张洋

2017 年 8 月 10 日

目录

0.1 Future Ruminations

http://sean-parent.stlab.cc/2017/07/10/future-ruminations.html

0.1.1 Background

This post is a lengthy answer to a question from Alisdair Meredith:

How do we know if the primitives are adequate until we know the feature set that must be supported? —Alisdair Meredith (@AlisdairMered) July 10, 2017

The question is regarding the numerous proposals for a better future class template for C++, including the proposal from Felix Petriconi, David Sankel, and myself.

It is a valid question for any endeavor. To answer it, we need to define what we mean by a future so we can place bounds on the solution. We also need to understand the problems that a future is trying to solve, so we can determine if a future is, in fact, a useful construct for solving those problems.

The proposal started with me trying to solve a fairly concrete problem; how to take a large, heavily threaded application, and make it run in a single threaded environment (specifically, compiled to asm.js with the Emscripten compiler) but also be able to scale to devices with many cores. I found the current standard and boost implementation of futures to be lacking. I open sourced my work on a better solution, and discussed this in my Better Code: Concurrency talk. Felix heard my CppCast interview on the topic, and became the primary contributor to the project.

0.1.2 Futures

Conceptually, a future is a way to separate the result of a function from the execution of the function. The task (the function packaged so it returns void) can be executed in a different context (the execution context is controlled by executors in some of the proposals) and the result will become available via the future.

A future also serves as a handle to the associated task, and may provide some operation to control the task.

The primary advantage of a future over a simple callback is that a callback requires you to provide the subsequent operation in advance. Where a future allows you to provide a continuation, via a then() member function, at some later point. This feature makes futures easier to compose, easier to integrate into an existing system, and more powerful as they can be stored and the continuation can be attached as the result of another action, later. However, this flexibility comes with inherent cost, it requires an atomic test when the continuation is attached to

determine if the value is already available. Because of this cost, for many library operations it makes sense to provide a form taking a callback as well as one returning a future. Although at first glance it may appear that a callback from is easily adapted to a future form, that is not the case for reasons discussed below.

0.1.3 Design Issues

Because a future is a token for a type T, we can look at the regular properties of types in evaluating if we have an efficient basis for a future. In order to be a reasonable proxy for a value of type T, we must be able to pass a future to a function, return it from a function, and store it as the member of an object. But should a future be copyable if T is copyable? One thought is that copying should not be handled specially, and should be done on the value of T within a continuation. To illustrate, here is an initial attempt to split a value using the current WG21 proposal for non-copyable futures.

```
// 'split' fut to call two functions, f & g
auto a = fut.then([](auto x){
        auto value = x.get();
        return make_pair(f(value), g(value));
});
```

Immediately we can see the problem with the approach. Even though f and g are independent operations, they will be serialized and the result will not be available until both f and g are complete. If we want to continue calculations down separate paths, those paths will continue to be packaged together.

A better option is to write a form of copy to split or fork the future. Here is a simple implementation (which ignores exception for illustrative purposes only):

```
template <class T>
auto split(future<T>& x) {
    auto tmp = std::move(x);
    promise<T> p;
    x = p.get_future(); // replace x with new future
    return tmp.then([_p = move(p)](auto _tmp) mutable {
        auto value = _tmp.get();
        _p.set_value(value); // assign to new "x" future
        return value; // return value through future result
    });
}
```

This can then be called as:

```
// 'split' fut to call two functions, f & g
auto a = split(fut).then([](auto x) { return f(x); });
auto b = fut.then([](auto x){ return g(x); });
```

Using split() we can fan out our parallel computations. However, split has a significant disadvantage over a copy, it requires that the argument being split is mutable. Being able to copy immutable objects is a fundamental building block for reasoning about parallel computations, having split() be our only way to copy means that copying larger constructs containing futures requires additional synchronization.

Another approach is to introduce a new type such as `shared_future` (in my opinion this is a horrible misnomer). It is a potential option if implementing copy, even when not called, requires additional overhead. I do not know that this has been demonstrated (and do not believe it to be true). An approach more inline with the current language is to optimize for the rvalue case to provide performance.

There may well be good reasons to have a concept for a future with many type models. And then a type erased `any_future<>` similar to the relationship between function objects and `std::function<>`. However such type erasure comes at a cost, and it needs to be demonstrated that there are enough useful models of a Future concept and potential optimization benefits to have them be distinct types to warrant such an approach. I do not believe a copy basis operation rises to this level.

So for basis operations on a future we have:

1. default ctor
2. construct with a value
3. construct with associated task
4. move ctor and move assignment
5. copy ctor and copy assignment
6. equality/inequality

For the later, two futures are equal if they refer to the same shared state (copies are equal).

0.1.4 Future as Task Handle

If the result of the task is no longer necessary the task can be canceled, and if it hasn' t yet started to execute then it can be destructed releasing all subordinate resources. This is a way to provide RAAII to computation, frequently the most valuable resource in the machine. The progress of a task may be made available, specifically if it is complete (the future is ready), if it has started, and this attribute could be tracked through subordinate tasks to determine an estimate towards completion. Finally, the context within which a task is to be executed may

be changed, the most common change is to promote a task to execute with a higher priority or immediately within the current execution context.

0.1.5 Getting Value from Future

Since a future is a token for a value, there needs to be some mechanism to retrieve the value. Starting with C++11, futures have a get() function and various forms of wait() functions to retrieve the value. However, both get() and wait() are very problematic operations. Unless the associated task can be promoted to immediate execution, these operations may deadlock if the associated task is unable to run concurrently with the waiting task. This can happen for numerous reasons:

- The system doesn’ t have thread support and tasks are executed on a main serial queue.
- The tasks are scheduled within a thread pool and all threads within the pool are similarly blocked on get() or a condition variable.
- The number of available threads within the system is exhausted.

