overload 115

JUNE 2013 £3

Demons May Fly Out Of Your Nose

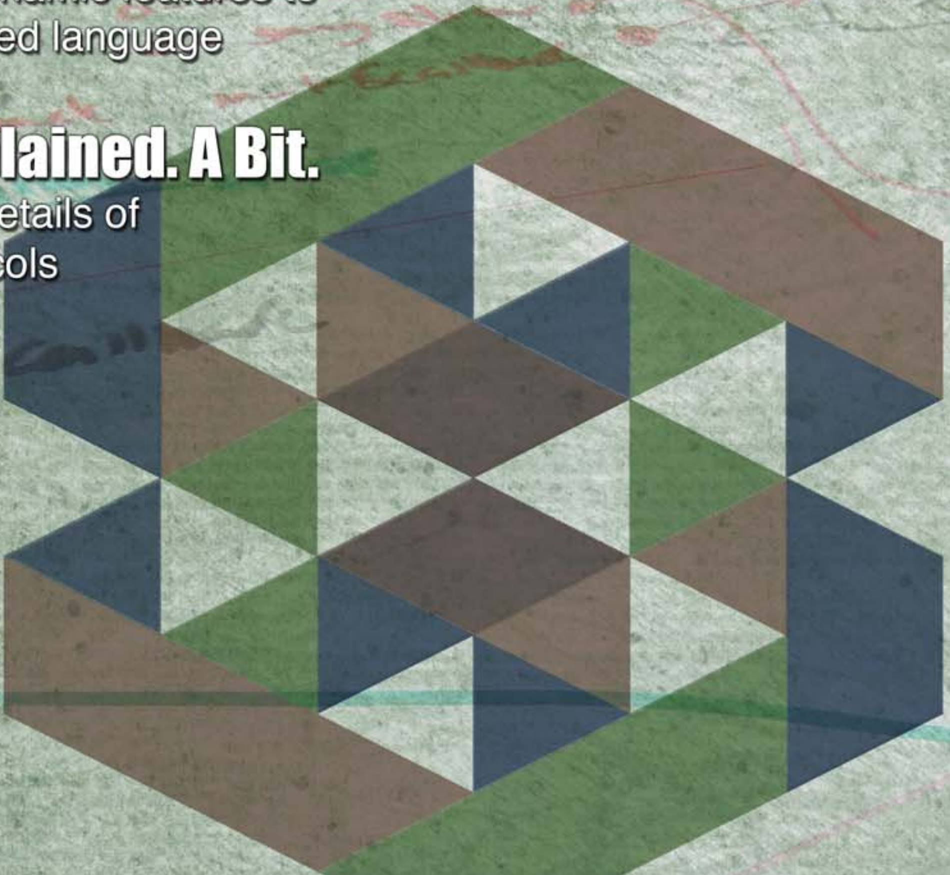
We investigate the murky world of Undefined Behaviour

Dynamic C++, Part 1

How to add dynamic features to a statically typed language

TCP/IP Explained. A Bit.

A look at the details of network protocols



Auto – A Necessary Evil?

We look at one of the new C++11 language features, investigating its advantages and disadvantages

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 115

June 2013

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Guest Editor

Ric Parkin
ric.parkin@gmail.com

Advisors

Matthew Jones
m@badcrumble.net

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simonsebright@hotmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 116 should be submitted by 1st July 2013 and for Overload 117 by 1st September 2013.

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

Fantasy Languages

Software is all about describing a solution to a computer. Ric Parkin imagines what his ideal dialect would sound like.

Hello there, and welcome to Overload 115. Firstly, to avoid confusion, Frances has been off to enjoy a well earned holiday and kindly asked me to step in and guest edit this issue. Thankfully the well-oiled ACCU publication machine is working as well as ever, and by taking advantage of the captive speakers at the conference in Bristol I've managed to find plenty of articles for this issue – and promises for the future.

It was the first conference that I've attended for a few years – my excuse was that having to write editorials meant I'd failed to come up with any decent talk ideas, but this year my company kindly paid for me to attend. And it was as good as ever with a packed programme where, as usual, there would be at least two or three talks at a time I'd like to attend. It was also a great opportunity to meet old friends and put faces to people I only knew as an email address.

C++: the past, the present, and the future

While there is always a wide range of subjects across the talks, each conference tends to have an overall theme – sometimes intentionally, sometimes it emerges naturally depending on what's new and interesting to people. This year's was definitely about the evolution of C++. Many talks looked at how the new features in C++11 work in practice, and how different features interact – always the bit where the unexpected appears for both good and bad.

A good example of these serendipitous synergies is from the previous standard, when the interaction of template specialisation, non-type template parameters, and some limited compile time evaluation resulted in the discovery of Template Meta-Programming (which some consider a mixed blessing!) For all its warts (and the error messages – sorry, essays – that can result are most definitely 'interesting' [Curse]) TMP has been hugely influential which has attracted attempts to improve it – for library writers, and for the users trying to use the resulting libraries (especially when working out why their attempt has failed.)

It is notable that a lot of the recent standardisation effort has gone into improving many TMP techniques, helping better compiler support, standardising utilities such as `enable_if`, and aiming to provide some sort of Concepts support. Sadly the original Concepts got dropped from C++11 when it was realised that it was becoming too big and unwieldy and risked jeopardising the next standard [Stroustrup09]. More happily, for the next standard – dubbed C++14 [C++14] – as well as mainly being some tidying up, a few new bits and pieces here and there, and relaxing some of the restrictions in C++11, there is also ongoing work towards one major new feature – Concepts-Lite – that should deliver many of the hoped for benefits of the original Concepts without becoming overly-complex. There's even a compiler that has already implemented it to get some idea of how it works in practice [Sutton].

Beyond that is even more work on C++17. You may have noticed that after the marathon to get C++11 out, the committee has decide that a shorter 3-year rolling release cycle was needed, which enables small tweaks to make it into the 'Official' standard rather than an interim Technical Report, while allowing ongoing work for large features to aim for a release further ahead. One of these ideas are to extend Concept-Lite even further to make the more complex cases even easier to write and use. Some of the possible ideas were shown at the conference, and valuable feedback was gleaned which resulted in some rapid updates to the syntax ideas.

A fantasy programming language

This is all tremendously exciting, although I do fear for my bookshelves that will have to deal with the resulting tomes. But it has made me wonder what sort of language features would be in my 'Ideal Language' – one which didn't have to deal with backwards compatibility with previous versions (although it can be persuasively argued that backward compatibility with C was a major reason for the take up of C++, despite it leading to some design decisions that you might have not have chosen from afresh). So this is my wish-list – it's not particularly comprehensive, or even thought through very much so forgive the inevitable clashes and contradictions between ideas. After all, that's where the interesting things happen.

The first thing to keep in mind is the difference between the Language, the Standard Library, other available libraries, the compiler and other tools. Many people cite their opinion that language X is better than Y, but often they mean it has more libraries out of the box, or it has a nice IDE, or lots of useful tools. These are indeed important but each are different.

The core language should be as lean as possible, but allow a rich set of tools to be built upon it, including the standard libraries and others. This implies several simple but orthogonal features, with tools for abstraction and encapsulation to build new types and libraries.

The Standard library should provide a decent set of tools and types that are likely to be common to most programmes on most platforms. A key idea would be to provide an extendable framework to allow different extensions to be combined in unforeseen ways. The design of the original STL is instructive here – by having iterators as the 'glue' between algorithms and containers, you could write new components that provided iterators e.g. number generators, then existing and future algorithms would work with them.

We would also like lots of libraries to be available from third parties – these would cover bits that we have inevitably missed, or more niche uses. These will depend a lot on how easy such libraries are to write and combine, and so depend on the whole language package.

Ric Parkin has been programming professionally for around 20 years, mostly in C++, for a range of companies from tiny startups to international corporations. Since joining ACCU in 2000, he's left a trail of new members behind him. He can be contacted at ric.parkin@gmail.com.

The compiler and tools are an odd one, but very important. We'd want our language to be easy to write the tools for, and easy to extend. C++ has traditionally been very hard to parse and slow to compile, and so has become neglected somewhat compared to other languages. More recently tools such as Clang [Clang] have attempted to rectify this, allowing better helpers to be written. Making our new langauge simple to parse would help to get a rich set of tools for it to fill any gaps, and a short compile time avoiding all the textual header includes is also very desirable.

Many languages have an underlying philosophy to guide the choices to be made. Here C++ has some good ones, e.g. trust the programmer; don't pay for what you don't use; support multiple programming paradigms. I'd also add some more: keep individual language features simple, but able to be combined with others; and avoid surprises.

So what did C++ get wrong? Well, for understandable historical reasons, values are mutable by default, similarly methods are non-const. We've learnt by experience and from functional languages that mutability is problematic – it can lead to programmes that are harder to reason about, and causes problems in multi-threaded code. So things should be immutable by default, and you have to ask for a modifiable value. This could even extend to free functions – if they promised to be 'const' by default, i.e. not modify global data or call non-const functions, then you don't have side effects, resulting in less for the programmer to worry about and the compiler has better optimisation opportunities.

So what did C++ get right? I'm a big fan of value-based programming, especially if the language gives you tools for making user defined types that look and feel 'natural' to use, using converting constructors, operator overloading, conversions, and overloaded functions. So these sorts of things should be in, and perhaps extended further. At the same time I'm very much *not* a fan of using built-in types for anything other than implementing user-types with more specific semantics. So in addition I'd like it to be very easy to define 'strong typedefs' – new types that act just like an existing one, but is a different type not an alias. e.g. instead of forename and surname both being strings, you can just declare

```
type Forename = String;
type Surname = String;
```

and you'd have two different types that can't be mixed up.

You may have noticed that this depends on a Strict type system, which is another good thing as it helps catch errors early. But many other languages are not strictly typed and have found the flexibility of dynamic type systems rather useful in certain areas. So I'd like this ability to be directly supported, but it should be optional to avoid unnecessary overhead – perhaps a keyword to tell the compiler to add the machinery for dynamic method dispatch and dynamically adding new methods. This last item is already hinting at the need for something along the lines of closures or lambdas, and this in turn hints at some sort of garbage collection, although again it should be optional on a per-type basis. Reflection can be very powerful too, but given some of the stories about its overhead in the program footprint this one definitely has to be optional!

Garbage Collection and C++ can't be mentioned without touching on what I think is C++'s great strength – deterministic lifetime and destructors. While GC is great for reclaiming abandoned memory, it fails badly at cleaning up resources that are more limited or have to be released in a timely manner – file handles, mutexes, sockets are just some examples. The fact GC doesn't deal with these very well can be seen by some of the solutions to these issues, such as the clunky Execute Around idioms, **`using`** and **`IDispose`**, or old fashioned **`close`**/**`teardown`** methods. But value-based objects on the stack and destructors shine for these situations, and it surprises me that such solutions are not used as much in other languages (even GC ones – why doesn't declaring an object on the stack automatically generate a **`new`** call and a **`dispose`** on block exit?)

C++ also has the very nice container/iterator/algorithm framework from the STL, and something similar would be excellent. This would require some sort of generics support, and I think C++'s template mechanism is an excellent start (and is better than the alternative interface based solutions), although improved concept and requirements checking would be needed. Not a trivial task! Strings are not so good though. A better solution would involved immutable strings, slices, mutable string builders, and ropes (bundles of strings that act like a single string – useful for concatenation without reallocation).

Parallelism is very important now and for the foreseeable future, so good support is vital. A good memory model is a vital bedrock upon which low level threading primitives can be based. But even these are too low level to work with directly, so some higher level parallel structures are needed, perhaps Erlang's message passing, coroutines, Actors or some such. This latter area is less obvious, so I suspect a library based solution may be preferable to a core language issue, although many other languages take a different approach.

Other ideas are from functional languages, such as list comprehensions or data pattern matching. Getting this to mesh with procedural style can be a challenge, but languages such as D show that it can be done. Adding map/reduce, and tuples with operations such as tie and zip can be very flexible too.

And finally what about errors? There are only a few choices, and none are totally appealing. I tend to favour exceptions but would like better support for writing exception safe code – deterministic destructors help here, but also **`nothrow`** qualifiers, and perhaps ones for Basic and Strong guarantees may help, or compiler to generate code to do the the right thing, e.g. generated swap functions.

These are just a few of my ideas off the top of my head, but is probably already too complex though. Which ones would you keep, and which have I missed?

References

[C++14] New draft at http://isocpp.org/files/papers/N3690.pdf

[Clang] http://clang.llvm.org/

[Curse] http://en.wikipedia.org/wiki/May_you_live_in_interesting_times

[Stroustrup09] 'No "Concepts" in C++0x', http://accu.org/index.php/journals/1576

[Sutton] http://concepts.axiomatics.org/~ans/

Auto – A Necessary Evil?

Superficially simple language features can be surprisingly complicated. Roger Orr explores a new one that is likely to be used widely.

To have a right to do a thing is not at all the same as to be right in doing it ~ G.K.Chesterton.

The keyword **`auto`** has a new use in C++11 – although the suggestion has been under discussion for a while, as we shall see. It was one of the early proposals for addition to what was then called C++0x and, since it was both useful and (relatively) non-controversial, some compilers added support for it well before the completion of C++11. This does have the advantage that it has had 'field testing' by a large number of programmers and so the form of the feature in the new International Standard seems to be pretty solid.

The keyword **`auto`** now lets you declare variables where the *compiler* provides the actual type and the programmer is either unwilling or unable to *name* the actual type. The keyword can also be used in function definitions to let you provide the return type *after* the rest of the function declaration, which is useful when the return type depends on the type of the arguments.

As with any new keyword there are questions about usage – at two levels. First of all, where and how are programmers *permitted* to use the new feature. Secondly, what guidance is there to sensible *adoption* of the new feature. I intend to start with by answering the first question and then subsequently focus on the second.

A bit of history

The word **`auto`** has been re-purposed in C++11 – it was inherited from C where it has been a keyword since the first days of *The C Programming Language* by Kernighan and Ritchie.

The old meaning of **`auto`** was defined as follows:

> Local objects explicitly declared **`auto`** or **`register`** or not explicitly declared **`static`** or **`extern`** have automatic storage duration. The storage for these objects lasts until the block in which they are created exits.

This meant that the keyword essentially added nothing over an implicit declaration:

```
{
  auto int i; // explicitly automatic
  int j;      // implicitly automatic
  // ...
} // end of life for both i and j
```

and so in practice **`auto`** was almost never used in production code.

When Bjarne Stroustrup started working on C++ his Cfront compiler originally allowed **`auto`** to be used for variable declarations in a very similar way to that now in C++11: "The auto feature has the distinction to be the earliest to be suggested and implemented: I had it working in my Cfront implementation in early 1984, but was forced to take it out because of C compatibility problems" [Stroustrup].

Many years later there was a discussion on the C++ committee email reflector about the difficulty of declaring variables resulting from complex template expressions. David Abrahams wrote (in ext-4278, 26 Oct 2001): "...the expression results in a very complicated nested template type which is difficult for a user to write down".

