Boolan
高端IT互联网教育平台

2017 CPP-Summit

C++及系统软件技术大会

C++ and System Software Summit

演讲稿合辑

www.cpp-summit.org

“全球大师”系列高端在线课程

《CTO必修课》

风靡美国技术界的CTO必修课，揭开技术管理迷雾

主讲老师：

Martin Abbott

全球软件架构大师
《架构即未来》作者

Martin 从 eBay 创业初期直到发展为全球500强企业，担任 eBay 高级技术副总裁以及首席技术执行官。在加入eBay之前，Martin 曾在 Gateway 以及 Motorola 担任技术、管理等重要职位。

面向技术人员和技术管理人员: CTO、技术VP、首席架构师、技术总监、开发主管、研发经理。

- 提升自身领导力及架构能力
- 解决技术团队及架构现存问题
- 提升团队凝聚力及创新力

10+小时课程时长 丰富准则、工具、经验案例。

- 凝聚Martin Abbott 20年一线实操精粹，历经350家企业实践检验。
- 历经数十年精心提炼打磨，专业在线课程团队全心策划制作。

2018 震撼推出！课程体验名额限量招募中。
扫码填写登记表。

2017 CPP-Summit

C++17: We have a new language. What does it mean to me?

Michael Spertus

Fellow/VP, Symantec
University of Chicago

C++17: We do have a new language

- A standard was agreed upon by the committee and unanimously approved by all nations participating in the C++ standards process
- Your compiler probably supports most of it now, so you can start using it right away
- This is much faster than for previous language standards
- Clang status: https://clang.llvm.org/cxx_status.html
- G++ status: https://gcc.gnu.org/projects/cxx-status.html
    - For some reason, this page calls it C++1z instead of C++17
- Visual Studio status: https://blogs.msdn.microsoft.com/vcblog/2017/05/10/c17-features-in-vs-2017-3/

OK. We have a new language. Do I Care?

- C++17 is the latest version of C++, but many people think it’s not really much of a change
- In fact, Mike Wong’s keynote tomorrow is “C++17 was not that great”

- With all due respect to Mike (and I look forward to his talk), maybe there is another way to look at it

What makes something great?

- Does adding the most features at the fastest rate make something great?
- Maybe not. Some people think C++ is too complicated already
- The important thing is to have the most capability made accessible through elegance of design
- A ming vase beats something complex and gimmicky any day

C++17 is about making C++ more powerful and cleaner at the same time

- Not a lot of highly-profile features adding power but more features to learn
- But bringing C++ closer to being a clean and elegant language where programming is natural
- In fact, I want to convince you that C++17 is quietly a lot cleaner and more powerful than any preceding C++ standard
- This is great gain for you today and builds a strong vision that will cleanly align with the “major language” features that have already been voted into the C++17 working paper
- In my template talk tomorrow, I will explain how C++17 template features not only help you know but help prepare your code for easier transition to concepts

A small but mighty example

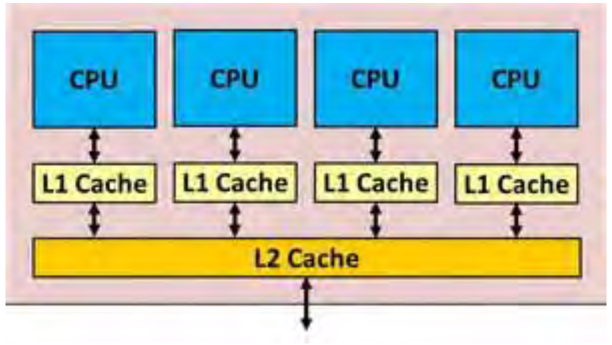
- C++17 adds `std::hardware_destructive_interference_size` and `std::hardware_constructive_interference_size` giving the maximum and minimum expected cache line sizes
- "**What?!?** I don't even know what that means"
- Virtually no articles on C++17 even mention this
- I expect that I will use it in almost every program I write and that it will make them run much faster
- In fact, I think it goes a long way to making C++ the best language for wherever performance matters, whether it is small embedded devices are large Big Data databases
- "**Really?!?** Okay, can you explain"

