Part VIII

C++17 specific

Outline

21 Expansion of `constexpr`

22 Miscellaneous

23 Template parameter deduction

if constexpr

- From C++17, you can now have `if` statements that are evaluated at compile-time, and avoid instantiation of branches other than the one that succeeds.
- For example, part of the `Variadic Templates` example now reads:

```
template<int N>
auto get()const{
  if constexpr(N > 0){
    return Tuple<OtherParams...>::template get<N−1>();
  }
  else{
    return param;
  }
}
```

- where omitting `constexpr` would result in `get<-1>` being instantiated, giving an error.

if constexpr

- Overall, this avoids the need for `std::enable_if` and a separate specialization of `get<0>()`.
- The `else` does not need a `constexpr`; it binds to the preceding `if`.
- Also, note the use of `auto`, which avoids the complex `EltType` construct.
- The return-type deduction ignores the non-followed `if constexpr` branch if `N==0` and takes its type from `param`.
- See `constexpr_if.C` for full code.

Const-evaluation for non-type template args

- Previously, non-type, non-integer template parameters had to be explicitly given:

```
double ODESolver::Euler(double x, double dt){
  return dt;
}

template<double (*f)(double, double)>
double mySolver(double x, double T){
  for(int i=0 ; i < 100 ; i++){
    x += T/100 * f(x, T/100);
  }
  return x;
}

double y = mySolver<ODESolvers::Euler>(x, T);
```

Const-evaluation for non-type template args

- However, since we can imagine `constexpr` functions that return a function pointer, consider:

```
constexpr auto pickSolver(ODE solver){
  if(solver == Euler){
    return &ODESolvers::Euler;
  }
  else if(solver == RK2){
    return &ODESolvers::RK2;
  }
}
```

- In C++17: `mySolver<pickSolver(Euler)>(0, 10)` is now valid.
- This a fairly contrived example, but more complex ways of picking a function pointer could be imagined, or indeed `constexpr` resulting in either pointers or references.
- See `Examples/template_arg.C` for full code.

constexpr lambda functions

- In C++17 lambda functions can be implicitly cast to constexpr function pointers.

```
template<typename T>
constexpr int smallest(bool (*lessThan)(int, int),
                       std::initializer_list<int> a) {
  int s = *a.begin();
  for(int i : a) {
    if( lessThan(i, s) ) {
      s = i;
    }
  }
  return s;
}
auto mod10 = [](int a, int b){return (a % 10) < (b % 10);};
static_assert(smallest(mod10, {19, 9, 22, 31}) == 31);
```

- We use a lambda-comparison function that compares the units digits of two elements of a list. This can be evaluated at compile-time.
- See `Examples/constexpr_lambda.C` for full code.

constexpr limits

- For anyone concerned (or disturbingly excited) that `constexpr` allows a lot of calculation to be done at compile-time, there are limits.
- The C++17 standard recommends allowing up to 512 recursive `constexpr` invocations and $1,048,576$ full-expressions in a `constexpr` evaluation.
- For `gcc` and `clang` the former can be changed by `-fconstexpr-depth`.
- The latter appears not to be modifiable in `gcc`, but can be in `clang` by `-fconstexpr-steps`.
- `gcc` also has `-fconstexpr-loop-limit=262144` by default.

Outline

Auto deduction from braced list

- I *think* the behaviour of:

```
auto x{func()};
auto x2 = {func()};
```

  should change at C++17, but it doesn't seem to.

- Also, I think:

```
auto x1{1, 2};
```

  should have been permitted before C++17, but neither gcc nor clang++ do so.

- Maybe C++17 just explicitly bans something that compilers banned anyway?
- See http://open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3681.html for background.
- See http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3922.html for solution.

typename in template template parameter

- Prior to C++14, template template parameters did not allow the use of `typename`.
- I tried the example at: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4051.html` but cannot get a failure to compile with earlier standard.

Nested namespaces

- Namespace definitions can now be nested with a more succinct syntax:

```
namespace boost{
  namespace tuples {
  ...
}
}
```

can now be replaced by

```
namespace boost::tuples{
  ...
}
```

- This is probably of most interest to library authors.

Guaranteed copy elision

- As noted in “Return Value Optimization”, compilers are allowed to make optimizations by ignoring an unnecessary copy/move.
- However, in C++14 an object without copy or move constructors could not undergo this optimization, even though the constructor was not necessary.
- In C++17 this restriction is removed, by reworking the definitions of what is being transferred, so that a copy/move would not be required anyway.

