Counting from 98 to 14 in C++

Version 1

Julien Jorge
julien.jorge@stuff-o-matic.com

Contents

Preface

1 About this Book

Once upon a time, as I was working on a large project with others C++ programmers, I was asked to set up a series of talks about the language and especially about what has changed since the arrival of C++11. It was in 2020.

So I started to write some slides with what seemed to be the key features from C++11, and quite soon I had to face the truth: it is a lot of content, and there was three additional major updates in the language that should be covered too.

In the end I did not do the talks. However, I kept working on the slides, until eventually I decided to switch the format. It is probably too much material for a talk, but what about a small book?

Hence this document.

The goal is to list many, if not all, essential features introduced in the C++ language since its first well-known deep update, known as C++11, up to the most recent version of the standard, which is C++20 by the time I am writing this.

These features are for the most part presented following a format where the pre-C++11 way is reminded to the reader, with a short explanation of why it may have been problematic or inefficient, then the new way of doing things is presented.

Some parts of the language are silenced, mostly the ones for which I don't know much, other parts are more thoroughly presented. In any case, this book won't go into the details and subtleties of any feature, nor into compiler-specific stuff. The reason being mostly time (as far as I can tell I have a limited amount of that in my life) and space (the book is already large enough). The reader is invited to satisfy his curiosity and complete is knowledge by reading other material. For example, the website `https://en.cppreference.com` has everything we need to know about any feature of the language.

Be advised that some critics may suddenly appear in these pages, about the language or the programmers. Keep in mind that those are personal opinions and may change suddenly!

Finally, a basic knowledge of the language is preferable for the reader to enjoy this book, as some notions will be used without being explained.

While this book is about C++, one should remember that C++ itself is just a tool in the programmer's toolbox. If you are focusing on learning C++ to become a programmer, a good programmer, I would suggest to rethink your plan and learn programming on a larger scale: computer architecture, algorithms, data structures, project management, packaging, dependency management, coding style, reviews, testing... There are many aspects to be familiar with in the daily life of a programmer, keep some place for them.

As a good starting point, every programmer should read Code Complete [McC04]. This book goes in detail in all aspects of software development, backed up by data coming from over decades of real-life projects, so if you read it you will also gain part of the knowledge from these people who tried, failed, and succeeded before you.

Clean Code [Mar08] is a good second book to read, even though I would not approve all suggestions. For example, it pushes for intensive factorization and the use of object-oriented programming everywhere, which are rather things I have painfully learnt to use parsimoniously. Still, the book is a reference in software development, so you should read it at least to make yourself an opinion and to know what is going on in the business.

Finally, remember that if reading is acquiring the experience of others, practicing is building our own experience. So I strongly recommend to find or start a side project, maybe even a rewrite of existing tools, just to try and get a grasp of the potential underlying complexity.

2 About the Author

Should one take this book's content at face? Is the author legit? It is normal to question the legitimacy of who pretend to give advice. So in order to help you gauging the credibility of this book, I think it is important to tell a bit about me.

First of all, I read a lot of code. I read code almost everyday, on GitHub, on blogs, on StackOverflow, Reddit, and on other forums. I read code written by me or others, from my personal projects, from my employer, or from random projects I find on the Internet. All this code is displayed either on my laptop, or in a terminal connected to a remote server, or on my phone. Reading is undoubtedly the main part of my programming activities.

On the productive side, I write code as a hobby since 1994, and professionally since 2005. I have at least half a million of C++ behind me, just counting the lines of code that survived in past projects I could find.

I also have coded a lot of Bash, a good share of Java, a bit of HTML and a bit of JavaScript. Additionally, in a more anecdotal way, I coded some C# programs, some ActionScript, some Pascal and Delphi ones, a small compiler in Eiffel, some pet projects in Visual Basic too, BASIC a long time ago, on a Commodore 128 and later under DOS. I also did a bit of Objective-C.

Let's face it though, a good share of this code was crap.

Some code did end up well nonetheless. One project I am proud of is a mobile game written in C++, which was played by more than 500'000 people every day during more than three years. Aside from that, I also took part in projects that were struggling to start and brought them into a viable product. So I guess I made stuff that does not suck.

When I code I tend to think about long term and architecture. I try not to take any shortcut and to answer the problem without attempting to solve the future. I code in small boxes, many, with the intent that they can be broken, removed, replaced, without changing everything. It wasn't always like that but that's how I work today.

Finally, I am certainly not the type to rush for the new thing. I like tools and practices that have been well tested, so you probably won't hear me telling you to use this new thing from C++42 because it's new and it will show that you are modern and blah blah blah.

Convinced? Anyway, I hope you will find something useful in this book :)

3 Where to find this book

This book is available as a PDF, always synchronized with the latest changes, at https://github.com/j-jorge/counting-in-cpp/releases/download/continuous/counting-in-c++-wip.pdf.

It is also provided as a website at https://julien.jorge.st/counting-in-cpp/, in good old HTML format.

Finally, the LaTeX source files are stored in a Git repository at https://github.com/j-jorge/counting-in-cpp.

4 License

The header file icon displayed in the margins, like the one on the side of this paragraph, is based on the New Document icon from the Tango Icon Theme [con05]. The original icon is in the public domain and in order to respect the intent of the source the variation made for this book, as in the SVG file available in the repository containing the sources of this book, is also released in the public domain.

This one.

Chapter 1

The C++ Language and its Community

C++is a quite old programming language: it was created by Bjarne Stroustrup in 1982. It was initially thought as an improved C.

The first official specifications were *The C++ Programming Language* [Str86]. This book was then updated multiple times [Str91], [Str97], [Str00], and [Str13].

Starting from 1998, the official specifications are described in *The Standard*.

1.1 The Standard

The standard from 1998 set the ground for a new era of compilers by defining how C++ programs should behave, and by describing the content of the standard library as well as the constraints on its implementation.

The standard is defined by *The ISO C++ Committee*, i.e. people from the industry: Google, HP, Intel, Oracle, and many more.

It is worth noting that no code is provided by the committee. The standard just describes the language, then independent developers (mostly compiler vendors) provide the implementation. Theoretically, developers can switch easily from one compiler to the other. Clients are not tied to the vendor's compiler anymore.

As I write there are six revisions for this document, informally identified by the year they came out:

- **C++98** defined the core features: the syntax, the memory model, templates, namespaces... And the Standard Template Library (STL): `std::vector`, `std::string`, `std::map`...
- **C++03** fixed some wording and inconsistencies.
- **C++11** the beginning of what is called *modern C++*. Initially expected during the 00's, the committee had to kiss goodbye to some awaited features. Better done than perfect.

- **C++14** mostly bug fixes but also nice features: e.g. variable templates.
- **C++17** nice new features: fold expressions, `if constexpr`, copy elision, `std::optional`, `string_view`...
- **C++20** `std::span`, concepts, modules... Also: your compiler still doesn't fully support C++17.

1.2 The Community

The C++ community is made of humans, so it is naturally composed of a lot of good things and many problems. Sometimes simultaneously. And since the language allows several paradigms and provides multiple tools, there are approximately as many programming styles than there are C++ programmers.

Two typical behaviours appeared very common to me in the recent years. The first one is about the tools provided by the standard library. First we hear people complaining: "The standard does not even contain *feature*. One has to use *libfeature* for it." Cue to the release of the aforementioned feature, and suddenly: "The specs for *feature* in the standard prevent efficient implementations. One has to use *libfeature* for it."

From where I stand, it looks like people's demands are ignored, and they complain about it. Then they receive what they asked for, and they obviously still complain about it. To be fair, it is probably not the same people who complain in each situation. So I hope. Are they?

The second behaviour is about build times. C++ is well known for being the language of quite long to compile programs, especially when the program contains templates or other metaprogramming techniques. This is a topic that regularly comes out as a major pain; it was even listed as the second most frustrating thing in the 2021 Annual C++ Developer Survey [Fou21].

Despite that, once we are on the field we meet developers being like "Hey, here's a header-only library for *feature* relying heavily on templates and metaprogramming stuff." Simultaneously, we hear complaints like "Compilation times are too long! The committee should do something about that!". And still, more developers continue asking for more header-only libraries because it is easy to integrate in a project[1].

This is a problem of resource management. People have a limited computing power, and they use every drop of it to compile and recompile the same heavily template-based stuff again and again, until it becomes unbearable. At this point, instead of reviewing their work, they ask others (the committee, the standard, compiler vendors) to solve their problem. Eventually they may also even buy more computing power... only to push until using every drop of it. Doesn't it look suspiciously like some other real-life important resource management problems?

Ah, humans...

Aside from these little problems, that are actually as much inherent to the language as they are the effect of human behaviour, there is a very large and diverse ecosystem surrounding C++. Many nice libraries and tools are published, often in a free software way, and every where we can see that people want to do their best. This is very stimulating.

[1] Dependency management in C++ is the main frustrating thing according to the aforementioned survey.

1.3 The Cost of Modernity

With all these nice updates coming every three years, as C++ developers we naturally wonder if we can use the most recent features or if we should stick with an older version.

There is a trend amongst us to push toward using the most recent features, visible in online discussions or by the amount of libraries whose description is along the line of "A C++20 library to do something or something else". This trend is called *"modern" C++ programming*, where "modern" unfortunately changes almost every day.

It is tempting to go for the most recent features for several reasons: to benefit from the increasing performance and support from compilers, to write more straightforward code, to make a difference with the old conservative folks… And obviously because it is fun. As developers we are subject to a common disease: the refactoring, also known as "just burn everything and rewrite from scratch".

On the other hand, using the bleeding-edge features has some downsides: their implementation did not go through as much testing than the old methods, and they may even not be available everywhere yet. Moreover, if we write libraries, we must take into account that not all projects can afford to use the most recent features[2]. Consequently, when we choose a recent standard, we explicitly exclude a bunch of project from using it.

On this topic, I would like to share a short experience I had.

Near 2014 I had been working on mobile games developed in C++ and available on iOS and Android (and unofficially on Linux and OSX for the developers). C++11 was ready since a long time when the first project was started so we naturally used it. Using `std::unordered_map` and the likes was quite common in the projects.

At some point the Android version of a game started to take forever to launch. It happened that the filling of a large `std::unordered_set` was the issue, and after digging a bit I found why it was problematic only on Android. It turned out that the Android Native Development Kit (NDK) was using GCC 4.7, for which there was a bug that caused poor performance of the insertions in unordered maps and sets[3].

Actually we probably had poor performance everywhere these containers were used, but only one pathological case made it visible. At the time we managed to work around the issue by using the equivalent containers from Boost but the taste was bitter. We were stuck with a broken compiler for Android and the solution we chose was to tighten an already questioned and large dependency.

Could we have solved this problem by switching to a different or more recent compiler? Probably later, but not at the moment. There was indeed some experimental support for Clang in the NDK even though the official way to go native was via GCC, so we did try that, and it required a lot of work to compile the small shared common part of the project. On the other hand, we were already using Boost so switching to their container was simple, plus it guaranteed equivalent performance on every platform.

Regarding the standard in use, in 2014 C++14 was barely ready and certainly not available in the NDK. Moreover, to put things in perspective, consider that even at the dawn of 2021, C++17 was still not fully

[2] Specialized tooling, weird architectures, politics… All of them may be valid explanations for inertia.

[3] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=54075

supported by the NDK[4]. So, for this platform at least, which I have heard is quite popular, pushing for the trending features is counter productive as it can actually prevent our tools to be used.

So should we stay with the oldest tools for maximum support? Look fook example at Curl or zlib to name a few, they are here since forever and work perfectly, should we come back to the *good old* C? Well, probably not, but I guess there is a compromise to make between cutting-edge and widely accepted.

> **Guideline**
>
> There is an old saying going by "use the right tool for the job". Even though it is nowadays used by programmers to dismiss the choices other have made, there is some wisdom in it.
>
> When facing the question of whether or not to use a feature from a somewhat recent version of the language, take the time to consider what it deprives you and your users it terms of usability; then weight the benefit you gain from its usage.
>
> Know that apart from some details, code written for all older versions of C++ are still valid in newer versions. On the contrary, the more recent are the features you use, the less your code can be imported into other projects.
>
> So, if you chose C++14 just to be able to write `auto foo() -> int {}` instead of `int foo() {}`, or if you chose C++20 just to be able to use a pair of concepts, please reconsider, as these features have no benefit for the final product.

[4]The revision r21e released in January lacks support for `std::filesystem`. This has been added in revision r22b, released in March the same year. Note that the former is a long term support (LTS) release, which means that it is expected to not have full C++17 for a long time.

Chapter 2

Nice Things from C++11

Ah, C++11, the long awaited update of the language. There was so much hope in it! Just imagine, back in the day the language had no automatic dynamic memory management, and not even a hash table in the standard library... in the year 2000!

At that time we were trying to compensate for the lack of features with Boost, eagerly waiting for the ever postponed upcoming C++0x that would change everything. We were going to have modules! And concepts too!

Looking back into these years, I am quite happy that the committee dropped the most complex features to finally publish a new version, because most of the parts that were ready are quite awesome.

2.1 At the Language Level

2.1.1 Range-based For Loops

What was the problem

Before C++11, if someone wanted to iterate over a container, he had to construct an iterator, of the correct type with respect to the container, and write the loop with the classical three steps: initialization (get an iterator on the begining of the container), stopping condition (the iterator has not reached the end), and the loop increment.

```
void multiply(std::vector<int>& v, int c)
{
  // I'm going to iterate over v, so I need an iterator.
  typedef std::vector<int>::iterator it_t;
  const it_t end(v.end());

  for (it_t it(v.begin()); it != end; ++it)
    *it *= c;
}
```

How the Problem is Solved

All of this was quite verbose and repetitive when using standard containers. Starting from C++11, all this administrative stuff can be avoided by using a ranged-based for loop.

```
void multiply(std::vector<int>& v, int c)
{
  // Just get all entries from v.
  for (int& vi : v)
    vi *= c;
}
```

This format uniformizes iteration over anything having a begining and an end. Moreover, it handles the typical subtleties of for loops for you: the end of the container is guaranteed to be computed only once and the increment use the preincrement operator.

2.1.2 `std::nullptr_t` and `nullptr`

What was the problem

The traditional way to set a pointer to zero before C++11 was via the `NULL` macro. So what would happen if we tried to compile the following program?

```
#include <cstddef>

void foo(int) {}
void foo(int*) {}

int main(int argc, char** argv)
{
  foo(NULL);
  return 0;
}
```

Did you expect the program to compile well and `foo(int*)` to be called? Too bad, we are in a good old ambiguous call situation:

```
error: call of overloaded 'foo(NULL')) is ambiguous
```

Since `NULL` is often defined as the integral value zero, the compiler cannot distinguish it from an integer. Thus the error.

How the Problem is Solved

<cstddef>

As an answer to this problem, C++11 introduces the `nullptr` keyword, which exactly represents a zero pointer. Its type is `std::nullptr_t` and it is implicitly convertible to any pointer. Consequently, the code below compiles as expected.

```
void foo(int) {}
void foo(int*) {}
```

```
int main(int argc, char** argv)
{
  foo(nullptr);

  return 0;
}
```

2.1.3 Scoped Enumerations

What was the problem

An enumeration in pre-C++11 code is just like a C enumeration: a list of constants of type `int` in the scope containing the enumeration. For example, note how `appliance::fan` can be accessed directly and assigned to an integer in the code below:

```
enum appliance
{
  fan,
  oven
};

int main()
{
  int v = fan;
  return 0;
}
```

While this is nice when we actually want the values of the enumeration to be an alias for integer constants, for example to store bit field flags, it can quickly become a mess when the entries of different enumerations share the same identifier in the same scope.

```
enum appliance
{
  fan,
  oven
};

enum follower
{
  fan,
  admirer
};
```

The above code will fail to compile with a message along the lines of "error: 'fan' conflicts with a previous declaration." Now the typical solution to that was to add a unique prefix to the enumerated values, but in the times of namespaces and so, is it really a solution?

How the Problem is Solved

A scoped enumeration as introduced in C++11 is declared by adding the `class` or `struct` keyword between `enum` and the identifier. Additionally the storage type of the values can be defined too:

```
enum class appliance
{
  fan,
  oven
};

enum class follower : char
{
  fan,
  admirer
};
```

With these declarations the values are scoped in their respective enumerations and do not conflict with each other; so if we want to access a value we must now prefix it with the name of the enumeration, like in `follower::fan`. Moreover, they are also not implicitly convertible to `int` anymore, which may or may not always be a good thing.