At the time the best suggestion was to write such variable declarations as something like:

```
typeof(<expression>) x = <expression>;
```

(**`typeof`** was an early name for what eventually became **`decltype`** in C++11).

This however meant that the (potentially rather complex) expression had to be written *twice*, for example in this simple case:

```
typeof(alpha*(u-v)*transpose(w))
   x = alpha*(u-v)*transpose(w);
```

which made the code harder to read – and was also a good source of bugs if and when the expression was changed.

He suggested this form of declaration could be replaced with something like:

```
template <class T> T x = <expression>;
```

The C++ template argument deduction rules could then come into play to work out the actual compile-time type of '**`x`**'.

In the subsequent discussion Andy Koenig wrote: "I would also like to see something like

```
auto x = <expression>;
```

I know we can't use **`auto`**, but you get the idea."

However, various people picked up on his, probably throwaway, suggestion and the idea gained momentum. Of course, a big concern was whether this change of use for the **`auto`** keyword would break a lot of code; the standards committee is understandably very reluctant to break existing valid code. A number of people spent some time searching internal company code bases they had access to and also using the now defunct Google Code Search. Daveed Vandevoorde reported that "Google Code Search finds less than 50 uses of **`auto`** in C++ code."

It turned out that most existing uses of **`auto`** were in test code (verifying that compilers, parsers or other tools handling C++ code correctly processed the keyword) and that a number of the remaining uses were in fact incorrect! The research gave the committee confidence that repurposing the keyword would not be a major problem. This confidence seems to have been well-founded.

The first formal paper for C++0x was N1478 (Apr 2003) [N1478]. The emphasis of this paper was in providing ways to make *generic* programming easier – the draft of this proposal (ext-5364) begins: "Proposal for "auto" and "typeof" to simplify the writing of templates".

The paper also proposed another new keyword, **`fun`**, which was used for declaring function return types. Over time this was replaced by an

Roger Orr has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

It isn’t the first time that a feature in C++ has had its use broadened well beyond the original expectations

overloaded use for `auto` (and jokes about how we lost the fun.) I do sometimes wonder whether `auto` is in danger of gaining multiple meanings in the same way that the keyword `static` has!

It is worth keeping this history in mind when looking at the use of `auto` as it might help distinguish the two main uses (one for variables and one for functions). It is also instructive to compare the original target design space – templates – with the range of uses finally allowed. It isn’t the first time that a feature in C++ has had its use broadened well beyond the original expectations.

So what did we end up with?

`auto` is repurposed and can be used in a variety of ways, such as:

- a placeholder for the type in a simple variable declaration:

```
auto x = 5; //'auto' here is equivalent to 'int'
```

- to declare a variable referring to a lambda:

```
auto lambda1 = [](int i){ return i * i; };
```

- in a `new` expression:

```
new auto(1.0); //'auto' equivalent to 'double'
```

- in function declarations (and definitions) allowing the return type to be specified at the end:

```
auto f()->int(*)[4];
```

- in function template declarations:

```
template <class T, class U>
auto add(T t, U u) -> decltype(t + u);
```

where this is considerably simpler than the equivalent *without* `auto`:

```
template <class T, class U>
decltype((*(T*)0) + (*(U*)0)) add(T t, U u);
```

In each case `auto` is a place holder for a *specific* compile time type – this type is ‘baked in’ by the compiler. This is worth highlighting, especially for those used to languages with dynamic types; there is no runtime overhead in using `auto`. Also note that the use of `auto` does not change the *meaning* of the code – it means exactly the same as the equivalent code with the deduced type written in full.

Once formally adopted into the working paper, `auto` became available for use in several compilers. Scott Meyer’s list [Meyers12a] of C++11 support shows `auto` was available in:

- gcc 4.4 (formal release Apr ’09)
- MSVC 10 (formal release Apr ’10)

and the examples given above all do compile successfully with both gcc and MSVC.

As the wording for `auto` was being polished for inclusion in C++11 (and as additional papers were written adding further new features to the language) there was a keen interest in avoiding any ‘special cases’ for `auto`. The committee followed the general principle of trying to make use of `auto` orthogonal to other choices: so for example `auto` for function return types is not restricted to function templates but can also be used for non-template functions.

Interactions with other items

r-value references

One of the new items added to C++11 was r-value references (designated with &&). As many of you will already be well aware this was principally added to support ‘move semantics’ which enables significant performance improvements when copying data out of temporary objects.

```
auto var1 = <expression>;
auto & var2 = <expression>;
auto && var3 = <expression>;
```

These are all valid (subject to constraints on the actual expression).

Note though the last in particular may not do *quite* what you expect … I will say more about this in the second article. (Scott Meyers covered this in his article on ‘Universal References in C++’ [Meyers12b].)

Lambda

The addition of lambda expressions to C++ was one of the motivating cases for `auto`. Passing a lambda to a function template works easily – for example:

```
template <typename T> void invoke(T t);
...
invoke([](int i){ return i; });
```

The call to `invoke` passes a (trivial in this example) lambda that takes an `int` and returns it. The compiler deals with instantiation of the correct template and so the programmer neither knows nor cares what the actual type of the lambda is.

But what if you want to hold the lambda in a variable?

```
<type> square = [](int i){ return i * i; };
int j = square(7);
```

The $64,000 question is: “What should replace `<type>` ?” The answer is `auto`.

NSDMI (non-static data member initialisers)

In C++11 values can be provided for non-static data members that will be used to provide the initial value (unless one is supplied in the initialisation list of the constructor). For example:

```
class x {
  int i = 128;
  double d = 2.71828;
};
```

Could you instead write:

```
class x {
  auto i = 128;
  auto d = 2.71828;
};
```

> **The basic principle behind auto is that the compiler knows the type ... but you either can’t describe it or don’t want to**

Short answer: no. This was rejected ... see ‘Where *can’t* you use it?’ below for a bit more detail about the reasons for this.

Range-based for

C++11 added syntactic sugar to support simple syntax for iteration over containers, for example:

```
for (std::string x : container) {
  // do something with 'x'
}
```

which is a simpler and safer way to write:

```
for (std::vector<std::string>::const_iterator it
    = container.begin(); it != container.end();
    ++it) {
   std::string x = *it;
   // do something with 'x'
}
```

The **auto** keyword is allowed in this context too, so you can write:

```
for (auto x : container) {
  ...
}
```

and the compiler will deduce the correct type for **x** to match the elements in the container.

The use of references and **const** allows more control over whether the loop variable is a value or a reference and whether or not it is constant:

```
for (auto & x : container) {
  x += ...
}
```

Or

```
for (auto const & x : container) {
  ...
}
```

(Note that in the first example the type of **x** is already a **const** reference if the container is **const**.)

Specification note

You may or may not care that range-based for is actually specified in terms of **auto** (see Listing 1).

The decltype keyword

The keyword **decltype** obtains the type of an expression. In C++03 there was no easy way to do this and various tricks were invented to provide various derived types – for example by using nested **typedef**s or associated traits classes. While **auto** allows you to declare a variable of the same type as an expression, **decltype** provides a more general technique. For example, declaring a variable without an initial value:

```
std::vector<int> vec;
decltype(vec.begin()) iter;
```

```
{
  auto && __range = range-init;
  for ( auto __begin = begin-expr,
             __end = end-expr;
        __begin != __end;
        ++__begin ) {
    for-range-declaration = *__begin;
    statement
  }
}
```

Listing 1

There are some subtle differences declaring a variable with **decltype** and with **auto**, which I will touch on later.

Where must you use it?

The basic principle behind **auto** is that the compiler knows the type … but you either can’t describe it or don’t want to. There is one primary use-case where you cannot name the type – with lambdas. Lambdas are most often used as arguments to other functions. However, if you want one as a local variable, the standard states (5.1.2p3) that the type of the lambda-expression “is a *unique, unnamed* nonunion class type – called the closure type” (my italics)

What this means is you the programmer **cannot** name the type (as the type is unnamed), nor can you even use **decltype** to declare a variable to hold the lamdba (as the type is unique so the type in the **decltype** *won’t* match the actual type of the expression).

Side note:

A small number of types in the standard are specified as *unspecified* so you cannot name them portably. **auto** gives you a way to create variables of those types; however this is almost never a genuine problem as the number of use cases when you genuinely need to do this is vanishingly small!

What is the actual type of a lambda variable?

Listing 2 is a simple example of a variable holding a lambda.

```
int main()
{
  auto sum = [] (int x, int y)
  { return x + y; };

  int i(1);
  int j(2);
  // ...
  std::cout << i << "+" << j << "="
    << sum(i, j) << std::endl;
}
```

Listing 2

In this contrived example the lambda is created and assigned to `sum` at the start of `main` and then invoked at the end of `main` in the output operation. But, if we are curious, we may be wondering what actually *is* the type of the variable holding the lambda.

We cannot *name* it in our code, but we are allowed to perform some other operations on the type.

We may for instance try to get some information by using `typeinfo`, for example with: t`ypeid(sum).name()`

The actual output is implementation specified, I obtain this with MSVC:

```
class <lambda_8f4bf0680d354484748e55d11883b00a>
```

and this with gcc:

```
Z4mainEUliiE_
```

(this name demangles to `main::{lambda(int, int)#1}`)

This gives some hint about possible implementation strategies in each case, but obviously code like this is of very limited practical utility.

An alternative solution

Very commonly of course we are not interested in the *precise* type of the variable but more in what we can *do* with it. We could then make use of the C++ `function` class to hold the variable:

```
std::function<int(int, int)> sum = [](int i,
                                      int j) ...
```

This technique employs *type erasure* behind the scenes – the actual lambda type is hidden inside the `std::function` object at the cost of a small runtime penalty. (`auto` avoids this penalty.)

This looks very similar to the following C# code:

```
Func<int, int, int> sum = (int x, int y) => {...}
```

Are lambdas the only place to use auto?

Declaring variables to hold lambda expressions is, I believe, the only time `auto` is mandatory in your code. However most people recommend you use `auto` in (at least some of) the cases where giving the name of type yourself is a valid option.

Herb Sutter, for example, wrote: "For example, virtually every five-line modern C++ code example will say "auto" somewhere." [Sutter]

As the quotation from G.K.Chesterton implies, being *allowed* to use `auto` does not mean this is always the right thing to do. I will look in the subsequent article about some of the forces involved in deciding when to use (and when not to use) the `auto` keyword.

Where can't you use it?

In C++11 you cannot use `auto`:

- As the type of lambda **arguments**:

  ```
  auto sum = [] (auto x, auto y)
  // not (currently) legal
    { /*...*/ }
  ```

  This however was voted into the next release – C++14 – at this April's WG21 meeting; and is already in some recent versions of gcc.

  What this generates is a lambda which can take different argument types – a sort of 'lambda template'. This has been named 'polymorphic lambda' and you may well have heard some of the discussion about this feature, which is one of the most common requests people make for extensions to lambda.

- To declare function return types without a trailing-return-type declaration

  ```
  auto func() { return 42; }
  // not (currently) legal
  ```

  This also was voted into C++14 – compilers will be able to deduce the return type of `func()` from the type of the returned expression (or expressions, if they are of equivalent type).

- To declare member data

  ```
  class X {
    auto field = 42; // error
    // ...
  };
  ```

  As mentioned earlier, this idea was floated during the discussions about `auto` for C++11, but there were concerns over whether this change might make the parsing of class definitions too complex and also over violations of the ODR (one definition rule) if the type of the initialisation expression was *different* in two different translation units.

  Discussion on supporting this one has resurfaced recently and it is possible there will be a proposal to add it to the language.

  I note that C#, where the `var` keyword has much the same purpose as `auto` for C++, also disallows fields being declared with `var`. Perhaps this common choice indicates some deeper problems with what at first sight seems to be a relatively straightforward extension.

- To declare function arguments

  ```
  void foo(auto i) { /*...*/ } // error
  ```

  The idea here is that this declares a function `foo` that behaves like a template and instantiates itself according to the type of argument provided – the code above would be effectively equivalent to:

  ```
  template <typename __T1>
  void foo(__T1 i) { /* ... */ }
  ```

  However, we *already* have function templates to do this job, and the use of explicitly named template arguments rather than `auto` allows you to express constraints between the arguments types more easily. However, there is some interest in supporting this syntax and so it may possibly be standardised at some time in the future, but is not currently in scope. It may be introduced as part of the 'concepts lite' development that is being formalised as a Technical Specification since this may provide the vocabulary to express constraints between the arguments.

Conclusion

C++11 contains a number of new features, some of which are somewhat complicated or obscure. The `auto` keyword though seems to be relatively safe and easy to use and allows complicated variable declarations to be greatly simplified. When used in conjunction with the range-based for loop the resultant code, to my mind at least, expresses intent much more clearly than the equivalent C++03 code and with very few downsides.

However, the use of `auto` is not always so cut and dried – and there are also some subtle interactions with `const` and r-value references. In the next article I will explore in more detail when you might wish to use `auto` and when you might prefer not to use it (and why). I will also cover some of the cases where `auto` produces different behaviour from what you might expect. ■

Acknowledgements

This article is based on the presentation with the same title at ACCU 2013.

Many thanks to Christof Meerwald, Irfan Butt, Sam Saariste and the *Overload* reviewers for their suggestions and corrections, which have helped to improve this article.

References

[Meyers12a] http://www.aristeia.com/C++11/C++11FeatureAvailability.htm

[Meyers12b] 'Universal References in C++', Scott Meyers (*Overload* 111)

[N1478] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1478.pdf

[Stroustrup] http://www.stroustrup.com/C++11FAQ.html#auto

[Sutter] http://herbsutter.com/elements-of-modern-c-style/

TCP/IP Explained. A Bit

Nowadays most programmers rely on network connectivity, often without really understanding the details. Sergey Ignatchenko compares and contrasts the two main protocols.

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

TCP/IP is widely used on one hand – most applications, apps and applets use some kind of connectivity these days – and yet paradoxically is not widely understood. While there is a strong temptation to just use TCP as a ‘magic box that works’, and most of the time it does indeed work as expected, there are still pitfalls in the cases when it doesn’t.

This article does not intend to discuss specific APIs in detail (those interested may refer to an appropriate book or reference; for example, for *nix APIs, [Stevens98/04] provides a great read and reference); rather, it attempts to describe some common issues with TCP/IP that might not be obvious from API references.

It should also be mentioned that this area is still evolving and there may be recent developments which are not reflected here; as usual, please take everything you read (including this article) with a pinch of salt.

TCP vs UDP

Everything which travels over the Internet is represented by an IP (Internet Protocol) packet (for the purposes of this article, there is no difference between IPv4 and IPv6 packets). As IP packets travel across the Internet any router on the way may drop them; recovery from such dropped packets must be handled by the client and server computers involved in sending and receiving the data.