In order to promote a task, the task must be fully resolved, all of it’ s arguments are known, if the tasks is required to be serialized with other tasks then the system must assure that all prior tasks, and the task cannot require execution within a context other than the current context.

These guarantees can be provided by `std::async()` which is why it is permissible to implement `std::async()` with the `std::launch::async` policy on a thread pool by implementing task promotion and promotion is the mechanism by which `std::launch::deferred` tasks work (deferred can be viewed as promoting from execute never to execute immediately). However, neither of this is implementable through the `std::promise` mechanism because `std::promise` does not provide a task handle and so has no way to implement promotion.

In the presence of continuations, `get()` is more problematic because the task associated with a continuation future is not resolved. Promotion of such a task would require first promoting all upstream tasks. This is why many of the proposals, including ours, for better futures with continuations remove the `get()` and `wait()` operations. Even if implementable, they may be undesirable (see discussion on why context matters).

0.1.6 Cancellation

Another question is what should happen when the future destructs? C++11 has three answers:

- Futures returned from `std::async` with `std::launch::async` will block (wait) on destruction until the task completes.

- Futures returned from `std::async` with `std::launch::deferred` will cancel the associated task and free the resources.
- Future obtained from an `std::promise` will detach the associated task, allowing it to run to completion but drop any result.

If we bias our design towards functions whose behavior is not defined by their side-effects then the option of canceling on destruction makes the most sense as a default behavior. A task may be canceled even if it is not resolved, so cancellation of a future associated with a continuation has the effect of "unraveling" the dependency graph through associated tasks. However, for this to be the default behavior, the default mechanism to create a future should not be through something like `std::promise` which cannot act as a task handle, but rather should be a function that splits a function into a task and future.

```
template <class Sig, class F>
auto package(F&& f) -> std::pair<task_t<Sig, F>, future<task_result_t<Sig, F>>
```

Often cancellation is discussed as being a mechanism which is separate from futures. Usually with some handwaving "just have a shared atomic flag you check for cancellation." The problem with such an approach is that in order to cancel a task, you have to know if the result of the task is needed or not. In the presence of splits or copies, that information cannot be known locally. In a large system, you do not know what to cancel, only what is not needed locally. Consider an animation system which is generating a stream of futures for frames. If a frame is not ready on time it is canceled (dropped), but any calculations for that frame which are needed by subsequent frames should continue.

Because `std::promise` does not provide a task handle, there is no way to implement cancellation or blocking behaviors with just `std::promise` as a primitive. Composing an `std::launch::deferred` task with a promise it is possible to build a task that cancels on destruction. This `cancelable_task()` function is called in a similar manner to `package()` defined above.

```
template <class Sig, class F>
auto cancelable_task(F&& f) {
    using shared_t = std::packaged_task<Sig>;

    auto p = std::make_shared<shared_t>(std::forward<F>(f));
    auto w = std::weak_ptr<shared_t>(p);
    auto r = p->get_future();

    return std::make_pair([_w = std::move(w)] (auto&&... args) {
        if (auto p = _w.lock()) (*p)(std::forward<decltype(args)>(args)...);
    },
```

```
    std::async(std::launch::deferred, [_p = std::move(p), _r = std::move(r)] () mutable {
        return _r.get();
    }));
}
```

The problem of promise not providing a task handle is why you cannot simply adapt a callback based interface into one returning a future by providing a promise in the callback. Although callbacks can be more efficient in some cases, they do not provide a cancellation mechanism and the transformation from “asynchronous function with callback” to “cancelable task with future” is more complicated.

0.1.7 Context Matters

The context within which a task is executed may be of significance for a number of reasons including:

- Side-effects which need to be serialized with other tasks (part of a sequential process).
- Interruption of a time critical system.
- Access to specific resources (such as access to thread local variable on a specific thread or access to hardware resources such as a GPU).

A common design for futures is to default to immediate (or inline) execution for continuations and consider execution context as something that can always be “composed in” later. The problem with this is that immediate execution on a continuation will execute either in the context of the associated task, or in the context of the future. In the case of an asynchronous task, the exact context will be determined by execution performance. In practice, for a given piece of code on a given machine, it will almost always execute in one of those contexts, which is determined by the latency between the asynchronous invocation and when the continuation is attached.

My experience in working with teams using such a mechanism, and even in discussion with experts, is that they will make assumptions about which of the two contexts the code will execute in, and introduce race conditions. Even in the absence of a direct race condition they may inadvertently introduce a performance issue by interrupting a time critical system with a long task. For example, let’ s say that I execute expensive operation as continuation. The continuation is attached within the main event loop of an application, in testing, the continuation always executes in the context of the associated task and there are no performance problems. After the application is shipped, new hardware, or a new OS version, is released which changes the timing of the asynchronous operation so now it complets before the continuation is attached. Suddenly the continuation is executed in the context of the main event loop, and delays processing of user events causing the interactive performance of the system to become unusable.

For someone coming from a background where they have used promises in a language such as ES6, where continuations are always queued, this can be very surprising behavior.

In practice, immediate execution is nearly always a poor default and is of value only when the continuation is "small" (will execute quickly) and has no side effects or external dependencies. It is better to default to execute the task inside of a task system (such as a task stealing thread pool) and force the developer to explicitly choose immediate execution. Having a bad default forces a lot of code like this:

```
auto a = fut.then([](auto x){
    return async(default_executor, [_x = x.get()]{ f(_x); });
});
```

Which could otherwise just be:

```
auto a = fut.then(f);
```

If the default is not immediate execution, then there must be a way to override the default, otherwise there is no way to obtain immediate execution. I do not believe that the issue of context (executors) can be meaningfully separated from futures with continuations.