A couple of dirty secrets of computers

- Processors have not gotten faster for a decade
  - Essentially all of the improvement in performance of computers for the last 10 years have come from adding more cores, not making individual cores faster
- Even if you have a really fast processor with many cores, memory is so slow that they won’t have enough data to act on
  - Just a single load from main memory takes hundreds of clock cycles
- If you ignore either of these problems, your program will likely run like it is on decade-old hardware
  - For example, if it is single-threaded, multiple cores won’t by you anything
  - If you constantly read memory, it will be like having a fancy sports car with no fuel

Caches

- To make memory fast enough, computers have *caches* that store high-performance copies of data close to each core



- These caches can be hundreds of times faster than ordinary memory
- The problem is that if you are using multiple cores (and you need to), each processor needs to agree on what memory looks like
- If two processors are caching the same data, then they have to coordinate so much (“cache coherency protocols”) that it is no better than main memory, and your program is slow again

C++17 to the rescue

- C++ has always given you great control over memory, but now it understands caches
- hardware_destructive_interference_size tells you how far you need to keep objects used by different threads apart from each other
- hardware_constructive_interference_size tells you how close you need to keep objects for a single thread to get both of them with a single read of main memory
- I will talk about how to make the most of this in another talk this conference, but for now, I'd just like to state that major high-profile projects have seen 10x performance improvement by reimplemented in C++ primarily from careful control of memory layout
- Far from being some minor tweak, this is a great feature
  - You don't have to learn any new language constructs
  - It was data you had to manually get yourself before
  - Now your programs run faster (often much faster) more portably
  - If you are interested, my memory management talk later this summit has a clear set of rules for how to benefit from this in almost any program

String Views

- Another great example of this Simplification philosophy is string_view
- Every C++ function that works with text has always had to choose whether to use C strings (`char *`), C++ strings (`std::string`), or even character buffers
- Whatever you choose, half your users will be mad at you!
- Since nearly every program works with text, this is a huge issue!
- In C++17, string views make C strings look like C++ strings
-
```
size_t count_occurrences(string_view sv, char c) {
  size_t occurrences{}
  for(x : sv) { if(x == c) occurrences++; }
  return occurrences;
}
```
- All of the following work just fine
```
cout << occurrences("foo", 'o');
cout << occurrences({"foo", 3}, 'o');
cout << occurrences(string("foo"), 'o');
```

Wait a minute! I could do that with strings!

- That’s true
- If I had used the signature
  ```
  size_t occurrences(string, char);
  ```
- All of the same code would have worked!
- So what’s the big deal?
- If occurrences takes a string by value, like above, it copies all of the string’s data every time you call the function, which makes things really inefficient
- `occurrences(million_char_string, 'x’); // copies a million chars!`
- To avoid this problem, programmers invariably take the string by const reference
  ```
  size_t occurrences(string const &, char);
  ```
- But now it won’t work with C strings anymore

```
occurrences("foo", 'o'); // Illegal!
```

`string_view` solves everything perfectly

- By using `string_view`, you don't have to learn any new interfaces, functions, or algorithms, because they act like const strings
- Your clients can pass any kind of textual data to you
- The data that is passed in is never needlessly copied

“Wow! C++17 is starting to look pretty good”

- This is worth repeating
- Pretty much every program I right uses concurrency, so it will benefit from proper cache usage
  - Again, if you don’t know what that means, come to my memory talk later
- Pretty much every program I write works with text, so it will benefit from string views
- And,…
- We haven’t even gotten to any of the major features of C++17 yet!