Both TCP and UDP are protocols which are implemented on top of the IP packet mechanism, so technically TCP and UDP are in the same ‘Transport layer’ of the ‘Internet Protocol Suite’. However, when looking at UDP we find that it is a basic IP packet with only simple additional information (like UDP port), and without any built-in mechanism to detect dropped packets. This means that if you’re using UDP you’re on your own with regards to detecting dropped packets and recovering from them – this is exactly why UDP is often referred to as an ‘unreliable’ protocol. In practice, the use of UDP is usually limited to scenarios when the delay of data is more harmful than the partial loss of data; one specific example is VoIP/video delivery protocols such as Real-time Transport Protocol (RTP) (while RTP may work over TCP, in practice UDP is usually used).

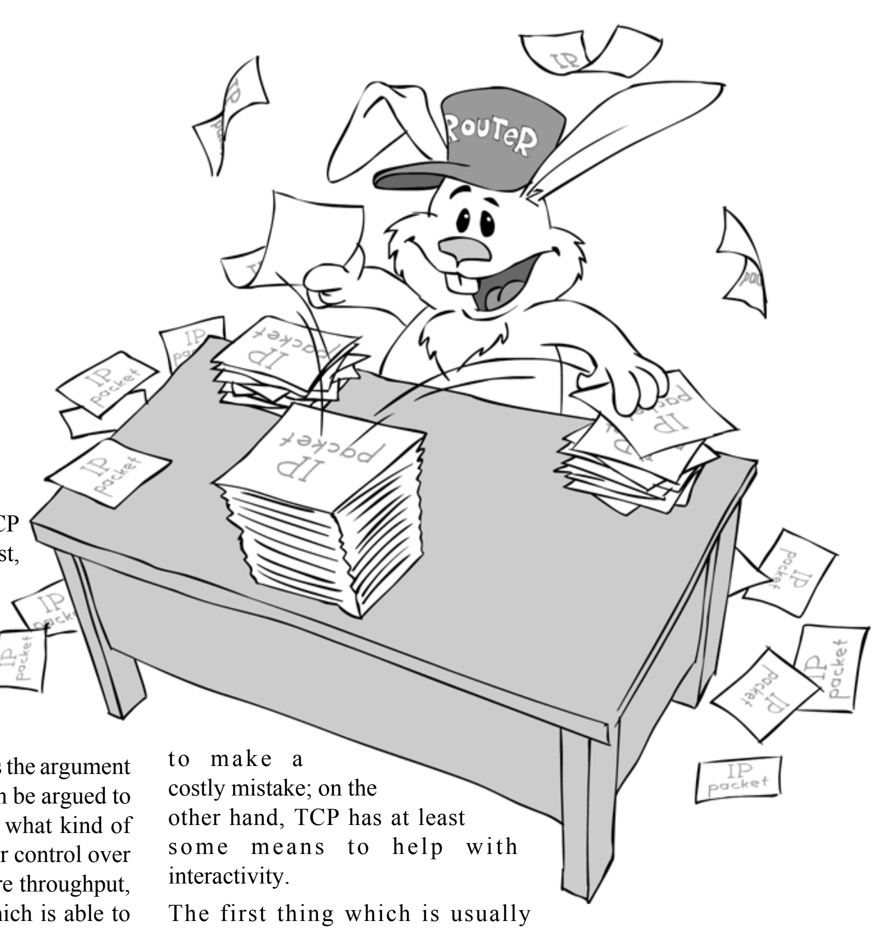
Comparing TCP and UDP

	TCP	UDP
Stands for...	Transmission Control Protocol	User Datagram Protocol
Is...	Stream-based	Message-based
Base for...	HTTP, HTTPS, FTP, SMTP, POP3, IMAP, Telnet, ...	DNS (most of), SNMP, TFTP, RTP (used for VoIP), ...
Ordering	Guarantees stream consistency	No guarantees on order of messages
Reliability	‘Reliable’ (there is an ACK packet; automatically handles retransmits) Reliability is limited by checksum being 16-bit	‘Unreliable’ (no ACK, no automated retransmits) If message is received, integrity is ensured by 16-bit checksum
Packet header size, (including typical IP header size)	40 bytes (60 bytes for IPv6)	28 bytes (48 bytes for IPv6)
Flow control	Present; if channel is busy, TCP slows down	Not present; if developer is not careful, can cause data loss due to sending rate being higher than the receiver processing rate
Congestion control	Present	Not present; if the developer is not careful, it may easily cause congestion
Is Internet-friendly	Usually yes	Depends on how it is used
Connection overhead	3 packets to establish connection, 4 packets to terminate gracefully	None
Delays	Potentially increased	Minimal

One feature which is present in UDP (but is not present in TCP) is multi-casting – when the same packet may be delivered to multiple locations, though these locations will still be identified by a single IP address. But while it is the case that one-to-many delivery looks interesting, a word of caution is necessary: the last time I checked, multi-cast wasn’t generally supported by Internet routers, and it didn’t look likely that this was going to change. This means that if you want to use multi-cast on an Intranet (with full control over routers and network administrators willing to help you, possibly including VPN-based network connections) it has a reasonably good chance of working, but if you need multi-cast over the public Internet you’re likely to be out of luck. If you’re desperate for multi-cast over the Internet by all means try it (things might have changed), but make sure that you’ve tested it in a real-world environment before committing to any large-scale development.

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

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



Unlike UDP, TCP is a ‘reliable’ protocol; this means that TCP detects IP packets that have been lost, re-transmits the request, and eventually gets the requested packet or the TCP connection becomes broken – and all of this happens almost invisibly to the developer. ‘Almost’ refers to the fact that nothing comes for free, and one needs to pay for the reliability with potentially increased delays which can be an observable effect at the application level.

One common misconception in ‘TCP vs UDP’ discussions is the argument that ‘UDP is faster’. This is not really a statement which can be argued to be right or wrong without further clarification – knowing what kind of ‘faster’ is needed. On the one hand UDP does provide better control over delays, but on the other hand, from the point of view of pure throughput, it is extremely difficult to build a UDP-based protocol which is able to compete with TCP over the Internet.

Overall, for applications which do not care about delays too much, TCP is usually a much better choice. However, there are still some caveats.

TCP caveat – reliability

While TCP is a ‘reliable’ protocol, it’s not absolute: as TCP checksums are only 16-bits long, if an IP packet is randomly corrupted on the way there is a 1 in 65536 (or ~0.0015%) chance that the checksum will be the same and the corruption will not be detected. In practice this has two implications. First one is: ‘never ever rely on the reliability of bare TCP transfers’; if one needs to transfer an important file it is necessary to do an extra check that the file has been transferred correctly (for example, by using SHA-1 or similar checksum on the whole file). While guarantees provided by SHA-1 are also not absolute, the probability of a corrupted file being undetected by SHA-1 is 1 in 2^{160}, which can be roughly translated as ‘not in your lifetime’ (even the long lifetime of an ithé[1] such as yourself). It should be noted that if SSL-over-TCP (or TLS-over-TCP) in which additional checks are used, the reliability of the transfer can usually be assumed. The second implication is that if, for example, one needs to transfer over a not-so-good link (and all links involving the ‘last mile’ to a home user should be deemed as potentially unreliable) a multi-gigabyte file with a SHA-1 checksum on the whole file (to guarantee integrity), it might be prudent to transfer the file in chunks with a checksum on each chunk; this way if TCP did allow a corrupted packet through, one will be able to re-transmit only the offending chunk instead of re-transmitting the whole multi-gigabyte file.

TCP caveat – interactivity

In general TCP has not been built for interactive communications, but mostly for long and steady file transfer; delays on the order of minutes have never been considered a problem for TCP. This means that a delay in the order of minutes is not a fault, it is a feature. The question is what to do when you need an interactive communication. While writing your own reliable protocol over UDP might sound like a good idea, it rarely is. On the one hand, any reliable protocol is highly complicated so it is very easy to make a costly mistake; on the other hand, TCP has at least some means to help with interactivity.

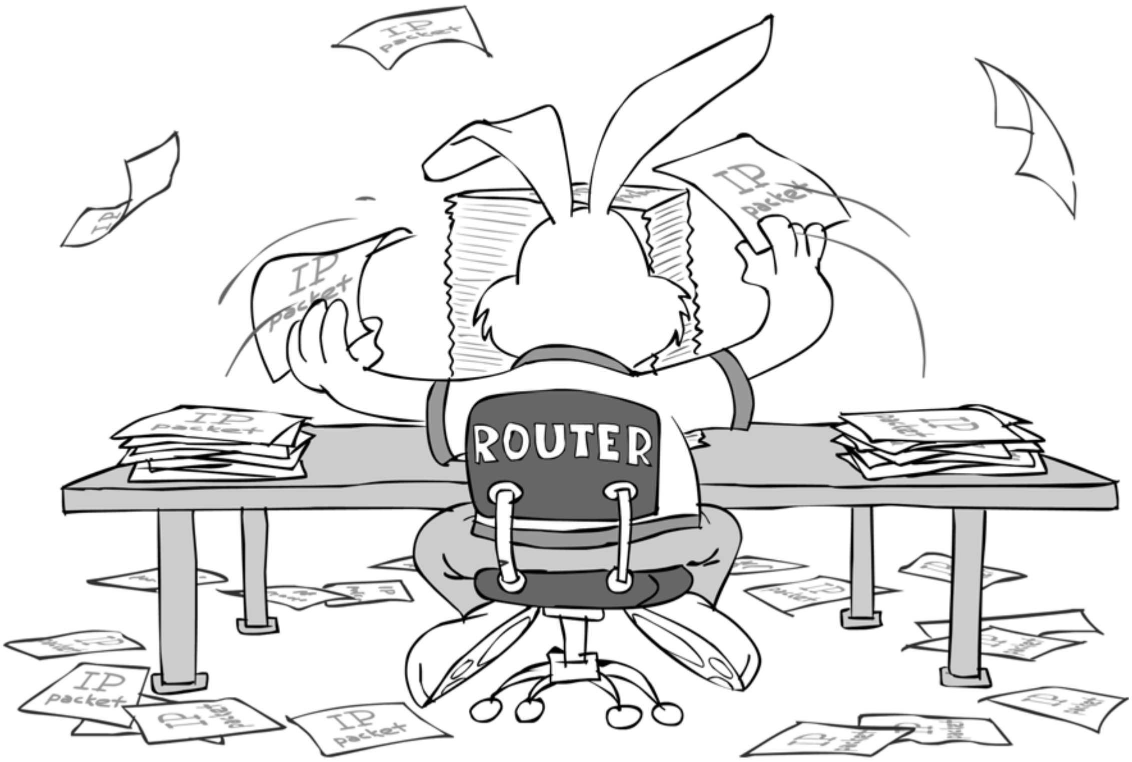
The first thing which is usually mentioned as a way to improve the interactivity of TCP connection is the `TCP_NODELAY` socket option. This might indeed help a bit, but one needs to keep in mind several issues:

- `TCP_NODELAY` behaviour varies significantly from one platform to another, so testing on all potential platforms is highly desirable; in particular, on some platforms it has been reportedly observed it affects the timing of re-transmissions in the case of dropped packets.
- It has been reported that it can affect the ‘PSH flag’, which might improve interactivity too, but the exact effects again need to be tested on all platforms.
- Usually `TCP_NODELAY` forces a packet to be sent immediately after `send()` is called. This means that if your code is written in a manner that calls `send()` for each single byte, then your code would work ok without `TCP_NODELAY` (as the TCP stack will wait before actually sending a packet, combining several `send()` calls together using Nagle’s algorithm), but with `TCP_NODELAY` enabled you’ll end up sending a 40-byte TCP+IP header for each call, leading to up to a 40x overhead! Ouch! On the other hand, if your code already combines all the available data before calling `send()` (which is often a good idea anyway), then `TCP_NODELAY` may indeed improve interactivity.

Hence if you don’t have problems with interactivity then don’t bother with `TCP_NODELAY`; it is a rather risky option which, unless carefully tested, may cause more problems than it solves.

Another thing which is not often mentioned but is at least as important for interactivity, is the handling of ‘hung’ TCP connections. Have you ever seen a web page which has stalled in the middle of being loaded, just to press ‘reload’ and voila – the page is there in no time? Chances are it was a ‘hung’ TCP connection. To make things worse, it might not be technically ‘hung’ from a TCP point of view (as mentioned above, TCP isn’t intended to care about a delay of a few minutes), but from the end-user’s point of view it certainly feels like it. For example, a compliant TCP stack is required to double the retry time each time, which means that if the first retry is 1 second (the default in the TCP standard), and then 7 subsequent retries fail (and if we have packet loss of a mere 0.01% then this will happen sooner or later given the number of packets in use

[1] ithé – n. man, human in general; from [Loganberry04]



nowadays), we're already in the 2-minute delay range; from TCP's point of view the connection is still alive and kicking, but for the end-user it is not so clear, and the user would probably just prefer it if the application detects the problem, cancels the old transfer and establishes a new connection to retrieve the data (which is a heresy from the network point of view, but forcing user to hit 'reload' to solve purely technical problem is an even worse heresy from the user interface point of view).

In addition, TCP as such does not really provide the means to detect connections which are *really* 'hung' even from the TCP point of view, e.g. when other side is not reachable at all; the socket was closed by the server but the RST response got lost on the way back; the server has been powercycled, etc. When I first saw the socket **`SO_KEEPALIVE`** option I thought 'hey, this is exactly what I need!'; however, my excitement soon faded when I realized that the default **`SO_KEEPALIVE`** timeout is 2 hours (!), and while on Windows it can be changed in the registry there is no way to change it programmatically. On Linux there are non-standard options such as **`TCP_KEEPIDLE`** and so on, but as many clients are on Windows it won't help us much.

All of the above may easily result in the need to design your own keep-alive subprotocol over a TCP connection, and doing it is quite an effort. Still, it is much less time-consuming and error-prone than writing your own reliable protocol over UDP (and if you don't need reliability – you may want to think about using UDP directly).

TCP caveat – single-channel throughput

While the original TCP (as specified in RFC 793) works over a transatlantic link with its signal delays (a round-trip time of ~100ms, although even worse are satellite links but these are rare in practice), there is a well-known problem that maximum throughput of a single TCP channel is limited; namely the Bandwidth-Delay Product [BW-D P] of TCP is limited to 64K,which with the RTT above corresponds to approx. 5MBit/s; it means that if TCP is used, over a single transatlantic connection it is not possible to obtain throughput over that even if all the paths between hosts are multi-gigabit. To deal with this, 'TCP window scaling' was introduced in RFC 1323 to increase this 64K limit. It does help, but there are still a few things to know: first, for TCP scaling to work both the client and server must support it; second, TCP window scaling is not enabled by default in pre-Vista Windows, so XP clients are usually still limited. Also, I know of people who were trying to establish a transfer in the gigabit/s range over a single transatlantic TCP link (they have had both servers close to the backbone, both servers had TCP window enabled, window scaling was used according to Wireshark, etc.); but they have found that a single TCP link is still limited to a speed in the order of a few hundred Mbit/s. They didn't manage to find out what was the underlying reason, but as a work around ended up using multiple connections which has solved the problem. My guess would be that at such speeds there was another bottleneck (perhaps the application wasn't able to write data with sufficient speed, and if encryption was used it would explain a lot). The lesson of this is that such bottlenecks are easy to run into, and if very high throughput is needed it must be carefully tested.