Unfortunately, this is not a universal answer, there are problems where futures are useful, which have nothing to do with asynchronous development and you want a fast implementation that doesn't require synchronization and always executes continuations immediately. A channel as a stream of futures for connecting co-routines as communicating sequential processes is one possible example. As of yet, I don' t have enough experience with implementing this to fully understand the impact on the design of futures.

0.1.8 Conclusions

There is a lot of industry experience with promise/future systems and they have become popular because they allow fairly straightforward transformation of sequential code into asynchronous code without needing to provide explicit synchronization. With continuations, futures can scale from single threaded environments to large grid systems with very high performance. This makes them an attractive basis upon which to build. By viewing futures as a token for a value of type T and as a handle for an associated task, we have a reasonable framework to explore a bounded feature set and make the engineering tradeoffs between performance and capabilities. In general, we shouldn' t drop a capability unless there are no compelling use cases, or there is a demonstrable performance implication, and in the later case the performance implications must be weighed against the value of the use cases and if there is reasonable alternative implementations for those use cases.

0.2 What does the [[carries_dependency]] attribute mean?

[[carries_dependency]] is used to allow dependencies to be carried across function calls. This potentially allows the compiler to generate better code when used with std::memory_order_consume for transferring values between threads on platforms with weakly-ordered architectures such as IBM's POWER architecture.

In particular, if a value read with memory_order_consume is passed in to a function, then without [[carries_dependency]], then the compiler may have to issue a memory fence instruction to guarantee that the appropriate memory ordering semantics are upheld. If the parameter is annotated with [[carries_dependency]] then the compiler can assume that the function body will correctly carry the dependency, and this fence may no longer be necessary.

Similarly, if a function returns a value loaded with memory_order_consume, or derived from such a value, then without [[carries_dependency]] the compiler may be required to insert a fence instruction to guarantee that the appropriate memory ordering semantics are upheld. With the [[carries_dependency]] annotation, this fence may no longer be necessary, as the caller is now responsible for maintaining the dependency tree.

```
void print(int * val)
{
    std::cout<<*p<<std::endl;
}

void print2(int * [[carries_dependency]] val)
{
    std::cout<<*p<<std::endl;
}

std::atomic<int*> p;
int* local=p.load(std::memory_order_consume);
if(local)
    std::cout<<*local<<std::endl; // 1

if(local)
    print(local); // 2

if(local)
    print2(local); // 3
```

In line (1), the dependency is explicit, so the compiler knows that local is dereferenced, and that it must ensure that the dependency chain is preserved in order to avoid a fence on POWER.

In line (2), the definition of print is opaque (assuming it isn't inlined), so the compiler must issue a fence in order to ensure that reading *p in print returns the correct value.

On line (3), the compiler can assume that although print2 is also opaque then the dependency from the parameter to the dereferenced value is preserved in the instruction stream, and no fence is necessary on POWER. Obviously, the definition of print2 must actually preserve this dependency, so the attribute will also impact the generated code for print2.

0.3 declare your variables as const

`http://www.bfilipek.com/2016/12/please-declare-your-variables-as-const.html`

I need to confess that for the last few years I' ve been a bit obsessed with the idea of making all variables const. Whenever I declare a variable in a function body, I try to think if I can make it constant. Let me explain why I think you should be doing the same.

0.3.1 What' s wrong?

What' s wrong with the following code?

```
int myVariable = 0;

// some code...

myVariable = ComputeFactor(params...);
```

Versus:

```
// some code...

const int myVariable = ComputeFactor(params...);
```

In the first sample, we' re just changing the value of some variable, and that' s typical thing in the code⋯isn' t it?

Let' s go through the list of benefits of the second approach.

Please note that I' ll focus only on variables used in function bodies, not parameters of functions, or class members.

0.3.2 Why it helps

0.3.2.1 Performance?

Several years ago, my colleague suggested using const for variables. I though the only reason for this was optimization and performance. Later, I understood that' s it' s not that obvious, and there are far more important reasons for using const.

In fact, a good C++ compiler can do the same kind of optimization no matter you use const or not. The compiler will deduce if a variable is changed or just initialized once at the start. So, is there any performance benefit here?

It's hard to show the real numbers here. Ideally, we could get a C++ project (let say minimum 10k LOC) and then use const whenever possible, and compare it against the same project without const.

In a synthetic, small examples like:

```
string str;
str = "Hello World";
```

VS

```
const string str = "Hello World";
```

There can be a performance increase of even 30%! Numbers from J. Turner talk "Practical Performance Practices". As one commenter noticed: the gain comes not from the const itself, but from the fact that we're not reassigning the value.

As we can see, there's a potential in getting some performance, but I wouldn't expect much across the whole project. It depends on the context. Maybe something like 1…or 2% max. As usual: measure measure measure! :)

Still, why not making life much easier for the compiler and have better code.

So, it seems that the "performance" is not the strongest reason for using const, read below for far more important aspects:

0.3.2.2 Variables are declared local to their use

If you want to declare a constant variable, you need to have all the required data available. That means you cannot just declare it at the beginning of a function (like in standard old C-way). Thus, it's a higher chance to have variables quite local to their actual usage.

```
void foo(int param)
{
    const int otherVariable = Compute(param);
    // code...

    // myVar cannot be declared before 'otherVariable'
    const int myVar = param * otherVariable;
}
```

Declaring variables local to their use is not only a good practice but can result in less memory usage (since not all variables might be allocated) and even safer code.