Constructor Template Argument Deduction (CTAD)

- That sounds like a mouthful. Can I just ignore it?
- Well, it's a simple idea, and it will save you a mouthful in your code
- Basically, it says that constructors for class templates can deduce their arguments like other template functions do
- This makes a million things easier. We all initialize containers from initializer lists
  - `vector<double> v14 = { 1.2, 3.4 }; // C++14`
  - `vector v17 = {1.2, 3.4}; // C++17 figures out it's a vector<double>`
- How about getting locks in multithreaded programs?
  - `lock_guard<shared_lock<shared_timed_mutex>> lck14{smtx}; // Yuck!`
  - `lock_guard lck17{smtx}; // C++17. That's more like it!`

CTAD can even do more amazing deductions

- Suppose I want to create a vector from two iterators
- set s = {1, 2, 3, 4};
  vector v(set.begin(), set.end()); // Correctly gets vector<int>
- “How does it do that? It doesn’t seem possible”
- Well, I’ll talk about that in my Templates talk, but for now it’s enough to say that template deduction works better than ever in C++17
- One way I like to think about is that many of the most popular features of C++11 were those that simplify initialization, like initializer lists and auto
- Constructor Template Argument Deduction simplifies many initializations, so I expect it will be just as useful
- “This is great! Does C++17 improve initialization in other ways?”

Structured bindings

- As you will hear from Kate Gregory in a later talk, all of the information returned by your function should be in the return type
- Don’t use “output parameters” to return values

```
int  f(~~double &d,~~ X x); // Bad way to return an int and a double
```

- If you follow this best practice, many of your functions will need to return tuples to return all of its outputs

```
tuple<int, double>  f(X x); // Right way to return an int and a double
```

- The problem with this is that the caller probably doesn’t want to deal with a tuple. They want the results in two variables, which is clumsy in C++14

```
int i;
double d;
tie(i,d) = f(x);
```

- In C++17, *Structured Bindings* let us do better

```
auto [i, d] = f(x);  // Much clearer
```

Structured Bindings

- Structured bindings can be used in all the ways you expect

-
  ```
  tuple<int, double> tid{1, 2.3)
  auto const &[i,d] = tid;
  int ia[2] = 2, 3;
  auto & [first, second] = ia; // Give names to the array elements
  ```
- Again, this is a simple and natural feature that you will use all the time (as long as you follow the best practice of not using output parameters)

A taste of parallel algorithms*

- As we mentioned, modern programs get their performance from leveraging concurrency
- A big challenge is that writing multi-threaded code is hard
- Wouldn’t it be great if we could write code using STL algorithms like normal and the compiler could automatically leverage parallelism for us?
- In C++17, we can! The following sort automatically runs in parallel

  -
    ```
    vector huge_vector = getHugeVector();
    sort(parallel::par, huge_vector.begin(), huge_vector.end());
    ```
- The following STL algorithms can all run in parallel now, just by requesting it
  - adjacent difference, adjacent find. all_of, any_of, none_of, copy, count, equal, fill, find, for_each (return type changed to void), generate, includes, inner product, in place merge, merge, is heap, is partitioned, is sorted, lexicographical_compare, min element, minmax element, mismatch, move, nth_ element, partial_sort, partial_sort_copy, partition, remove + variations, replace + variations, reverse / rotate, search, set difference / intersection / union /symmetric difference, sort, stable partition, swap ranges, transform, unique

*Adapted from http://www.bfilipek.com/2017/08/cpp17-details-parallel.html

New parallel algorithms

- In addition to the above, new parallel algorithms have been added that correspond to those in functional languages like Scala
- The following and their descriptions are taken from http://www.bfilipek.com/2017/08/cpp17-details-parallel.html
- `for_each` - similar to std::for_each except returns void.
- `for_each_n` - applies a function object to the first n elements of a sequence.
- `reduce` - similar to std::accumulate, except out of order execution.
- `exclusive_scan` - similar to std::partial_sum, excludes the i-th input element from the i-th sum.
- `inclusive_scan` - similar to std::partial_sum, includes the i-th input element in the i-th sum
- `transform_reduce` - applies a functor, then reduces out of order
- `transform_exclusive_scan` - applies a functor, then calculates exclusive scan
- `transform_inclusive_scan` - applies a functor, then calculates inclusive scan

REAL Concurrency

- The parallelism described above is great, but it doesn't begin to describe what a big deal this is
- As Sean Parent has pointed out
  - Less than 1% of the processing power on a modern computer is on a single core
    - Concurrency is essential!
  - Even multiple cores get you less than 10% of the processing power
    - Multithreading alone isn't enough
  - With vectorization like SSE, you can take advantage of about a quarter of the computing power on your system
  - When you add in code to leverage your GPU, you can fully utilize your system
- "I want to do this, but it sounds hard!"
- With parallel algorithms, all you need to do is change the execution policy to parallel::par_unseq, and the compiler is allowed to vectorize your code and leverage your GPU, potentially fully utilizing your system
- `sort(parallel::par_unseq, huge_vector.begin(), huge_vector.end());`
- **Warning:** Compilers are still learning how to do this optimally, but if you write your code this way now, it will automatically get faster as compilers improve

Libraries, libraries, libraries!

- One of the reasons that people use languages like Java is not necessarily for the language, but the comprehensive set of standard libraries that is much larger than C++
- A goal of C++17 was to start filling in the “missing” C++ libraries
- This will be a long process (it will take several C++ releases), but C++17 takes a big step on the way there
- Let’s touch on some of the best ones

Filesystem

- C++’ file handling has always been terrible
  - There has never been a portable way to list the files in a directory, etc.
  - I can’t begin to think of how many times I’ve wished for this
- No more!
- C++17’s filesystem library offers full support for working your way through the filesystem
  - It understands directories, symbolic links, hard links, named pipes, block vs. character device,…
  - In other words, all of the things that modern filesystems have
- Here is some code to find the oldest file in a directory (say if you are managing a directory of log files)