At one time so-called 'download accelerators' were quite popular; these were (and still are) quite efficient and often do improve download speeds in practice. Almost all of them simply establish multiple connections to the server, which apparently works well. The reason for the effectiveness of download accelerators has only a weak relation to the TCP window limit described above: while multiple connections from a client may indeed help to bypass the 5MBit/s limit, another issue is usually much more important: namely, if the server channel is limited, usually packets from all TCP connections are dropped and/or delayed in the same manner and therefore TCP connections are effectively throttled down proportionally. This means that during throttling a client having two TCP connections will get roughly twice as much data than a client having only one TCP connection, at the expense of the other clients.

The bottom line about throughput – in most cases, you can get away with a single TCP channel, but if getting the highest possible throughput is an issue you need to be ready to investigate problems, and in extreme cases may still need to use multiple TCP connections.

TCP caveat – packet loss resilience

One thing which should be noted about TCP is that, as a rule of thumb, it becomes virtually unusable when packet loss exceeds a certain percentage, in many cases within 5–10% range. Such a packet loss rate is usually considered abnormal (normal values even for the last mile should be within 0.01–0.1%), though I've personally experienced ISP support who told me "hey, 10% loss is ok, you still have 90% of the stuff going through, so we won't do anything about it". It is unlikely to become a problem in practice, and it is not clear if anything can be done about it, except for developing our own reliable protocol over UDP, which is unlikely to be worth it for all but very special applications.

TCP caveat – developers without a clue

One very common bug in TCP programs (probably the most common for beginners) is related to the incorrect use of streaming APIs. By its very nature TCP is a stream, so if on the sending side there is a single call to

send(), on the receiving side there is absolutely no guarantee that there will be exactly one successful call to **recv()**. In general the boundaries between **send()** calls are not seen on the server side at all, so for any number of **send()** calls there can be any number of **recv()** i.e. there is no 1–1 correspondence.

To make matters worse, when testing a program on the same computer or in a LAN the 1-to-1 relation between **send()** and **recv()** calls may happen to be observed, but when going into a WAN, things can start to fail from time to time. The only way to avoid it is to remember that TCP is always a stream, and if one needs boundaries between messages within this stream they must be introduced on top of TCP by the developer.

Another common problem with network programs (which applies both to TCP and UDP), is developers sending C/C++ structures over the network without marshaling. While this might work at first, in a project which aims to live for more than a few days, it is a time bomb. If sending/receiving C/C++ structures without marshaling, you do not really have a well-defined protocol. Instead, you're implicitly relying not only on the specific platform (because of little-endian/big-endian stuff), but also relying on the way a specific compiler applies alignment rules, and on stuff like **#pragma** pack in a specific place where the header which defines structure was included. If you don't use marshaling, and then, at any point down the road, you'll decide to go cross-platform or even to use different compiler for the same platform – the scale of the resulting problems due to the lack of marshaling might easily prevent you from doing it. Think more than twice before deciding not to marshal your data over the network.

TCP caveat – PMTUD

One of the very many features of TCP is 'Path MTU Discovery', or PMTUD in short. It is a nice feature which aims to detect the maximum packet size over the connection between client and server, and then to use this information to improve throughput. Unfortunately, one misconfigured router or firewall on the way may break it easily, leading to TCP connections which work normally when packets are small, and hanging forever when a large packet is seen. This was a big problem back 10 years ago, although is now less of an issue but it still happens from time to time. Usually it is considered a misconfiguration issue, but if it becomes a real problem (in other words too many customers are complaining), there is a chance to resolve it by using **TCP_NODELAY** and ensuring that all calls to **send()** are limited to at most 512 bytes in size (disclaimer: this is a guess from my side, and I've never tried such way of handling PMTUD myself; also note that strictly speaking, formally it is guaranteed to fix the issue only if the size is at most 28 bytes, but in practice 512 bytes should do nicely).

Troubleshooting, testing and Wireshark

If you have problems with a TCP connection, or if you're using any of the not-so-common TCP options (and this includes **TCP_NODELAY**), it is highly recommended to use a network analyzer to see what exactly is going on. If your application is used over the public Internet, it is highly recommended to test it with the server being as close to a real world one as possible and with all the likely clients (the behaviour of different TCP features may vary greatly from platform to platform). Testing over a link with a high delay is highly desirable even if the application is expected to be deployed over an Intranet. It should be noted that testing with high-delay links does not necessarily require special hardware or servers on the other side of the Atlantic. For example, for a low-traffic but highly-critical application, we've ended up purchasing a dial-up connection with the hope that if we can make it work reliably over that, it will work reliably under all realistic scenarios; it turned out that we've indeed chosen a very good way of testing.

When testing and analyzing TCP connectivity, a packet analyzer can be of great help. One I can recommend is Wireshark; it is free, and does its job wonderfully. One of the features I like the most, is the ability to analyze tcpdump logs. This means that if I have a Linux or BSD server and a real-life problem with one of the clients, I can, without installing Wireshark on the server, run a tcpdump on the server (it's usually part of a default installation), filtering by the client I'm interested in by IP using tcpdump's options, then download tcpdump's log to my desktop where Wireshark is installed, and then see what was going on using Wireshark's GUI. This allows us to use full-scale analysis for real-world problems. One thing to remember when using tcpdump for this purpose,is to use the **-s 65535** option, otherwise on some platforms tcpdump packets may be truncated, which might complicate analysis by Wireshark.

Firewall considerations

If one tries to build an application for the public Internet, a rabbit needs to think about the entire path all the way from the server to the client. This is likely to include firewalls at least for some of the clients. Usually, firewalls out there are statistically very friendly to TCP connections, and statistically a bit less friendly to UDP connections; in addition, UDP may cause issues when a client is behind certain types of NAT.

On the other hand, if a developer tries to use port 80 (the usual port for HTTP) for non-HTTP traffic in a naive attempt to bypass over-eager firewalls, there is another potential issue – many ISPs (especially in 3rd-world countries) use 'transparent caching proxies' on port 80, parsing requests in an attempt to save on network traffic; what happens with such proxies when a non-standard request comes in over port 80 is not defined (in practice the result may vary from forwarding the request 'as is' to hanging the whole proxy), so using port 80 for non-HTTP traffic cannot be regarded as safe.

On HTTP

One protocol implemented on top of TCP is HTTP. In general, HTTP is even more firewall-friendly than bare TCP, and if your conversation over TCP is limited to a request-response pattern, in some cases it might be worth to consider using HTTP instead (usually over port 80). In simple cases you may limit your HTTP to HTTP 1.0, which is trivial to implement at least on the client side. On the other hand, if you need multiple requests over the same TCP connection (in order to avoid penalties of re-establishing TCP, which might be quite large in case of multiple small requests), you might need to implement HTTP 1.1, which is doable but is a little bit more tricky. Alternatively, you may want to use HTTP APIs which are already available on many platforms. When using APIs, it is important to realize that as HTTP is implemented on top of TCP, so most of the TCP caveats also apply to HTTP connections.

Epilogue

With TCP and UDP being cornerstones of the Internet, lots of developers are bound to use them, either explicitly or implicitly. In many cases these protocols (especially TCP) do their job marvelously without the need for the developer to understand how they work. However, as there are only two of these protocols for all the myriad of usage scenarios on the Internet – sometimes they're used under conditions for which they were not designed; in such cases it may become necessary to understand how this low-level stuff works under the hood. I hope that this article is a good starting point. ■

References

[BW-D P] http://en.wikipedia.org/wiki/Bandwidth-delay_product

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

[Stevens98/04] *UNIX Network Programming: Networking APIs: Sockets and XTI*; Volume 1 W. Richard Stevens

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

Demons May Fly Out Of Your Nose

Language standards give guarantees about valid program behaviour. Olve Maudel discovers what happens if you break your end of the bargain.

C is one of the most widely used programming languages of all time. It is still the preferred language in many domains. With C you get direct access to the hardware and you only get exactly what you ask for. Often this is what you need. But, as we all know, with great power comes great responsibility. One of the responsibilities is to learn and understand the contract between yourself and the compiler. If you break the contract, then anything can, and will, happen.

```
#include <stdio.h>

int a() { printf("3"); return 3; }
int b() { printf("4"); return 4; }

int main(void)
{
  int c = a() + b();
  printf("%d\n", c);
}
```

Listing 1

Consider the C snippet in Listing 1. You might say that this program prints "347", but are you sure about that? Could the code print "437" instead? Do you understand C well enough to defend your answer? Most programmers assume that expressions are evaluated in a certain order, usually from left to right. This is a valid assumption for most modern programming languages, and if you rewrite the code above into, say Java, C#, Python or Ruby, then you are guaranteed to get "347". C is unlike modern programming languages in many ways. One of them being that in C the evaluation order of expressions is mostly unspecified. For the code snippet above, both "347" and "437" are valid answers.

Is this a bug in the language specification? No, it is a feature, an important feature actually. With loose evaluation, the compiler is able to create very efficient code on a wide range of hardware platforms. This is exactly one of the design goals for C – portability and speed.

Sequence points

Let’s take a look at another C snippet:

```
int  v[]  = {2,4,6,8,10,12};
int i = 1;
int n  = ++i  + v[++i];
// what is the value of n?
```

A similar code snippet in, say Java, is perfectly OK, well defined, and the value of `n` will be 2+`v`[3]=2+8=10. Most C compilers will happily create an executable from this code snippet, you will typically not get any warnings, and when you execute the code you might get 9, 10, 11, 12, 42 or whatever. This is a classic example of undefined behaviour in C. Not only can you end up with any random value, but when you have undefined behaviour anywhere in your code, the whole program is invalid – anything can happen! Really! One of the basic design principles in C is that the compiler is allowed to assume that you as a programmer only write correct code and therefore the compiler can, and will, make all kinds of ‘shortcuts’ when compiling your code. Or, in other words, it will certainly not waste any CPU-cycles to create a safety net around your code. If you break the contract, then the whole execution state of the program will be corrupt. In a one million line C program, a snippet like the one above invalidates the entire program. In theory, the program might end up formatting your hard drive or shut down the cooling system of the nuclear reactor, or, as they say on `comp.std.c` [JargonFile] – “When the compiler encounters [a given undefined construct] it is legal for it to make demons fly out of your nose”.

Olve Maudal works for Cisco Systems in Normay where he involved in developing videoconference and telepresence systems. Olve has been working a lot with C and C++ in the last 20 years: previous experience includes systems for mobile payments, bank transactions and seismic exploration. He can be contacted at oma@pvv.org

You may argue that you never write code like this (of course you don’t, I believe you). However, it is not enough to just say that you never write code like this, it is not enough to know that this is invalid code, you need to have an understanding of your programming language that is deep enough to also explain why this is invalid C code. To program correct C you need a deep understanding of the language [Maudal11].

If you reflect on the fact that the evaluation order of expressions in C is mostly unspecified, you might come up with a reasonable argument for why

```
int n = ++i + v[++i];
```

gives unpredictable result. As you cannot assume a certain evaluation order, then you do not know if the side effect of the first `++i` will happen before or after evaluating `v[++i]`. There is a contract between the programmer and the compiler that has now been broken. The contract, as defined in the C standard, says that if you update a variable twice between two sequence points, then you get undefined behaviour. Sequence points are like heart beats of a C program. There will be a sequence point at the end of a full expression, before a function is entered in a function call, and a few other places, but sequence points are surprisingly sparse in C. There is often a lot of code that needs to be evaluated between two sequence points. At a sequence point in the code, all side effects of previous evaluations will be completed, but between sequence points you are not allowed to make any assumptions about the state of involved variables. This also means that in C, unlike most other languages, the following expression leads to undefined behaviour

```
v[i] = i++;
```

because the assignment operator does not represent a sequence point in C.

Having few sequence points gives the compiler much freedom when optimizing the code. The compiler is allowed, and expected, to just evaluate the expression in a way that is optimal for the current hardware without even considering the possibility that a variable might be updated

```
load_a       $b                 ; load value of b into register A
compare_a    0                  ; compare register A to 0
jump_equal   label1             ; skip next statement if A == 0
call         print_b_is_true    ; print "b is true"
label1:
  load_a     $b                 ; load value of b into register A
  xor_a      1                  ; xor register A with 1
  compare_a  0                  ; compare register A to 0
  jump_equal label2             ; skip next statement if A == 0
  call       print_b_is_false   ; print "b is false"
label2:
```

Listing 2

twice between sequence points. A complex expression for example, can result in the compiler building up a long pipeline of powerful machine instructions to be churned through the CPU in the most efficient way happily ignoring all potential conflicting side effects.

Signed integer overflow

In C, a signed integer overflow also gives undefined behaviour. Hardware architectures handle integer overflow in different ways and C responds by saying that if that happens, then the compiler is not to blame. So the following function can give undefined behaviour and invalidate the whole program:

```
int the_answer(int seed)
{
  int answer = seed + 42;
  return answer - seed;
}
```

It is not the function itself that is the issue here, suppose this function is called with

```
the_answer(INT_MAX)
```

then the contract is broken and the compiler is not responsible for the consequences. Of course, in this particular case, the compiler will probably not be able to see what is going on. Benign code will be created and if you know the underlying hardware well enough, you might get a result you expected. The whole program might work just fine – until you try to compile it for another hardware platform, or you change the optimization level.

C has many similar cases of undefined behaviour. Is this a good thing as well? Is it a feature? Let's turn it around instead. Can you imagine how many extra machine code instructions would be needed to make edge cases like this portable across many hardware platforms? Perhaps you would need a virtual machine that the code could run on? Perhaps you would need fancy exception handling mechanisms that could flag run time errors like this? Well, there are languages that do these kinds of things, but C is not one of them. If you appreciate execution speed and direct access to native hardware, then declaring certain corner cases as undefined behaviour is often a good solution.

Surprising results

Believe it or not, there are examples of compilers that do pull pranks on you when encountering undefined behaviour [Wikipedia], but no reports so far confirms that a compiler actually try to make nasal demons fly out of your nose. The compilers usually try to be friendly and do the best they can. However, sometimes the compiler makes assumptions that can give very surprising results.

Here is a dramatic illustration of what might happen when breaking the rules of the language. As you know, in C all variables with non-static storage duration must be given an explicit value before they can be used, otherwise you break the contract and the compiler is no longer to blame for the consequences.

```
bool b;
if (b)
  printf("b is true\n");
if (!b)
  printf("b is false\n");
```

Suppose you found this somewhere in your code. You immediately see that this is undefined behaviour since `b` is not properly initialized before use, and from a theoretical point of view anything can happen now (for example, nasal demons). You might, however, from a practical point of view, expect that the code will always either print `b is true` or `b is false`, but even that is not guaranteed. A compiler might use a byte in memory to represent a bool, and at the same time assume that this byte in memory, since it is a bool, can only have the bit pattern 00000000 or 00000001. With that assumption the compiler might generate machine instructions similar to the pseudo-code in Listing 2, which assumes that `$b` is either `0` or `1`.

