attack is still possible

attack wouldn't be feasible at all (i.e. all the computers currently available on the Earth wouldn't be able to crack it until the Sun reaches the end of its lifetime).[3]

However, the number of possible passwords (known as the 'size of search space') is relatively low, which opens the door for a brute-force attack. If we consider a search space consisting of all 8-character passwords, then (assuming that both-case letters and digits are possible), we'll get $(26+26+10)^8$~=2.2e14 potential passwords to try. While this might seem a large enough number, it is not.

Modern GPUs are notoriously good in calculating hashes; also note that the search task is inherently trivial to parallelise. In practice, it has been reported that on a single stock GPU the number of SHA256's calculated per second is of the order of 1e9 [HashCat]. It means that to try all the 8-character passwords within our 2.2e14 search space (and therefore, to get an 8-character password for a single user for sure), it will take only about 2.5 days on a single stock GPU ☹. As mentioned in [SgtMaj], this means that the upper-bound of the cost of breaking the password is mere $39. This is despite having used an industry-standard (and supposedly unbreakable) hash function, and despite the whole thing being salted ☹.

Note that the attack above doesn't depend on the nature of the hashing function. The attack doesn't depend on any vulnerability in SHA256; the only thing which the attack relies on is that SHA256 is a *reasonably fast hash function*.

Mitigation #1: Enforce long passwords

What can be done about these brute-force attacks? Two approaches are known in this field. The first approach is to enforce a minimum password length of longer than 8. This can be illustrated by Table 1, which shows that if we can enforce all users having relatively long passwords (at least 10–12 characters depending on the value of the information we're trying to protect), we might be OK. However, with users being notoriously reluctant to remember passwords which are longer than 8 characters, this might be infeasible; moreover, with the power of computers still growing pretty much exponentially, soon we'd need to increase the password length even more, causing even more frustration for users ☹.

Going beyond a password length of 12 isn't currently worthwhile; IMNSHO (in my not-so-humble opinion), any security professional who is trying to protect information which is worth spending half a billion to get with a mere password (i.e.without so-called 'two-factor authentication') should be fired on the spot.

Mitigation #2: Use intentionally slow hash functions

As noted above, to mount a brute-force attack, an attacker needs our hash function to be reasonably fast. If the hash function is, say, 100,000 times slower than SHA256, then the attack costs for the attacker go up 100,000-fold.

That's exactly what people are commonly doing to protect themselves from a brute-force attack on a stolen password DB – *they're using hash functions which are intentionally slow*.

Several intentionally slow hash functions have been developed for exactly this purpose, with the most popular ones being PBKDF2, bcrypt, and (more recently) scrypt. As of now (mid-2015), I would suggest scrypt – which, in addition to being intentionally slow, is specially designed to use quite a lot of RAM and to run quite poorly on GPUs while being not-so-bad for CPUs – or PBKDF2, if you need to keep your crypto NIST-standardized.

All such intentionally slow functions will have some kind of parameter(s) to indicate how slow you want your function to be (in the sense of 'how many computations it needs to perform'). Using these functions makes sense only if the requested number of computations is reasonably high.

The next obvious question is, 'Well, how big is this 'reasonably high' number of calculations?' The answer, as of now, is quite frustrating: 'as high as you can afford without overloading your server'. ☹

Note that when choosing load parameters for your intentionally slow hash function, you need to account for the worst-possible case. As noted in [SgtMaj], in many cases with an installable client-app (rather than client-browser) this worst-case scenario happens when you've got a massive disconnect of all your users, with a subsequent massive reconnect. In this case, if you've got 50,000 users per server, the load caused by intentionally slow hash functions can be quite high, and may significantly slow down the rate with which you're admitting your users back.[4]

Password Length	Search Space	Time on a Single GPU	Cost of Brute-Force Attack
8	2.2e14	~2.5 days	~$40
9	1.3e16	~5 months	~$2,400
10	8.4e17	~27 years	~$150,000
11	5.2e19	~1648 years	~$9.3M
12	3.2e21	~100,000 years	~$580M

Table 1

3. Rough calculation: 2^{128}=3.4e38. Now let's assume that there is a billion (1e9) cores on Earth, each able to calculate a billion hashes per second. It would mean that going through the whole 2^{128} search space will take 3.4e38/1e9/1e9=3.4e20 seconds, or approx. 1e13 years. As the lifetime of the Sun is currently estimated at about 5e9 years, it means that the sun will have enough time to die 2000 times before the search space is exhausted. And for 2^{256}, the situation becomes absolutely hopeless even if each and every atom of the Earth is converted to a core calculating a billion hashes per second.