The fact that we can also define the type of their values allows for smaller memory consumption when needed, but also add the possibility to use longer-than-`int` values. Nevertheless, I usually don't specify this type unless necessary since it has to be repeated when the enumeration is forward declared. This repetition adds coupling and complicates any change in the type, for something that look like implementation details to me.

2.1.4 `constexpr`

What was the problem

Let's say we have some complex computation, like for example counting the number of bits set to one in an integer:

```
int popcount(unsigned n)
{
  return (n == 0) ? 0 : ((n & 1) + popcount(n >> 1));
}
```

What happens when we want to be able to call this function both with run-time values and compile-time constant as arguments? In the code below we would want the size of the array to be a constant, but as it is written its size will be computed at run-time, which makes it a non constant-sized array, which is not standard compliant.

```
int main(int argc, char**)
{
  int array[popcount(45)];
  printf("%d\n", popcount(argc));

  return 0;
}
```

A typical solution for this problem is to implement the computation via template classes and meta-programming:

```
#include<cstdio>

template<unsigned N>
struct popcount_t;

template<>
struct popcount_t<0>
{
  enum { value = 0 };
};

template<unsigned N>
struct popcount_t
{
  enum
    {
      value = (N & 1) + popcount_t<(N >> 1)>::value
    };
};

int popcount(unsigned n)
{
  return (n == 0) ? 0 : ((n & 1) + popcount(n >> 1));
}

int main(int argc, char**)
{
  int array[popcount_t<45>::value];
  printf("%d\n", popcount(argc));

  return 0;
}
```

This implementation works but has two major problems: first it is incredibly verbose, second it forces us to implement the same algorithm twice, respectively for run time and compile time computations, doubling the risk of bugs and errors.

How the Problem is Solved

The `constexpr` keyword introduced in C++11 allows us to use the same implementation for both compile-time and run-time computations.

```
#include<cstdio>

constexpr int popcount(unsigned n)
{
  return (n == 0) ? 0 : ((n & 1) + popcount(n >> 1));
}
```

```
int main(int argc, char**)
{
  int array[popcount(45)];
  printf("%d\n", popcount(argc));

  return 0;
}
```

This keyword can be applied to a variable or a function to explicitly tell the compiler that it can and should be computed at compile-time when it appears in constant expressions. It is for example totally possible to call the `constexpr popcount()` function as a template argument, like in `popcount<popcount<42>>()`.

2.1.5 Uniform initialization

What was the problem

There are so many ways to initialize a variable in C++ that it became a running gag, so it's natural that a new initialization syntax was added into C++11.

Anyway, let's jump to the problem. The following code is a recurring issue in C++, so much that even experienced programmers stumble upon it once in a while.

```
struct foo {};

struct bar
{
  bar(foo f);

  int i;
};

int main()
{
  bar foobar(foo());
  printf("%d\n", foobar.i);

  return 0;
}
```

The question is "what is `foobar` in the above code?" Most C++ programmers will say it is a `bar` constructed with a default-constructed `foo`. Unfortunately it is actually the declaration of a function returning `bar` and taking a function returning a `foo` as its single argument. This problem is known as *the most vexing parse*.

A workaround for this issue is to declare a temporary of type `foo` and pass it to the constructor of `bar`:

```
int main()
{
  foo f;
  bar foobar(f);
```

```
  printf("%d\n", foobar.i);
}
```

This is not very convenient, and suddenly the scope is polluted by a useless variable. Moreover it can quickly become hard to implement when a variable number of arguments are passed to `foobar` (e.g. by forwarding the arguments of a variadic macro, or a variadic template 2.3.3).

How the Problem is Solved

Enters the uniform initialization from C++11. By replacing the parentheses with brackets in the construction of the argument, the ambiguity is lifted.

```
int main()
{
  bar foobar(foo{});
  printf("%d\n", foobar.i);
}
```

Used like that, the brackets mean something like "a default-created `foo`". It can also be used to zero-initialize any variable, which is especially nice in a template context:

```
template<typename T>
void many_tees()
{
  // If T is a class, calls the default constructor.
  // If T is a fundamental type (e.g. int), its value is
  // whatever is in memory at &t1.
  T t1;

  // Declares a function t2 with no argument and returning
  // a T.
  T t2();

  // Seems to work. Does it?
  T t3 = T();

  // If T is a class, calls the default constructor.
  // If T is a fundamental type (e.g. int), its value is
  // the zero of this type.
  T t4{};
}
```

When to Use the Uniform Initialization Syntax

Consider the code below:

```
#include <cstdio>
#include <utility>

struct foo_struct
```

```
{
  int a;
  int b;
};

struct foo_constructor
{
  foo_constructor(int v1, int v2)
    // Note that arguments and fields are swapped.
    : a(v2), b(v1)
  {}

  int a;
  int b;
};

struct foo_initializer_list
{
  foo_initializer_list(int v1, int v2)
    // Note that arguments and fields are still swapped.
    : a(v2), b(v1)
  {}

  // An initializer list allows to construct an object using an aggregate-like
  // syntax. We will come back to it in 2.5.6 :)
  foo_initializer_list(const std::initializer_list<int>& i)
    // Note that the fields are set to the same value.
    : a(*i.begin()),
      b(a)
  {}

  int a;
  int b;
};

template<typename Foo>
void build_foo(int x, int y)
{
  Foo foo{x, y};

  printf(".a=%d, .b=%d\n", foo.a, foo.b);
}

int main()
{
  build_foo<foo_struct>(24, 42);
  build_foo<foo_constructor>(24, 42);
  build_foo<foo_initializer_list>(24, 42);

  return 0;
```

```
}
```

What would be the output of this program? The highlighted line constructs a variant of `foo` with what looks like the aggregate initialization syntax, or is it uniform initialization? Maybe is it a call to a constructor? Which one?

Here is the output of this program:

```
$ a.out
.a=24, .b=42
.a=42, .b=24
.a=24, .b=24
```

So the first call is without surprise an aggregate initialization.

The second one is a call to the constructor; since there is one defined for `foo_constructor` then the aggregate initialization is disabled. Note that before C++11 the compiler would have reported an error, saying that the constructor should be used. Here it calls the constructor silently even though it looks like an aggregate initialization.

The last one is the worst. It creates an `std::initializer_list` [1] with the values, then pass it to the corresponding constructor.

This is painfully ambiguous. Unfortunately, the so-called "modern" C++ programming trend is pushing for using bracket initialization, maintaining ambiguities everywhere. I prefer to use it parsimoniously. In particular, uses of `std::initializer_list` should certainly be avoided as they lure the programmer into thinking that the assignment is an efficient aggregate initialization while it is actually copying stuff around.

> **Guideline**
>
> Use bracket initialization if:
>
> - you want to initialize an aggregate,
> - or you want to zero-initialize something in a context where you don't know for sure what the type of the thing is.
>
> *Do not* use bracket initialization if you want to call a non-default constructor.

As a side note, the bracket initialization can be used without specifying the type, for example to assign a variable or for the return statement of a function:

```
struct interval
{
  interval();
  interval(int a, int b);
};

interval build_interval(bool f)
{
```

[1] See section 2.5.6 for details about that.

```
  interval b;

  if (f)
    b = {23, 23};

  return {42, 32};
}
```

This is terrible.

> **Guideline**
>
> Mind the next reader, write what you mean.
>
> While the compiler can effortlessly find the type of a variable, or the type returned by a function, a human will either have to scan back toward the declarations or be helped by a tool, if they have one.
>
> Constructing a value without saying the type is ambiguous for a human. This is a lot of effort to put on the reader to save some characters on the writer's side.
>
> Don't build complex types without explicitly tell what you want to build.

2.1.6 Compile-Time Assertions

What was the problem

Compile-time assertions, declared with a `static_assert`, are a way to verify a property at compile time.

For example, the code below declares a type with an array whose size, with respect to the type's name, must be even:

```
template<typename T, unsigned N>
struct even_sized_array
{
  typedef T type[N];
};
```

How can we ensure that `N` is always even? Before C++11 a solution to that would have been to enter some template dance such that passing an odd value would instantiate an incomplete type.

```
// The third template argument is expected to receive a flag
// telling if N is even (i.e. valid).
template<typename T, unsigned N, bool IsEven>
struct even_sized_array_impl;

// And we implement this type only if the flag is true.
template<typename T, unsigned N>
struct even_sized_array_impl<T, N, true>
{
  typedef T type[N];
};
```

```
template<typename T, unsigned N>
struct even_sized_array
{
  // Now we trigger the check by actually instantiating the
  // helper type, hoping that the flag will be true.
  typedef
  typename even_sized_array_impl<T, N, N % 2 == 0>::type
  type;
};

void test()
{
  even_sized_array<int, 8>::type ok;
  even_sized_array<int, 9>::type nok;
}
```

Not only this is verbose, but these kind of template instantiation errors are well known for producing kilometer-long unbearable error messages.

How the Problem is Solved

Enters `static_assert` in C++11. Just like the good old `assert(condition)` would check that the given condition is true at run time, `static_assert(condition, message)` will check the condition during the compilation.

```
template<typename T, unsigned N>
struct even_sized_array
{
  // We can also put a message to repeat the condition, like
  // professionals. Don't forget the exclamation mark to
  // show it is serious.
  static_assert(N % 2 == 0, "The size must be even!");
  typedef T type[N];
};

// So much less code <3
void test()
{
  even_sized_array<int, 8>::type ok;
  even_sized_array<int, 9>::type nok;
}
```

This is very less verbose than previously and it actually carries the intent. The error message is also clearer, as it is just something like “static assertion failed: the message”.

2.1.7 Lambdas

What was the problem

How would we pass a custom comparator to `std::max_element` before C++11, say to compare strings by increasing size? Certainly by writing a free or static function taking two strings as arguments and returning the result of the comparison.

```
// Comparator for strings by increasing size.
// Declared as a free function.
static bool string_size_less_equal
(const std::string& lhs, const std::string& rhs)
{
  return lhs.size() < rhs.size();
}

std::size_t
largest_string_size(const std::vector<std::string>& strings)
{
  // The comparator is outside the scope of this function.
  return
    std::max_element
      (strings.begin(), strings.end(), &string_size_less_equal)
    ->size();
}
```

Now what if we needed a parameterized comparator, for example to call `std::find_if` to search for a string of a specific size? The free function would not allow to store the size, so we would certainly write a functor object, i.e. a `struct` storing the size parameter and defining an `operator()` receiving a string and returning the result of the comparison.

```
// Need a function object if the comparator has parameters.
struct string_size_equals
{
  std::size_t size;

  bool operator()(const std::string& string) const
  {
    return string.size() == size;
  }
};

bool has_string_of_size
(const std::vector<std::string>& strings, std::size_t s)
{
  string_size_equals comparator = {s};

  return
    std::find_if(strings.begin(), strings.end(), comparator)
      != strings.end();
}
```

How the Problem is Solved

Having the comparators as independent objects or functions is nice if they are used in many places, but for a single use it is undoubtedly too verbose. And confusing too. Wouldn't it be clearer if the single-use comparator was declared next to where it is used? Hopefully this is something we can use with lambdas, starting with C++11:

```
bool has_string_of_size
(const std::vector<std::string>& strings, std::size_t s)
{
  // Only three lines for the equivalent of the type
  // declaration, definition and the instantiation.
  // The third argument is a lambda.
  return std::find_if
    (strings.begin(), strings.end(),
     [=](const std::string& string) -> bool
     {
       return string.size() == s;
     })
    != strings.end();
}
```

> **Guideline**
>
> Mind the next reader: keep your lambdas small.

The Internals of Lambdas

A declaration like

```
[a, b, c]( /* arguments */ ) -> T { /* statements */ }
```

is equivalent to

```
struct something
{
  T operator()( /* arguments */ ) const { /* statements */ }

  /* deduced type */ a;
  /* deduced type */ b;
  /* deduced type */ c;
};
```

Actually the compiler will create a unique type like this struct for every lambda we write.

More conceptually, the parts of a lambda are the following:

[*capture*](*arguments*) *specifier* ->*return_type* { *body* }

Where:

- *capture* is a list of variables from the parent scope that must be accessible inside the lambda. Use `[=]` to tell the compiler to automatically copy any variable used by the lambda, or `[&]` to keep a reference to the corresponding variables from the parent scope. Variables can also be captured in a fine-grained way, e.g. `[=a, &b]` to copy the value of `a` but store a reference to `b`.
- *arguments* are the arguments of the function.
- By default the variables captured by value cannot be assigned to in the body, unless we put the mutable keyword in *specifier*. In effect, it removes the `const` from `operator()`.
- *return_type* is the type of the value returned by this function, if any, or void otherwise. Contrary to any other function, the return type appear after the argument list instead of before.
- *body* is the body of the function.

2.1.8 Type Deduction with `decltype`

What was the problem

During the pre-C++11 era every variable, member, argument, etc. has to be explicitly typed. For example, if we were writing a template function operating on a range, how could we declare a variable of the type of its items?

```
template
<
  typename InputIt,
  typename OutputIt,
  typename Predicate,
  typename Transform,
>
void transform_if
(InputIt first, InputIt last, OutputIt out,
 Predicate& predicate, Transform& transform)
{
  for (; first != last; ++first)
  {
    /* some_type */ v = *first;
    if (predicate(v))
    {
      *out = transform(v);
      ++out;
    }
  }
}
```

How the Problem is Solved

Getting the correct type to put in place of `some_type` was not obvious, and required a fair share of template metaprogramming. Now, with the `decltype` specifier introduced in C++11, the programmer can tell the compiler to use "the type of this expression".

```
template
<
  typename InputIt,
  typename OutputIt,
  typename Predicate,
  typename Transform,
>
void transform_if
(InputIt first, InputIt last, OutputIt out,
 Predicate& predicate, Transform& transform)
{
  for (; first != last; ++first)
  {
    decltype(*first)& v = *first;
    if (predicate(v))
    {
      *out = transform(v);
      ++out;
    }
  }
}
```

In the example above, `decltype(*first)` is the type of the result of dereferencing `first`.

2.1.9 Auto

Section 2.1.8 explained how one could deduce the type of an expression with `decltype` to set the type of a local variable to match the type of an expression.

Now what about storing a lambda in a local variable, what would be the type of the variable? As seen in 2.1.7, the type of the lambda is generated by the compiler, and thus out of reach for the programmer. The solution is then found in the `auto` keyword, introduced in C++11.

```
bool has_string_of_size
(const std::vector<std::string>& strings, std::size_t s)
{
  auto predicate =
     [=](const std::string& string) -> bool
     {
       return string.size() == s;
     };

  return
    std::find_if(strings.begin(), strings.end(), predicate)
    != strings.end();
}
```

Auto as a Type Placeholder

The `auto` keyword tells the compiler to deduce the actual type of a variable from whatever is assigned to it. It can be used as long as there is an expression the compiler can use to find the type. It can be augmented with `const` or `&`.

A typical use is for iterating over an associative container in a for loop:

```
template<typename F>
void for_each_entry(const std::map<int, int>& m, F& f)
{
  for (const auto& e : m)
    f(e.first, e.second);
}
```

In the spirit of 2.1.1, the type of `e` is deduced to `std::map<int, int>::value_type`, to which are added the `const` and the `&`.

When used as a return type, the `auto` keyword allows to defer the declaration of the actual return type after the argument list. For example, if we don't know what is the type but we know it is exactly the one of a given expression, we can combine this with `decltype`:

```
template<typename F>
auto indirect_call(F&& f) -> decltype(f())
{
  return f();
}
```

When not to Use `auto`

It is tempting to use the `auto` keyword everywhere, especially for programmers coming from loosely typed languages such as Python or JavaScript. Moreover, using `auto` gives the satisfying feeling of writing seemingly "generic" code that can work with whatever type is deduced.

In practice, the use of this keyword has lead to very painful to read code, where nothing can be understood without going through every expression assigned to an `auto` variable, and where entire functions have to be interpreted to eventually find the returned type. This is a very high load to pass to the next reader.

> **Guideline**
>
> Mind the next reader; write what you mean.
>
> Declaring a variable or a function as `auto` is like writing an innuendo, and innuendos make the communication more difficult; if you see what I mean.
>
> As a rule of thumb, use `auto` only if:
>
> - there is no other way to write the type (e.g. assigning a lambda to a variable),
> - maybe if the type is a well known idiom (e.g. `auto it = some_container.begin()`, or for a loop variable in a range-based for loop over an associative container), but I would argue that writing the actual type would still be more explanatory for the reader.