Try to follow the code: If the value of `b` is 1, then the code will print `"b is true"`, if `b` is 0 then it will print `"b is false"`, but if `b` is a random value, say 42, then this code snippet will print:

```
b is true
b is false
```

Indeed, what I just described, is exactly what happens with a very popular C compiler if the bit pattern of the byte used to represent bool `b` happens to become anything but 0 or 1. (see [Shroyer12] for more info)

This never happens in real code? Does it? Yes it happens! All the time! You typically observe the effects of undefined behaviour when you increase the optimization level, change the compiler or just update to a new version of the same compiler. If you have code that works just fine in one optimization level, but not when you change the optimization level – then you might want to start looking for undefined behaviour in your code. The compiler is not only allowed, but also expected, to generate code that is as efficient as legally possible. This means for example, that it will not, and should not, create any machine code instructions to check for or compensate for the possibility of invalid data.

Conclusion

To program correct C you need to have a deep understanding of the programming language. You need to understand the contract between the programmer and the compiler, because if you break the contract then the compiler can, and probably will, create code that give unexpected results. ■

References

[JargonFile] 'nasal demons', http://www.catb.org/jargon/html/N/nasal-demons.html

[Maudal11] 'Deep C (and C++)' http://www.slideshare.net/olvemaudal/deep-c

[Shroyer12] Mark Shroyer 'Both true and false: a Zen moment with C' http://http://markshroyer.com/2012/06/c-both-true-and-false/

[Wikipedia] 'Undefined behavior', http://en.wikipedia.org/wiki/Undefined_behavior

Wallpaper Rotation on Ubuntu using Ruby and Flickr

Repetitive tasks are ideal candidates for scripting. Filip van Laenen walks us through a simple example.

One of the benefits of being a programmer is that you can set up your computer to do things just the way you want, even if there's no program for it. I once wrote a podcatcher that downloads and manipulates podcasts so that I can listen to them in the right order. One reason for doing that was that I wanted to learn more about the technology involved, but I also felt that the podcatching programs that I had tried out didn't really do what I wanted them to do. May's issue of *C Vu* [vanLaenen13] contains another example: a script to back up my computer files just the way I want it. In this article, I'll explain a little program [WRUF] that I wrote to rotate the wallpaper on my Ubuntu laptop because I didn't feel other wallpaper rotation tools did what I wanted them to do.

The requirements

Let's start by sketching out the requirements for our little program. Basically, what I wanted was something that could change the wallpaper on my laptop once in a while. If possible, the wallpaper should be decorated with a small calendar, and some information about what the picture is about and where it comes from. An obvious choice to look for interesting pictures is Flickr [Flickr], a photo sharing website that I use to share my personal pictures with family and friends.

There are of course many other sources one could use to fetch interesting pictures from. Competitors of Flickr like Instagram and Picasa spring to mind, but also NASA's picture of the day, or press agencies like Reuters are good sources of pictures that could serve as wallpaper.

Flickr REST API

Flickr provides a REST API [FlickrAPI] through which you can search for pictures using keywords, one of the search modes being searching for the most 'interesting' pictures [Google]. You can use the same REST API to fetch pictures from groups, your private photostream, your favorites, or the photostream of friends and family. If you authenticate yourself, you can also use the same API to fetch private pictures, update information or upload pictures.

If you're interested in learning more about REST and how you can document a REST API, I think the Flickr REST API website is a great place to start. I think the format they use works really well, and when you browse through the documentation, you'll get a lot of inspiration about how resources and parameters should be named, default values, etc.

Accessing Flickr from Ruby

If you want to access Flickr from Ruby, there are basically two alternatives. One alternative is to use the REST API directly, but it's also possible to use one of the many specialized libraries (or gems in Ruby-speak). Both have advantages and drawbacks. If you only need to access a very limited set of services (resources) of the Flickr REST API, and you're not already used to using one of the Ruby Flickr libraries, accessing the Flickr REST API directly is not a bad choice. Just finding out which library is the right one for your project, e.g. based on activity in the project, the documentation and the API, may in itself take more time than implementing a simple REST call or two. On the other hand, if you're going to access many of the Flickr REST API resources, using one of the libraries may be a better idea.

```
FlickRestServicesUri =
    'http://api.flickr.com/services/rest/'
def do_rest_request(form_data)
  uri = URI.parse(FlickRestServicesUri)
  http = Net::HTTP.new(uri.host, uri.port)
  request = Net::HTTP::Get.new(uri.path)
  request.set_form_data(form_data)
  request = Net::HTTP::Get.new(uri.path + '?' +
     request.body)
  response = http.request(request)
  case response
  when Net::HTTPSuccess, Net::HTTPRedirection
    return REXML::Document.new(response.body)
  else
    raise "An error occured while trying to
access Flickr."
  end
end
```

Listing 1

In the case of WRUF, I chose to access the Flickr REST API directly in a class called **FlickrSearcher**. Listing 1 shows the class's method that deals with executing a basic **REST** call and returning its result. Accessing a Flickr REST API resource is then as simple as building up the correct form data, invoking the **do_rest_request** method, and filtering the data we're interested in from the result that's returned by the method.

Listing 2 shows how we build up the form data in order to search for an interesting picture. [Flickr2] The first parameter we have to set is the method name parameter, which is the search method in this case. Next, we add the API key for our application. Flickr uses this key to keep track of the applications that use its API (and probably also to blacklist you if you don't behave properly). Applying for a key doesn't take much time, and is free as long as you're not going to use it for commercial activities. [FlickrKey]

The search method doesn't require any authentication, so we're not adding our **user_id** or any other authentication information. The rest of the form data then controls how the search is performed. First, the extras parameter lists the additional information we'd like to see included in the search result. We need the original dimensions of the picture in order to filter the ones that are too small, and in addition we need to know the original format (typically JPEG) in order to build up the picture URL correctly. We specify

Filip van Laenen is a chief technologist at the Norwegian software company Computas. He has a special interest in software engineering, security, Java and Ruby, and likes to do some hacking on his Ubuntu laptop in his spare time. He can be contacted at f.a.vanlaenen@ieee.org

it's sufficient to **store the URLs** of all the photos we've used as wallpaper so far, and **match any potential wallpaper candidates** against the list

```
PhotosSearchMethod = 'flickr.photos.search'
ApiKey = <Your Application's API Key>

def create_form_data_to_search_photos(tags, i)
  form_data = {'method' => PhotosSearchMethod,
           'api_key' => ApiKey,
           'extras' => 'o_dims,original_format',
           'format' => 'rest',
           'media' => 'photos',
           'page' => i.to_s,
           'safe_search' => '1',
           'sort' => 'interestingness-desc',
           'tag_mode' => 'any'}
  if (tags != nil)
    form_data['tags'] = tags.join(',')
  end
  return form_data
end
```

Listing 2

REST as the format for the response in the format parameter, so that we can use XPath to extract information from the search result. Media is set to photos, so we don't get any videos in our search result. The page parameter is used to specify the search result page we want to return. The parameter **`safe_search`** is set to **`1`**, in order to filter out pictures that would be 'too interesting'. Notice that since we call the search method unauthenticated, search results will already be filtered to be safe, so this is just a precaution. The **`sort`** parameter is set to descending by interestingness, so we get the most interesting pictures first. Finally, if the method is called with a set of tags, we add them as a comma-separated list. Notice that tags that are prefixed with a minus sign (**`-`**) will be used to exclude matches from the result. We also set the tag mode to **`any`**, which results in an OR combination of the tags (the default). If you want an AND combination, you have to set this parameter to **`all`**.

```
def get_photo_info(info_set, history)
  return info_set.get_elements ('rsp/photos/photo') \
   .select{|e| e.attributes['o_width'].to_i >= @width} \
   .select{|e| e.attributes['o_height'].to_i >= @height} \
   .select{|e| (e.attributes['o_height'].to_f / \
               e.attributes['o_width'].to_f) / \
               (@height.to_f / @width.to_f) < \
               1.to_f + @tolerance} \
   .select{|e| (@height.to_f / @width.to_f) / \
               (e.attributes['o_height'].to_f / \
               e.attributes['o_width'].to_f) < \
               1.to_f + @tolerance} \
   .reject{|e| history.include?(get_photo_url(e))} \
   .first
end
```

Listing 3

Framing a picture

Unless we've chosen some very particular keywords, a search on Flickr will return a vast number of photos. Of course, these photos will have a wide range of dimensions, and not all of them will fit the desired desktop size. How do we select the photos that do fit?

In plain words, the rule to select photos is not so difficult. First of all, the photo should be larger than the desktop size in absolute terms. Furthermore, the ratio between height and width for the photo and the desktop should be equal within a given margin of tolerance. Finally, the photo should be one that we haven't used before.

Listing 3 shows how this selection process is implemented in WRUF. One of the great things about Ruby is that it has blocks (and lambda expressions). These blocks can be used as parameters, and probably the most common way to use them is in the API for collections. In this listing, blocks are used as a parameter for the methods select and reject. They are used by these methods to filter the initial collection of search results down to a collection of photos that can be used as wallpapers. Have a look at the first call, which says that only those elements (**`e`**) should be retained (**`select`**ed) which have an attribute called **`o_width`** that when converted to an integer is larger or equal to our desired field width. Notice also that the select method returns the resulting collection, so that we can chain all calls together without having to assign and reassign to a local variable.

In order to get the initial collection, we use an XPath expression on the XML object that was returned by the Flickr search method call. And once we've narrowed the collection of photos down to the ones that can be used as a wallpaper, we simply call **`first`** to return the first element. If the resulting collection turns out to be empty, the method will return **`null`**, and a new call to the Flickr search method should be issued in order to get the next search page.

I'm sure there are more efficient ways to filter out the photos with the right ratio than the two-pass filtering I use. However, performance hasn't been an issue yet, so I haven't cared to look into it more deeply. Considering that computers are terribly fast at the simple arithmetic involved in the calculations in Listing 3, that the program will run in the background anyway, and that it won't do the calculations for more than a couple of hundreds, or at worst a couple of thousands photos once a day or so, just writing the lines in this paragraph probably cost me more time than I'll ever be able to save.

Keeping track of history

Talking about time, our little program also keeps track of history. In our case, it's sufficient to store the URLs of all the photos we've used as wallpaper so far, and match any potential wallpaper candidates against the list. If we're going to switch wallpaper only once a day, the list won't be longer than a couple of hundred URLs in the course of a year. We can therefore store the URLs in a simple flat file, one URL on every line. Adding a URL to the history file is then as simple as appending it to the end. Reading the history file is simple too: just create an empty array, and add every line as a new element to it.

our particular requirement for handling history –never ever reuse a photo as a wallpaper– simplified matters substantially

There are of course alternatives to storing the URLs in a flat file. One option would have been to use an XML file, but that would only have made sense if the data structure would have been more complicated. The same is true for storing the URLs in a database, but in addition to that, using a database would have added a dependency to the system, and complicated matters substantially. Not using a database at all is a big feature, especially at installation time.

At the same time, it should be noted that our particular requirement for handling history –never ever reuse a photo as a wallpaper– simplified matters substantially. Other users may prefer to be able to reuse photos as wallpaper after a certain number of days. I've found out that as long as your keywords are 'normal', Flickr has such a vast amount of interesting photos that there's really no need to reuse any of them. You could probably change wallpaper every hour or every minute, and there would still be enough photos to chose from.

```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/
Graphics/SVG/1.1/DTD/svg11.dtd">
<svg height='768' version='1.1' width='1366' xmlns:xlink='http://
www.w3.org/1999/xlink' xmlns='http://www.w3.org/2000/svg'>
  <image height='911'
xlink:href='0ac9b2fc1f8ab4b73b26608259e11683a03f90dd.jpg'
width='1366.0' x='-0.0' y='-71'/>
  <g id='photo_info'>
    <text fill='#FFCC11' font-family='Ubuntu' font-size='16' font-
weight='bold' x='68' y='649'>Man Made Cascade, Virginia Water</text>
    <text fill='#FFCC11' font-family='Ubuntu' font-size='12' x='68'
y='673'>flatworldsedge @ Flickr</text>
    <text fill='#FFCC11' font-family='Ubuntu' font-size='12' x='68'
y='691'>http://www.flickr.com/photos/flatworldsedge/5252778530/</
text>
  </g>
  <g font-family='Ubuntu' font-size='32' font-weight='bold'
id='calendar' text-anchor='middle' transform='translate(1297,76)'>
    <g id='last_week' opacity='0.2'>
      <text fill='#FFCC11' x='-268' y='0'>29</text>
      <text fill='#FFCC11' x='-224' y='0'>30</text>
      <text fill='#FFCC11' x='-179' y='0'>1</text>
      <text fill='#FFCC11' x='-134' y='0'>2</text>
      <text fill='#FFCC11' x='-89' y='0'>3</text>
      <text fill='#FFCC11' x='-44' y='0'>4</text>
      <text fill='#FF0000' x='0' y='0'>5</text>
    </g>
    <g id='this_week'>
      <text fill='#FFCC11' opacity='0.2' x='-268' y='44'>6</text>
      <text fill='#FFCC11' opacity='0.2' x='-224' y='44'>7</text>
      <text fill='#FFCC11' opacity='0.2' x='-179' y='44'>8</text>
      <text fill='#FFCC11' opacity='1' x='-134' y='44'>9</text>
      <text fill='#FFCC11' opacity='0.5' x='-89' y='44'>10</text>
      <text fill='#FFCC11' opacity='0.5' x='-44' y='44'>11</text>
      <text fill='#FF0000' opacity='0.5' x='0' y='44'>12</text>
    </g>
    <g id='next_two_weeks' opacity='0.5'>
      <text fill='#FFCC11' x='-268' y='89'>13</text>
      …
      <text fill='#FFCC11' x='-44' y='134'>25</text>
      <text fill='#FF0000' x='0' y='134'>26</text>
    </g>
  </g>
</svg>
```

Listing 4

Decorating a picture through SVG

Before I set a photo as a wallpaper, I would like to decorate it with its title, its author, its URL, and a little calendar. Title, author and URL are useful in case I want to look up the photo on the internet (e.g. because I like it and would like to favorite it in Flickr). Of maybe I just want to see what the picture is about and who created it. The calendar is more of a gimmick, but I like to have it on my wallpaper for quick reference.

The strategy I chose to decorate the photo is to include it in an SVG image, and add the texts on top of it. SVG [SVG] stands for Scalable Vector Graphics, an XML format to define, well, vector graphics. Listing 4 shows how an SVG file to decorate a wallpaper photo typically looks like. Let's walk through it.