  -
    ```
    max_element(directory_iterator("/foo/bar"), directory_iterator(),
        [](directory_entry const &e1, directory_entry const &e2)
            { return e1. last_write_time() > e2.last_write_time(); }
    ```

Library support for better object management

- C++17 provides three libraries that solve important problems that constantly arise when working with values
  - `variant`
  - `optional`
  - `any`
- Together, they solve problems for manipulating values that occur in every program
- Let’s take a look

Variants

- C++ inherited unions from C, but they are low level and dangerous
  - You will likely crash if you access a union with the wrong type
  - You can’t find out what type of object is in the union
  - Doesn’t do the right thing if an exception is thrown when assigning a value
  - etc.
- However, we have all used unions just because they are so darn useful!
  - It has pretty much been the only way to get an object whose type can vary dynamically at runtime in a way that is not just inheritance
  - While it is great that C++ is statically typed, sometimes you do need dynamically typed objects
- C++17 has added a safe, strongly-typed, discriminated union called `variant`

Variants: Example from cppreference.com

```
int main()
{
    std::variant<int, float> v, w;
    v = 12; // v contains int
    int i = std::get<int>(v);
    w = std::get<int>(v);
    w = std::get<0>(v); // same effect as the previous line
    w = v; // same effect as the previous line

  //  std::get<double>(v); // error: no double in [int, float]
  //  std::get<3>(v);      // error: valid index values are 0 and 1
    try {
      std::get<float>(w); // w contains int, not float: will throw
    }
    catch (const std::bad_variant_access&) {}

    std::variant<std::string> x("abc"); // converting constructors work when unambiguous
    x = "def"; // converting assignment also works when unambiguous


    std::variant<std::string, bool> y("abc"); // casts to bool when passed a char const *
    assert(std::holds_alternative<bool>(y)); // succeeds
    y = "xyz"s;
    assert(std::holds_alternative<std::string>(y)); //succeeds
}
```

Optional

- One often wants to know if a variable has been initialized with a value
- The way to do this in C++14 is to use a pointer (or smart pointer), so we can check if it is null
- Frankly, this is an abuse of pointers, since we aren’t really interested in pointing to objects
- Pointers (even smart ones) are one of the scariest concepts in C++ and delay new programmers ability to start producing usable code
- What we really need here is the notion of an “optional” value in C++, which is exactly what C++17 gives us
- `std::optional<Foo>` represents an optional object of type Foo
- You can check if it is there, and then use it if it is

Optional: Example

-
  ```
  optional<string> ostr = "foo"; // Contains a foo
  cout << ostr.value(); // Prints ostr
  ostr.reset(); // No longer contains a value
  cout << ostr.value_or("empty"); // Prints "empty"
  cout << ostr.value(); // Throws exception due to missing value
  ```