> Absolutely never use `auto` in place where you could have used basic types like `int` or `bool`, or by laziness. It is not because it is shorter that it improves the readability. On the contrary, the readability is improved by being clear about what is going on.

2.1.10 Rvalue References

What was the problem

In the function `build_widget()` below, a temporary string is created by the code `label + "def"`, then a reference to this string is passed to the constructor of `widget`, where it is then copied in `widget::m_label`. Finally, the temporary is deleted.

```
struct widget
{
  widget(const std::string& label)
    : m_label(label)
  {}

  std::string m_label;
};

void build_widget()
{
  std::string label("abc");
  widget w(label + "def");
}
```

Temporaries have the interesting property that they have no use but to be consumed by an expression building another value. In other words, in the example above, the purpose of the result of `label + "def"` is to end up in `widget::m_label`. So why do we need a copy? It would be more efficient to just pass the instance created by the concatenation directly to the final variable.

How the Problem is Solved

To solve this problem, C++11 introduces the concept of rvalue references, which are, to put it simply, references to temporaries. These references are denoted with a double ampersand `&&`. See the updated example, below:

```
struct widget
{
  widget(std::string&& label)
    : m_label(label)
  {}

  std::string m_label;
};

void build_widget()
{
  std::string label("abc");
```

```
  widget f(label + "def");
}
```

The signature of the constructor indicates that it accept temporaries as arguments. However, this code is not enough to avoid copies of the string yet, as the instruction `m_label(label)` still copies the string. The next step is to replace this instruction with `m_label(std::move(label))`, and then we will have no copy. This is the topic of section 2.2.

Note that if the constructor's argument was `widget`, like in `widget(widget&& that)`, then this would declare what is called a move-constructor. This is a distinct constructor than the copy-constructor and it is explicitly allowed to steal the internals of its argument, as long as the instance behind the argument stays in a valid state.

2.2 Move Semantics

What was the problem

Copies! Copies everywhere! And dynamic allocations too!

So was the world before C++11. In these old days, if we had to pass a large object to a function that needed its own instance of it, then we would have certainly created a copy of it.

```
struct catalog
{
  catalog(const std::vector<item>& entries)
    // Here we have one allocaction, for the storage,
    // plus copies of the elements from the vector.
    : m_entries(entries)
  {}

private:
  std::vector<item> m_entries;
};

void create_catalog(int n)
{
  // One allocation of n items, plus their initialization.
  std::vector<item> entries(n);

  // ...

  catalog c(entries);
  // I don't need the entries anymore, but they are still here.
}
```

How the Problem is Solved

<utility>

C++11 introduces the move semantics and the `std::move()` function. In practice it means far fewer copies, as the instances' data is transfered from one place to another.

```
struct catalog
{
  // Note the pass-by-value for the argument.
  catalog(std::vector<item> entries)
    // Explicit transfer into m_entries, no allocation.
    : m_entries(std::move(entries))
  {}

private:
  std::vector<item> m_entries;
};

void create_catalog(int n)
{
  // The single allocation in this program.
  std::vector<item> entries(n);

  // ...

  // I don't need the entries anymore, so I transfer them.
  catalog c(std::move(entries));
}
```

In the example above, there is only one allocation, for the vector of entries in `create_catalog()`, and zero copies. The ownership of the allocated space is actually transfered from one vector to another every time `std::move()` is used, until it ends up in `m_entries`.

2.2.1 The Move Constructor and Assignment Operator

In order to make it work, a new type of reference was added to the language; and classes can use this reference in constructors and assignment operators to implement the actual transfer of data from one instance to another. So to mark an argument as moveable, use &&:

```
struct foo
{
  foo(foo&& that)
    : m_resource(that.m_resource)
  {
    that.m_resource = nullptr;
  }

  foo& operator=(foo&& that)
  {
    std::swap(m_resource, that.m_resource);
  }
};

// All of these call the constructor defined above.
foo f(foo());
foo f(construct_a_foo());
```

```
foo f(std::move(g));

// All of these call the assignment operator defined above.
f = foo();
f = construct_a_foo();
f = std::move(g);
```

The parameter of such functions is considered a temporary on the call site. They are actually not restricted to constructors or assignment operators, and can be used in any function.

The semantics of such arguments must be interpreted as "I *may* steal your data". The *may* is important here, as there is no guarantee on the call site that the instance going through an `std::move` is actually moved.

2.2.2 `std::move`, the Subtleties

Naming quality: ★☆☆☆☆. Would have put zero stars if I could.

As a great example of "naming things is hard", `std::move` does not move anything. It actually just marks the value as transferable, by casting it to an rvalue-reference. See this actual implementation exctracted from GCC 9:

```
template<typename _Tp>
constexpr typename std::remove_reference<_Tp>::type&&
move(_Tp&& __t) noexcept
{
  return static_cast
  <
    typename std::remove_reference<_Tp>::type&&
  >(__t);
}
```

> **Guideline**
>
> As a rule of thumb of its usage, consider two use cases:
>
> 1. If callee always takes ownership of the value: prefer arguments passed by value.
> 2. If callee may take ownership of the value: use an rvalue reference.

```
// "I need my own bar."
void foo_copy(bar b) { /* ... */ }

// "I may take ownership of b...
//  ...Unless I don't."
void foo_rvalue(bar&& b) { /* ... */ }

void baz()
{
  bar b1;
  // Valid, b1 is unchanged in baz.
```

```
  foo_copy(b1);

  // Valid, we can forget about b1 now.
  foo_copy(std::move(b1));

  bar b2;
  // Invalid, b2 is not an rvalue
  // foo_rvalue(b2);

  // Valid, but is b2 actually moved?
  foo_rvalue(std::move(b2));
}
```

Note that the rules above do not apply directly for functions that assign an existing resource, like a setter. In this case, always passing by value may be a pessimization if the caller does not pass an rvalue and the member assigned to releases its resource before taking the one from the argument.

```
// This example does not consider small string optimization.

class foo
{
  std::string name;

public:
  void set_name_value(std::string n)
  {
    // The internals of the name member are released, then the resources from
    // n are transfered.
    name = std::move(n);
  }

  void set_name_reference(const std::string& n)
  {
    // The value in n is copied in this->name. No allocation is done if there
    // is enough room for it.
    name = n;
  }
};

void test_foo()
{
  // Set up.
  foo f;
  f.set_name_value("abc");

  // Behind the following call:
  // 1. allocate a temporary std::string on the stack.
  // 2. allocate the internal buffer of the temporary in the heap.
  // 3. copy "def" in the internal buffer.
  // 4. swap f.name's internals with the ones of the temporary.
  // 5. release the temporary's internals (previously f.name's).
```

```
  f.set_name_value("def");

  // Behind the following call:
  // 1. allocate a temporary std::string on the stack.
  // 2. allocate the internal buffer of the temporary in the heap.
  // 3. copy "ghi" in the internal buffer.
  // 4. copy the temporary's internals into f.name.
  // 5. release the temporary's internals.
  f.set_name_reference("ghi");

  // Behind the following call:
  // 1. allocate a temporary std::string on the stack.
  // 2. allocate the internal buffer of the temporary in the heap.
  // 3. copy the internals of name in the temporary's internals.
  // 4. swap f.name's internals with the ones of the temporary.
  // 5. release the internals of the temporary (previously f.name's).
  f.set_name_value(name);

  // Behind the following call:
  // 1. check that f.name is large enough for name, realloc if needed.
  // 2. copy the internals of name in f.name.
  f.set_name_reference(name)
}
```

In all cases in the example above, it is better to pass by address. If the allocation of the temporaries is an issue, one can decide to override the setter to take an rvalue reference.

2.2.3 `std::forward` and Universal References

When `&&` is used in association with a type to be deduced, like in the function below:

```
template<typename T>
void foo(T&& arg)
{
  /* ... */
}
```

Then it refers neither to an rvalue-reference nor a reference. In this context, it is called a *universal reference*. Depending on how the function is called then `T&&` will be either deduced to an rvalue-reference, if the argument is an rvalue, or else the references are collapsed and `T&&` becomes an lvalue-reference, just like `T&`.

```
void bar()
{
  std::string s;

  // calls foo(std::string&)
  foo(s);

  // calls foo(std::string&&)
```

```
  foo(s + "abc");
}
```

Now how one should pass to another function an argument received as a universal reference? If it is deduced as an rvalue-reference, then `std::move()` should be used, otherwise it should be passed as is.

Fortunately C++11 provides `std::forward()` to do the check for us:

<utility>

```
template<typename T>
void foo(T&& arg)
{
  // Pass as an rvalue-reference or by address, whichever fits.
  foobar(std::forward<T>(arg));
}
```

2.3 Template-Related Features

2.3.1 Template Aliases

What was the problem

Before C++11, we could simply not declare a typedef with a parameterized type. For example, the following did not work:

```
// We cannot do that.
template<typename T>
typedef std::vector<T> collection;

// We cannot do that either.
template<typename V>
typedef std::map<std::string, V> string_map;
```

The alternative was to use a struct to receive the type, then declare the typedef inside the struct:

```
// Workaround: nest the type.
template<typename V>
struct string_map
{
  typedef std::map<std::string, V> type;
};

// Easy to use, isn't it?
string_map<int>::type string_to_int;
```

How the Problem is Solved

Starting with C++11, the `using` keywork allows to declare templated type aliases:

```
// We can do that.
template<typename T>
```

```
using collection = std::vector<T>;

// We can also do that.
template<typename V>
using string_map = std::map<std::string, V>;

// Looks like a first-class type.
string_map<int> string_to_int;
```

The using keyword is also a replacement for typedef in general.

2.3.2 External Templates

What was the problem

External templates are one of my favorite features from C++11.

When we were writing template code before C++11, for example a template function, then every time the code was included in a compilation unit the compiler would instantiate all used symbols. Check for example the header below:

```
// factorial-98.hpp
#pragma once

template<typename T>
T factorial(T n)
{
  T r = 1;

  for (T i = 1; i <= n; ++i)
    r += r * i;

  return r;
}
```

If this header was included in two .cpp files, and its function actually called, then its compiled code would be present in the object file of each .cpp; something we can check with `nm`.

Following the example, let's compile the two files below:

```
// foo-98.cpp
#include "factorial-98.hpp"

int foo(int i)
{
  return factorial(i);
}

// bar-98.cpp
#include "factorial-98.hpp"
```

```
int bar(int i)
{
  return factorial(i);
}
```

Then `nm` would print:

```
$ nm bar-98.cpp.o foo-98.cpp.o

bar-98.cpp.o:
000000000000001b T foo(int)
0000000000000036 W int factorial<int>(int)

foo-98.cpp.o:
000000000000001b T bar(int)
0000000000000036 W int factorial<int>(int)
```

Not only does this consume space (54 bytes per file here) but it also costs a lot of work for the compiler and the linker. All of this adds up, even for medium projects. Imagine that for each template function or class the compiler parses the code, then it compiles it, then writes all these bytes on disk; then all these bytes are read again by the linker, who sorts all these duplicate symbols to keep only one.

At the end of this process, literally all the work done by the compiler has been useless. Wouldn't it have been better not to do it in the first place? The linker then spent more time removing the crap rather than actually linking, and the programmer was wondering if he could get a more powerful computer. Again.

How the Problem is Solved

Thankfully the `extern template` from C++11 allows us to skip all this useless work. First we have to tell the compiler not to instantiate the template for a subset of valid types by adding a single line to our header:

```
// factorial-11.hpp
#pragma once

template<typename T>
T factorial(T n)
{
  T r = 1;

  for (T i = 1; i <= n; ++i)
    r += r * i;

  return r;
}

extern template int factorial<int>(int);
```

This line tells the compiler not to instantiate `factorial()` when `T = int`. Then we add a .cpp file where we explicitly ask for the instantiation:

```
// factorial-11.cpp
#include "factorial-11.hpp"

template int factorial<int>(int);
```

That's it. Let's check with nm:

```
$ nm bar-11.cpp.o foo-11.cpp.o factorial.cpp

bar-11.cpp.o:
000000000000001b T bar(int)

foo-11.cpp.o:
000000000000001b T foo(int)

factorial-11.cpp.o:
0000000000000036 W int factorial<int>(int)
```

The template function is indeed compiled in a single file, and absent from the others[2].

> **Guideline**
>
> If you are writing template code for which you know some or all instantiations, then add an `extern template` line for them in the header, and explicitly instantiate them in an implementation file. This is not always feasible, but when it can be done it should be done

2.3.3 Variadic Templates

What was the problem

Let's design a some kind of message dispatcher using features available before C++11. The use case is presented by the code below.

```
// Some functions that will be called by our dispatcher
void on_event();
void also_on_event();
void on_message(const char* message);
void also_on_message(const char* message);
void on_long_message(const char* message_1, const char* message_2);
void also_on_long_message(const char* message_1, const char* message_2);

int main()
{
  // A dispatcher with no arguments and two scheduled functions.
  dispatcher<> zero;
  zero.queue(&on_event);
  zero.queue(&also_on_event);
```

[2] We can push even further in this case, by having the whole body of `factorial()` in the .cpp file and only its signature in the header. It would not only avoid the parsing of the code but, more importantly, allow to remove from the header all include directives related to implementation details. Note that in this case the extern keyword can even be omitted.

```
  // A dispatcher with one argument and two scheduled functions.
  dispatcher<const char*> one;
  one.queue(&on_message);
  one.queue(&also_on_message);

  // A dispatcher with two argument and two scheduled functions.
  dispatcher<const char*, const char*> two;
  two.queue(&on_long_message);
  two.queue(&also_on_long_message);

  // Now we trigger the calls with the given arguments.
  zero.dispatch();
  one.dispatch("hello");
  two.dispatch("hello", "world");

  return 0;
}
```

The idea is to have a `dispatcher` whose role is to store functions to be called with arguments provided later. It is parameterized by the type of the arguments. If we want to accept any kind of function, we must accept from zero to n arguments.

A typical solution for that would have been to declare the type with multiple template parameters and to default these parameters to a type representing the state of not being used. Then the actual implementation would have been selected by template specialization of a super type. In order to keep this example short we are going to limit our implementation to functions with up to two arguments.

Here is the dispatcher with the defaulted parameters.

```
template<typename Arg1 = void, typename Arg2 = void>
struct dispatcher: dispatcher_impl<Arg1, Arg2>
{};
```

Let's have a look at the dispatcher implementation with two arguments.

```
template<typename Arg1, typename Arg2>
struct dispatcher_impl
{
  typedef void (*function_type)(Arg1, Arg2);

  void queue(function_type f)
  {
    m_scheduled.push_back(f);
  }

  template<typename A1, typename A2>
  void dispatch(A1 a1, A2 a2)
  {
    const std::size_t count = m_scheduled.size();

    for (std::size_t i(0); i != count; ++i)
```

```
      m_scheduled[i](a1, a2);
  }

private:
  // The functions that will be called on the next dispatch.
  std::vector<function_type> m_scheduled;
};
```

Nothing special here. Let's look at the implementation with one argument.

```
template<typename Arg>
struct dispatcher_impl<Arg, void>
{
  typedef void (*function_type)(Arg);

  void queue(function_type f)
  {
    m_scheduled.push_back(f);
  }

  template<typename A>
  void dispatch(A a)
  {
    const std::size_t count = m_scheduled.size();

    for (std::size_t i(0); i != count; ++i)
      m_scheduled[i](a);
  }

private:
  // The functions that will be called on the next dispatch.
  std::vector<function_type> m_scheduled;
};
```

This is very similar to the previous implementation. We could factorize some parts (`queue()` and `m_scheduled`) by moving them in a parent class parameterized with `function_type`, but for now let's just keep it as is.

The `dispatch()` function cannot be factorized though, as its signature and the `m_scheduled[i](/*args*/)` line is specific to each version.

The implementation with no arguments is equivalently similar, so I won't include it here. I think you got the point: this is a lot of code, redundant code, for a feature limited to only three use cases amongst many. Wouldn't it be better if we could put all of that in a single implementation that would handle any number of arguments?