4. While caching users' credentials to avoid overload at this point is possible, it is quite difficult to implement such a feature without introducing major security holes, and therefore I do not recommend it in general.

when choosing load parameters for your intentionally slow hash function, you need to account for the worst-possible case

Mitigation-for-mitigation #2.1: Client + Server hashing

To mitigate this server-overload in case of massive reconnects, several solutions (known as 'server relief') have been proposed. Most of these solutions (such as [Catena]), however, imply using a new crypto-primitive[5], which is not exactly practical for application development (that is, until such primitives are implemented in a reputable crypto library).

One very simple but (supposedly) efficient solution is to combine both client-side and server-side hashing. This approach, AFAIK, was first described in a StackExchange question [paj28], with an analysis provided in [SgtMaj].

Client + Server hashing

The 'Client + Server' password hashing schema works as follows:

1. User enters password P
2. P' is calculated (still on the client) as:
   `client_slow_hash(SiteID||UserID||P)`
   where `SiteID` is unique per-site string, `UserID` is the same ID which is used for logging in, `||` denotes concatenation, and `client_slow_hash` is any of the intentionally slow hash functions described above.
3. P' is transferred over the wire
4. on the server side, P'' is calculated as:
   `server_slow_hash(S||P')`
   where `server_slow_hash` may be either the same as or different from `client_slow_hash`, and `S` is a crypto-random salt stored within user DB for each user.
5. P'' is compared to P'' stored within DB. P' is never stored in database.

This approach shifts some of the server load to the client. While you still need to have both of your hash functions as slow as feasible, Client + Server hashing (when you have an installable client app rather than a browser-based app) may allow an increase from 10x to 100x of the brute-force-attack-cost-for-the-attacker [SgtMaj], which is not that small an improvement security-wise.

Note that while this Client + Server hashing might seem to go against the 'no-double hashing' recommendation in [Crackstation], in fact it is not: with Client + Server it is not about creating our own crypto-primitive (which is discouraged by Crackstation, and for a good reason), but rather about providing 'server relief' at minimal cost (and re-using existing crypto-primitives).

On the other hand (unless/until [WebCrypto] is accepted and widely implemented), this Client + Server hashing won't be helpful for browser-based apps; the reason for this is simple – any purely Javascript-based crypto would be way too slow to create enough additional load to bother the attacker.

5. A basic number-crunching crypto-algorithm, acting as a building block for higher-level protocols. Examples of crypto-primitives include AES, SHA256, and [Catena]. The problem with introducing a new crypto-primitive is that they're usually quite difficult to implement properly, so for application-level programmer it is usually better to wait until a crypto library does it for you.

WebCrypto

WebCrypto, more formally Web Cryptography API, is a W3C candidate recommendation, essentially aiming to provide access from JavaScript to fast browser-implemented crypto-primitives.

What MIGHT happen in the future

In the future, things might change. Very recently, a 'Password Hashing Competition' has been held [PHC], looking for new crypto-primitives which allow for better ways of password hashing; while they don't seem to apply any magic (so being intentionally slow will still be a requirement for them), there is a chance that one of them will become a standard (and becomes implemented by the crypto-library-you're-using) sooner or later. When/if it happens, most likely it will be better to use this new standard mechanism.

Bottom line

As long as a new standard for password-hashing is not here yet, we (as app developers) need to use those crypto-primitives we already have. Fortunately, it is possible and is reasonably secure using the approaches described above.