Any type

- Sometimes optional isn’t enough, and you need a fully dynamic type
- C++17 provides that with std::any
- any a = 7; // a contains an int
  cout << any_cast<int>(a); // Prints 7
  a = "foo"s; // a contains a string
  cout << any_cast<string>(a); // prints "foo"
  cout << any_cast<int>(a); // Throws an exception

Scoped locks

- A lot of times, you need to acquire multiple locks at one
- This can be an enormous headache in C++14

  -
    ```
    void swap(MyType const &l, MyType const &r)
    {
        std::lock(l.mtx, r.mtx);
        std::lock_guard<std::mutex> llck(l.mtx, std::adopt_lock);
        std::lock_guard<std::mutex> rlck(r.mtx, std::adopt_lock);
        /* ... */
    }
    ```
- But is a snap in C++17

  -
    ```
    void swap(MyType const &l, MyType const &r)
    {
        std::scoped_lock lck(l.mtx, r.mtx);
        /* ... */
    }
    ```

Many more language features!

- Template features like template<auto>, fold expressions, better support for constexpr evaluation of non-type template arguments, constexpr if, etc.
  - I will say more about these in my template talk
- Making `static_assert` more usable
- Copy elision
  - avoids unnecessary copying of arguments, transparently making many programs perform faster
- Nested namespaces: namespace X::Y { ... }
- Initialize variables in `if` and `switch`
  - `if(int i = foo()) do_something_with_i(i)`
- Inline variables
- Check if a header file exists, so you don't get a `#include` error
- While I can't cover all of these, they generally follow the principle I stated above of making the language more powerful without requiring the programmer to relearn the language

Many more library features

- Uniform container access with `std::size`, `std::empty`, and `std::data`
- New algorithm to `sample` data
  - Expect to be really handy in todays world of data science and statistical learning
- `invoke` allows you to directly call any callable object on a list of arguments
- `void_t` revolutionizes metaprogramming but takes a lecture to explain
  - Best way is to watch a video of Walter Brown explaining his awesome invention
- More type traits
- Efficient calculation of greatest common divisor, and least common multiple
- Polymorphic allocators
- Fast string searching like Boyer-Moore
- Many more
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0636r2.html#new-lib

Removals

- One other way that C++17 is simplifying the language is by removing old features that are no longer needed
- This is controversial
  - Helps simplify the language, frees up design space, makes it easier for programmers not to stumble on the wrong thing, makes "bad code" get fixed, etc.
  - But can break compatibility
  - Let me know what you think
- Binders like bind1st have been removed because lambdas are a better approach
- auto_ptr is gone
  - It is broken and unique_ptr is better
  - You can use clang::tidy to automatically update your code
- random_shuffle, allocator support in function, the register keyword, etc. were all removed
  - These are all broken, and you should remove any places you are using them

My thoughts and path forward

- C++17 really does make C++ both much more powerful and simpler
- A successful release that is well worth learning
- **But,** we can't afford another successful release like this
- C++20 needs to get in some of the transformative features like Concepts, Modules, Coroutines, and more to move the language forward
- Please go to Mike Wong's keynote tomorrow for a great perspective on how to move future releases of C++ where then need to go