How the Problem is Solved

The variadic template syntax introduced in C++11 provides a solution to this problem.

```
// The template accepts any number of arguments.
```

```
template<typename... Args>
struct dispatcher
{
  // We can list the template parameters by appending '...'.
  typedef void (*function_type)(Args...);

  void queue(function_type f)
  {
    m_scheduled.push_back(f);
  }

  // args represent any sequence of arguments here.
  template<typename... A>
  void dispatch(A&&... args)
  {
    const std::size_t count = m_scheduled.size();

    // Then we can list the arguments by appending '...'
    // to the function argument.
    for (std::size_t i(0); i != count; ++i)
      m_scheduled[i](args...);
  }

private:
  std::vector<function_type> m_scheduled;
};
```

Here we have a single implementation accepting any number of arguments, and thus covering all use cases from the previous implementation and more.

The syntax of `typename... Args` defines a *parameter pack*. This is like a template parameter but with an ellipsis.

A template with a parameter pack is called a *variadic template*. Note that it is allowed to mix a parameter pack with non-pack template arguments, as in `template<typename Head, typename... Tail>`.

When an expression containing a parameter pack ends with an ellipsis, it is replaced by a comma-separated repetition of the same expression applied to each type from the pack. For example:

```
template<typename F, typename... Args)
void invoke(F&& f, Args&&... args)
{
  // this is equivalent to
  //   f(std::forward<Args0>(args_0),
  //     std::forward<Args1>(args_1),
  //     etc);
  f(std::forward<Args>(args)...);
}
```

Or, for another example, here is a function creating an array of the names of the types passed as template parameters:

```
template<typename... T>
void collect_type_names()
{
  const char* names[] = {typeid(T).name()...};
  // ...
}
```

2.4 Objects and Classes

2.4.1 Delegated Constructors

What was the problem

Before C++11, constructors could not call other constructors. Consider the example below:

```
struct foo
{
  foo(bar* b, float f, int i)
    : m_bar(b),
      m_f(f),
      m_i(i)
  {}

  foo(bar* b, int i)
    // can't I call foo(b, 0, i) directly?
    : m_bar(b),
      m_f(0),
      m_i(i)
    {}

private:
  bar* const m_bar;
  const float m_f
  const int m_i;
};
```

If we want to share initialization code between multiple constructors, then we have to put it in some separate member function called by the constructor, like this:

```
struct foo
{
  foo(bar* b, float f, int i)
  {
    init(b, f, i);
  }

  foo(bar* b, int i)
```

```
  {
    init(b, 0, i);
  }

private:
  void init(bar* b, float f, int i)
  {
    m_bar = b;
    m_f = f;
    m_i = i;
  }

  // We cannot make any of these members const anymore.
  bar* m_bar;
  float m_f;
  int m_i;
};
```

This approach was kind of error-prone. Stuff may happen before and after the call to the `init()` function, and actually nothing prevents it to be called at any point in the life of the instance. Finally, this is incompatible with `const` members.

How the Problem is Solved

Starting from C++11, a constructor can call another constructor:

```
struct foo
{
  foo(bar* b, float f, int i)
    : m_bar(b),
      m_f(f),
      m_i(i)
  {}

  foo(bar* b, int i)
    : foo(b, 0, i)
  {}

private:
  bar* const m_bar;
  float m_f;
  int m_i;
};
```

This solves all problems.

2.4.2 Deleted Constructors

What was the problem

Delegating constructors is a nice feature, but wat about totally removing a constructor?

Consider the class below:

```
struct scoped_listener
{
  scoped_listener(dispatcher& d, callback c)
    : m_dispatcher(&d)
  {
    m_id = m_dispatcher->connect(c);
  }

  ~scoped_listener()
  {
    m_dispatcher->disconnect(m_id)
  }

private:
  dispatcher* m_dispatcher;
  int m_id;
};
```

Creating copies of `scoped_listener` does not make any sense, as all copies would share the same `m_id` and will thus trigger the same call to `disconnect(int)` when destructed. See for example its usage below:

```
void foo()
{
  /* ... */
  scoped_listener listener(d, c1);
  scoped_listener copy(listener);
  scoped_listener other(d, c2);

  {
    // This assignment does not call disconnect(c2).
    other = listener;
    // disconnect(c1) is called here,
    // when other goes out of scope.
  }
  // disconnect(c1) is called twice here: in the
  // destruction of listener and copy.
}
```

One would typically want to forbid copies of `scoped_listener` by disabling its copy constructor.

Before C++11, one solution we would find here and there was to declare the copy constructor and assignment operator as private. The problem was that it was still available for the class and its friends. So the programmer would then either implement an always failing body for this constructor, emitting an error at run time, or would just not implement the constructor, thus triggering an error at link time.

These solutions were kind of weak, in the sense that the error, if any, was presented quite late for the programmer, and with a not obvious explanation.

How the Problem is Solved

Now, starting with C++11, the constructor and operators can be explicitly deleted:

```
struct scoped_listener
{
  scoped_listener(const scoped_listener&) = delete;
  scoped_listener& operator=(const scoped_listener&) = delete;

  scoped_listener(dispatcher& d, callback c)
    : m_dispatcher(&d)
  {
    m_id = m_dispatcher->connect(c);
  }

  ~scoped_listener()
  {
    m_dispatcher->disconnect(m_id)
  }

private:
  dispatcher* m_dispatcher;
  int m_id;
};
```

Using a deleted function will trigger a clear error from the compiler when the call is encountered.

2.4.3 Defaulted Constructors

What was the problem

Let's continue with constructors. Before C++11, each class would have implicit constructors unless stated otherwise. One example of a situation where an implicit constructor would not have been created was the explicit declaration of a custom constructor by the programmer. For example:

```
struct foo
{
  foo(int) {}
};

int main()
{
  foo f1;      // fail
  foo f2(24); // ok
  foo f3(f2); // ok

  return 0;
}
```

In the above example, `foo` has no default constructor (but has an implicit copy constructor). In order to have the default constructor, the programmer had to implement one. The main problem becomes mainte-

nance: when new fields are added in the class, we have to remember to update the constructor to initialize them.

How the Problem is Solved

Starting with C++11, the programmer can tell the compiler to implement the constructor with what would have been the default implementation if it was not deleted.

```
struct foo
{
  foo() = default;
  foo(int) {}
};

int main()
{
  foo f1;      // ok
  foo f2(24); // ok
  foo f3(f2); // ok

  return 0;
}
```

2.4.4 Override and Final

What was the problem

So we have a class in our C++-98 code base that looks like that:

```
struct abstract_concrete_base
{
  virtual ~abstract_concrete_base();
  virtual void consume();
};
```

Down the hierarchy we have some class that overrides the `consume()` function[3]:

```
struct serious_business : abstract_concrete_base
{
  void consume();
};
```

Suddenly, the base class has some work to do in `consume()` regardless of the details of the derived classes. Also, it is well known that one should prefer non-virtual public functions calling virtual private functions, so let's update our code:

```
struct abstract_concrete_base
{
```

[3] You may also find versions of this pattern in the wild where the `virtual` keyword is repeated in the derived classes, so the reader can suppose that it is an overridden function without looking at the declaration of the parent class.

```
    virtual ~abstract_concrete_base();

    void consume()
    {
      stuff_from_base_class();
      do_consume();
    }

private:
    void stuff_from_base_class();
    virtual void do_consume();
};
```

Nice. Everything compiles well but all derived classes are broken. It turns out that since `do_consume()` is not a pure virtual function, there is no problem to instantiate the class. All derived will use the default `do_consume()` and their old `consume()` is now just another function of theirs.

Wouldn't it be nice if there was a way to detect the problem?

How the Problem is Solved

That's why the `override` keyword has been introduced in C++11. By modifying the initial implementation of `serious_business` as follows:

```
struct serious_business : abstract_concrete_base
{
  void consume() override;
};
```

The programmer explicitly tells the compiler that this function is expected to override the same function from a parent class. So when `abstract_concrete_base` is updated and its `consume()` function becomes non-virtual, the compiler will immediately complain that the function from `serious_business` does not override anything. Good!

Another keyword related to inheritance and function overriding has been introduced in C++11: the `final` keyword. When used in place of `override`, it tells the compiler that the function should not be overridden in the derived classes. Previously a virtual function could be overridden anywhere from the top to the bottom of the hierarchy. Now if one declaration is marked as final, the derived classes are not allowed to redefine the function anymore.

Additionally, one can place the `final` keyword after the name of a class to indicate that this class cannot be derived at all:

```
struct serious_business final : abstract_concrete_base
{
  void consume() override;
};
```

2.4.5 Using parent functions

What was the problem

Here is an actual problem I encountered when developing mobile games in C++. When targeting Android devices, the application has to do some calls from the C++ part to the Java part. These calls are done via the Java Native Interface (JNI), which is accessed via an object of type `JNIEnv`.

Calling a Java method from Java requires the identifier of the class, of type `jclass`, and the identifier of the method, of type `jmethodID`. When all of them are known, we can call for example a static method returning an integer as follows:

```
JNIenv* env = /* ... */;
jclass the_class = /* ... */;
jmethodID the_method = /* ... */;

const jint result =
  env->CallStaticIntMethod(the_class, the_method, /* arguments */);
```

I certainly did not want to write this stuff everywhere and was hoping for something more like:

```
// This variable represents the function to call.
static_method<jint> method = /* ... */;

// Then we call it like a function.
const jint result = method(/* arguments */);
```

Since all method calls need the same parameters (i.e. the JNI environment, the class, and the method), I put them in a base class from which specialization for the returned type would be created[4]:

```
class static_method_base
{
public:
  static_method_base
  ( JNIEnv* env, jclass class_id, jmethodID method_id );

protected:
  JNIEnv* m_env;
  jclass m_class;
  jmethodID m_method;
};
```

Then I would inherit from this class to create a function object for each supported return type. Here for a static method returning an integer:

```
template< typename R >
class static_method;

template<>
class static_method<jint>:
```

[4]This is simplified for the example. Check https://github.com/IsCoolEntertainment/iscool-core/ for the actual code.

```
  public static_method_base
{
public:
  // This constructor seems useless, doesn't it?
  static_method_base
  (JNIEnv* env, jclass class_id, jmethodID method_id)
    : static_method_base(env, class_id, method_id)
  {}

  template<typename... Arg>
  jint operator()(Arg&&... args) const;
};
```

At this point we have something weird: the only reason we have defined a constructor in `static_method<jint>` is to be pass its argument to the parent class. Wouldn't it be better if we could just say "reuse the constructor from the parent"?

How the Problem is Solved

This is possible thanks to the `using` keyword from C++11:

```
template< typename R >
class static_method;

template<>
class static_method<jint>:
  public static_method_base
{
public:
  // This is concise and precise.
  using static_method_base::static_method_base;

  template<typename... Arg>
  jint operator()(Arg&&... args) const;
};
```

When used like this, the `using` keyword imports the function into the current class. It is also a way to solve the problem of parent functions hidden by overloads in the child class:

```
struct base
{
  void foo();
};

struct derived: base
{
  // This declaration hides base::foo().
  void foo(int);
};

void bar()
```

```
{
  derived d;
  // This fails, derived::foo requires an int argument.
  d.foo();
}
```

With the `using` keyword:

```
struct base
{
  void foo();
};

struct derived: base
{
  using base::foo;
  void foo(int);
};

void bar()
{
  derived d;
  // This is ok, foo is a member function in derived.
  d.foo();
}
```

2.5 In the Standard Library

2.5.1 Beginning and End of Sequence

What was the problem

All containers from the STL have a `begin()` and `end()` member function to get an iterator on the first element in the sequence or, respectively, just after the last element. Unfortunately, there is no such function for the most basic sequences, i.e. C-like arrays, so code like that was sure to fail before C++11:

```
template<typename Sequence, typename T>
void replace_existing
(Sequence& s, const T& old_value, const T& new_value)
{
  *std::find(s.begin(), s.end(), old_value) = new_value;
}

void foo()
{
  int a[] = { 1, 4, 3 };
  replace_existing(a, 4, 2);
}
```

How the Problem is Solved

<iterator>

Fortunately, C++11 introduces the `std::begin()` and `std::end()` free functions which accepts a C-like array[5]. Now, this code will work in all cases:

```
template<typename Sequence, typename T>
void replace_existing
(Sequence& s, const T& old_value, const T& new_value)
{
  *std::find(std::begin(s), std::end(s), old_value) = new_value;
}

void foo()
{
  int a[] = { 1, 4, 3 };
  replace_existing(a, 4, 2);
}
```

2.5.2 Iterator Successors and Predecessors

What was the problem

There are many kind of iterators: forward iterators, that can be incremented one step at a time with `operator++`, bidirectional iterators, that can additionally be decremented with `operator--`, and random access iterators, that can be incremented or decremented by any amount at once.

When writing a function taking an iterator whose type is templated, incrementing an iterator by more than one unit cannot be done directly, as it would fail for non-random iterators. Prior to C++11, this was done with `std::advance()`.

```
template<typename Iterator, typename F>
void every_n_items
(Iterator it, std::size_t count, std::size_t step, F f)
{
  for (; count > step; count -= step)
  {
    f(*it);
    std::advance(it, step);
  }
}
```

`std::advance()` accepts a negative distance, in which case it will advance... hum... backwards. Note that the function modifies its argument, so using it to get another iterator from a given one is cumbersome:

```
template<typename Iterator>
void inplace_adjacent_sums(Iterator first, const Iterator& last)
{
  if (first == last)
```

[5] as long as we include <array>

```
    return;

  // Two steps to get an iterator on the second element. Ideally we
  // would have wanted to limit its scope to the loop too.
  Iterator second = first;
  std::advance(second, 1);

  for (; second != last; )
  {
    *first += *second;
    first = second;
    std::advance(second, 1);
  }
}
```

How the Problem is Solved

<iterator>

C++11 introduces the replacement functions `std::prev()` and `std::next()`, which are mostly here to make things clear and more convenient. The former is to be used to move the iterator backwards, while the latter is used to move it forward. If no distance is passed, then it defaults to one.

Note that they return a copy of the new iterator, instead of modifying its argument.

```
template<typename Iterator>
void inplace_adjacent_sums(Iterator first, const Iterator& last)
{
  if (first == last)
    return;

  // Here we can initialize the second iterator in the loop, reducing
  // its scope.
  for (Iterator second = std::next(first); second != last; )
  {
    *first += *second;
    first = second;
    std::advance(second, 1);
  }
}
```

2.5.3 Arrays

What was the problem

How do we usually pass a raw array to a function in C++? By passing both a pointer to the first element and the number of elements, that is how.

And when we need to copy such array, or to return such array from a function, it becomes quite tricky.

```
void replace_zeros_copy(int* out, int* in, std::size_t size, int r)
{
  // Can I trust the caller that the target array is large enough to
```

```
  // store the result? Is it a valid use case to pass null pointers
  // here? So many questions :(

  std::transform
    (in, in + size, out,
     [r](int v) -> int
     {
       return (v == 0) ? r : v;
     });
}

// We certainly cannot properly return a raw array unless we use
// some dynamic allocation or other techniques with their own
// problems.
int* replace_zeros_copy(int* array, std::size_t size, int r) { /* ... */ }
```

How the Problem is Solved

With C++11 came `std::array` a template type to be used as an alternative to raw arrays, with two parameters: the type of the elements and their count.

<array>

```
template<std::size_t N>
void replace_zeros_copy
(std::array<int, N>& out, const std::array<int, N>& in, int r)
{
  // Forget about the null pointers problem and the size issues,
  // everything fits perfectly here.

  std::transform
    (in.begin(), in.end(), out.begin(),
     [r](int v) -> int
     {
       return (v == 0) ? r : v;
     });
}

// We can also directly return the array.
template<std::size_t N>
std::array<int, N> replace_zeros_copy(const std::array<int, N>& array, int r)
{
  std::array<int, N> result;
  replace_zeros_copy(result, array, r);
  return result;
}
```

An `std::array` is as cheap as a raw array. It has no fancy constructor or any subtleties, and I can think of only two downsides to using it:

1. the cost of including an extra header [Tre20] [Dou20], not negligible in this case as <array> pulls <utility> and more,

2. the need to carry the size as a template parameter.

> **Guideline**
>
> Because of the downsides associated with the type and its header, using `std::array` should always be preceded by two considerations:
>
> 1. Will the header propagate to many files?
> 2. Will I have to templatize everything?
> 3. Should I implement iterator-based algorithms instead?

Also, note that `std::array` is not a fixed-capacity vector. All its entries are default initialized as soon as the array is constructed. So if it contains complex types, it means that the default constructor of this type is called for each entry.

2.5.4 Random

What was the problem

Getting random numbers in C++ was historically done by using functions from the C library. First we initialized the a global generator via a call to `srand()` then the values were generated via `rand()`.

Unfortunately this generator was known for being a quite poor one.