When implementing login for an installable client-app, I would suggest to do the following:

- Encrypt the whole communication with your client. If you're communicate using TCP, use TLS; if you're communicating using UDP, use DTLS; for further details see [NoBugs]. Sending password over an unprotected connection is something you should never do, NEVER EVER.
- Implement Client + Server Hashing as described above, configuring both client-side and server-side functions to be as slow as feasible
  - As of now, scrypt is recommended to be used on both client-side and server-side; if following NIST standards is a requirement, PBKDF2 is recommended.
  - For the client side, load parameters which are based on the maximum-allowable delay for the slowest-supported client hardware should be used.
  - For the server side, load parameters which are based on the maximum-allowable delay in the worst-possible-case (for example, in case of massive reconnect if applicable) for the server-hardware-currently-in-use.
- Set the minimum password length to at least 8
- Allow a maximum password length of at least 12, preferably 16
- Prohibit passwords which are in well-known password databases (and enforce this prohibition)

- Enforce password changes (which will be a separate and quite painful story)
- Do think how you will provide 'password recovery' when you're asked about it (and you will, there is absolutely no doubt about it). While 'password recovery' is a fallacy from a security point of view, there is 99% chance that you will be forced to do it anyway, so at least try to avoid the most heinous things such as sending password over e-mail (and if you cannot avoid it due to 'overriding business considerations', at the least limit the password validity time slot, and enforce that the user changes such a password as soon as she logs in for the first time).
- Implement two-factor authentication at least for privileged users, such as admins.
- Implement a login rate limit (to prevent online brute-force attacks)
  - With the precautions listed above, pretty much any reasonable limit will protect from brute-force as such (even limiting logins from the same user to once per 1 second will do the trick).
  - On the other hand, to avoid one user attacking another one in a DoS manner, it is better to have two limits: one being a global limit, and this one can be, say, one login per second. The second limit may be a per-user-per-IP limit, and this needs to be higher than the first one (and also may grow as number of unsuccessful attempts increases). With these two limits in place, the whole schema will be quite difficult to DoS.

Phew, this is quite a long list, but unfortunately these are the minimum things which you MUST do if you want to provide your users with the (questionable) convenience of using passwords. Of course, certificate-based authentication (or even better, two-factor authentication) would be much better, and if you can push your management to push your users to use it – it is certainly the way to go, but honestly, this is not likely to happen for 99% of the projects out there ☹. Another way is to rely on Facebook/whatever-other-service-everybody-already-has logins – and this is preferable for most of the apps out there, but most likely you will still need to provide an option for the user to use a local account on your own site, and then all the considerations above will still apply ☹.

For browser-based apps, the schema would be almost the same, except for replacing 'Implement Client + Server hashing...' with:

- Implement Client + Server hashing without `client_slow_hash`, i.e. with P'=P. Configure server-side function to be 'as slow as feasible'
  - As of now, scrypt is recommended to be used on both client-side and server-side; if following standards such as NIST is a requirement, PBKDF2 is recommended.
  - For the server side, load parameters which are based on the maximum-allowable delay in the worst-possible-case (for example, in the case of a massive reconnect if applicable) for the server-hardware-currently-in-use.

Note that when/if WebCrypto is widely adopted, browser-based apps should also move towards fully implemented Client + Server hashing as described for installable client-apps. ■

References

[Catena] Forler, Christian, Stefan Lucks, and Jakob Wenzel., 'Catena: A Memory-Consuming Password Scrambler.', IACR Cryptology ePrint Archive, 2013

[Contini] Scott Contini, 'Method to Protect Passwords in Databases for Web Applications', Cryptology ePrint Archive: Report 2015/387, https://eprint.iacr.org/2015/387

[CppReference] http://en.cppreference.com/w/cpp/numeric/random/rand

[Crackstation] 'Salted Password Hashing – Doing It Right', https://crackstation.net/hashing-security.htm

[Fortuna] https://en.wikipedia.org/wiki/Fortuna_%28PRNG%29

[HashCat] 'hashcat. advanced password recovery', http://hashcat.net/oclhashcat/

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

[NoBugs] 'No Bugs' Hare, '64 Network DO's and DON'Ts for Multi-Player Game Developers.Part VIIa: Security (TLS/SSL)', http://ithare.com/64-network-dos-and-donts-for-multi-player-game-developers-part-viia-security-tls-ssl/

[paj28] http://security.stackexchange.com/questions/58704/can-client-side-hashing-reduce-the-denial-of-service-risk-with-slow-hashes