The file starts with an XML header, defining it as an SVG 1.1 document. The **svg** element is the root element, and it also sets the dimensions of the image as 768 × 1366. In SVG, elements are drawn on top of each other in the same order as they appear in the XML document, so the first thing we want to draw is the photo. The image element does just that, referring to the file name where the photo can be found. The photo is scaled proportionally so that either the height or the width match the size of the screen, with the non-matching dimension being slightly larger. Using the x and y attributes, the photo is also the positioned such that its middle will be in the middle of the SVG image, and therefore also of the screen.

Notice that the name of the photo file isn't the real name of the photo as it can be found on Flickr, but a SHA-1 digest of its URL. There are two reasons for doing so. First of all, since the URLs will be

I don't know in advance what will be **the dominant colour of the photo**, and I don't know of a **method to inspect the photo** in that sense either

unique, this will result in unique file names. In addition to that, there won't be any issues with problematic characters, since the file name will consist of hexadecimal characters only.

Groups of elements can be defined using the **`g`** element. This SVG file has five of them, and as Listing 4 illustrates, groups can be nested. Besides having an id to identify them, groups can be used to set common attributes to all its elements in a DRY-fashion. Of course, these attributes will only apply to those elements in the group for which they are relevant.

The first group prints some information about the photo on the wallpaper image, using text elements. Notice that the **`fill`** and **`font-family`** attributes could have been defined on the **`g`** element instead of on each text element. Consider it a bit of technical debt, a consequence of my experimenting with different colors and fonts when I was writing the program. The attributes on the text elements are pretty self-explanatory: **`font-family`**, **`font-size`** and **`font-weight`** define the family, the size and the weight of the font to be used, and **`x`** and **`y`** where the text should be positioned. The **`fill`** attribute sets the color of text. Finding the right colour turned out to be a bit of a challenge though.

I don't know in advance what will be the dominant colour of the photo, and I don't know of a method to inspect the photo in that sense either. (I'm sure there exist tools for that, and using one of them would be an obvious nice feature to add when I have more time.) I therefore had to pick a colour that would work well in most occasions, and found out that FFCC11, a colour close to gold (FFD700), was a good choice. An initial thought was to use a contrasting stroke too (e.g. black stroke with yellow fill), but that turned out not to work. The text is too small in order for the stroke to have a good effect. A better alternative would probably have been to put a semi-transparent rectangle behind the text, but since that would have hidden part of the photo too, I didn't want to do that. Besides, using a semi-transparent rectangle wouldn't have worked for the calendar anyway. The calendar is too big, so it would have hidden a rather large part of the photo.

My calendar consists of the current week, last week, and the next two weeks – four weeks in total. Days in the past are made almost completely transparent (opacity 20%), and days in the future half-transparent (opacity 50%). Weekdays, including Saturdays, have the same colour as the other texts, but Sundays are marked in red (FF0000). An obvious improvement would be to mark bank holidays in red too. Notice that for the calendar, I did use group attributes to set common attributes across all elements. In addition to that, I used the transform attribute to translate the calendar to the right place in the image. Alternatively I could have added the two numbers to the x and y coordinated of every text element, but I think my solution makes it more clear where which element of the calendar goes.

It should be noted that I chose to generate the content of the SVG file as an XML document using **`REXML::Document`**. Just as there exist specialized libraries to access Flickr, there exist specialized libraries to create SVG files. My feeling is that it's easier to create SVG files through XML, as it gives you full flexibility and you need to know SVG anyway to use the SVG libraries. Your mileage may of course vary…

Converting SVG to PNG

One drawback of using SVG to decorate the wallpaper is that it has to be converted back to JPG or PNG. Decorating the JPG photo directly would probably have been the most elegant solution. An alternative approach would have been to convert the photo to PNG, and do the decorating in PNG. But that would have involved a conversion too, so it probably wouldn't have saved us much compared to using SVG. In either case, I didn't find a Ruby library to draw text directly on JPG or PNG images, so that's also a reason why I used SVG.

The conversion from SVG to PNG is the part that I'm the least satisfied with in WRUF. I never managed to find a good Ruby library that could do the job, so I had no choice but to make a system call to rsvg-convert. Rsvg-convert is one of the tools provided by the librsvg2-bin package, and it can convert SVG images into PNG raster images.

The conversion tool works fine, and the system call to rsvg-convert in itself isn't a problem either. But if I would like to port the program to another operating system in the future, this will be one of the issues. Preferably I should migrate to a Ruby library that can convert SVG images into PNG, the alternative being to find similar tool in the target operating system and make a system call to that.

Setting the wallpaper

Setting the resulting image as the current wallpaper is done through a system call too, and will therefore have to be adjusted to specific operating systems too. But contrary to the conversion of SVG images into PNG images, this is something that can be expected. Even within the same operating system there may be differences from one version to another, as I discovered myself. Ubuntu 11.10 and newer versions use e.g. **`gsettings set org.gnome.desktop.background`** to set the wallpaper, whereas earlier versions used gconftool-2.

Command-line user interface

So far we've described how the main program works, but now we still have to get it started. This is done from a Shell script that calls the main Ruby program. But the Shell script can do more than just calling the main program: it can also start the initialization, or print some help text, version, copyright and warranty information. But let's start with just running the program.

There are basically two ways I want to start the program from: manually from the command-line, or automatically from Cron (e.g. every hour). I have therefore linked `/usr/bin/wruf` to wherever the main Shell script resides, so that I don't need to remember where I've put it. Running the program is therefore as simple as typing **`wruf run`** on the command-line, or adding a line with a call to **`/usr/bin/wruf run`** to crontab. It is then the task of the Shell script to find out what all the local directories are, and to call the Ruby program with the correct parameters. Listing 5 shows most of the main Shell script.

First, the script saves the first parameter as **`ACTION`**. Then it stores where it expects the local directories to be in some local variables, and how to

I was a bit surprised that I needed **both a warranty and a license text**—I thought a license text was all I needed

call Ruby. After that, it does some magic to make WRUF run from Cron. As it turns out, setting the wallpaper doesn't work just like that if you run the program in the background. Finally, it sets the version number and the copyright year to be used in the various messages further down.

The rest of the Shell script is a case statement on the **`ACTION`** variable. For every action, it either calls another Shell script to perform a specific task, or prints out some text. I suppose the help, version, copyright and warranty information could have been printed out by the main Ruby program too, but it seems like overkill to start a Ruby program just to print out some text. A big argument against putting this in the Shell script is that it makes the whole program more dependent on Shell scripting. This may again make it harder to port WRUF to a different operating system. A big argument in favour of it is that this way, printing out the help message doesn't depend on having the local directories initialized correctly, or even having Ruby installed.

When I wrote the program, I was a bit surprised that I needed both a warranty and a license text—I thought a license text was all I needed. But when you think of it, the purpose of the license text is to handle how the program can be used and reused by others. The warranty, however, makes sure that nobody can come after me if my program decides to delete a user's disk. If you want to open source your code, you should probably have both.

```
#!/bin/sh
# (Header with copyright information omitted.)

ACTION="$1"

export WRUFDIR="/opt/wruf"
export LOCALWRUFDIR="${HOME}/.wruf"
export RUBY="ruby"

# (Some magic to make WRUF run from Cron
# omitted.)
VERSION="1.1a1"
COPYRIGHTYEAR="2011"

case "$ACTION" in
  init)
    ${WRUFDIR}/wruf_init.sh
    ;;
  run)
    ${WRUFDIR}/wruf_run.sh
    ;;
  tags)
    ${WRUFDIR}/wruf_tags.sh
    ;;
  current)
    ${WRUFDIR}/wruf_current.sh $2
    ;;
```

Listing 5

YAML ain't markup language

Now that we have a running program, we still need to initialize it. In order to run the program properly we need to know the dimensions of the screen we're going to produce wallpaper for, the tolerance for how much the dimensions of a picture can deviate, the minimum number of hours between the rotation of the wallpaper, and the tags WRUF should use when searching on Flickr.

Notice that WRUF keeps control over when it's time to rotate the wallpaper. Since I don't keep my laptop running the whole day, I can't set up a Cron job at a specific hour to rotate the wallpaper once a day. Instead, I have a Cron job that runs WRUF once every hour, so that it rotates the wallpaper whenever time's up. This also means that I can run WRUF at start-up, without it causing the wallpaper to be rotated a second or even a third time during the same day just because I had to reboot. On the other hand, sometimes you'll want to change the wallpaper immediately, e.g. because WRUF happened to pick a photo you don't like. This is why I needed to implement the **`current dislike`** function too, as mentioned in the help text in Listing 5, in addition to **`run`**.

In order to keep things simple, I use YAML [YAML] to store the settings in a settings file. YAML is "a human friendly data serialization standard for all programming languages", as the official YAML Web Site defines it. Human friendliness is not a big issue for using YAML in this case, even though it's always nice to be able to inspect what's stored in the settings file during development or debugging. The biggest reason for using it in WRUF is that it has been included in the standard library for Ruby since version 1.8, and the API for using it is very compact and easy to understand. Listing 6 shows how the settings file is stored and read, together with an example of how such a settings file looks.

The method **`to_yaml`** converts a Ruby object into a YAML string, which can then be stored directly in a file. The **`load`** method from the YAML module does the reverse: it creates a Ruby object from a YAML string or an IO stream. The sample settings file shows how primitive attributes (the hours, an integer, and the tolerance, a float) are stored, but also arrays (the dimensions and the tags).

Installation

Finally some words on installing and creating an installation script for WRUF. Since this program is relatively simple, I simply pack everything that's needed to run WRUF together in a tar-file. Right now, this includes the Ruby files, the Shell scripts, and the license text. Installation is then as simple as unpacking the tar-file, and then running the installation script.

The installation script first deletes the WRUF installation directory if it already exists, and then makes a clean copy of all Ruby files and Shell scripts. Then it makes the Shell scripts executable, and links `/usr/bin/wruf` to the main Shell script. At the end, it also installs the Log4r gem if it's not already installed.

I chose to use `/opt/wruf` as the WRUF installation directory. According to the Linux Filesystem Hierarchy [Linux], "[t]his directory is reserved for all the software and add-on packages that are not part of the default

I can run WRUF at start-up, without it causing the wallpaper to be rotated a second or even a third time during the same day just because I had to reboot

```
  help)
    echo "Wallpaper Rotator Using Flickr (WRUF) v${VERSION}"
    echo "Copyright © ${COPYRIGHTYEAR} Filip van Laenen <f.a.vanlaenen@ieee.org>"
    echo
    echo "Usage:"
    echo "  wruf action [parameters]"
    echo
    echo "where actions and parameters include:"
    echo "  init              initialize WRUF"
    echo "  run               run WRUF"
    echo "  tags              manage the tags used by WRUF in an interactive dialogue"
    echo "  current dislike   rotate the wallpaper regardless of when it was rotated last"
    echo "  help              show this message"
    echo "  version           show the version information"
    echo "  copyright         show the copyright information"
    echo "  warranty          show the warranty information"
    ;;
  version)
    echo "Wallpaper Rotator Using Flickr (WRUF) v${VERSION}"
    echo "Copyright © ${COPYRIGHTYEAR} Filip van Laenen <f.a.vanlaenen@ieee.org>"
    echo "This program comes with ABSOLUTELY NO WARRANTY; for details run 'wruf warranty'."
    echo "This is free software, and you are welcome to redistribute it"
    echo "under certain conditions; run 'wruf copyright' for details."
    ;;
  copyright)
    echo "Wallpaper Rotator Using Flickr (WRUF) v${VERSION}"
    echo "Copyright © ${COPYRIGHTYEAR} Filip van Laenen <f.a.vanlaenen@ieee.org>"
    echo
    echo "This program is free software: you can redistribute it and/or modify"
    echo "it under the terms of the GNU General Public License as published by"
    echo "the Free Software Foundation, either version 3 of the License, or"
    echo "(at your option) any later version."
    echo
    echo "This program is distributed in the hope that it will be useful,"
    echo "but WITHOUT ANY WARRANTY; without even the implied warranty of"
    echo "MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the"
    echo "GNU General Public License for more details."
    echo
    echo "You should have received a copy of the GNU General Public License"
    echo "along with this program.  If not, see <http://www.gnu.org/licenses/>."
    ;;
  warranty)
    echo "Wallpaper Rotator Using Flickr (WRUF) v${VERSION}"
    echo "Copyright © ${COPYRIGHTYEAR} Filip van Laenen <f.a.vanlaenen@ieee.org>"
    echo
    echo "There is no warranty for the program, to the extent permitted by applicable law."
# (The rest of the warranty text omitted.)
    ;;
  *)
    echo "Wallpaper Rotator Using Flickr (WRUF) v${VERSION}"
    echo "Copyright © ${COPYRIGHTYEAR} Filip van Laenen <f.a.vanlaenen@ieee.org>"
```

Listing 5 (cont'd)

> **Just like most hobby projects, WRUF is not complete, and it will probably never be**

```
    echo
    echo "Usage: wruf {init|run|tags|current|help|version|warranty|copyright}" >&2
    echo "Type 'wruf help' to get more information."
    exit 1
    ;;
esac
```

Listing 5 (cont'd)

installation." This makes WRUF system-wide available, but it also requires that the person installing WRUF has administrator access rights (and uses e.g. sudo to install the program). User settings, history and cached wallpaper photos and files are then stored in `${HOME}/.wruf`, so that each user can have his own set of tags and other settings.

Feature backlog

Just like most hobby projects, WRUF is not complete, and it will probably never be. I already mentioned that there are lots of other sources of good photos that could be used as wallpaper, but for now, WRUF only searches through Flickr. But there are other alternatives too, like using local photos, or even creating random drawings in SVG. Other features that could be added include 'liking' the current wallpaper, e.g. by adding the photo to the user's favorites in Flickr, putting other information on the wallpaper, like geo-information, updating the calendar regardless of whether the source photo should be rotated or not, or using dynamic tags like the current month or season. Auto-detection of the current desktop size would also be nice.

```
# Storing the settings file using a
# settings object:
open(file_name, "w") { |file|
  file.write(settings.to_yaml)
}
# Reading the settings file into a settings object:
settings = YAML::load(read_file(file_name))
# Sample settings file:
--- !ruby/object:WrufSettings
dimensions:
- 1366
- 768
hours: 18
tags:
- landscape
- forest
- sea
- mountain
- mountains
- river
- clouds
tolerance: 0.25
```

Listing 6

Creating a wallpaper rotator for Ubuntu wasn't a difficult task, but it involved many different technologies. First of all, we needed REST to access Flickr and find a good background photo for the wallpaper. Then we used SVG to decorate the photo, and converted it to PNG. In order to do the conversion, we had to make a system call from Ruby, just like for setting the resulting image as the new wallpaper. We kept track of history through a simple text file, and stored the settings using YAML. Finally, the core program was written in Ruby, with some Shell scripts on top of it to get it running and installed.