How the Problem is Solved

<random>

This has been greatly improved in C++11. For the initialization, we now have `std::random_device` to access a hardware-based generator[6], then there are many pseudo-random number generators, `std::mt19937` being a quite popular one.

```
int main()
{
  // The random device will get a random value from the system. It is
  // certainly the best random source we can have here.
  std::random_device seed;

  // This is a pseudo-random number generator, here initialized by
  // a value obtained from the random device.
  std::mt19937 generator(seed());

  // Finally this distribution will project the random values in the
  // [1, 24] range with equiprobability for each value.
  std::uniform_int_distribution<int> range(1, 10);

  for (int i = 0; i != 1000; ++i)
    printf("%d\n", range(generator));

  return 0;
}
```

[6] If there is one, otherwise it would be software-based.

Unfortunately the generators available in the standard library are known to be both inefficient and not very good at randomness. As an alternative, we can check for a permuted congruential generator (PCG) [O'N14], at https://www.pcg-random.org/.

2.5.5 Smart Pointers

Memory allocation in C++ is a tough subject, dynamic allocation being the hardest part.

Before C++11, the programmer had to be very careful with the life span of dynamically allocated memory in order to, first, be sure that it is released at some point and, second, that no access is made to it once it has been released, not even another release.

See how many problems could occur with this short snippet:

```
struct foo
{
  foo(int* p)
    : m_p(p)
  {}

  ~foo()
  {
    delete m_p;
  }

private:
  int* m_p;
};

void bar()
{
  foo f1(new int);
  foo f2(f1);

  int* p1(new int);
}
```

There are two problems with this code. First, `foo` does not define nor disable its copy constructor, so, by default, its `m_p` pointer will be copied to the new instance. Then, when the original instance and its copy will be destroyed, both will call `delete` on the same pointer. This is what will happen with `f1` and its copy `f2`. In the best scenario the program would crash here.

The second problem is `p1`. This pointer points to a dynamically allocated `int` for which no `delete` is written. When the pointer will go out of scope then there will be no way to release the allocated memory.

Self-Deleting Non-Shared Pointer

C++11 introduces a pointer wrapper named `std::unique_ptr`, which has the merit of automatically calling `delete` on the pointer upon destruction. Applied to the previous example, it would solve one problem and force us to find a solution for the other:

<memory>

```
struct foo
{
  foo(std::unique_ptr<int> p)
    : m_p(std::move(p))
  {}

  // The default destructor does the job.

private:
  std::unique_ptr<int> m_p;
};

void bar()
{
  std::unique_ptr<int> p1(new int);

  foo f1(std::unique_ptr<int>(new int));
  foo f2(std::move(f1));
}
```

This program is undoubtedly safer than the previous one. The copy constructor of `foo` is still not defined, but it is for sure deleted since `std::unique_ptr` has no copy constructor neither. So, by default, we cannot share the resource between two instances, which is great. The only solution here is to either allocate a new int for `f2` or steal the one from f1. The latter is implemented here.

Then, for the release of `p1`, it is automatically done when the variable leaves the scope, so no memory is leaked.

One of the best features of `std::unique_ptr` is the possibility to use a custom deleter to release the pointer. This makes this smart pointer a tool of choice when using C-like resources.

Check for example the use case of libavcodec's `AVFormatContext`. The format context is obtained via a call to `avformat_open_input(AVFormatContext**, const char*, AVInputFormat*, AVDictionary**)` and must be released by a call to `avformat_close_input(AVFormatContext**)`. With the help of `std::unique_ptr` this could be done as follows:

```
namespace detail
{
  static void close_format_context(AVFormatContext* context);
}

void foo(const char* path)
{
  AVFormatContext* raw_context_pointer(nullptr);

  const int open_result =
    avformat_open_input
      (&raw_context_pointer, path, nullptr, nullptr);

  std::unique_ptr
```

```
    <
      AVFormatContext,
      decltype(&detail::close_format_context)
    >
    context_pointer
    (raw_context_pointer, &detail::close_format_context);

    // ...
  }
```

With this approach, the format context will be released via a call to `detail::close_format_context` as soon as `context_pointer` leaves the scope. Do we know if the memory pointed by `raw_context_pointer` is dynamically allocated? We do not, and it does not matter. What we have here is a simple robust way to attach a release function to an acquired resource.

Self-Deleting Shared Pointer

<memory>

I have a hard time trying to find a use case where `std::shared_ptr` is the best solution, so let's just focus on a good-enough solution.

This smart pointer is the answer to the problem of having a dynamically allocated resource that may outlive its creator, and that may also be accessed from two independent owners. In this case, the ownership of the resource is unclear, as it is *shared*, so the idea is to keep the resource alive until all its owner release it.

For example, let's say we have a function whose role is to dispatch a message to multiple listeners, whom will not process it right now. For some reason the message cannot be copied, so we cannot send a copy of it to everyone:

```
void dispatch_message
(const std::vector<listener*>& listeners,
 const std::string& raw)
{
  message m = parse_message(raw);
  std::shared_ptr<message> p
    (std::make_shared<message>(std::move(m)));

  for(listener* l : listeners)
    listener->add_to_queue(m);
}
```

With this implementation the message outlives the scope of `dispatch_message()` and will remain in memory until all listeners have released it.

Is it the best solution for this problem? Honestly I doubt that. Actually, most uses of `std::shared_ptr` I have seen, including mine, looked a bit like a lack of reasoning about resource management.

Wouldn't it have been better if the dispatching and the listeners were scheduled in a loop and if the dispatcher would own the messages for some iterations, or until they would be marked as processed by the listeners? Do we really have to pay for dynamic allocations here?

Guideline

When you find yourself thinking that a shared resource with no clear life span — a `std::shared_ptr` — is a good answer to your problem, please double check your solution, and ask for a second opinion.

It's a trap

Many documentations declare, rightfully, that `std::shared_ptr` is thread-safe, and I have seen people using it to *safely* access the allocated resource in a concurrent way.

It must be said that the only thread-safe part in a `std::shared_ptr` is its reference counter.

Nothing is done — nothing can be done — to provide an automatic thread-safe access to the allocated resource. Actually, even having two threads doing respectively a copy (i.e. `std::shared_ptr<T> p(sp)`) and a deletion (i.e. `sp.release()`), for the same shared pointer `sp`, has no defined outcome without additional synchronization. The only guarantee is that the increment of the counter in the copy won't be done between the decrement and the deletion from the release.

Shared Pointer Observer

<memory>

What happens when the instance pointed by a `std::shared_ptr` owns a `std::shared_ptr` pointing to the owner of the former? Then we have a cycle and none of the instances will be released.

```
struct foo;

struct bar
{
  std::shared_ptr<foo> m_foo;
};

struct foo
{
  std::shared_ptr<bar> m_bar;
};

void foobar()
{
  std::shared_ptr<foo> f(std::make_shared<foo>());
  std::shared_ptr<bar> b(std::make_shared<bar>());
  b->m_foo = f;
  f->m_bar = b;

  // The instances pointed by foo and foo->bar won't be deleted.
}
```

In order to break this kind of dependency loop, one pointer should be declared as an `std::weak_ptr`. Just like `std::shared_ptr`, this smart pointer is here to point to a shared resource, except that it does not count as an owner of the resource. Additionally, it provides a way to test if the resource is still available via

the `std::weak_ptr::lock()` function, which returns a shared pointer on the resource if it is available, or `nullptr` otherwise.

```
struct foo;

struct bar
{
  std::weak_ptr<foo> m_foo;
};

struct foo
{
  std::shared_ptr<bar> m_bar;
};

void foobar()
{
  std::shared_ptr<foo> f(std::make_shared<foo>());
  std::shared_ptr<bar> b(std::make_shared<bar>());
  b->m_foo = f;
  f->m_bar = b;

  {
    std::shared_ptr<foo> resource(b->m_foo.lock());
    if (resource)
      printf("This message is printed.\n");
  }

  f.reset();

  std::shared_ptr<foo> resource(b->m_foo.lock());

  if (resource)
    printf("This message is not.\n");
}
```

2.5.6 Initializer list

What was the problem

Initializing a container like `std::vector` with a predefined sequence of values was quite verbose before C++11, as we had to insert the values one by one:

```
void transform
(std::vector<int>& values, const std::vector<int>& multipliers);

void up_down_transform(std::vector<int>& values)
{
  // Pfff it is not even const.
  std::vector<int> pattern(4);
  pattern[0] = 0;
```

```
  pattern[1] = 1;
  pattern[2] = 0;
  pattern[3] = -1;

  // Can't we have simple initialization, like we have for arrays?
  // const int pattern[4] = {0, 1, 0, -1};

  transform(values, pattern);
}
```

How the Problem is Solved

<initializer_list>

However, thanks to the the introduction of `std::initializer_list` in C++11, we can now initialize our containers wich bracket enclosed values:

```
void transform
(std::vector<int>& values, const std::vector<int>& multipliers);

void up_down_transform(std::vector<int>& values)
{
  const std::vector<int> pattern = {0, 1, 0, -1};
  transform(values, pattern);

  // This one also works:
  // transform(values, {0, 1, 0, -1});
}
```

It's just like uniform initialization [2.1.5]! Or is it?

The Problem with `std::initializer_list`

Instances of `std::initializer_list` are created by the compiler when it encounters a list of values between brackets and if the target to which these values are assigned is, or can be constructed from, an `std::initializer_list`.

In the example above, we can create a vector from the values because `std::vector` defines a constructor taking an `std::initializer_list` as it sole argument. This constructor then copies the values from the list into the vector[7].

I think it is important to emphasize that: the constructor *copies* the values from the list. There is no way it can move them, let alone wrap them.

So we have something that looks surreptitiously like the good old simple and efficient aggregate initialization, that is consequently identical in its syntax to uniform initialization [2.1.5], and that is actually a quite inefficient way to initialize something as soon as the element type is non trivial.

The worse part is that bracket initialization is pushed by the so-called "modern" C++ trend, propagating this inefficiency everywhere.

[7] Check [Bri20] for an illustration of the problem.

Guideline

Avoid `std::initializer_list`.

If you really, really, want the target container to be const, then use an immediately invoked lambda to construct it:

```
const std::vector<int> pattern =
  []() -> std::vector<int>
  {
    std::vector<int> result(4);
    result[0] = 0;
    result[1] = 1;
    result[2] = 0;
    result[3] = -1;

    return result;
  }(); // Watch out for the parentheses here,
       // it is a call to the lambda.
```

2.5.7 Threads

What was the problem

Before C++11 there was just nothing in the standard library to execute a thread, so we had to either go native with pthreads, Windows threads, or whatever fit our need, or else we could use an independent library that would do the system abstraction for us, like Boost.Thread.

How the Problem is Solved

Thankfully, C++11 came with `std::thread`, which does the system abstraction for us and allows us to write portable threaded code[8]. Launching a thread is as simple as this:

<thread>

```
auto expensive_computation =
  []() -> void
  {
    // This infinite loop takes forever to complete! It slows down
    // our awesome app, better put it in a thread.
    while (true);
  };

std::thread t(expensive_computation);
// Here we go, the thread is running.
```

With the threads come the mutexes, condition variables, and more, that will help us write synchronization points. Here is a more complete example with two threads accessing shared data:

<mutex>,
<condition_variable>

```
#include <cstdio>
#include <mutex>
```

[8] When the targeted platform supports it. I'm looking at you WebAssembly!

```
#include <thread>

struct flood_state
{
  int counter;
  std::mutex mutex;
};

int main()
{
  flood_state state;
  state.counter = 0;

  // std::thread's constructor takes the function to be executed in the thread
  // as its argument. Here we use a lambda [2.1.7].
  std::thread increase
    ([&state]() -> void
     {
       while (true)
         {
           // Wait until the mutex is available and lock it.  std::unique_lock
           // automatically unlocks it when going out of scope. Good old RAII.
           const std::unique_lock<std::mutex> lock(state.mutex);
           ++state.counter;
         }
     });
  std::thread decrease
    ([&state]() -> void
     {
       while (true)
         {
           int counter;

           {
             // We limit the scope of the lock to the access to the shared
             // state.
             std::unique_lock<std::mutex> lock(state.mutex);
             counter = state.counter;
             state.counter -= 2;
           }

           // This is executed after the release of the mutex, so the state can
           // be accessed while printing.
           printf("%d\n", counter);
         }
     });

  // std::thread::join() waits until the thread is over (which will not happen
  // in our case as nothing breaks outside the infinite loops.
  increase.join();
```

```
  decrease.join();

  return 0;
}
```

2.5.8 Tuple

What was the problem

The type `std::pair` is a nice utility class available in the STL since forever. It is a struct accepting two template parameters and defining two fields of these types, named "first" and "second". Approximately like:

```
template<typename T1, typename T2>
struct pair
{
  T1 first;
  T2 second;
};
```

It is for example the value type of associative containers like `std::map`, and now we are stuck with map entries with fields named "first" and "second" while something like "key" and "value" would have carried the semantics better. Meh. Well, it may not be the most expressive but at least we are reusing code via this hyper generic reusable type, hooray!

Sorry, I inadvertently switched the rant-mode button on[9].

How the Problem is Solved

Generalizing this structure to any number of fields/types was quite complex, as before the arrival of variadic templates [2.3.3] in C++11 we had to go through some type lists and other metaprogramming dances.

<tuple>

Thanks to their introduction, generalizing `std::pair` to any number of elements is now somewhat simpler, and it is already done in the STL via `std::tuple`. Additionally, `std::make_tuple()` is a utility function that can create a tuple from its values, deducing the actual tuple type from its arguments. Finally, the main companion function to `std::tuple` is `std::get()`, which allows to access a tuple element by its index.

```
// Creating a tuple explicitly.
std::tuple<int, float, bool> t(24, 42.f, true);

// Creating a tuple from its values.
t = std::make_tuple(42, 24.f, false);

// Accessing the elements in a tuple.
printf("%f\n", std::get<1>(t));
std::get<2>(t) = false;
```

[9] That being said, I actually love the fact that even for small types like that effort is made by C++ developers to build more efficient implementations. Search for *compressed pair* or *tight pair* for example.

Additionally, `std::tie()` is a helper function which creates a tuple whose elements are references to the arguments passed to the function.

```
// Swap the first mismatching elements from two vectors.
void swap_first_mismatch(std::vector<int>& v1, std::vector<int>& v2)
{
  using iterator = std::vector<int>::iterator;

  iterator it_v1;
  iterator it_v2;

  // std::mismatch() returns a pair of iterators. We assign this pair
  // to a tuple whose elements are references to the two variables
  // declared above, meaning that we actually assign the variables.
  std::tie(it_v1, it_v2) = std::mismatch(v1.begin(), v1.end(), v2.begin());

  std::swap(*it_v1, *it_v2);
}
```

Declaring a tuple in a C++ program typically happen for few reasons:

- By laziness where a struct could have been used[10],
- To return multiple values from a function: `std::tuple<int, int> get_dimensions()`[11].
- To lure Python developers into writing C++[12].
- To group types, to carry type lists, in metaprogramming context.

> **Guideline**
>
> If you ever end up in a situation where `std::tuple` seems to be the best type for an aggregate value, please reconsider.
>
> Naming things is hard, but having a named struct will be better to carry the meaning to the next readers than presenting them a bunch of data thrown in a bag.