Guaranteed copy elision ctd

```
struct NonMoveable
{
  NonMoveable(int){}
  NonMoveable(NonMoveable&) = delete;
  NonMoveable(NonMoveable&&) = delete;
  std::array<int, 1024> arr;
};
NonMoveable make(){
  return NonMoveable(42);
}
```

- This is not allowed in C++14 but is allowed in C++17, even though there is no copy/move constructor.
- See `Examples/copy_elision.C`
- See `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0135r0.html`

Outline

21 Expansion of `constexpr`

22 Miscellaneous

23 Template parameter deduction

Automatic template deduction

- In C++17 you can omit some template parameters when constructing objects:

```
std::pair a(1.0, 2); // OK
std::vector b{1, 4, 7, 0}; // OK
std::vector c{1, 4.2, 7, 0}; // Not OK.
```

- The compiler deduces the element types `double` and `int` for the `pair`, `int` for the first `vector`, and fails to find a consistent type for the last case.
- This results from a change in the standard (rather than header files).
- The standard specifies that the compiler attempt to find a suitable constructor based on the explicit argument types passed in.

User-defined template deduction guides

- The above does not work for:

```
int d[5] = {1,2,3,4,5};
std::vector d2(d, d+5);
```

  because the compiler cannot deduce the `vector` template parameter from the iterator/pointer arguments.

- Here we have to use user-defined deduction guides.
- At a scope outside the `vector` class you can define:

```
template<typename Iter,
typename ValType = typename iterator_traits<Iter>::value_type>
vector(Iter, Iter) -> vector<ValType>;
```

  (slightly simplified from actual STL definition)

- The compiler now adds this to the set of constructors it attempts to match. It can easily deduce the type of `Iter`, from which it finds the default parameter `ValType`, which then forwards to the constructor for `vector<ValType>` that takes two iterators.

User-defined template deduction guides

- Full code is available at `Examples/template_deduction.C`.
- See `https://en.cppreference.com/w/cpp/language/class_template_argument_deduction` for more details
- and for the standards paper describing the issue: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0091r3.html`

Structured bindings

- If a function returns a `std::tuple` then since C++14 we have been able to do:

```
std::tuple<int, double, std::string> func()
{
  return std::tuple(42, 3.14159, "Ford");
}
int x; double y; std::string z;
std::tie(x, y, z) = func();
```

- However, from C++17 we can now also do any of:

```
auto [x, y, z] = func();
const auto [x, y, z] = func();
const auto& [x, y, z] = func();
auto& [x, y, z] = func();
```

- See `Examples/structured_bindings.C` where we demonstrate that the last example fails if a non-lvalue is returned from `func`.

Range-based for loop

- From C++17 it is not necessary for the begin and end of a for-loop range to be of the same type.
- Previously, the begin and end were iterators and had to be of the same type.
- See `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0184r0.html`
- It is unclear whether this is of much use within C++17? Seems to be useful for new Ranges TS library.

Omitted

- I have worked from gcc's list of C++ features. From these, I have omitted the following:
- Pack expansions in using-declarations: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0195r2.html
- Inheriting constructors: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0136r1.html

Part IX

C++20 specific

C++20 upcoming

- Although C++20 is still (maybe nearly) 2 years away, we have some indication of which major features will be in the standard.
- Some compilers have begun to implement these.
- `gcc` has the `-std=c++2a` option, but support is "highly experimental" as of February 2019.
- Interesting prospects include:
- Contracts - more complex and feature-rich form of `assert` and `static_assert` - see `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html`
- Operator `<=>` - see `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0515r3.pdf`

C++20 major features

Feature-set agreed in February 2019: `https://herbsutter.com/2019/02/23/trip-report-winter-iso-c-standards-meeting-kona/`

- Modules: Ability to encapsulate program components without polluting main scope (with macros, for example). May be familiar from Python and Fortran. Can deliver a set of variables, functions, and classes.
- Coroutines: Allows ability to suspend functions, pass control to another one, and return to the original function later.
- Concepts: Support for explicit Requires, Constraints, Expects, and Mandates specifications for functions. These are various conditions that the function needs in order to function correctly. Intended to encapsulate `enable_if` and similar constructs in a more readable way.