0.3.2.3 Clear Intent

When you declare something as constant you make it clear "I won't change the value of that variable."

Such practice is vital when you read the code. For example:

```
int myVar = 0;

// code...

// code...
```

When you see such a thing, you're not sure if myVar will change or not. It might not be a problem in small functions, but what about longer, complex methods?

While having:

```
const int myVar = ...;

// code...
```

You're at least sure that nothing happens with myVar. You get one parameter less to track.

0.3.2.4 Clean Code

Sometimes the initialization of a variable won't be just a simple assignment. Several lines (or more) might be used to give a proper value. In that case making the variable const will force you to move such initialization to a separate place.

As I described in IIFE for Complex Initialization you might enclose the initialization in IIFE or a different method. Anyway, you'll avoid code looking like that:

```
int myVariable = 0;

// code...

// complex initialization of 'myVariable'
if (bCondition)
    myVariable = bCond ? computeFunc(inputParam) : 0;
else
    myVariable = inputParam * 2;

// more code of the current function...
```

No matter what you use, you'll end up with only one place where the variable gets its value.

0.3.2.5 Fewer bugs

When a variable is const you cannot change it, so some unwanted bugs are less likely to happen.

Accidental problems might easily happen when there’ s some long function and variables tend to be reused in some cases. You change the value of a variable, and it works for your case, but the old case where it was used now stops working. Again, declaring a variable as const will at least protect you from such stupid bugs. Not to mention that debugging such errors might be a real pain.

BTW: for an example please see this blog posts from Andrzej Krzemienski: More const—fewer bugs

0.3.2.6 Moving towards functional languages

Functional style is probably a topic worth a separate article, but in general having immutable objects is an essential thing in functional languages.

Immutable objects are thread safe by their nature. When a thread processes that kind of objects, we can be sure that no other threads are changing the objects. Lots of data races can be avoided. That opens many ways to parallelize the algorithm relatively easy.

0.3.3 Exceptions

‘A constant variable’ isn’ t that an oxymoron?

Of course, there are situations where a variable need to be a ‘normal.’ In fact, you might argue that most of the cases involve the need to modify a value. So unless you’ re trying to write functional code (that likes immutability), you’ ll end up with tons of examples when you need to change a value (or just part of an object).

Simple examples: calculating a sum of an array, iterators, small functions, changing health param in GameActor, setting a part of GPU pipeline.

Still, bear in mind that the most of the above examples could be rewritten into an ‘immutable’ version as well. For example, you can use higher-order functions like Fold/Reduce, and recursion to implement many ‘standard’ algorithms. But that’ s going into functional languages area.

One remark: while I was writing this article I realized I make a distinction here: variables vs. larger objects. In theory, those are the same, but for practical reasons, it’ s easier to use const on smaller, ‘atomic’ types. So, I try to use const for smaller types: like numerics, strings, Vector2d, etc⋯but when I have some large custom class, I just skip const and allow to mutate its state (if needed). Maybe in my next iteration of my ‘const correctness’ I’ ll try to apply that rule also on larger objects⋯so this would be a more functional style of programming.

第一章　C++ 17 Features

http://www.bfilipek.com/2017/01/cpp17features.html

This year we’ ll get a new version of C++: C++17!

In this mega long article I’ ve built a list of all features of the new standard.

Have a look and see what we get!

1.1 Intro

Updated: This post was updated 16th June 2017.

The list is mostly done! Still some descriptions could be improved or more example could be provided.

If you have code examples, better explanations or any ideas, let me know! I am happy to update the current post so that it has some real value for others.

The plan is to have a list of features with some basic explanation, little example (if possible) and some additional resources, plus a note about availability in compilers. Probably, most of the features might require separate articles or even whole chapters in books, so the list here will be only a jump start.

See this github repo: github/fenbf/cpp17features. Add a pull request to update the content.

The list comes from the following resources:

1. SO: What are the new features in C++17?
2. cppreference.com/C++ compiler support.
3. AnthonyCalandra/modern-cpp-features cheat sheet - unfortunately it doesn’ t include all the features of C++17.
4. plus other findings and mentions

And one of the most important resource: Working Draft, Standard for Programming Language C++

Plus there’ s an official list of changes: P0636r0: Changes between C++14 and C++17 DIS

I am also working on a bit detailed series:

1. Fixes and deprecation

2. Language clarification
3. Templates
4. Attributes
5. Simplification
6. Library changes - Filesystem
7. Library changes 2 (soon)
8. Library changes 3 (soon + 1)
9. Wrap up, Bonus

1.2 Language Features

1.2.1 New auto rules for direct-list-initialization

Fixes some cases with auto type deduction. The full background can be found in Auto and braced-init-lists, by Ville Voutilainen.

It fixes the problem of deducing `std::initializer_list` like:

```
auto x = foo(); // copy-initialization
auto x{foo}; // direct-initialization, initializes an \mintinline{c++}{initializer_list}
int x = foo(); // copy-initialization
int x{foo}; // direct-initialization
```

And for the direct initialization, new rules are:

1. For a braced-init-list with only a single element, auto deduction will deduce from that entry;
2. For a braced-init-list with more than one element, auto deduction will be ill-formed.

Basically, auto x 1 ; will be now deduced as int, but before it was an initializer list.

1.2.2 `static_assert` with no message

Self-explanatory. It allows just to have the condition without passing the message, version with the message will also be available. It will be compatible with other asserts like `BOOST_STATIC_ASSERT` (that didn't take any message from the start).