It is thus quite difficult to find a good short example for `std::tuple`. One good example could be argument binding, but it's a quite long example that needs many extra features. Another example is found in one constructor of `std::pair`, which allows to directly pass the arguments to use to construct its fields:

```
template<typename T, typename U>
struct pair
{
  // Simplified for the example.
  template<typename... FirstArgs, typename... SecondArgs>
  pair
```

[10] Don't do that.
[11] Don't do that either, just write a meaningful type.
[12] But... why should we lure them if it's not their kind of stuff?

```
(std::tuple<FirstArgs...> first_args, std::tuple<SecondArgs...> second_args)
  : first(/* here we pass the content of first_args */),
    second(/* here we pass the content of second_args */)
{}
```

Yet another example is a metaprogramming use case where multiple parameter packs must be passed to a type or function:

```
// This won't work since the compiler cannot tell where to split the
// As and Bs
template<typename... As, typename... Bs>
struct failing_multi_pack
{};

// The code below will work though.

// This one is just the base template declaration, not defined.
template<typename As, typename Bs>
struct working_multi_pack;

// And we can specialize it for tuples. Now the compiler can split As
// and Bs.
template<typename... As, typename... Bs>
struct working_multi_pack<std::tuple<As>, std::tuple<Bs>>
{};
```

Well, I can see that you are a bit disappointed. Let's have a look at the binding stuff. Hold on, it is not simple (still incomplete however):

```
#include <tuple>
#include <cstdio>

// This structure is here to "store" a sequence of integers in its template
// parameters. There is nothing like that in C++11 but it is already available
// in C++14 3.2.2.
template<unsigned... I>
struct integer_sequence {};

// This one is used to create a sequence of N consecutive integers, from 0 to
// N-1, given N.
//
// Remaining is the number of integers that still have to be generated.
// Computed is the sequence we have computed so far.
template<unsigned Remaining, unsigned... Computed>
struct make_integer_sequence;

template<unsigned... Computed>
struct make_integer_sequence<0, Computed...>
{
  using type = integer_sequence<Computed...>;
};
```

```
template<unsigned Remaining, unsigned... Computed>
struct make_integer_sequence
{
  using type =
    typename make_integer_sequence
    <
      Remaining - 1,
      Remaining - 1,
      Computed...
    >::type;
};

// The call_helper will help us to get the elements of a tuple, because we
// cannot get them by type.
template<typename IntegerSequence>
struct call_helper;

template<unsigned... I>
struct call_helper<integer_sequence<I...>>
{
  template<typename F, typename... Args>
  static void call(F&& function, std::tuple<Args...>& arguments)
  {
    // Here we unpack I... to get the arguments from the tuple.
    // See [2.3.3].
    //
    // Note that it does not work with member functions, as they require another
    // call syntax. This is left as an exercise for the reader.
    function(std::get<I>(arguments)...);
  }
};

// A binding is a function object that can be called with no arguments and that,
// when called, will pass the arguments given to its constructor to the function
// given to its constructor.
template<typename F, typename... Args>
struct binding
{
  binding(F&& f, Args&&... arguments)
    : m_function(std::forward<F>(f)),
      m_arguments(std::forward<Args>(arguments)...)
  {}

  void operator()()
  {
    // Here there is no way to get the elements from the m_arguments tuple by
    // unpacking Args..., so we indirectly build an integer pack that will
    // ultimately be used to call std::get<I>(m_arguments) for each I.
    call_helper
```

```
      <
        typename make_integer_sequence<sizeof...(Args)>::type
      >::call
      (m_function, m_arguments);
  }

private:
  F m_function;
  std::tuple<Args...> m_arguments;
};

// This function is just here to build a binding without specifying all its
// template parameters.
template<typename F, typename... Args>
binding<F, Args...> bind(F&& f, Args&&... arguments)
{
  return
    binding<F, Args...>
    (std::forward<F>(f), std::forward<Args>(arguments)...);
}

// The function that will be bound.
void print(int a, int b)
{
  printf("%d, %d\n", a, b);
}

int main()
{
  // This creates the binding. Is it a valid use case of auto [2.1.9]?
  auto f(bind(&print, 24, 42));

  // And now we can call print with the provided arguments.
  f();

  return 0;
}
```

2.5.9 Argument Bindings

What was the problem

The bindings example from 2.5.8 was quite complex, and it does not even handle member functions. Since it is already using C++11 features, it is left to the reader to consider how to implement something equivalent with pre-C++11 features.

How the Problem is Solved

However, a better implementation of a binding is available in C++11, via `std::bind()`. This function creates a function object, of an unspecified type, that will forward its arguments to the bound function. This

<utility>

is something I would typically use in event handling.

```
struct message_queue { /* ... */ };

struct message_counter
{
  void count_message();
};

void connect_handlers(message_queue& queue)
{
  message_counter counter;

  // Here std::bind() creates a function object that calls
  // counter.process_message() when invoked.
  //
  // Note that the arguments of the function object are deduced from
  // the signature of message_dispatcher::count_message.
  queue.on_message(std::bind(&message_counter::process_message, &counter));
}
```

Interestingly, it is also possible to either force the value of an argument, or to redirect an argument from the caller to a specific argument of the called.

```
struct message_queue { /* ... */ };

void log_error(int queue_id, error e);

void connect_error_handler(message_queue& queue, int queue_id)
{
  // This one creates a function object accepting a single argument,
  // that calls log_error(queue_id, e), for a given argument e.
  //
  // Note that we must explicitly tell what to do with the argument
  // given by the caller here, via the placeholder.
  queue.on_message(std::bind(&log_error, queue_id, std::placeholders::_1));
}
```

In the above code, `std::placeholders::_1` tells `std::bind` to pass whatever is received as the first argument (because the _1) to the second argument of `log_error` (second because it is the second following the function when the binding is created).

Note that the standard does not define how many placeholders must be defined.

2.5.10 Reference Wrapper

What was the problem

Be it C++11 or before, template argument deduction never pick a reference. So, for example, if we wanted to create an `std::pair<int, int&>`, we could not use `std::make_pair`:

```
int a;
int b;

// Ok in C++11, not before. The members of the pair are references to
// a and b.
std::pair<int, int&> pair_1(a, b);

// Not ok, std::make_pair returns an std::pair<int, int>.
std::pair<int, int&> pair_2 = std::make_pair(a, b);

// Not ok in C++11 and before, for different reasons.
std::pair<int, int&> pair_3 = std::make_pair<int, int&>(a, b);
```

How the Problem is Solved

The `std::ref()` function (as well as `std::cref()`) introduced in C++11 will help us with this problem. They both create a reference wrapper for their argument (non const or const, respectively), that is implicitly convertible to a raw reference.

<utility>

```
int a;
int b;

// Ok, std::make_pair returns an
// std::pair<int, std::reference_wrapper<int>>, which is itself
// convertible to std::pair<int, int&>.
std::pair<int, int&> pair_make = std::make_pair(a, std::ref(b));
```

This is especially useful when binding variables that we don't want to copy.

```
#include <algorithm>
#include <cstdio>
#include <functional>

struct capacity_tracker
{
  // Decrease the capacity by the given value if it is not too much. Returns
  // false if the capacity has been decreased, true otherwise.
  //
  // Yes, the meaning of the return value is crap. It is just for the example,
  // so it matches the expectations of std::find() below. Don't do that at home.
  bool operator()(int value)
  {
    if (capacity < value)
      return true;

    capacity -= value;
    return false;
  }

  int capacity;
```

```
};

int main()
{
  capacity_tracker tracker = { 10 };
  const int values[] = { 3, 1, 2, 4, 5, 8, 6, 7 };

  // Decrease the capacity by the given values until the capacity becomes lower
  // than the value. The capacity is decreased along the way and thus reflects
  // the final value at the end.
  //
  // Even though the returned value is correct, the final capacity in the
  // tracker will be incorrect. Indeed, it is copied in the call to
  // std::find_if(), so any change has no impact on the local variable.
  const int* overflow = std::find_if(values, values + 8, tracker);

  printf
    ("remaining capacity: %d, overflow with %d\n", tracker.capacity, *overflow);

  // This one will work correctly, because a reference to the tracker is passed
  // to std::find_if.
  overflow = std::find_if(values, values + 8, std::ref(tracker));

  printf
    ("remaining capacity: %d, overflow with %d\n", tracker.capacity, *overflow);

  return 0;
}
```

2.5.11 Functions

What was the problem

Storing functions in a variable in C++ used to be a pain. For example, how would we implement `callable` such that the code below works?

```
void foo();
void bar(int arg);

struct some_object
{
  void some_method();
};

void test();
{
  std::vector<callable> scheduled;

  scheduled.push_back(&foo);
  // bar with arg = 5.
```

```
  scheduled.push_back(std::bind(&bar, 5));

  some_object c;
  scheduled.push_back(std::bind(&some_object::some_method, &c));

  // All stored functions can be called as if they were void().
  scheduled[0]();
  scheduled[1]();
  scheduled[2]();
}
```

Having a working type for all these use cases is a huge task, and before C++11 our only hope were Boost.Function, some other libraries[13], or and homemade solution.

A binding like presented in Section 2.5.8 is a partial answer to that but first, it does not even handle member functions, and two, it is already C++11.

How the Problem is Solved

The `std::function` type introduced in C++11, in combination with `std::bind` [**??**], is exactly what we need to fix our previous example. The former is an object that represents a function, and that can be called to invoke this function. It can be a function object or a free function, everything works.

<utility>

```
void foo();
void bar(int arg);

struct some_object
{
  void some_method();
};

void test();
{
  std::vector<std::function<void()>> scheduled;

  scheduled.push_back(&foo);
  // bar with arg = 5.
  scheduled.push_back(std::bind(&bar, 5));

  some_object c;
  scheduled.push_back(std::bind(&some_object::some_method, &c));

  // All stored functions can be called as if they were void().
  scheduled[0]();
  scheduled[1]();
  scheduled[2]();
}
```

[13] Search for FastDelegate, the Impossibly Fast C++ Delegates, More Fasterest Delegates, and Ultimate Fast C++ Delegates II'. Some of them may not exist.

As far as can tell there is only one downside to `std::function`, it is that every call begins with a test checking if a function is set. If we care about performance, it can be an issue[14].

2.5.12 Hash Tables

What was the problem

There were two main associative containers in C++ before C++11: `std::vector` and the likes, to associate integer keys with almost any value type, and `std::map`, to use any key type whose values can be strictly ordered. Similar to the latter, `std::set` is an ordered set of values. In practice, `std::map` is an `std::set` where the value type is `std::pair`, with the key as the first field (with a const modifier), and the value as the second field.

These last two collections had several problems, the main one being that they require a total order on their entries. However, it is not exceptional to have a type for which `operator<()` is not defined and where providing one would be weird.

```
struct color
{
  uint8_t red;
  uint8_t green;
  uint8_t blue;

  // Does it make sense to say that a color is less than another color?
  bool operator<(const color& that) const
  {
    // Note that in C++11 we could have used std::tie() to create a
    // tuple from the fields [2.5.8], then used
    // std::tuple::operator<() to compare them lexicographically.

    if (red != that.red)
      return red < that.red;

    if (green != that.green)
      return green < that.green;

    return blue < that.blue;
  }
};
```

A way to avoid the weird operator is to declare the comparison operator outside the type, as an independent function object. It has the effect of not pretending that there is a meaningful order on the given type. The type of the function object is then passed to `std::set` or `std::map`; it becomes a property of the container (i.e. the way its entries are ordered) instead of a property of the data.

```
struct color
{
```

[14] This test is used to throw an `std::bad_function_call` if the invocation is done on an empty instance. If you wonder why `std::function` does not reference by default a function that throws the exception, such that no test is done and the exception is still thrown when an empty function is invoked, know that I wonder too. If you have insight about it, I would love to know.

```
  uint8_t red;
  uint8_t green;
  uint8_t blue;
};

// This is one way to order, amongst many. The advantage here is that
// there is no inherent ordering attached to color.
struct color_lexicographical_order
{
  bool operator()(const color& left, const color& right) const
  {
    if (left.red != right.red)
      return left.red < right.red;

    if (left.green != right.green)
      return left.green < right.green;

    return left.blue < right.blue;
  }
};

std::set<color, color_lexicographical_order> used_colors;
```

Another problem with `std::set` and `std::map` is that they are implemented as a balanced tree, typically a red-black tree, where the nodes are dynamically allocated. This is not very performance-friendly since the data ends up scattered in memory, with an additional cost of three pointers per entry. Additionally, look-up is logarithmic.

How the Problem is Solved

Since C++11, these containers are advantageously replaced by `std::unordered_set` and `std::unordered_map`, which provide the same service but with an implementation based on hash tables. Now, look-up is more often constant-time than anything else. The difficulty being to find a good hash function for the stored type.

<unordered_map>, <unordered_set>

```
struct color
{
  // Both std::unordered_set and std::unordered_map need a way to tell
  // if two instances are equal, in case of a hash collision.
  bool operator==(const color& that) const

  uint8_t red;
  uint8_t green;
  uint8_t blue;
};

namespace std
{
  // We can specialize std::hash for our own types. It is one of the
  // few symbols from the STL that we are allowed to specialize.
  template<>
```

```
  struct hash<color>
  {
    std::size_t operator()(const color& c) const
    {
      return
        ((std::size_t)c.red << 16)
        | ((std::size_t)c.green << 8)
        | (std::size_t)c.blue;
    }
  };
}

// No need to define the hash since we defined it via
// std::hash. Alternatively, we could have set the second template
// parameter std::unordered_set<color, some_hash_type>.
std::unordered_set<color> used_colors;
```

Moreover, a new way to insert elements in a set or a map has been introduced, via the `emplace()` method. This method will create the item in-place, without copies, while the previous `insert()` approach would copy its argument into the container. It is available both for the ordered and unordered variants:

```
struct color
{
  color(uint8_t r, uint8_t g, uint8_t b);

  bool operator==(const color& that) const

  uint8_t red;
  uint8_t green;
  uint8_t blue;
};

std::unordered_set<color> used_colors;

// The arguments are forwarded to the constructor of color.
used_colors.emplace(218, 13, 13);
```

For the map version, since the entries are instances of `std::pair`, we cannot pass all parameters in a single argument list, as the compiler would not know where the constructor arguments for `std::pair::first` would end and where the ones for `std::pair::second` would start. Consequently, we must use tuples [**??**] and the piecewise constructor:

```
std::unordered_map<color, unsigned> color_count;

// The std::piecewise_construct is here to select the corresponding
// constructor from std::pair. The triplet argument will be forwarded
// directly to the construction of the first member; The second
// singleton argument will be forwarded to the second member.
color_count.emplace
  (std::piecewise_construct,
   std::forward_as_tuple(218, 13, 13),
```

```
    std::forward_as_tuple(1));
```

The main problem with `std::unordered_set` and `std::unordered_map` is that they perform quite poorly. Unfortunately, due to the constraints imposed by the standard, they cannot be implemented in another way.

> **Guideline**
>
> If you need an hash table in the internals of your application or library, i.e. it won't be visible by anything outside, then you probably want to switch to another implementation. Otherwise, if your library or application exposes a hash table, you have to consider:
>
> 1. if it makes sense to pass the exposed table as-is to another third-party application, then use the standard containers,
> 2. if the exposed table is expected not to go anywhere, then use a better container.
>
> So far, I have been quite satisfied with the robin hood hashing from Martin Ankerl, a.k.a martinus, at https://github.com/martinus/robin-hood-hashing.

2.5.13 Type Traits

What was the problem

Let's have a look at this small function:

```
template<typename T>
T mix(T a, T b, T r)
{
  return r * a + (1 - r) * b;
}
```

This is a quite basic function that just combine two values with a weighted sum. Its implementation tells us that r should probably be in $[0, 1]$, but most importantly, the computation makes sense only if T is a float-like type. If we want to prevent the compiler or the programmer to call this function with an incompatible type, we can add some SFINAE[15], for example by introducing a type deduced from T that would not be defined if T is not a float-like type. In C++98 the implementation could have been similar to:

```
// This struct declares a type identical to T if and only if T is
// a float-like type. We are going to specialize it only for the
// valid types. For the other types, the missing type declaration
// in the struct will trigger a substitution failure.
template<typename T>
struct only_if_float_like;

template<>
struct only_if_float_like<float>
{
```

[15] Substitution Failure Is Not An Error, is a principle in template instantiations according to which the compiler must not emit an error if it fails to instantiate a template. Instead it should try another template candidate. This feature is often used and abused to provide multiple implementation behind the same function signature.

```
  typedef float type;
};

template<>
struct only_if_float_like<double>
{
  typedef double type;
};

template<typename T>
typename only_if_float_like<T>::type mix(T a, T b, T r)
{
  return r * a + (1 - r) * b;
}
```

Aside from the fact that `long double` is not handled, does it work as expected?

```
void test()
{
  // float and double are handled as expected.
  printf("%f\n", mix(1.f, 3.f, 0.5f));
  printf("%f\n", mix(1.d, 3.d, 0.5d));

  // Using integers triggers an error, we can say that it fails successfully!
  // printf("%d\n", mix(1, 3, 2));
}
```

Typing custom type like `only_if_float_like` for every use case is repetitive, so we would certainly end up splitting it into two types, that could be used like `only_if<is_float_like<T>::value, T>::type`.

How the Problem is Solved

<type_traits>

Lucky us, C++11 greatly simplify this work by introducing the required types, in the form of `std::enable_if` and `std::is_floating_point_type`. Using these types the whole implementation is reduced to the following:

```
#include <type_traits>

template<typename T>
typename std::enable_if<std::is_floating_point<T>::value, T>::type
mix(T a, T b, T r)
{
  return r * a + (1 - r) * b;
}
```

There are many other predicates and operations added in <type_traits> in C++11, that can greatly help for metaprogramming. I can only suggest to have a look at them on a nice online reference.