[PHC] Password Hashing Competition, https://password-hashing.net/index.html

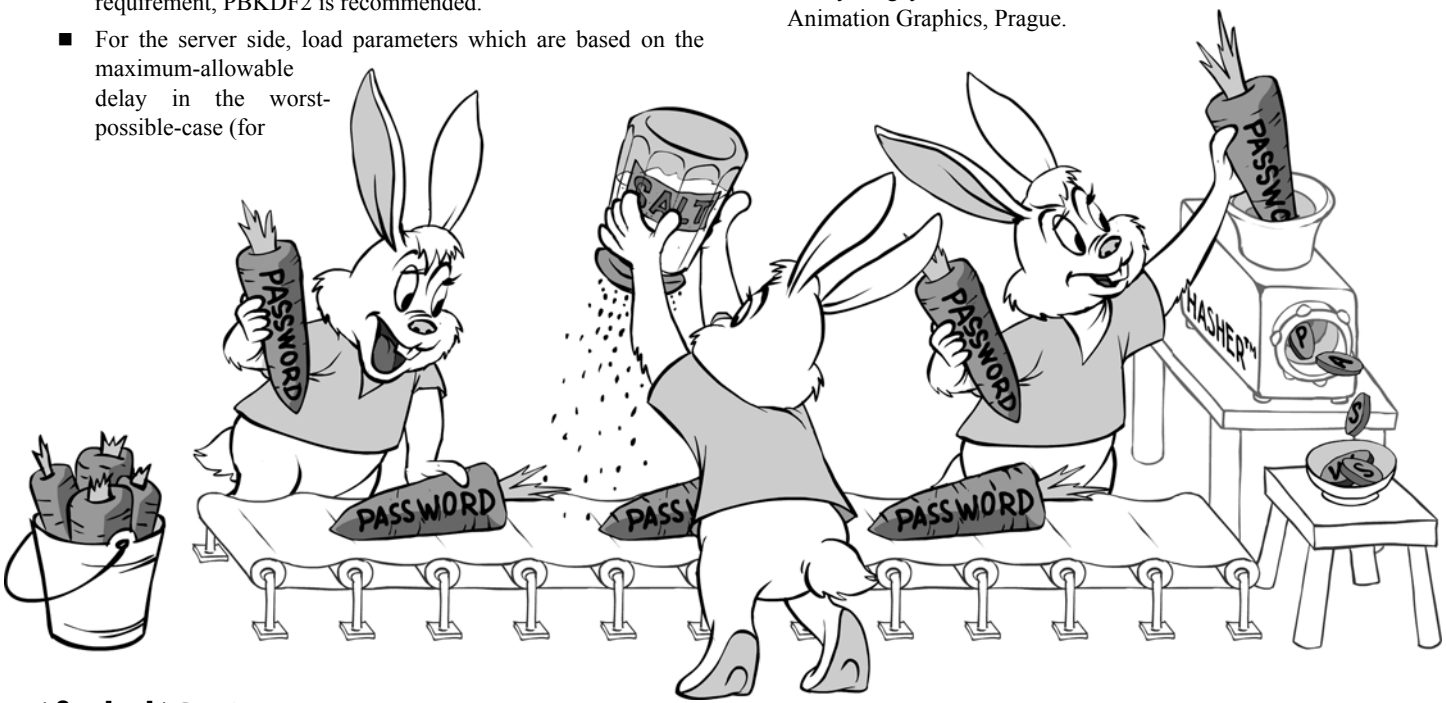
[SgtMaj] 'Sergeant Major' Hare, 'Client-Plus-Server Password Hashing as a Potential Way to Improve Security Against Brute Force Attacks without Overloading the Server', http://ithare.com/client-plus-server-password-hashing-as-a-potential-way-to-improve-security-against-brute-force-attacks-without-overloading-server/

[WebCrypto] http://www.w3.org/TR/WebCryptoAPI/

[WikiRainbow] 'Rainbow tables', Wikipedia, https://en.wikipedia.org/wiki/Rainbow_table

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.



An Inline-variant-visitor with C++ Concepts

Concepts are abstract. Jonathan Coe and Andrew Sutton provide us with a concrete example of their use.