1.2.3 typename in a template template parameter

Allows you to use typename instead of class when declaring a template template parameter. Normal type parameters can use them interchangeably, but template template parameters were restricted to class, so this change unifies these forms somewhat.

```
template <template <typename...> typename Container>
//            used to be invalid ^^^^^^^^
struct foo;

foo<std::vector> my_foo;
```

1.2.4 Removing trigraphs

Removes ??=, ??(, ??>, ···

Makes the implementation a bit simpler, see MSDN Trigraphs

1.2.5 Nested namespace definition

Allows to write:

```
namespace A::B::C {
   //...
}
```

Rather than:

```
namespace A {
    namespace B {
        namespace C {
            //...
        }
    }
}
```

1.2.6 Attributes for namespaces and enumerators

Permits attributes on enumerators and namespaces. More details in N4196.

```
enum E {
  foobar = 0,
  foobat [[deprecated]] = foobar
};
```

```
E e = foobat; // Emits warning

namespace [[deprecated]] old_stuff{
    void legacy();
}

old_stuff::legacy(); // Emits warning
```

1.2.7 u8 character literals

UTF-8 character literal, e.g. u8'a'. Such literal has type char and the value equal to ISO 10646 code point value of c-char, provided that the code point value is representable with a single UTF-8 code unit. If c-char is not in Basic Latin or C0 Controls Unicode block, the program is ill-formed.

The compiler will report errors if character cannot fit inside u8 ASCII range.

Reference:

1. cppreference.com/character literal
2. SO: What is the point of the UTF-8 character literals proposed for C++17?

1.2.8 Allow constant evaluation for all non-type template arguments

Remove syntactic restrictions for pointers, references, and pointers to members that appears as non-type template parameters:

For instance:

```
template<int *p> struct A {};
int n;
A<&n> a; // ok

constexpr int *p() { return &n; }
A<p()> b; // error before C++17
```

1.2.9 Fold Expressions

More background here in P0036

Allows to write compact code with variadic templates without using explicit recursion.

Example:

```
template<typename... Args>
auto SumWithOne(Args... args){
    return (1 + ... + args);
}
```

Articles:

1. C++ Truths: Folding Monadic Functions
2. Simon Brand: Exploding tuples with fold expressions
3. Baptiste Wicht: C++17 Fold Expressions
4. Fold Expressions - ModernesCpp.com

1.2.10 Unary fold expressions and empty parameter packs

If the parameter pack is empty then the value of the fold is:

Operator	Value
&&	true
||	false
,	void()

For any operator not listed above, an unary fold expression with an empty parameter pack is ill-formed.

1.2.11 Remove Deprecated Use of the register Keyword

The register keyword was deprecated in the 2011 C++ standard. C++17 tries to clear the standard, so the keyword is now removed. This keyword is reserved now and might be repurposed in the future revisions.

1.2.12 Remove Deprecated operator++(bool)

The ++ operator for bool was deprecated in the original 1998 C++ standard, and it is past time to remove it formally.

1.2.13 Removing Deprecated Exception Specifications from C++17

Dynamic exception specifications were deprecated in C++11. This paper formally proposes removing the feature from C++17, while retaining the (still) deprecated throw() specification strictly as an alias for noexcept(true).

1.2.14 Make exception specifications part of the type system

Previously exception specifications for a function didn' t belong to the type of the function, but it will be part of it.

We' ll get an error in the case:

```
void (*p)();
void (**pp)() noexcept = &p;   // error: cannot convert to pointer to noexcept function
```

```
struct S { typedef void (*p)(); operator p(); };
void (*q)() noexcept = S();   // error: cannot convert to pointer to noexcept function
```

1.2.15 Aggregate initialization of classes with base classes

If a class was derived from some other type you couldn't use aggregate initialization. But now the restriction is removed.

```
struct base { int a1, a2; };
struct derived : base { int b1; };

derived d1{{1, 2}, 3};       // full explicit initialization
derived d1{{}, 1};           // the base is value initialized
```

To sum up: from the standard: An aggregate is an array or a class with:

1. * no user-provided constructors (including those inherited from a base class),
2. * no private or protected non-static data members (Clause 11),
3. * no base classes (Clause 10) and // removed now!
4. * no virtual functions (10.3), and
5. * no virtual, private or protected base classes (10.1).

1.2.16 Lambda capture of *this

this pointer is implicitly captured by lambdas inside member functions (if you use a default capture, like [&] or [=]). Member variables are always accessed by this pointer.

Example:

```
struct S {
   int x ;
   void f() {
      // The following lambda captures are currently identical
      auto a = [&]() { x = 42 ; } // OK: transformed to (*this).x
      auto b = [=]() { x = 43 ; } // OK: transformed to (*this).x
      a();
      assert( x == 42 );
      b();
      assert( x == 43 );
   }
};
```

Now you can use *this when declaring a lambda, for example `auto b = [=, *this]() { x = 43 ; }`. That way this is captured by value. Note that the form `[&,this]` is redundant but accepted for compatibility with ISO C++14.

Capturing by value might be especially important for async invocation, paraller processing.

1.2.17 Using attribute namespaces without repetition

Other name for this feature was "Using non-standard attributes" in P0028R3 and PDF: P0028R2 (rationale, examples).

Simplifies the case where you want to use multiple attributes, like:

```
void f() {
    [[rpr::kernel, rpr::target(cpu,gpu)]] // repetition
    do-task();
}
```

Proposed change:

```
void f() {
    [[using rpr: kernel, target(cpu,gpu)]]
    do-task();
}
```

That simplification might help when building tools that automatically translate annotated such code into a different programming models.