The program has been running on my laptop for more than a year now, and it has been working fine for me. Often it fetches great photos from Flickr, taken at amazing places like the Denali National Park & Reserve in Alaska. WRUF wasn't only a good way to explore some interesting technologies, but also to see some of the most interesting photos on Flickr, and to learn about the extraordinary places where they've been taken. ■

References

[Flickr] See http://www.flickr.com

[Flickr2] For a detailed overview of the parameters that can be set, see http://www.flickr.com/services/api/flickr.photos.search.html

[FlickrAPI] See http://www.flickr.com/services/api/ for the Flickr API.

[FlickrKey] See http://www.flickr.com/services/api/misc.api_keys.html for more information about Flickr API keys and how to apply for one.

[Google] Patent submitted as United States Patent Application 20060242139 and at the time of writing still pending. http://www.google.com/patents/US20060242139

[Linux] See http://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/

[SVG] See http://www.w3.org/Graphics/SVG/ for more information about SVG, including the standards

[vanLaenen13] 'Tar-based Back-ups' *C Vu* Vol 25 Issue 2, May 2013.

[WRUF] The source code can be downloaded from https://github.com/filipvanlaenen/wruf. Feel free to copy the code and create your own branch as long as you respect the software license.

[YAML] See http://yaml.org/ and http://en.wikipedia.org/wiki/YAML for more information about YAML.

Dynamic C++, Part 1

Static and dynamic languages have different trade-off. Alex Fabijanic attempts to get the best of both worlds.

> *As to which is more important, Dynamic or static, both are absolutely essential, even when they are in conflict.*
> ~ Robert Pirsig, Metaphysics of Quality [Pirsig]

C++ is a statically-typed language. The static nature of the C++ type system provides a data integrity 'safety net'. The compiler is an indispensable runtime-surprise-prevention tool and the static nature of C++ provides runtime performance gain. Before we go any further, let's clarify the nomenclature – static typing here should not be confused with its close relative strong-typing, see the definitions on the right for details. It is precisely the 'weaknesses' of the type system (in combination with polymorphism and templates) that provides the functionality needed for dynamic-like behavior within a statically typed language such as C++. And there are circumstances calling for a 'softened' type system, where a degree of performance sacrifice is acceptable and runtime type system relaxation desirable (or even necessary). To provide generic functionality automatically adaptable to different data types at runtime within the confines of standard C++, the runtime type detection system does not suffice – one has to resort to library solutions based on various techniques described later.

Static vs. dynamic

This classification has to do with the timing of value-to-type attachment. Static means values are attached to types ('compiled') at compile time. Dynamic means they are attached ('interpreted') at runtime. Since C++ attaches values to types at compile, it follows that C++ is a statically typed language.

Strong vs. weak

This classification has to do with 'loopholes' the programming language type system leaves open for its type system to be 'subverted'. Both C and C++ allow different types and pointers thereof to be cast to each other. While C++ is stricter than C, it is also backward compatible. But even without the C compatibility, C++ provides ways to subvert the type system and therefore can not be considered a strongly typed language. As a (non-exhaustive) example, void* and union disqualify C and C++ from strongly-typed qualification.

C++ Dynamism

To get a broader perspective, let's look at the need for and benefits of 'dynamic' typing in C++. Even the dynamic language environments are ultimately built on a statically typed foundation. Yet, when the static/dynamic langugage interaction need arises, we embed those 'foreign' language environments and we must speak to them in a 'foreign tongue' (i.e. through a specialized translation layer). Wouldn't it be nice to (a) smooth the rough edge between the two in a reusable way, while also (b) addressing the concern of dealing with external data of different types and (c) gain a generic-purpose standard C++ 'dynamic' environment natively and seamlessly, as a side-effect?

While standard C++ claims to be a general-purpose language, it stops abruptly at the point where (among other things) dynamic-language-like behavior is needed – as things stand at the time of this writing, even a well-known, half-way-there oldie like **`boost::any`** could only make it to 2014 Technical Specification (a pre-standardization mechanism for almost-there-but-not-yet-standard-ready libraries and language features), which means it will not be standardized will not be standardized until 2017 at least.

The original spark triggering this systematic overview (**`boost::any`** port to POCO some years ago) was decidedly not about C++ as a 'dynamic' language but rather about a way to work around the rigidly static C++ type system in a reasonably efficient and reusable way – through a library solution. But libraries are languages, so the C++ library solutions for type dynamics are C++'s native 'dynamic' languages of sorts. Although the underlying types are still statically defined, due to the weakness of C++ type system, they can be dynamically held; and one can coax them to be readily available at runtime, holding values of different types, that they very much 'quack and walk' as a dynamic language... Suddenly, the idea of a native C++ 'dynamic language' does not sound so outlandish ...

Data from external data sources arrives in a variety of types and brings along the need for efficient and transparent datatype conversion. The proliferation of web-based interfaces and databases with the addition of popular textual formats such as JSON and XML exacerbates the need for a relaxed type system with transparent and safe conversion facilities. This is a domain where dynamic languages have gained a significant footing. In order to clarify the premises for this writing, we must take a brief detour here (see 'C$$++ Dynamism').

So, back on track – is it possible to provide dynamic-like behavior within the constraints of standard ANSI/ISO C++? How can a C++ programmer accurately and efficiently transfer data from a database to XML, JSON or HTML without stumbling over the rigid C++ static type-checking mechanism at compile time while ensuring accuracy at runtime? Can type-erasure and (checked) type-conversion techniques fit the bill? Given both historical (ANSI C union and **`void*`**, MS COM Variant, **`boost::`**[variant, any, **`lexical_cast`**], Qt QVariant, **`adobe::any_regular`**) and recent (**`Boost.TypeErasure`**, Facebook **`folly::dynamic`**) development trends (including the pending Boost.Any C++ standard proposal), the need for a way around the static nature of C++ language is obvious. Since the DynamicAny [Fabijanic08a, Fabijanic08b] article, some new solutions [Folly] have appeared, POCO [POCO] has seen several release cycles and **`Poco::DynamicAny`** is now known under a new name – **`Poco::Dynamic::Var`**. Additionally, the performance and type-safety of number/string conversion has been

Alex Fabijanic has been a professional programmer since 1992, specializing in industrial automation and process control software using C and C++ since 1998. He leads the POCO (C++ POrtable COmponents, http://pocoproject.org) project and occasionally writes Javascript and Python code. He can be contacted at alex@pocoproject.org.

Data from external data sources arrives in a variety of types and brings along the need for efficient and transparent datatype conversion

improved by replacing **sscanf**/**sprintf**-based conversion with double-conversion [DoubleConversion] (also used by **folly::dynamic**).

POCO

In this article series, both externals and internals of **boost::**[**variant**, **any**, **type_erasure**], **folly::dynamic**, **Poco::Dynamic::Var**, Qt QVariant and **adobe::any_regular** are explored and compared. Design, capabilities, ease of use as well as pros and cons of each solution will be examined. Performance benchmark comparisons results will be provided as well.

We will start our journey through Dynamic C++ world with a smooth sail – simple, minimalistic and well-known **boost::any**, a 'bipolar' class with deceptively soft, entirely type-agnostic conception and surprisingly rigid, ultra-strongly typed delivery interface (or, should we say, lack thereof). As we move on, the journey takes us into the rough waters of solutions that endeavor, each in its own way, to provide dynamic facilities within the confines of standard C++ and its static type system. The solutions gradually build on existing foundations, attacking the problem from various angles while trying to keep size, performance and datatype integrity under control.

But, first things first – let us start by looking at the concerns shaping the solutions and the ingredients they're made of.

Dynamic concerns

What are the concerns involved with dynamic behavior and how are they solved? Let's enumerate, disect and analyze them ...

Storing value

This concern has to do with the location where the actual bits representing the value reside. Within the C++ memory model, there are two distinct choices – heap and stack – and a hybrid between the two; more on this later. There are various memory allocation optimization methods that look just like heap allocation from programmer's standpoint but actually allocate from different places; such constructs are beyond the scope of this article. Let us just mention here that the term 'stack' above should be used cautiously; it is very common to refer to *placement new* techniques constructing objects in a dedicated storage inside a class as 'stack-based'; that convention is used in this article as well. It is, however, important to remember that there is nothing preventing an object of such class to be allocated on the heap.

Performing operations

The most frequently encountered operations are type conversions, between string and numeric or other values. Furthermore, there are assignment, arithmetic and logical operators. There are other language operations such as bitwise but those are not of concern for this article's theme. Finally, there are various conversions or transformations; as we will see later, some solutions even provide capability to add custom operations to types at compile time.

POCO

POrtable COmponents C++ Libraries are:

- A collection of C++ class libraries, concpetually similar to the Java Class Library, the .NET Framework or Apple's Cocoa.
- Focused on solutions to frequently-encountered practical problems.
- Focused on 'internet-age' network-centric applications.
- Written in efficient, modern, 100% ANSI/ISO Standard C++.
- Based on and complementing the C++ Standard Library/STL.
- Highly portable and available on many different platforms.
- Open Source, licensed under the Boost Software License.

In regards to Boost, in spite of some functional overlapping, POCO is best thought of as a Boost complement (rather than replacement). Side-by-side use of Boost and POCO is a very common occurence.

Retrieving value

Value retrieval ranges from a strict requirement to explicitly specify the held type, to transparent conversion between different types, sometimes with runtime exceptions thrown if conversion is impossible; with some solutions, it is very easy to venture into undefined behavior if the user is not careful. Sometimes, built-in value retrieval is readily available, while in some cases the user is required to use pre-existing or provide custom external 'scaffolding' in order to extract the held value.

Runtime performance

From the runtime performance standpoint, there will typically be two concerns: heap memory allocation and conversion/transformation costs. From this aspect, anything that could be done at compile time, should. Additionally, as mentioned above, small object optimization affects runtime performance in both ways – positively when heap allocation is avoided and negatively every time the value is retrieved.

Memory usage

Memory usage will vary, from the exact type size (plus platform-dependent alignment, if applicable) to a fixed size, large enough to hold the largest stack-based type supported.

Code size

The binary code size generated by various solutions will mostly be proportional to the functionality provided. For example, **boost::any** code will be small due to non-existent conversion logic. **Poco::Dynamic::Var** code will be the largest, due to exhaustive involvement in type conversions and accuracy checks. The rest of the solutions are somewhere in between.

Ease of use

Last but not least, this concerns the user experience when dealing with dynamic functionality. Some solutions have rigid compile-time constraints, while some others may exhibit surprising runtime behavior.

if only smaller types are used, sometimes there may be some space **not effectively used but consumed** nevertheless

If the user has to understand its implementation details in order to use a software component correctly, the total value of the abstraction is diminished regardless of the implementation quality. Or, as Scott Meyers succintly puts it, “Make interfaces easy to use correctly and hard to use incorrectly”.

Data storage

We have several choices for where and how we store the data values. Again, each comes with its own implementation and concerns.

Storing the value on the heap

If the value resides on the heap, we will pay in runtime performance for memory allocation. However, the amount of memory will be variable, commensurate with the size of held type plus platform-dependent padding/alignment.

Implementation:

- **`void*` and `operator new`**

  This technique provides dynamic-like behavior by virtue of `void*`, a C language construct allowing pointers to unknown types. The default `operator new` allocates memory on the heap. Due to the type-independent nature of `void*`, the newly created entity can be of any type, so the `new operator` can construct the type needed in the allocated memory; note the difference in word order – `new operator` first calls `operator new` and, after the memory is allocated, constructs the object. From that point on, it is up to the ‘dynamic’ solution and programmer to ensure the newly created type value is properly treated. There are some variations on this theme in later described solutions and we will examine them in due course.

Concerns:

- **Allocation overhead**

  Memory allocation on the heap can be an expensive runtime operation; for optimization purposes the language allows overloading of the `operator new`. This allows for various schemes of memory (pre)allocation that alleviate the performance hit imposed by the default `operator new`.

- **Memory cleanup**

  Memory that was allocated on the heap by `new` must be released with `delete`.

Storing the value on the stack

If the value resides on the stack, we will invariably pay the storage size of the largest value we wish to store. As mentioned earlier, although commonly referred to as ‘stack’, a more appropriate name would be ‘internal’ because there is nothing preventing the creation of object on the heap.

Implementation:

- **`union` + tag**

  This technique utilizes the C++ `union` facility. Unlike `struct`, whose members are laid out in memory next to each other, `union` can only hold one value at a time because its data members overlap. This `union` feature provides the same storage location for different types – a feature that can be exploited for dynamic-type-like behavior at runtime without paying the full sum-of-storage price for all the types supported. Additional tag is needed to indicate the currently active union member. In C++03 standard, the limitation is that only POD and classes with trivial construction/destruction can be used. C++11 standard relaxes the only-trivial-construction/destruction limitation.

- **`union` + `placement new`**

  This technique utilizes the C++ `union` in combination with placement `new`. Placement `new` does not allocate memory but only constructs object in pre-allocated storage. The reason for the use of `union` is twofold:

  - there is a need for a special-purpose union member ensuring proper alignment for the largest type held when it is not known at compile time
  - there are ‘hybrid’ solutions, mixing types known at compile time with ‘raw’ storage for the unknown types (placement-new-constructed at runtime); the tag indicating currently active type is necessary in this case

Concerns:

- **Size**

  Since a `union` must accomodate the largest type supported, it has to occupy at least the largest type size.

- **Alignment**

  In practice, the amount of space needed is often more than largest `union` member size due to platform-dependent alignment requirements. This means that, if only smaller types are used, sometimes there may be some space not effectively used but consumed nevertheless. Alignment requirements and details are beyond the scope of this writing, but let us just mention here that it is a fairly complex topic, especially in the C++03 context; for details, see [Sutter].

- **Destruction**

  When the held object is placement-new constructed in pre-allocated storage, there is no need to explicitly call `delete`. This, however, means that the destructor has to be called explicitly by the programmer.

Using a hybrid solution

The hybrid solution (also known as small object optimization, configurable at compile time) compromises, to an extent, the stack size an area where things get really complicated – ‘dynamically’ attaching operations to types that do not ‘natively’support them

concern in order to avoid the heap allocation penalty for types under certain size; this solution, however, imposes runtime penalties of size inspection (a) before instantiation and (b) at every value retrieval.

Implementation:

- **Small Object Optimization**

  This is a combined technique of heap- and stack-based storage strategies. The programmer decides and specifies at compile time the maximum object size that can be created on the stack. At runtime, based on the compile-time value, the decision is made whether the new object will be constructed on the stack or the storage for it to be constructed will be allocated on the heap.

Concerns:

- **Runtime detection performance**

  Obviously, every creation and retrieval of the value will incur the penalty of the value location detection. There are additional difficulties with assignment and swap operations as well as with exception safety.

- **Stack use**

  The fixed stack space is used indiscriminately, even when the value is allocated on the heap (in which case, the stack space usually serves as the pointer storage).