Variants allow run-time handling of different types, without inheritance or virtual dispatch, and can be useful when designing loosely coupled systems. J.B.Coe and R.J.Mill [Mill14] previously showed a technique to avoid writing boilerplate code when writing visitors for the traditional inheritance-based VISITOR pattern. We modify the inline-visitor technique to handle variant-based visitation and use Concepts [Sutton13] from the Concepts TS [Concepts] to switch run-time behaviour depending on syntactic properties of the class encountered.

Variants

A variant is a composite data type formed from the union of other types. At any point the type of the data contained within a variant can only correspond to one of the composite types. Variants are useful when data flow through a program needs to correspond to multiple possible types that are not logically related by inheritance. For instance, the data within a spreadsheet cell could be an integer, a floating point value or a text string. We can represent such a spreadsheet cell by a variant of **`int`**, **`double`** and **`string`**. We can use this variant type within our program as a function argument type or return type and defer type-specific handling to a later point.

The boost libraries offer a variant type [Boost] as does the eggs.variant library [Eggs-1]. There are differences between the different offerings of variant but for the purposes of our inline visitor they are much the same and could be implemented as a union of types with an index used to determine which type is active (see Listing 1).

Our code examples focus on eggs.variant which is a lot more sophisticated than the exposition-only variant class above (especially where **`constexpr`** methods are concerned) [Eggs-2].

Design decisions about exception safety and default construction are interesting and, at the time of writing, under much discussion among the ISO C++ standards body, but beyond the scope of this paper.

```
class ExpositionOnlyVariant
{
  union Data_t
  {
    int i_;
    double d_;
    string s_;
  };

  Data_t data_;
  size_t active_type_;

  size_t which() const { return active_type_; }

  // Other non-illustrative methods and
  // declarations
};
```

Listing 1

```
using Cell = variant<int, double, string>;

template<typename R, typename Visitor>
R apply(const Cell& c, Visitor&& v)
{
  switch (c.which()) {
    case 0: return v(get<0>(c));
    case 1: return v(get<1>(c));
    case 2: return v(get<2>(c));
    default: throw std::logic_error
      ("Bad variant type");
  }
}

struct DoubleCell_t
{
  int operator()(int n) { return n*2; }
  double operator()(double d) { return d*2; }
  string operator()(string s) {
    return s.append(s); }
};

Cell DoubleCell(const Cell& c)
{
  return apply<Cell>(c, DoubleCell_t());
}
```

Listing 2

A visitor For variants

The Gang of Four [GoF] describe the VISITOR pattern as "Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."

We can use visitors to implement the type-specific handling our variant may require.

Writing a visitor on a variant type is a straightforward matter that entails determining the value of the discriminator and dispatching to a function that accepts the corresponding concrete type. For example, we could explicitly build a function that applies a visitor for our previously mentioned spreadsheet. See Listing 2.

Jonathan Coe has been programming commercially for about 7 years. He has worked in the energy industry on process simulation and optimisation and is currently employed in the financial sector. You can contact Jonathan at jbcoe@me.com

Andrew Sutton is an assistant professor at the University of Akron in Ohio where he teaches and researches programming software, programming languages, and computer networking. He is also project editor for the ISO Technical Specification, 'C++ Extensions for Concepts'. You can contact Andrew at asutton@uakron.edu.

if we could predetermine sets of abstract data types to which the visitor can be applied, then we could define a small set of generic functions to accommodate those behaviours

Generic code with templates

Templates in C++ allow the definition of functions and classes with generic types [Vandevoorde02]. We can, for instance, define a generic function that doubles its argument.

```
template<typename T>
T double(const T& t)
{
  return t*2;
}

assert(8.0 == double(4.0));
```

We can define an `operator()` template for a visitor which will mean it can be invoked on any type. This allows us to specify default behaviour for types that are not explicitly handled (function templates have lower precedence when it comes to overload resolution so a more specific function will be picked where available). See Listing 3.

This is appealing; we can define default behaviour for our visitor (the generic member function) and define exceptions to this default behaviour (with specific overloads). In this case, the generic version uses `operator*` for all non-`string` members of the variant, and the `string` version resorts to an `append` member function.