1.2.18 Dynamic memory allocation for over-aligned data

In the following example:

```
class alignas(16) float4 {
    float f[4];
};
float4 *p = new float4[1000];
```

C++11/14 did not specify any mechanism by which over-aligned data can be dynamically allocated correctly (i.e. respecting the alignment of the data). In the example above, not only is an implementation of C++ not required to allocate properly-aligned memory for the array, for practical purposes it is very nearly required to do the allocation incorrectly.

C++17 fixes that hole by introducing additional memory allocation functions that use align parameter:

```
void* operator new(std::size_t, std::align_val_t);
void* operator new[](std::size_t, std::align_val_t);
```

```
void operator delete(void*, std::align_val_t);
void operator delete[](void*, std::align_val_t);
void operator delete(void*, std::size_t, std::align_val_t);
void operator delete[](void*, std::size_t, std::align_val_t);
```

1.2.19 __has_include in preprocessor conditionals

This feature allows a C++ program to directly, reliably and portably determine whether or not a library header is available for inclusion.

Example: This demonstrates a way to use a library optional facility only if it is available.

```
#if __has_include(<optional>)
#  include <optional>
#  define have_optional 1
#elif __has_include(<experimental/optional>)
#  include <experimental/optional>
#  define have_optional 1
#  define experimental_optional 1
#else
#  define have_optional 0
#endif
```

1.2.20 Template argument deduction for class templates

Before C++17, template deduction worked for functions but not for classes. For instance, the following code was legal:

```
void f(std::pair<int, char>);

f(std::make_pair(42, 'z'));
```

because `std::make_pair` is a template function (so we can perform template deduction).

But the following wasn't:

```
void f(std::pair<int, char>);

f(std::pair(42, 'z'));
```

Although it is semantically equivalent. This was not legal because std::pair is a template class, and template classes could not apply type deduction in their initialization.

So before C++17 one has to write out the types explicitly, even though this does not add any new information:

```
void f(std::pair<int, char>);

f(std::pair<int, char>(42, 'z'));
```

This is fixed in C++17 where template class constructors can deduce type parameters. The syntax for constructing such template classes is therefore consistent with the syntax for constructing non-template classes.

todo: deduction guides.

1. A 4 min episode of C++ Weekly on class template argument type deduction
2. A 4 min episode of C++ Weekly on deduction guides
3. Modern C++ Features - Class Template Argument Deduction -

1.2.21 Non-type template parameters with auto type

Automatically deduce type on non-type template parameters.

```
template <auto value> void f() { }
f<10>();                // deduces int
```

Trip report: Summer ISO C++ standards meeting (Oulu) | Sutter's Mill

1.2.22 Guaranteed copy elision

Articles:

1. Jonas Devlieghere: Guaranteed Copy Elision

1.2.23 New specification for inheriting constructors (DR1941 et al)

More description and reasoning in P0136R0. Some excerpts below:

An inheriting constructor does not act like any other form of using-declaration. All other using-declarations make some set of declarations visible to name lookup in another context, but an inheriting constructor declaration declares a new constructor that merely delegates to the original.

This feature changes inheriting constructor declaration from declaring a set of new constructors, to making a set of base class constructors visible in a derived class as if they were derived class constructors. (When such a constructor is used, the additional derived class subobjects will also be implicitly constructed as if by a defaulted default constructor). Put another way: make inheriting a constructor act just like inheriting any other base class member, to the extent possible.

This change does affect the meaning and validity of some programs, but these changes improve the consistency and comprehensibility of C++.

```
// Hiding works the same as for other member
// using-declarations in the presence of default arguments
struct A {
  A(int a, int b = 0);
  void f(int a, int b = 0);
};
struct B : A {
  B(int a);      using A::A;
  void f(int a); using A::f;
};
struct C : A {
  C(int a, int b = 0);      using A::A;
  void f(int a, int b = 0); using A::f;
};

B b(0); // was ok, now ambiguous
b.f(0); // ambiguous (unchanged)

C c(0); // was ambiguous, now ok
c.f(0); // ok (unchanged)

// Inheriting constructor parameters are no longer copied
struct A { A(const A&) = delete; A(int); };
struct B { B(A); void f(A); };
struct C : B { using B::B; using B::f; };
C c({0}); // was ill-formed, now ok (no copy made)
c.f({0}); // ok (unchanged)
```

1.2.24 Direct-list-initialization of enumerations

Allows to initialize enum class with a fixed underlying type:

```
enum class Handle : uint32_t { Invalid = 0 };
Handle h { 42 }; // OK
```

Allows to create ‘strong types’ that are easy to use…

1.2.25 Stricter expression evaluation order

In a nutshell, given an expression such as f(a, b, c), the order in which the sub-expressions f, a, b, c (which are of arbitrary shapes) are evaluated is left unspecified by the standard.

```
// unspecified behaviour below!
f(i++, i);

v[i] = i++;

std::map<int, int> m;
m[0] = m.size(); // {{0, 0}} or {{0, 1}} ?
```

Summary of changes:

1. Postfix expressions are evaluated from left to right. This includes functions calls and member selection expressions.
2. Assignment expressions are evaluated from right to left. This includes compound assignments.
3. Operands to shift operators are evaluated from left to right.

