C++ 14 & C++ 17

What‘s new?

C++14 Features

Function return type deduction

VS2015

```
auto toStdString(const QString& val) {
  return val.toStdString();
}
```

- If multiple return types are available, all must deduce to the same type
- Recursions can be used with a function, but the recursive call must happen happen after at least one return statement in the definiton of the function.

Generic lambdas

VS2015

```
auto plus = [](auto x, auto y) {
  return x + y;
}
```

- For auto type deduction, generic lambdas follow the rules of template argument deduction.

Lambda capture expressions

VS2015

```
auto buffer = std::make_unique<int>(42);

auto plus = [_buffer = std::move(buffer)](auto x) {
  // use _buffer here
}
// be careful when use buffer here!
```

- C++11 only allows capture by reference or value copy
- C++14 allows captured members to be initialized with arbitrary expressions

Library extensions

`std::make_unique<T>` Creates a unique_ptr of T

`std::make_unique<T[]>(size)` Creates a unique_ptr of an array of size elements

`std::cbegin()/std::cend()`

`std::rbegin()/std::rend()`

`std::crbegin()/std::crend()`

Function to access the corresponding container functions

The attribute [[deprecated]]

VS2015

```
struct foo {
  [[deprecated]] int a();
  [[deprecated("Advice what to use instead")]] int b();
  int c();
};
```

- This results in warnings during compilation that a certain usage is deprecated.
- In Visual Studio, warning 4996 must not be disabled

More changes

VS2015

- Variable templates

```
template <typename T>
constexpr T pi = T(3.14159265358979323846264383);
```

- Binary literals

```
0b11 ⇔ 3
```

- Digit separators

```
auto bigValue = 1‘000‘000‘000
```

C++17 Features

Deprecation & Removal

- Removing trigraphs — VS2010
- Removing register keyword — VS2017
- Removing deprecated operator++(bool) — VS2017
- Cannot inherit from std::iterator
- Removing std::auto_ptr — VS2017
- Removing deprecated exception specification — VS2017
  - `void foo() throw() => void foo() noexcept;`
- Removed std::shared_ptr::unique() — VS2017

Fixes

```
auto foo {42};
```

VS2015

type of foo is now int, not std::initializer_list anymore

`std::shared_ptr<T[]>, std::shared_ptr<T[N]>` is supported

VS2017

VS2017

Structured Binding

```
double values[] = {1.0, 2.0, 3.0};
auto [a, b, c] = values;
```

```
std::pair<int, std::string> foo{42,"Don‘t panic!"};
auto [d, e] = foo;

struct Bar{ int x; double y; };
Bar createBar();
const auto[x, y] = createBar();
```

```
std::map<int, std::string> myMap;
for (const auto& [key, value] : myMap) {
  std::cout << key << "|" << value << '\n';
}
```

VS2017

Init-statement for if/switch

Old:

```
{
  auto f = foo();
  if (condition(f))
    // on success
  else
    // on false
}
```

New:

```
if (auto f = foo(); condition(f))
  // on success
else
  // on false
```

VS2017

Combining structured binding & init if

```
std::map<int, std::string> myMap;

if (auto [it, succeeded] = myMap.insert(value); succeeded) {
  use(it); // ok
} // iter and succeded are destroyed here
```

VS2015

Nested namespace definitions

Old:

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

New:

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

VS2017

STL Extensions

std::any	Similar to boost::any
std::optional	Similar to boost::optional (MeVis‘ CondVar)
std::string_view	
std::variant	Similar to boost::variant
. . .	

VS2017

std::variant Usage

```
std::variant<std::string, std::u16string> value;
value = "Foo";
auto s1 = std::get<std::string>(value); // decltype(s1) == std::string
auto s2 = std::get<0>(value);           // decltype(s1) == std::string
auto e = std::get<std::u16string>(value); => throws

value = u"Foo";
auto s3 = std::get<std::u16string>(value); // decltype(s1) == std::u16string
```

VS2017

Usage std::variant with std::visit

```
auto toQString(const std::string& val) {
  return QString(val.c_str());
}
auto toQString(const st::u16string& val) {
  return QString::fromUtf16(reinterpret_cast<wchar_t*>(val.c_str()));
}

std::variant<std::string, std::u16string> value;

QLabel label;

label.setText(
  std::visit([&label](const auto& v) { return toQString(v); }, value)
);
```

VS2017

Fold expressions

Old:

```
void print(std::ostream&) {}

template <typename T, typename... R>
void print(std::ostream& s, const T& t, const R&... r) {
  s << t;
  print(s, r...);
}
```

VS2017

Fold expressions

New:

```
template <typename... R>
void print(std::ostream& s, const R&... r) {
  (s << ... << r); // Don't forget the '(' and ')'!
}
```

VS2017

if constexpr()

Old:

```
class person {
  int &get_id();
  std::string &get_name();
  int &get_age();
private:
  int _id;
  std::string _name;
  int _age;
};

person p;

// does not compile
auto [id, name, age] = p;
```

```
template <std::size_t I>
auto& get(person& p);

template <>
auto& get<0>(person &p) { return p.get_id(); }

template <>
auto& get<1>(person &p) { return p.get_name(); }

template <>
auto& get<2>(person &p) { return p.get_age(); }

template <>
class std::tuple_size<person> : public
std::integral_constant<std::size_t, 3> {}

auto [id, name, age] = p; // now it compiles
```

VS2017

if constexpr()

New:

```
template <std::size_t I>
auto& get(person& p) {
  if constexpr (I == 0)
    return p.get_id();
  else if constexpr (I == 1)
    return p.get_name();
  else if constexpr (I == 2)
    return p.get_age();
}

// this compiles as well
auto [id, name, age] = p;
auto [id, name, _] = p;
```

New better:

```
template <std::size_t I, typename U>
decltype(auto) get(U&& p) {
  if constexpr (I == 0)
    return std::forward<U>(p).get_id();
  else if constexpr (I == 1)
    return std::forward<U>(p).get_name();
  else if constexpr (I == 2)
    return std::forward<U>(p).get_age();
}


auto [id, name, age] = p;
auto [id, name, _] = p;
```

VS2017

[[fallthrough]] attribute

```
void g();
void h();
void i();
switch (n) {
  case 1:
  case 2:
    g();
    [[fallthrough]];
  case 3: // warning on fallthrough discouraged
    h();
  case 4: // implementation may warn on fallthrough
    i();
    [[fallthrough]]; // illformed
}
```

VS2017

[[nodiscard]] attribute

```
enum class [[nodiscard]] ErrorCode {
  OK,
  Fatal,
  System
};

ErrorCode doWhatIsNeededToBeDone();

doWhatIsNeededToBeDone();                // warning encouraged
auto result = doWhatIsNeededToBeDone();  // no warning
```

VS2017

[[maybe_unused]] attribute

```
void foo([[maybe_unused]] int bar)
{
  // no usage of bar, should not trigger a warning
}
```

Even more …

- Template argument deduction for class
- Filesystem
- …