This could become irksome if there were many exceptions, and we have to know about types which need special handling ahead of time. For example, if we later extend `Cell` to include support for wide-character strings (e.g., `std::u16string`), then we would need to add overloads to each visitor function to provide the required functionality. In addition, there's no way we could reasonably expect our visitor to cope with arbitrary user-defined types. Adding date or time representations to `Cell` would require even more overloads.

However, if we could predetermine sets of abstract data types to which the visitor can be applied, then we could define a small set of generic functions to accommodate those behaviours. That is, some methods would use `operator*`, while others used `append`, some might throw exceptions, and so on. Concepts make this easy.

```
struct DoubleCellWithGenerics_t {
  template<typename T>
  T operator()(const T& t) { return t*2; }

  string operator()(string s) {
    return s.append(s); }
};

Cell DoubleCellWithGenerics(const Cell& c) {
  return apply<Cell>(c,
  DoubleCellWithGenerics_t());
}
```

Listing 3

```
struct DoubleCellWithConcepts_t {
  int operator()(const Numeric& n) { return n*2; }
  string operator()(Appendable s) {
  return s.append(s); }
};

Cell DoubleCellWithConcepts(const Cell& c) {
  return apply<Cell>(c,
  DoubleCellWithConcepts_t());
}
```

Listing 4

Constraining templates with Concepts

Concepts are an extension to the C++ Programming Language, defined as an ISO Technical Specification [Concepts]. The Concepts extension allows the specification of constraints on template arguments in a straightforward, and often minimal way. While, templates can be restricted using SFINAE tricks with `enable_if` [Vandevoorde02], Concepts provides a more readable notation and vastly improved support for overloading.

Listing 4 is a fully generic visitor written using concepts.

We only need two overloads to fully accommodate the breadth of abstractions included in our `Cell` representation. The first overload takes an argument of type `Numeric` and the second an argument of type `Appendable`. These are concepts. A concept is a compile-time predicate that defines a set of requirements on a template argument.

When a concept name is used in this way, it declares a function that accepts any type that satisfies the requirements of that concept. To help make that more explicit, we could have declared the first overload like this:

```
template<typename T>
  requires Numeric<T>()
int operator()(T n) { return n*2; }
```

Here, `Numeric` is explicitly evaluated by a `requires` clause. Of course, this is more verbose so we prefer notation above.

With this visitor, the `Numeric` overload defines the behaviour for the `int` and `double` members of `Cell`, For that matter, this will be the behaviour for any numeric type that might include in the future (e.g., vectors or matrices?). Similarly, the `Appendable` overload defines the behaviour for all sequence-like objects.

In this example, `Numeric` and `Appendable` are function templates, declared with the `concept` specifier. Here is the definition of `Appendable` that we use above:

```
template<typename T>
concept bool Appendable() {
   return requires(T& t, const T& u) {
     { t.append(u) } -> T&;
   };
}
```

Concepts are effective ways of specifying syntactic requirements. However, a meaningful concept definition also needs semantic requirements

The `requires` expression in the body of the function enumerates the syntactic requirements to be satisfied by a type `T`. Here, we require `T` to have a member function `append`, taking an argument of the same type. The return value, specified by the `-> T&` must be implicitly convertible to `T&`. If any of those requirements are not satisfied, the concept is not satisfied.

Concepts are effective ways of specifying *syntactic* requirements. However, a meaningful concept definition also needs *semantic* requirements. What is `append` required to do? The concept used in this example is not especially well-designed because it isn't obvious how to specify the semantics of an `Appendable` type, when we have only a single operation. Additional requirements like `size` and `back` would be helpful. In the future, we will endeavour to present only complete concepts.

Concepts are a very powerful tool and their first-class status in terms of compiler diagnostics make their consumption by non-expert users much more straightforward. Furthermore, the use of concepts will never add runtime overhead to your program. A technique like type-erasure could be used instead, but it can be difficult to implement, and it adds hidden runtime costs.

The Concepts TS has been sent for publication by ISO and is implemented in GCC trunk [GCC].