2.6 Miscellaneous

In the previous sections we saw many nice features from C++11. There are actually many more! As I did not use all of them, here come a short description for the missing ones.

2.6.1 Regular Expressions

Regular expressions are now first class citizen in the STL. We can finally easily write an HTML parser in a few lines of C++[16]!

<regex>

Interestingly, the format for the expressions can be of almost any existing syntax: ECMAScript, grep, awk...

2.6.2 Explicit Conversion Operators

Conversion operators can now be declared as explicit, to avoid unexpected conversions:

```
// This struct wraps an 8 bits non-negative integer value.
struct custom_uint8 { /* ... */ };

// This struct wraps a 64 bits signed integer value.
struct custom_int64 { /* ... */ };

// This struct wraps a 32 bits integer value.
struct custom_int32
{
  // This operator is explicit, as information may be lost when
  // truncating to 8 bits. Thus we don't want it to happen silently.
  explicit operator custom_uint8() const
  {
    return custom_uint8(m_value);
  }

  // This operator is implicit, as all signed 32 bits values can be
  // represented in a signed 64 bits.
  operator custom_int64() const
  {
    return custom_int64(m_value);
  }

private:
  int32_t m_value;
};

void foo()
{
  custom_uint8 i8;
  custom_int32 i32;
```

[16] Reference for the joke: https://stackoverflow.com/a/1732454/1171783. If you don't follow this link, just know that one cannot correctly parse random HTML with a regular expression.

```
  custom_int64 i64;

  // This is ok thanks to the implicit conversion operator.
  i64 = i32;

  // This won't work since there is an implicit conversion.
  i8 = i32;

  // Here the conversion is explicit, so it will work.
  i8 = static_cast<custom_uint8>(i32);
}
```

2.6.3 Unrestricted Unions

Unions can now contain non-POD members. I do not know where it is useful but it can be done.

2.6.4 String Literals

New string literals have been introduced to represent UTF-8, UTF-16, UTF-32 strings, as well as raw strings:

```
const char* utf_8 = u8"I'm UTF-8.";
const char* utf_16 = u"I'm UTF-16.";
const char* utf_32 = U"I'm UTF-32.";

const char* raw = R"_(This text is stored as is,
line breaks included,
  spaces included.
It is thus a four-lines text.)_";
```

2.6.5 User Defined Literals

Have you ever mixed some units like in the declartion of `m` below?

```
struct mass
{
  mass(float kg)
    : kilograms(kg)
  {}

  float kilograms;
};

void foo()
{
  // A mass of 2000 grams.
  mass m(2000);

  // Oops, mass() takes the value in kilograms :(
}
```

If you have already made this mistake, or just if you use this kind of code, you may be happy to learn that you can now use custom suffixes for literals, which may help to explicit the unit.

```
// We define the 'g' suffix for float numbers, to represent grams.
mass operator"" _g(float v)
{
  return mass(v / 1000);
}

void foo()
{
  // A mass of 2000 grams.
  mass m = 2000_g;

  // m.kilograms is equal to 2, everything is fine.
}
```

2.6.6 Very Long Integers

Sometimes an `int` is a bit short to store large values. Then we switch to `long int`, but sometimes it is still not enough. Now with C++11 we can also switch to `long long int`[17].

This type is guaranteed to be made of at least 64 bits, while `int` and `long int` are made of at least 16 and 32 bits respectively.

Actually, I doubt I will ever use it, because when I need to store a value in at least n bits, I usually choose `int`n`_t` type from <cstdint>.

2.6.7 Size of Members

Computing the size of a struct member in the old days of C++ required to use an instance of the struct, like in:

```
struct foo
{
  int m;
};

int main()
{
  // There is no way to apply sizeof directly to m, so we "create"
  // an instance just to get the member. Fortunately the expression
  // on which sizeof is applied is not evaluated.
  printf("%ld\n", sizeof(foo().m));

  return 0;
}
```

[17] Also available as `long int long`, or `int long long`, or `signed long int long`, or `long int signed long`, etc.

This is doable for simple structs but clearly becomes difficult if the constructor needs arguments.

In C++11 there is now a syntax to get the size of a member:

```
int main()
{
  printf("%ld\n", sizeof(foo::m));
  return 0;
}
```

2.6.8 Type Alignment

If you are the kind of person who write allocators or who use placement new in a buffer, then you will be happy to learn about `alignas(T)`, which sets a type alignment to the value `T`, if it is an integer, or to match the alignment of the type `T` otherwise. Similarly, `alignof(T)` is the alignement of the type `T`.

```
char buffer[1024];
int consumed = 0;

template<typename T>
T* allocate()
{
  const int shift = consumed % alignof(T);
  const int offset = (shift == 0) ? 0 : (alignof(T) - shift);

  T* const result = new (buffer + consumed + offset) T();
  consumed += offset + sizeof(T);

  return result;
}
```

2.6.9 Attributes

Maybe have you already seen the `__attribute__` token in some C++ code? This is used to annotate the code with hints or directives about some code. Unfortunately it is a compiler extension, so it was not usable in portable code.

Starting with C++11, we now have a standard way to specify attributes with the `[[some_attribute]]` syntax. Their semantic is still implementation-defined but at least it is syntaxically portable.

One of the only two well defined attributes in the C++11 standard is `[[noreturn]]`, which tells that a function never returns to the caller:

```
[[noreturn]] void fatal_error(const char* m)
{
  printf("%s\n", m);
  exit(1);
}
```

The second one is `[[carries_dependency]]` and I won't talk about it.

Chapter 3

Nice Things from C++14

C++14 is frequently qualified as a bugfix version of C++11, indeed most features are improvements or extensions of things introduced in the latter.

Nevertheless, it does not mean that these improvements are not worth it. Let's see.

3.1 At the Language Level

3.1.1 Number separator

What was the problem

Long numbers are hard to read. Check how long you need to get the numbers in this C++11 code:

```
int wisconsin_population = 5822434;
int california_population = 39512223;
```

My guess is that you grouped the digits by three in your mind in order to get the numbers right. Didn't you? Usually when we write large numbers like that, we group the digits with a separator to make the numbers easier to read, like in 1'234'567.

How the Problem is Solved

Since C++14 we can use this syntax in the code:

```
int wisconsin_population = 5'822'434;
int california_population = 39'512'223;

// It works with float numbers too.
const float f = 1'111.222'222;

// And also with for non-decimal numbers. Funny thing, the quote can
// appear anywhere between two digits.
```

```
const unsigned mask = 0xfff'0'00'ea;
```

3.1.2 Binary Suffix

What was the problem

In C++11 and before, we used to declare bit patterns as hexadecimal values, or shifts. For example, if one wanted to access the flags stored in the packed fields from the image descriptor of a GIF 87a file [Com87], the code could have been similar to:

```
void parse_image_descriptor_fields(std::uint8_t packed_fields)
{
  // Bitmask with shifts.
  const bool use_local_color_table = packed_fields & (1 << 7);

  // Bitmasks with hexadecimal values.
  const bool is_interlaced = packed_fields & 0x40;
  const std::int8_t bits_for_color_palette_size_minus_one =
    packed_fields & 0x07;
}
```

While not really obscure, this syntax requires some effort from the reader, and a good understanding of the binary representation of numbers. What if we could directly write the binary?

How the Problem is Solved

That's possible with C++14. Just like the 0x prefix introduces an hexadecimal value, 0b introduces a binary value:

```
void parse_image_descriptor_fields(std::uint8_t packed_fields)
{
  // Bitmask with binary literal.
  const bool use_local_color_table = packed_fields & 0b10000000;

  // Note that we can use the number separator [3.1.1] to split the mask.
  const bool is_interlaced = packed_fields & 0b0100'0000;
  const std::int8_t bits_for_color_palette_size_minus_one =
    packed_fields & 0b0111;
}
```

3.1.3 `[[deprecated]]` Attribute

What was the problem

As code evolves, some parts naturally become obsolete. Most of the time, obsolete code can be simply removed, but when it is published as a public API, great care must be taken in order not to break client code.

In such situation, the publisher would typically provide both the old and the new API for a period of time long enough for the clients to migrate. Which they may do only if they know which parts are obsolete!

So the issue here is actually to pass the information to the developers that the code is going to be deleted. How can we do that? Maybe by adding a paragraph in the release notes? I've heard that some people read them. Or maybe a warning in the header? Here we are going to rely on compiler-specific features, and the warning will be displayed as soon as the file is included, even if the deprecated code is not called. What about printing the deprecation message at run time, when the code is executed[1]?

How the Problem is Solved

Starting with C++14, the `[[deprecated]]` attribute can be used to target a specific symbol for deprecation, with an explicit message that will be displayed to the user, if and only if the symbol is used.

```
[[deprecated("Use the button class instead.")]]
class toggle_button
{
  /* ... */
};

class button
{
public:
  [[deprecated("Use set_font(const font&) instead.")]]
  void set_font(const std::string& font_name);
};

void create_buttons()
{
  // This line will trigger a deprecation warning.
  toggle_button toggle;

  // This one doesn't.
  button b;

  // But calling the deprecated method will trigger the warning.
  b.set_font("sans");
}
```

3.1.4 Generic Lambdas

What was the problem

The introduction of lambdas [2.1.7] in C++11 unlocked many possibilities for the programmer. Despite that, some use cases are still a bit difficult to tackle. Take for example a unique predicate that should be applied to two collections of different types, as in the example below:

```
#include <vector>
#include <algorithm>

bool same_count_by_sign
```

[1] Please don't.

```
(const std::vector<int>& ints, const std::vector<float>& floats)
{
  if (ints.size() != floats.size())
    return false;

  const auto non_positive_integer =
    [](int v) -> bool
    {
      return v <= 0;
    };
  const auto non_positive_float =
    [](float v) -> bool
    {
      // This is the same body as above. Do I really need two lambdas?
      return v <= 0;
    };

  const std::size_t count_ints =
    std::count_if(ints.begin(), ints.end(), non_positive_integer);
  const std::size_t count_floats =
    std::count_if(floats.begin(), floats.end(), non_positive_float);

  return count_ints == count_floats;
}
```

It is a bit disappointing to have two lambdas with the same body. Intuitively, one would want to templatize it. Unfortunately template lambdas do not exist, so we have to switch back to the pre-C++11 way with free functions or function objects.

```
#include <vector>
#include <algorithm>

namespace
{
  // There we go, a simple predicate function declared far from its use.
  template<typename T>
  bool non_positive(T v)
  {
    return v <= 0;
  }
}

bool same_count_by_sign
(const std::vector<int>& ints, const std::vector<float>& floats)
{
  if (ints.size() != floats.size())
    return false;

  const std::size_t count_ints =
    std::count_if(ints.begin(), ints.end(), &non_positive<int>);
  const std::size_t count_floats =
```

```
    std::count_if(floats.begin(), floats.end(), &non_positive<float>);

  return count_ints == count_floats;
}
```

It works, for sure, but it is not what we expect. Moreover, as soon as a variable has to be captured, we are back to the extremely verbose functor objects.

How the Problem is Solved

Enters C++14 and the generic lambdas. By declaring the lambda's arguments as `auto`, we can declare what is effectively the equivalent of a template lambda.

```
#include <vector>
#include <algorithm>

bool same_count_by_sign
(const std::vector<int>& ints, const std::vector<float>& floats)
{
  if (ints.size() != floats.size())
    return false;

  const auto non_positive =
    [](auto v) -> bool
    {
      return v <= 0;
    };

  // It's the same predicate in both calls.
  const std::size_t count_ints =
    std::count_if(ints.begin(), ints.end(), non_positive);
  const std::size_t count_floats =
    std::count_if(floats.begin(), floats.end(), non_positive);

  return count_ints == count_floats;
}
```

We saw in 2.1.7 what was the equivalent function object of a lambda. Here the corresponding transformation, given a lambda declaration like

```
[](auto a1, ..., auto an ) -> /* return type */ { /* statements */ }
```

would be something like

```
struct something
{
  template<typename T1, ..., typename Tn>
  /* return type */ operator()(T1 a1, ..., Tn an) const { /* statements */ }
};
```

Note that the type of the arguments are not related. Each use of `auto` here corresponds to a single template parameter. Also, it is totally possible to use declare a parameter pack [2.3.3], for example by using the `auto...` syntax.

> **Guideline**
>
> Mandatory `auto`-related notice: mind the next reader; write what you mean.
>
> As usual with the `auto` keyword, one may be tempted to use it to not have to think about the actual needed type, or because it looks "generic" or "future proof". Don't do that.
>
> The `auto` keyword is here to tell the compiler and the reader that the variable can be of many types. If we use it even when a single type is expected, it loses its meaning, and the code becomes harder to understand.

3.1.5 Lambda Capture Expressions

What was the problem

When we are defining a lambda, we have the possibility to capture variables of the enclosing scope such that they become available in the body of the lambda [2.1.7]. In practice, such variables are either copied (like `var1` below, due to the = syntax), or referenced (like `var2` below, with the `&` syntax).

```
const auto foo = [=var1, &var2]() -> void {};
```

Unfortunately there is no other variant of the capture, which leads to situations like this:

```
void wait_for_new_players(const std::string& group)
{
  std::string display_label = group + ": ";

  // Copies are striking again. Here display_label is copied into a
  // field of the function object associated with this lambda; not
  // moved nor initialized on construction.
  auto add_player_to_list =
    [display_label](player_id id, const std::string& player alias)
    {
      m_player_list->add(id, display_label + alias);
    };

  m_team_service.wait_for_players(group, add_player_to_list);
}
```

In the above example, we would want to avoid the copy of the display label into the equivalent member variable of the lambda. In C++11 there is only two ways to achieve that: either by creating the display label in the callback's body, thus creating a new string on each call[2], or by using a custom function object, C++98-style.

[2] We would also have a copy of the captured group variable anyway.

Additionally, we can note that this behavior prevent any use of a move-only type for a captured variable, such as `std::unique_ptr`. Indeed, there is no way to move the variable from the outer scope into the lambda.

How the Problem is Solved

In C++14, capture lists have been extended to allow the initialization of a variable by an expression. We can use that to initialize the display label with the final string directly:

```
void wait_for_new_players(std::string group)
{
  // display_label is initialized directly in the construction of the
  // lambda, thus saving a copy.
  auto add_player_to_list =
    [display_label = group + ":  "]
    (player_id id, const std::string& alias)
    {
      m_player_list.add(id, display_label + alias);
    };

  m_team_service.wait_for_players(group, add_player_to_list);
}
```

This also solves the move-only type situation since we can now move the variable:

```
void apply_filter(std::vector<int>& values, std::unique_ptr<filter> filter)
{
  std::for_each
    (values.begin(), values.end(),
     [f = std::move(filter)](int& v) -> void
     {
       v = f->transform(v);
     });
}
```

3.1.6 Return Type Deduction

What was the problem

The `decltype` specifier was added in C++11 to identify the type of an expression [2.1.8]. One of its typical use cases is the declaration of the return type in a template function whose result type must be deduced from the template arguments.

```
// This function is a proxy to call a given function with the given
// arguments. Its return type depends on the provided function.
template<typename F, typename... Args>
auto invoke(F&& f, Args&&... args) -> decltype(f(std::forward<Args>(args)...))
{
  return f(std::forward<Args>(args)...);
}
```

Note how the expression passed to `decltype` is exactly the body of the function.

How the Problem is Solved

To avoid this verbosity, C++14 allows to omit the trailing return type to let the compiler deduce the type from the function's body.

```
template<typename F, typename... Args>
auto invoke(F&& f, Args&&... args)
{
  return f(std::forward<Args>(args)...);
}
```

An excellent use case for this feature is a function returning a lambda [2.1.7]. Have you ever tried to write one?

```
auto make_comparator(int v) -> /* what should I write here? */
{
  return [v](int c) -> bool
  {
    return v == c;
  };
}
```

As far as I know there is no way to write such a function in C++11. In C++14, though, we can simply drop the return type and let the compiler deduce it from the body.

> **Guideline**
>
> One could be tempted to always omit the return type under the false assumption that less code is more readable. Don't do that.
>
> Whenever we omit the return type, the reader has to parse the whole body to find the actual type. This is needlessly complex and makes painful code reviews. What should be more readable actually requires to process more code.
>
> Implied information are bad in programming. Don't follow the "modern C++" trend which pushes for using every new feature everywhere; use them only when you need them.