Operations

Finally, we get to choose what sort of operations can be applied to these types. These choices will strongly affect the usability of the types so care and a deep understanding must be used.

Type conversions

Type conversions are the most frequently encountered operations, the most frequent conversions being those between numbers and strings. Conversions between compatible types (e.g. `short` to `int`) can often be performed statically. If static conversion is not possible, then dynamic functionality must take its place; this typically involves parsing a string to generate a corresponding number or vice-versa – formatting a number into string. Not all solutions described here are equally cooperative in this area; they range from those not providing any (no pun intended) conversions, via those providing accompanying mechanisms for defining conversion facilities to those providing built-in conversions.

Standard language operations (+, -, ==, ...)

These operations are indispensible for built-in types. They can also be brittle due to many runtime cases where they may make no sense for the held types/values. Therefore the choice is to either not provide them at all, or provide them and throw exception at runtime if the attempted operation makes no sense for the current values. The latter behavior is consistent with the way a dynamic language would behave.

Custom operations

This is an area where things get really complicated – ‘dynamically’ attaching operations to types that do not ‘natively’support them. There are some solutions that provide this functionality. There are also some pitfalls. These will be analyzed and discussed later.

Ingredients

The ‘ingredients’ for the dynamic functionality within C++ ‘recipe’ are summarized in the following list:

- new
- placement new
- void*
- union
- virtual functions
- templates

From the entities listed above, we already discussed **`new`** and **`union`**; the ones that were not touched on so far are virtual functions and templates.

- **Virtual functions**

  Virtual functions are, of course, an indispensable mechanism for runtime polymorphism, providing objects with identical interface that behave differently. They help tremendously in defining conversions and other operations, where it is very convenient to provide default behavior (often throwing an exception) in the parent class and appropriately override it in descendants. Virtual functions inflict both size and performance penalty.

- **Templates**

  Templates are another powerful C++ mechanism providing compile-time genericity. When combined with other facilities described here, templates can produce very powerful (but often complicated) programming constructs.

Boost.Any

This well-known class has been around for a long time; at the time of this writing, it is an active proposal for standardization [Dawes12]. According to proposal authors, **`std::any`** is a container for “Discriminated types that contain values of different types but do not attempt conversion between them”. This classifies any as a generic (in the sense of ‘general’, not template-based) solution for the first half of the problem – how to accommodate any type in a single container. The ‘syntactic sugar’ is avoided template syntax – any itself is not a template class but it has a template constructor and assignment operator; this is conveniently used to avoid the aesthetically displeasing angle brackets:

```
any a = "42";
any b(42);
```

> assignment will incur a **performance penalty** due to heap allocation, and a **size/ performance penalty due** to virtual inheritance of the internal placeholder

```
template<typename ValueType>
any(const ValueType & value):content
  (new holder<ValueType>(value))
{
}
```

Listing 1

What happens 'under the hood' is:

- at compile time, assignment (or construction) code for the appropriate type is generated
- at run time, the value is assigned to a polymorphic holder instantiated on the heap.

See Listing 1 for an example.

Runtime dynamism is achieved through polymorphism as shown in Listing 2.

Right away, it is obvious that assignment will incur a performance penalty due to heap allocation, and a size/performance penalty due to virtual inheritance of the internal placeholder. The convenience of **`any`** extends from the construction/assignment moment during its lifetime and stops the moment one wants to retrieve the value. Until then, **`any`** looks and acts like, well – any value. While it works in a wonderfully transparent manner on the assignment side, the data extraction side is out of **`any`**'s 'scope of supply' – the class does not offer value retrieval or type conversion functionality; the only way to retrieve the value is through **`any_cast`** – a set of free-standing functions that either return the value of the exact held type or throw if something else is requested. **`Poco::Dynamic::Var`** takes off where **`any`** stops, providing user-extensible conversion facilities for non pre-specialized types; the design, rationale, use and performance of this class hierarchy is described in a later installment of this series of articles.

```
class placeholder
{
public:
  virtual ~placeholder()
  {
  }
// ...
  virtual const std::type_info & type()
    const = 0;
// ...
}

template<typename ValueType>
class holder : public placeholder
{
public:
  holder(const ValueType & value):held(value)
  {
  }
  // ...
  ValueType held;
};
```

Listing 2

`Poco::Any` [POCO.Any] is a port of **`Boost.Any`** to POCO.

Boost.Variant

According to the authors [Boost.Variant], **`Boost.Variant`** class template is "a safe, generic, stack-based discriminated union container, offering a simple solution for manipulating an object from a heterogeneous set of types in a uniform manner". It determines the needed storage at compile time, uses **`boost::mpl`** and limits the runtime capabilities to types defined at compile time.

The performance penalty of Boost.Any creation and polymorphic nature, as well as its incapability to provide reliable compile-time type detection, were the motivating factors for **`boost::variant`** authors. For that reason, variant is stack-based and provides reliable compile-time type detection and value extraction. There is a caveat – to enforce the 'never empty' requirement, variant may temporarily allocate storage on the heap to keep the old value for the case when an exception being thrown during assignment. The authors claim to have plans for alleviating this shortcoming.

Faced with a **`boost::variant`**, hoping it comes with built-in (or at least accompanying) type conversion facilities, a naïve user may try something like this:

```
variant<int, string> v = 1;
string s = v; // compile error
boost::get<std::string>(v); // throws
```

While **`Boost.Variant`** offers slightly more cooperation than Boost.Any on the extraction side, it is not seamless or without dangers – intuitive code won't compile, while the next simplest way is brittle. Authors admit the shortcomings and brittleness of the above approach and provide a visitor mechanism as a vehicle to unleash the full strength of Boost.Variant. The visitor is created by inheriting from the **`boost::static_visitor<>`** class template (see Listing 3).

In order to provide the type conversions however, user must define a visitor per destination type, e.g. to facilitate the most common conversion between numbers and strings, the following minimal set of classes is needed, as seen in Listing 4.

Internally, the variant data is in-place constructed into the storage allocated at compile time and large enough to accommodate the largest datatype specified; storage is a union of char array plus alignment padding (see Listing 5).

The most significant constraint of Boost.Variant is that it can only accept a predefined set of types

```
class my_visitor
    : public boost::static_visitor<int>
{
  public:
  int operator()(int i) const
  { return i; }

  int operator()(const std::string & str) const
    return str.length(); }
};

int main()
{
  boost::variant< int, std::string > u
     ("hello world");
  std::cout << u; // output: hello world

  int result =
     boost::apply_visitor( my_visitor(), u );
  std::cout << result;
  // output: 11 (i.e., length of "hello world")
}
```

Listing 3

```
struct string_int_converter
  : public boost::static_visitor<int>
{
  int operator()(int i) const;
  int operator()(const std::string & str) const;
  int operator()(double d) const;
};

struct string_dbl_converter
  : public boost::static_visitor<double>
{
  double operator()(int i) const;
  double operator()
      (const std::string & str) const;
  double operator()(double d) const;
};

struct num_string_converter
  : public boost::static_visitor<std::string>
{
  std::string operator()(int i) const;
  std::string operator()
      (const std::string& str) const;
  std::string operator()(double d) const;
};
```

Listing 4

```
variant<char>: (1) 8
variant<int>: (4) 8
variant<float>: (4) 8
variant<double>: (8) 16
variant<std::string>: (32) 40
```

Listing 6

Typically, depending on the size of the held type, there will be some extra space used (e.g. on 64-bit Win8/VS2012), **`variant<char>`** will occupy 8 bytes, while **`variant<std::string>`** will occupy 40 (see Listing 6).

The most significant constraint of Boost.Variant is that it can only accept a predefined set of types. If a type is not explicitly listed in the declaration of the variant variable via a template instantiation, that type can not be assigned to it. The never-empty guarantee imposes some significant limitations and there is a discussion going on as to whether it is a reasonable constraint to start with. Default constructing variant as empty would alleviate this problem but it would also introduce the problem of always having to deal with empty in the visitors.

Comparison between Boost.Variant and Boost.Any

For easier understanding of the concepts behind the two classes described so far, **`boost::any`** is often compared to 'type-safe **`void*`**' whereas **`boost:variant`** is compared to 'type-safe **`union`**'. While there are certainly similarities, this comparison should be taken cautiously.

```
template <std::size_t size_,
          std::size_t alignment_>
struct aligned_storage_imp
{
  union data_t
  {
    char buf[size_];

    typename mpl::eval_if_c<
        alignment_ == std::size_t(-1)
      , mpl::identity<detail::max_align>
      , type_with_alignment<alignment_>
      >::type align_;
  }
  data_;
  void* address() const
  {
    return
       const_cast<aligned_storage_imp*>(this);
  }
};
```

Listing 5

The never-empty guarantee imposes some significant limitations

Boost.Variant advantages over Boost.Any:

- guarantees the type of its content is one of a finite, user-specified set of types
- provides compile-time checked generic visitation of its content (Boost.Any provides no visitation mechanism at all; even if it did, it would need to be checked at run-time)
- offers an efficient, stack-based storage scheme (avoiding the overhead of dynamic allocation).

Boost.Any advantages over Boost.Variant:

- allows any type for its content, providing great flexibility
- provides the no-throw guarantee of exception safety for its swap operation
- no template meta-programming techniques, which avoids potentially hard-to-read error messages and significant compile-time processor and memory demands.

Conclusion

The analysis of the available techniques, solutions 'ingredients' and trade-offs for dynamic-language-like functionality within standard C++ was provided. Two existing solutions, **`boost::variant`** and **`boost::any`** were described and compared. In the next installment, we will look into more existing solutions with designs/funactionality comparisons and performance benchmarks. Stay tuned ... ■

Credits

Help in assembling and systematizing the information in this article came from numerous sources. Kevlin Henney provided feedback and constructive discussions on **`boost::any`** and the topic in general. Steven Watanabe provided valuable help and guidance on **`boost::variant`** and **`boost::type_erasure`**, which will be presented in the Part II. Günter Obiltschnig and Andrei Alexandrescu provided valuable feedback and encouragement. The list is, of course, not exhaustive – many other people, discussions, libraries and code samples were an indispensable source of help in gathering and systematizing the information provided in this article.

References

Boost.Any] 'Boost.Any', Boost C++ libraries, Kevlin Henney, http://www.boost.org/doc/libs/1_53_0/doc/html/any.html

[Boost.Variant] 'Boost.Variant', Boost C++ libraries, Eric Friedman and Itay Maman, http://www.boost.org/doc/libs/1_52_0/doc/html/variant.html

[Dawes12] 'Any Library Proposal', Revision 1, Beman Dawes and Kevlin Henney (2012) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3390.html

[DoubleConversion] 'Double-conversion library', https://code.google.com/p/double-conversion/

[Fabijanic08a] 'DynamicAny', Part I, Alex Fabijanic, *Overload* 86, August 2008, http://accu.org/index.php/journals/1502

[Fabijanic08b] 'DynamicAny', Part II, Alex Fabijanic, *Overload* 87, October 2008, http://accu.org/index.php/journals/1511

[Folly] Facebook folly library, dynamic class, https://github.com/facebook/folly/blob/master/folly/docs/Dynamic.md

[Pirsig] 'A brief summary of the Metaphysics of Quality', Robert Pirsig (2005) http://robertpirsig.org/MOQSummary.htm

[POCO] POCO C++ libraries: http://pocoproject.org/

[POCO.Any] Poco::Any, http://pocoproject.org/docs/Poco.Any.html

[Sutter] 'Construction Unions: A C++ Challenge', http://www.informit.com/articles/article.aspx?p=360435

More information

[ACCU13] 'Dynamic C++', *ACCU 2013 Conference*, http://www.slideshare.net/aleks-f/dynamic-caccu2013

[C#] 'Using Type dynamic', *C# Programming Guide*, http://msdn.microsoft.com/en-us/library/dd264736.aspx

The Uncertainty Principle

Not being sure of something is usually thought of as a problem. Kevlin Henney argues to the contrary.

No, I'm not talking about Heisenberg, his lack of certainty or what he may (or may not) have thought about Schrödinger and the virtual vivisection of his cat. I'm also not referring to Heisenbugs, those peculiar defects that seem to disappear or reappear somewhere unexpected when you rerun code in a different environment, such as under a debugger, on your machine (it worked on mine) or at the customer's site.

The principle is that, in software development, a lack of certainty about something can be part of the solution rather than part of the problem. This point of view can, to many, seem a little counterintuitive and more than a little disturbing. There is a strong tendency for humans to feel unsure about uncertainty, in two minds over ambiguity and a little wobbly with instability. Whether over technology choice, implementation options, requirements or schedule, uncertainty is normally seen as something you must either suppress or avoid. Of this many people appear, well, certain. That you should embrace it and use it to help determine schedule and design is not immediately obvious.

Does an iterative approach to development embrace uncertainty? It can do, but not the way that most practitioners justify or use iterations. Starting from a position of incomplete knowledge and gradually iterating through hypothesis, experiment and discovery towards – one would hope – working software addresses part of the question of moving from the unknown to the known. But this view of iterative development accommodates and seeks just to reduce uncertainty over time rather than genuinely embracing it and using it to drive everything from critical project and product decisions to the detail of code. It is a subtle but important distinction.

Uncertainty arises when you are aware that at some point a decision about something may need to be made, and it is not clear what the best option is, but it is clear that it could have a significant influence. This may relate to an implementation detail (list or lookup table?), a broader technical choice (off-the-shelf database or custom, in-memory data model?) or a customer decision based on market direction (blue pill or red pill?). The decision point may appear to be now, but now may not be the best time to commit one way or another, even though you want to make progress.

Lean thinking offers options thinking as part of its toolkit, the act of taking a decision to take a decision, deferring a decision to a later point. This is not a matter of hesitantly wavering over a decision; it is based on identifying the last responsible moment a decision can be made, one that balances maximum knowledge with maximum opportunity.

But uncertainty is not just a driver for the schedule: it influences code. When there is more than one way to do something, many developers take that as a cue that what they must do is choose one. They may find themselves debating the choices with a colleague. In such a situation, it turns out that the real challenge is actually not to choose one of them, but to restructure the code so it's not as important which option is chosen. Add an object, an interface or a wrapper layer that reduces the significance of the actual choice. Then, should the decision need to be retaken, either because circumstances change or new information becomes available, the impact of change is diminished.

Uncertainty need not be unprincipled. Use it to help mark out the boundaries in a software system and loosen the coupling. Entertaining more than one option is not an act of indecision: it is a reflection of understanding, a way of uncovering possible areas of instability and seams of change in a software architecture. As Émile-Auguste Chartier noted, "nothing is more dangerous than an idea, when you have but one idea". ■

(First published in *NDC Magazine*, June 2009)

Kevlin Henney is an independent consultant, speaker, writer, trainer and long-standing ACCU member. He is co-author of *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*, two volumes in the Pattern-Oriented Software Architecture series, and editor of the *97 Things Every Programmer Should Know* book and site. He can be contacted at kevlin@curbralan.com