An inline-variant-visitor with Concepts

Defining the visitor outside the function in which it is used is necessary if it has generic functions. The inline-visitor pattern for the traditional visitor can be adapted to variant visitors and allow definition of concept-constrained generic functions inline at the point of use.

```
auto DoubleCell(const Cell& c)
{
  auto inline_cell_visitor
     = begin_variant_visitor()
    .on([](Numeric n) { return n*2; })
    .on([](Appendable a) { return a.append(a); })
    .end_visitor();
  return apply<Cell>(c, inline_cell_visitor);
}
```

This function is now hardened against future changes in the definition of the variant, `Cell`. Adding a new `Appendable` member to that variant would not require changes to this function. In fact, unless a future change adds entirely new categories (i.e., concepts) of members to the variant, we should never have to modify this function again (unless it contains a bug).

The inline-variant-visitor is implemented in much the same way as the inline-visitor [Mill14]: an inner composable function object is recursively built up and finally instantiated.

Conclusion

We have presented a method for generating visitors inline for variants where the run-time type-specific handling can be specified in generic terms depending on syntactic properties of the run-time type. Concept-based handling of variants can facilitate the writing of generic header-only libraries that make use of `variant<Ts...>` arguments. Type-specific handling of user-defined types, unknown to the library author, can be simply specified in terms of supported concepts.

Although there are slight differences in optimised assembler output with and without the inline-variant-visitor, there is no measurable run-time penalty for its use. ■

Appendix

Implementation of a concept-enabled inline-variant-visitor (Listing 5).

```
#include <string>
#include <utility>

// These using declarations are for publication
// brevity only
using std::make_pair;
using std::move;
using std::pair;
using std::nullptr_t;

template <typename F, typename T> concept bool
UnaryFunction() {
  return requires(const F &f, const T &t) {
    { f(t) }
  };
}

template <typename F, typename BaseInner,
  typename ArgsT>
struct ComposeVariantVisitor {
  struct Inner : BaseInner {
    Inner(ArgsT &&a) : BaseInner(move(a.second)),
    f_(move(a.first)) {}

    using BaseInner::operator();

    template <typename T>
    requires UnaryFunction<F, T>()
      auto operator()(const T &t) {
      return f_(t);
    }

  private:
    F f_;
  };

  ComposeVariantVisitor(ArgsT &&args) :
    m_args(move(args)) {}
```

Listing 5

```
  template <typename Fadd> auto on(Fadd &&f) {
    return ComposeVariantVisitor<Fadd, Inner,
        pair<Fadd, ArgsT>>(
        make_pair(move(f), move(m_args)));
  }

  auto end_visitor() { return Inner(move(m_args)); }
}

  ArgsT m_args;
};

struct EmptyVariantVisitor {
  struct Inner {
    struct detail_t {};

    Inner(nullptr_t) {}

    void operator()(detail_t &) const {}
  };

  template <typename Fadd> auto on(Fadd &&f) {
    return ComposeVariantVisitor<Fadd, Inner,
        pair<Fadd, nullptr_t>>(
        make_pair(move(f), nullptr));
  }
};

EmptyVariantVisitor begin_variant_visitor() {
   return EmptyVariantVisitor(); }
```

Listing 5 (cont'd)

References

[Boost] http://www.boost.org/doc/libs/1_59_0/doc/html/variant.html

[Concepts] Sutton, Andrew (ed). ISO/IEC Technical Specification 19217. Programming Languages – C++ Extensions for Concepts, 2015.

[Eggs-1] http://eggs-cpp.github.io/variant/

[Eggs-2] http://talesofcpp.fusionfenix.com/post-20/eggs.variant---part-ii-the-constexpr-experience

[GCC] svn://gcc.gnu.org/svn/gcc/trunk

[GoF] E. Gamma et al., *Design Patterns*. Addison-Wesley Longman, 1995.

[Mill14] Robert Mill and Jonathan Coe, 'Defining Visitors Inline in Modern C++'. *Overload* 123, October 2014.

[Sutton13] Andrew Sutton, Bjarne Stroustrup, 'Concepts Lite: Constraining Templates with Predicates'. https://isocpp.org/blog/2013/02/concepts-lite-constraining-templates-with-predicates-andrew-sutton-bjarne-s

[Vandevoorde02] Vandevoorde, David; Nicolai M. Josuttis (2002). *C++ Templates: The Complete Guide* Addison-Wesley Professional. ISBN 0-201-73484-2.