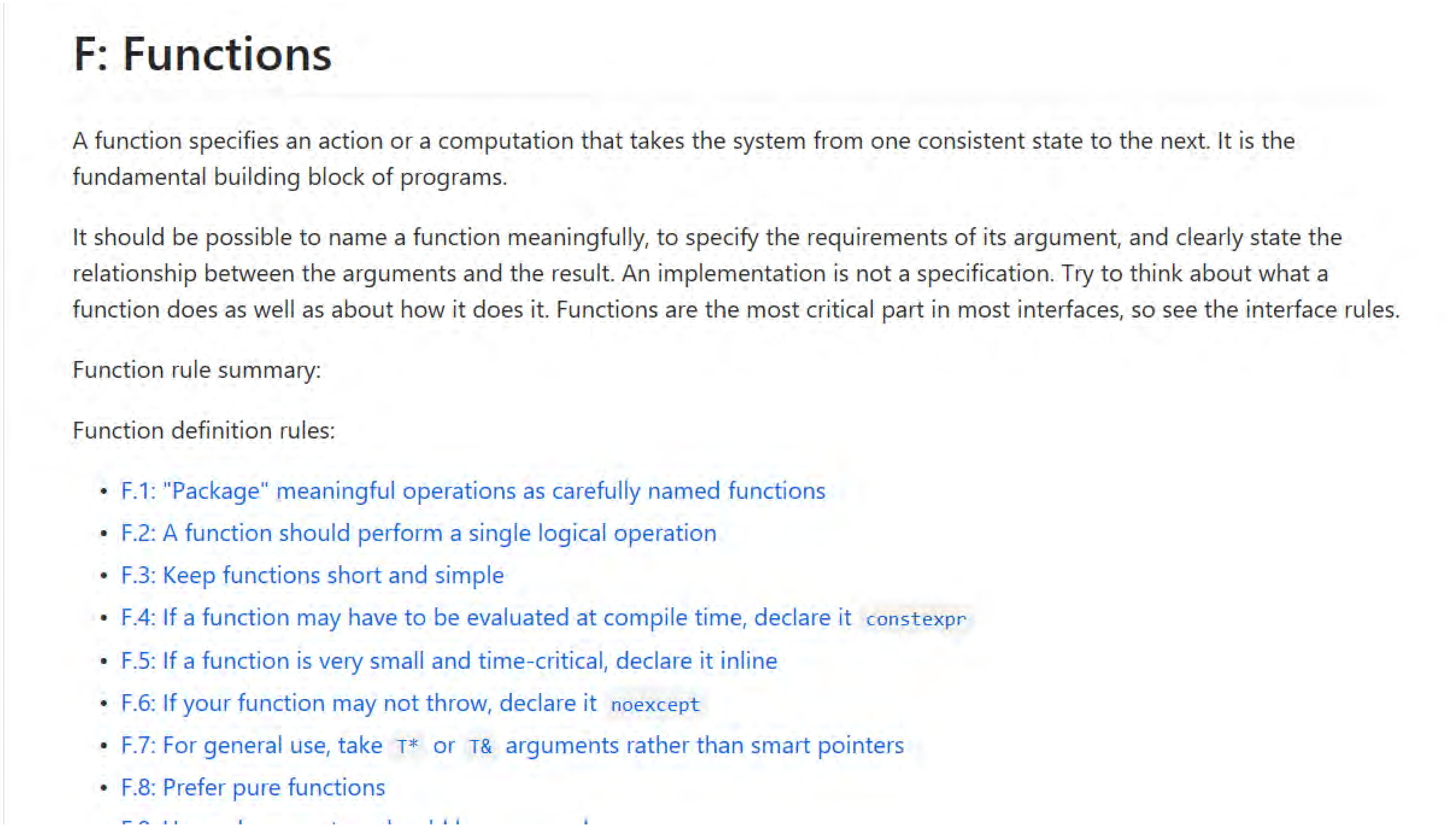
2017 CPP-Summit

10 Core Guidelines You Need to Start Using Now

Kate Gregory
@gregcons

CORE GUIDELINES

- Announced two years ago, at CppCon
- https://github.com/isocpp/CppCoreGuidelines
- CppCoreGuidelines.md is the “good stuff”
- http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines slightly more readable form



F: Functions

A function specifies an action or a computation that takes the system from one consistent state to the next. It is the fundamental building block of programs.

It should be possible to name a function meaningfully, to specify the requirements of its argument, and clearly state the relationship between the arguments and the result. An implementation is not a specification. Try to think about what a function does as well as about how it does it. Functions are the most critical part in most interfaces, so see the interface rules.

Function rule summary:

Function definition rules:

- F.1: "Package" meaningful operations as carefully named functions
- F.2: A function should perform a single logical operation
- F.3: Keep functions short and simple
- F.4: If a function may have to be evaluated at compile time, declare it `constexpr`
- F.5: If a function is very small and time-critical, declare it inline
- F.6: If your function may not throw, declare it `noexcept`
- F.7: For general use, take `T*` or `T&` arguments rather than smart pointers
- F.8: Prefer pure functions