Reference:

1. C++ Order of evaluation, cppreference
2. SO: What are the evaluation order guarantees introduced by C++17?
3. How compact code can become buggy code: getting caught by the order of evaluations, Fluent C++

1.2.26 constexpr lambda expressions

consexpr can be used in the context of lambdas.

```
constexpr auto ID = [] (int n)  { return n; };
constexpr int I = ID(3);
static_assert(I == 3);

constexpr int AddEleven(int n) {
  // Initialization of the 'data member' for n can
  // occur within a constant expression since 'n' is
  // of literal type.
  return [n] { return n + 11; }();
}
static_assert(AddEleven(5) == 16);
```

Articles

1. * A 5 min episode of Jason Turner’ s C++ Weekly about constexpr lambdas
2. * Lambda expression comparison between C++11, C++14 and C++17

1.2.27 Different begin and end types in range-based for

Changing the definition of range based for from:

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

Into:

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

Types of `__begin` and `__end` might be different; only the comparison operator is required. This little change allows Range TS users a better experience.

1.2.28 [[fallthrough]] attribute

Indicates that a fallthrough in a switch statement is intentional and a warning should not be issued for it. More details in P0068R0.

```
switch (c) {
case 'a':
    f(); // Warning emitted, fallthrough is perhaps a programmer error
case 'b':
    g();
[[fallthrough]]; // Warning suppressed, fallthrough is intentional
case 'c':
    h();
}
```

1.2.29 [[nodiscard]] attribute

[[nodiscard]] is used to stress that the return value of a function is not to be discarded, on pain of a compiler warning. More details in P0068R0.

```
[[nodiscard]] int foo();
void bar() {
    foo(); // Warning emitted, return value of a nodiscard function is discarded
}
```

This attribute can also be applied to types in order to mark all functions which return that type as [[nodiscard]]:

```
[[nodiscard]] struct DoNotThrowMeAway{};
DoNotThrowMeAway i_promise();
void oops() {
    i_promise(); // Warning emitted, return value of a nodiscard function is discarded
}
```

[A 4 min video about `[[nodiscard]]` in Jason Turner' s C++ Weekly](`https://www.youtube.com/watch?v=1_5PF3GQLKc`)

1.2.30 [[maybe_unused]] attribute

Suppresses compiler warnings about unused entities when they are declared with `[[maybe_unused]]`. More details in P0068R0.

```
                static void impl1() { ... } // Compilers may warn about this
[[maybe_unused]] static void impl2() { ... } // Warning suppressed


void foo() {
                      int x = 42; // Compilers may warn about this
     [[maybe_unused]] int y = 42; // Warning suppressed
}
```

[A 3 min video about `[[maybe_unused]]` in Jason Turner's C++ Weekly](https://www.youtube.com/watc

1.2.31 Ignore unknown attributes

Clarifies that implementations should ignore any attribute namespaces which they do not support, as this used to be unspecified. More details in P0283R1.

```
//compilers which don't support MyCompilerSpecificNamespace will ignore this attribute
[[MyCompilerSpecificNamespace::do_special_thing]]
void foo();
```

1.2.32 Pack expansions in using-declarations

Allows you to inject names with using-declarations from all types in a parameter pack.

In order to expose operator() from all base classes in a variadic template, we used to have to resort to recursion:

```
template <typename T, typename... Ts>
struct Overloader : T, Overloader<Ts...> {
    using T::operator();
    using Overloader<Ts...>::operator();
    // [...]
};

template <typename T> struct Overloader<T> : T {
    using T::operator();
};
```

Now we can simply expand the parameter pack in the using-declaration:

```
template <typename... Ts>
struct Overloader : Ts... {
    using Ts::operator()...;
    // [...]
};
```

Remarks

1. Implemented in GCC 7.0, see this change.

1.2.33 Decomposition declarations

Helps when using tuples as a return type. It will automatically create variables and tie them. More details in P0144R0. Was originally called “structured bindings”.

For example:

```
std::tie(a, b, c) = tuple; // a, b, c need to be declared first
```

Now we can write:

```
auto [ a, b, c ] = tuple;
```

Such expressions also work on structs, pairs, and arrays.

Articles:

1. Steve Lorimer, C++17 Structured Bindings
2. jrb-programming, Emulating C++17 Structured Bindings in C++14
3. Simon Brand, Adding C++17 decomposition declaration support to your classes

1.2.34 Hexadecimal floating-point literals

Allows to express some special floating point values, for example, the smallest normal IEEE-754 single precision value is readily written as 0x1.0p-126.

1.2.35 init-statements for if and switch

New versions of the if and switch statements for C++: if (init; condition) and switch (init; condition).

This should simplify the code. For example, previously you had to write:

```
{
    auto val = GetValue();
    if (condition(val))
        // on success
    else
        // on false...
}
```

Look, that val has a separate scope, without it it will ‘leak’.

Now you can write:

```
if (auto val = GetValue(); condition(val))
    // on success
else
    // on false...
```

val is visible only inside the if and else statements, so it doesn’t ‘leak’. condition might be any condition, not only if val is true/false.

Examples:

C++ Weekly - Ep 21 C++17’s if and switch Init Statements

1.2.36 Inline variables

Previously only methods/functions could be specified as inline, now you can do the same with variables, inside a header file.

A variable declared inline has the same semantics as a function declared inline: it can be defined, identically, in multiple translation units, must be defined in every translation unit in which it is used, and the behavior of the program is as if there is exactly one variable.

```
struct MyClass
{
    static const int sValue;
};
```

```
inline int const MyClass::sValue = 777;
```

Or even:

```
struct MyClass
{
    inline static const int sValue = 777;
};
```

Articles

SO: What is an inline variable and what is it useful for?

1.2.37 DR: Matching of template template-arguments excludes compatible

templates

This feature resolves Core issue CWG 150.

From the paper:

This paper allows a template template-parameter to bind to a template argument whenever the template parameter is at least as specialized as the template argument. This implies that any template argument list that can legitimately be applied to the template template-parameter is also applicable to the argument template.