3.1.7 Variable Templates

What was the problem

Up to C++11, the only things that could be templated were types and functions. So if you had a variable whose value was dependent on a template parameter, you had to wrap the variable in a type. Take for example this code that computes the number of bits set to 1 in a constant at compile time:

```
#include <cstdio>

template<unsigned N>
struct popcount;
```

```
template<>
struct popcount<0>
{
  static constexpr unsigned value = 0;
};

template<unsigned N>
struct popcount
{
  static constexpr unsigned value = N % 2 + popcount<(N >> 1)>::value;
};

int main()
{
  // In order to actually do the computation, we have to instantiate
  // a type and get the value declared inside this type.
  printf("%d\n", popcount<24>::value);
}
```

How the Problem is Solved

Starting with C++14, all the types in the above code can be removed and replaced by a much more concise template variable, which also carries the intent more accurately.

```
#include <cstdio>

// This is a variable, whose value depends upon a template argument.
template<unsigned N>
unsigned popcount = N % 2 + popcount<(N >> 1)>;

// The variable can be specialized too!
template<>
unsigned popcount<0> = 0;

int main()
{
  // Here we can use a variable to represent a value. Much more clear.
  printf("%d\n", popcount<24>);
}
```

Note than in this specific case a constexpr function [2.1.4] is more suited to the task; plus it can also be called at run time.

You will probably not see many variable templates in the wild, as they are quite a niche feature. Mostly, they are used to store some domain-specific constants to the precision of a given type.

3.1.8 `decltype(auto)`

What was the problem

We saw in [3.1.6] that we can now omit the return type of a function. So can you tell why the assert won't pass in the code below?

```
#include <type_traits>
#include <utility>

template<typename F, typename... Args>
auto invoke(F&& f, Args&&... args)
{
  return f(std::forward<Args>(args)...);
}

int& at(int* v, int i)
{
  return v[i];
}

// Note that no call occurs here since decltype(expression) produces
// the the type at compile-time, _as-if_ the expression was
// evaluated. So it is safe to use nullptr in the arguments.
static_assert
  (std::is_same
   <
     decltype(invoke(&at, nullptr, 42)),
     decltype(at(nullptr, 42))
   >::value,
   "");
```

If you ever encounter one of the few use cases where `auto` it is actually useful, you may be surprised that the deduced type does not match your expectation. In the above code one could read the return type of `invoke` as "whatever is returned by `f`". Unfortunately, `auto` does not work like that.

The `auto` keyword does not take into account the reference nor the constness of the expression. That means that the type of `auto_value` in the code below is actually `int`, not `const int&` like `reference`.

```
#include <type_traits>

int i;
const int& reference = i;
auto auto_value = reference;
static_assert(std::is_same<int, decltype(auto_value)>::value, "");
```

How the Problem is Solved

Most of the time, `auto` can be replaced by a more explicit type, but in the case of the `invoke` function the explicit version requires to repeat the whole body (see [3.1.6]). As a workaround, C++14 introduces the

`decltype(auto)` specifier, which keeps all the properties of the type of the statement from which the type is deduced:

```
#include <type_traits>
#include <utility>

template<typename F, typename... Args>
decltype(auto) invoke(F&& f, Args&&... args)
{
  return f(std::forward<Args>(args)...);
}

int& at(int* v, int i)
{
  return v[i];
}

// It works!
static_assert
  (std::is_same
   <
     decltype(invoke(&at, nullptr, 42)),
     decltype(at(nullptr, 42))
   >::value,
   "");
```

As usual, `decltype(auto)` can be used anywhere a type is required, as long as there is a way for the compiler to deduce the type.

As usual, there is no good reason to prefer that when a more explicit type can be used.

3.1.9 Relaxed `constexpr`

What was the problem

The introduction of `constexpr` in C++11 [2.1.4] greatly simplified some compile-time complex computations, even though they were some limitations. The most frustrating of which was certainly the constraint that `constexpr` functions should contain a single statement (a `return` statement). In practice this means no variable, no branch or control flow other than the ternary `?:` operator and using recursion for any loop-based algorithm. See for example this implementation of `popcount()` we wrote in section [2.1.4]:

```
#include<cstdio>

constexpr int popcount(unsigned n)
{
  return (n == 0) ? 0 : ((n & 1) + popcount(n >> 1));
}

int main(int argc, char**)
{
  int array[popcount(45)];
```

```
  printf("%d\n", popcount(argc));

  return 0;
}
```

How the Problem is Solved

In C++14, these restrictions are lifted and we can implement this function in a less convoluted way[3]:

```
#include<cstdio>

constexpr int popcount(unsigned n)
{
  int result = 0;

  for (; n != 0; n >>= 1)
    if ((n & 1) != 0)
      ++result;

  return result;
}

int main(int argc, char**)
{
  int array[popcount(45)];
  printf("%d\n", popcount(argc));

  return 0;
}
```

A subtle change in the transition from C++11 to C++14 is that `constexpr` used on a member function does not implies `const` anymore. For example, this code would not compile in C++11 but does in C++14:

```
struct s
{
  int value;

  constexpr int increment(int v)
  {
    // Can't do that in C++11 as the function signature is actually
    //
    //   constexpr int increment(int) const.
    //
    // Okay in C++14, where the function signature is as declared.
    return value += v;
  }
};
```

[3]Not that recursion is inherently convoluted. Some algorithms work very well when implemented in a recursive way, both regarding the readability and the performance, but even though any programmer worth is money must be know the practice I would argue that most code, i.e. the code we are familiar with, is non-recursive.

This can come as a surprise when switching to a newer standard since calling a `constexpr`-only member function on a `const` instance will suddenly trigger errors like “passing ‘const x’ as ‘this’ argument discards qualifiers”.

3.2 In the Standard Library

3.2.1 `make_unique`

What was the problem

<memory>

C++11 introduced smart pointers in the standard library, with `std::unique_ptr` and `std::shared_ptr`. For the latter the idiomatic creation code would use `std::make_shared`, which would have the benefit to create storage for the control block and the actual object into a single allocation. On the other hand, `std::unique_ptr` has no control block, so it made sense at the time not to have specialized creation functions for it.

There is another use case though, where such function can help:

```
create_form
  (std::unique_ptr<widget>(new label("Everything is fine.")),
   std::unique_ptr<widget>(new icon("resources/alert.png")));
```

See, since there is no guarantee on the order of evaluation of function arguments, the compiler is free to sequence them as follows:

1. `new label(...)`
2. `new icon(...)`
3. `std::unique_ptr(...)`
4. `std::unique_ptr(...)`

Now, if the constructor of `icon` throws an exception, then the `label` won’t ever be destroyed, nor its memory released during the program execution.

How the Problem is Solved

This can be solved by wrapping the allocation in a function, a function provided by the standard library for example.

```
create_form
  (std::make_unique<label>("Everything is fine."),
   std::make_unique<icon>("resources/alert.png"));
```

With this code, the order of evaluation does not matter. If the first argument is built before the second, then its memory already managed by the smart pointer, so even if the construction of the second argument throws, the object will be destroyed and the memory will be released. The reasoning is the same if the second argument is built before the first.

As a bonus, this is also less verbose.

3.2.2 Compile-time Integer Sequences

What was the problem

<utility>

If you start playing with parameter packs [2.3.3] and tuples [2.5.8], you will soon encounter a case where you will need to so something "for each item in the tuple".

A typical use case consists in calling a function with its arguments taken from a tuple, something we already tried in the previous pages [2.5.8], in an argument-bindings example. Go check, and see how much code we wrote to store an sequence of integers in a type.

How the Problem is Solved

Starting with C++14, this code can be replaced by `std::integer_sequence<typename T, T... Ints>` and `std::make_integer_sequence<typename T, T N>`. Writing a function that runs a function with its arguments taken from a tuple is approximately as simple as this:

```
// The call_helper will help us to get the elements of a tuple, because we
// cannot get them by type due to possibly duplicate types.
template<typename IntegerSequence>
struct call_helper;

template<unsigned... I>
struct call_helper<std::integer_sequence<std::size_t, I...>>
{
  template<typename F, typename... Args>
  static void call(F&& function, std::tuple<Args...>&& arguments)
  {
    // Here we unpack I... to get the arguments from the tuple.
    // See [2.3.3].
    //
    // Note that it does not work with member functions, as they require another
    // call syntax. This is still left as an exercise for the reader.
    function(std::get<I>(arguments)...);
  }
};

template<typename F, typename... Args>
decltype(auto) apply(F&& function, std::tuple<Args...>&& arguments)
{
  return call_helper
    <
      std::make_integer_sequence<std::size_t, sizeof...(Args)>
    >::call(std::forward<F>(function), std::move(arguments));
}
```

Isn't it clean? Hold on, we'll soon be able to remove this code too [**??**][4]!

[4] And it will be for the best, as this example code misses many features, like calling member functions, accepting a const tuple as argument, and probably more.

3.2.3 `std::exchange`

What was the problem

<utility>

If you are reading this, you certainly had to write a move constructor at some point. When doing so, the programmer has to "steal" the resources from an instance, and leave the aforementioned instance in a valid state. In the example from section 2.2.1, copied below for convenience, the resource from `that` is transferred to `this`, then the field from `that` is set to `nullptr`.

```
struct foo
{
  foo(foo&& that)
    : m_resource(that.m_resource)
  {
    that.m_resource = nullptr;
  }
};
```

As a move constructor grows with the number of fields, the distance between the transfer of the resource in the constructor's initializer list and the reset in the constructor's body increases. This blurs the relationship between the two statements, which actually form a single semantic operation. This is also error-prone, as one may easily forget one of the two instructions.

How the Problem is Solved

Hopefully C++14 introduces a nice small utility with `std::exchange` to solve this issue.

```
struct foo
{
  foo(foo&& that)
    // Assign that.m_resource to m_resource and set that.m_resource to
    // nullptr, in a single statement.
    : m_resource(std::exchange(that.m_resource, nullptr))
  {}
};
```

The `std::exchange` function assigns its second parameter to its first parameter, then returns the previous value of its first parameter. Yes, there is no actual exchange here... As with `std::move`, the name does not carry the meaning very well, but at least it solves a problem.

3.2.4 Tuple Addressing By Type

What was the problem

C++11 introduced tuples [2.5.8] which are quite handy if you are too lazy to write a struct, or if you come from toy languages like Python or JavaScript.

In order to access an element in a tuple, one must know the index of the element to pass to `std::get`. This is a bit problematic if we add an entry in the tuple; then we have to update all accesses to match the new indices[5].

[5] Something that would have not happen if only we used a struct. Let that sink in!

In a metaprogramming context this can also be very limiting. Check for example this map allowing to store values of different types with a key (this is just an aggregate of maps whose key and value types are passed as template arguments):

```
#include <unordered_map>

template<typename K, typename... V>
class heterogeneous_map
{
public:
  template<typename T>
  void set(const K& k, const T& v)
  {
    // What should I use for the index?
    std::get</* index of T in V... */>(m_maps)[k] = v;
  }

private:
  std::tuple<std::unordered_map<K, V>...> m_maps;
};
```

In C++11 one would have to write a helper to find the index of a type in a parameter pack, which is not very difficult but still a lot of work compared with doing nothing.

How the Problem is Solved

In C++14, The `std::get` function has been extended and now accepts a type as its template arguments, allowing to address a tuple element by its type. This only works if the tuple declaration has a single entry of the given type, which is good enough:

```
#include <unordered_map>
#include <tuple>

template<typename K, typename... V>
class heterogeneous_map
{
public:
  template<typename T>
  void set(const K& k, const T& v)
  {
    // No index needed!
    std::get<std::unordered_map<K, T>>(m_maps)[k] = v;
  }

private:
  std::tuple<std::unordered_map<K, V>...> m_maps;
};
```

Chapter 4

Bibliography

[Bri20] Tristan Brindle. *Beware of copies with std::initializer_list!* https://tristanbrindle.com/posts/beware-copies-initializer-list, 2020.

[Com87] Incorporated Compuserve. *Graphics Interchange Format Specifications*. https://www.w3.org/Graphics/GIF/spec-gif87.txt, 1987.

[con05] Tango Project contributors. *Tango Desktop Project*. https://tango.freedesktop.org, Since 2005.

[Dou20] Niall Douglas. *STL Header Heft*. https://github.com/ned14/stl-header-heft, 2020.

[Fou21] Standard C++ Foundation. *2021 Annual C++ Developer Survey "Lite"*, 2021. https://isocpp.org/files/papers/CppDevSurvey-2021-04-summary.pdf.

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, 1 edition, 2008.

[McC04] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, USA, 2004.

[O'N14] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, September 2014.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass, 1986.

[Str91] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, Mass, 1991.

[Str97] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, Mass, 1997.

[Str00] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Boston, 2000.

[Str13] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Upper Saddle River, NJ, 2013.

[Tre20] Philip Trettner. *C++ Compile Health Watchdog*. https://artificial-mind.net/projects/compile-health/, 2020.

Appendix A

Creative Commons Attribution-ShareAlike 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution-ShareAlike 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

1 Definitions

a. Adapted Material means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. Adapter's License means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. BY-SA Compatible License means a license listed at https://creativecommons.org/compatiblelicenses, approved by Creative Commons as essentially the equivalent of this Public License.

d. Copyright and Similar Rights means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

e. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

f. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

g. License Elements means the license attributes listed in the name of a Creative Commons Public License. The License Elements of this Public License are Attribution and ShareAlike.

h. Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

i. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

j. Licensor means the individual(s) or entity(ies) granting rights under this Public License.

k. Share means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

l. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

m. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

2 Scope

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

   a. reproduce and Share the Licensed Material, in whole or in part; and

   b. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make

technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a) (4) never produces Adapted Material.

5. Downstream recipients.

   a. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

   b. Additional offer from the Licensor – Adapted Material. Every recipient of Adapted Material from You automatically receives an offer from the Licensor to exercise the Licensed Rights in the Adapted Material under the conditions of the Adapter's License You apply.

   c. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

   d. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

   1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

   2. Patent and trademark rights are not licensed under this Public License.

   3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

3 License Conditions

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. . Attribution.

   1. If You Share the Licensed Material (including in modified form), You must:

      a. retain the following if it is supplied by the Licensor with the Licensed Material:

         i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

b. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

c. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

b. ShareAlike.

In addition to the conditions in Section 3(a), if You Share Adapted Material You produce, the following conditions also apply.

1. The Adapter's License You apply must be a Creative Commons license with the same License Elements, this version or later, or a BY-SA Compatible License.

2. You must include the text of, or the URI or hyperlink to, the Adapter's License You apply. You may satisfy this condition in any reasonable manner based on the medium, means, and context in which You Share Adapted Material.

3. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, Adapted Material that restrict exercise of the rights granted under the Adapter's License You apply.

4 Sui Generis Database Rights

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material, including for purposes of Section 3(b); and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

5 Disclaimer of Warranties and Limitation of Liability

a. UNLESS OTHERWISE SEPARATELY UNDERTAKEN BY THE LICENSOR, TO THE EXTENT POSSIBLE, THE LICENSOR OFFERS THE LICENSED MATERIAL AS-IS AND AS-AVAILABLE, AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE LICENSED MATERIAL, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHER. THIS INCLUDES, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OR ABSENCE OF ERRORS, WHETHER OR NOT KNOWN OR DISCOVERABLE. WHERE DISCLAIMERS OF WARRANTIES ARE NOT ALLOWED IN FULL OR IN PART, THIS DISCLAIMER MAY NOT APPLY TO YOU.

b. TO THE EXTENT POSSIBLE, IN NO EVENT WILL THE LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY (INCLUDING, WITHOUT LIMITATION, NEGLIGENCE) OR OTHERWISE FOR ANY DIRECT, SPECIAL, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, EXEMPLARY, OR OTHER LOSSES, COSTS, EXPENSES, OR DAMAGES ARISING OUT OF THIS PUBLIC LICENSE OR USE OF THE LICENSED MATERIAL, EVEN IF THE LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSSES, COSTS, EXPENSES, OR DAMAGES. WHERE A LIMITATION OF LIABILITY IS NOT ALLOWED IN FULL OR IN PART, THIS LIMITATION MAY NOT APPLY TO YOU.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

6 Term and Termination

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

   1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

   2. upon express reinstatement by the Licensor.

   For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

7 Other Terms and Conditions

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

8 Interpretation

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.