Example:

```
template <template <int> class> void FI();
template <template <auto> class> void FA();
template <auto> struct SA { /* ... */ };
template <int> struct SI { /* ... */ };
FI<SA>();  // OK; error before this paper
FA<SI>();  // error

template <template <typename> class> void FD();
template <typename, typename = int> struct SD { /* ... */ };
FD<SD>();  // OK; error before this paper (CWG 150)
```

(Adapted from the comment by IncongruentModulo1) For a useful example, consider something like this:

```
template <template <typename> typename Container>
struct A
{
    Container<int>    m_ints;
    Container<double> m_doubles;
};
```

In C++14 and earlier, A<std::vector> wouldn't be valid (ignoring the typename and not class before container) since std::vector is declared as:

```
template <typename T, typename Allocator = std::allocator<T>>
class vector;
```

This change resolves that issue. Before, you would need to declare template <template <typename...> ty which is more permissive and moves the error to a less explicit line (namely the declaration of `m_ints` wherever the struct A is implemented / declared, instead of where the struct is instantiated with the wrong template type.

1.2.38 `std::uncaught_exceptions()`

More background in the original paper: PDF: N4152 and GOTW issue 47: Uncaught Exceptions.

The function returns the number of uncaught exception objects in the current thread.

This might be useful when implementing proper Scope Guards that works also during stack unwinding.

A type that wants to know whether its destructor is being run to unwind this object can query `uncaught_exceptions`

in its constructor and store the result, then query `uncaught_exceptions` again in its destructor; if the result is different,

then this destructor is being invoked as part of stack unwinding due to a new exception that was thrown later than the object's construction

The above quote comes from PDF: N4152.

1.2.39 constexpr if-statements

The static-if for C++! This allows you to discard branches of an if statement at compile-time based on a constant expression condition.

```
if constexpr(cond)
     statement1; // Discarded if cond is false
else
     statement2; // Discarded if cond is true
```

This removes a lot of the necessity for tag dispatching and SFINAE:

1.2.40 SFINAE

```
template <typename T, std::enable_if_t<std::is_arithmetic<T>{}>* = nullptr>
auto get_value(T t) {/*...*/}

template <typename T, std::enable_if_t<!std::is_arithmetic<T>{}>* = nullptr>
auto get_value(T t) {/*...*/}
```

1.2.41 Tag dispatching

```
template <typename T>
auto get_value(T t, std::true_type) {/*...*/}

template <typename T>
auto get_value(T t, std::false_type) {/*...*/}

template <typename T>
auto get_value(T t) {
    return get_value(t, std::is_arithmetic<T>{});
}
```

1.2.42 if constexpr

```
template <typename T>
auto get_value(T t) {
     if constexpr (std::is_arithmetic_v<T>) {
         //...
     }
     else {
         //...
     }
}
```

Articles:

1. LoopPerfect Blog, C++17 vs C++14 - Round 1 - if-constexpr
2. SO: constexpr if and `static_assert`
3. Simon Brand: Simplifying templates and `#ifdefs` with if constexpr

1.3 Library Features

To get more details about library implementation I suggest those links:

1. VS 2015 Update 2’ s STL is C++17-so-far Feature Complete - Jan 2016
2. libstdc++, C++ 201z status
3. libc++ C++1z Status

This section only mentions some of the most important parts of library changes, it would be too impractical to go into details of every little change.

1.3.1 Merged: The Library Fundamentals 1 TS (most parts)

We get the following items:

1. Tuples - Calling a function with a tuple of arguments
2. Functional Objects - Searchers
3. Optional objects
4. Class any
5. `string_view`
6. Memory:
   (a) Shared-ownership pointers
   (b) Class `memory_resource`
   (c) Class `memory_resource`
   (d) Access to program-wide `memory_resource` objects
   (e) Pool resource classes
   (f) Class `monotonic_buffer_resource`
   (g) Alias templates using polymorphic memory resources
7. Algorithms:
   (a) Search
   (b) Sampling
8. `shared_ptr` natively handles arrays: see Merging `shared_ptr` changes from Library Fundamentals to C++17

The wording from those components comes from Library Fundamentals V2 to ensure the wording includes the latest corrections.

Resources:

Marco Arena, `string_view` odi et amo

1.3.2 Removal of some deprecated types and functions, including `std::auto_ptr`, `std::random_shuffle`, and old function adaptors

1. Function objects - `unary_function`/`binary_function`, `ptr_fun()`, and `mem_fun()`/`mem_fun_ref()`
2. Binders - `bind1st()`/`bind2nd()`
3. `auto_ptr`
4. Random shuffle - `random_shuffle(first, last)` and `random_shuffle(first, last, rng)`

1.3.3 Merged: The Parallelism TS, a.k.a. “Parallel STL.” ,

Articles: Parallel Algorithm of the Standard Template Library - ModernesCpp.com

1.3.4 Merged: File System TS,

P0218R1

1.3.5 Merged: The Mathematical Special Functions IS,

1.3.6 Improving std::pair and std::tuple

1.3.7 `std::shared_mutex` (untimed)

1.3.8 Variant

Variant is a typesafe union that will report errors when you want to access something that’ s not currently inside the object.

1. cppreference/variant
2. IsoCpp: The Variant Saga: A happy ending?

1.3.9 Splicing Maps and Sets

From Herb Sutter, Oulu trip report:

You will now be able to directly move internal nodes from one node-based container directly into another container of the same type. Why is that important? Because it guarantees no memory allocation overhead, no copying of keys or values, and even no exceptions if the container’ s comparison function doesn’ t throw.

1.4 Summary

Thanks for all the support with the list!

There are still items that should be updated, but the list is mostly done.