C++17 Language New Features Cheatsheet

Template argument deduction for class templates

```
pair p1(1, 2.0);
// vs
pair<int, double> p2(1, 2.0);
```

Declaring non-type template parameters with auto

```
template <auto ... seq>
struct my_integer_sequence {
  // Implementation here ...
};
// Explicitly pass type `int` as template argument.
auto seq = std::integer_sequence<int, 0, 1, 2>();
// Type is deduced to be `int`.
auto seq2 = my_integer_sequence<0, 1, 2>();
```

Folding expressions

```
template<typename ... Ts>
auto sum_fold_exp(const Ts& ... ts) {
  return (ts + ...);
}
template<typename ... Ts>
auto print_fold(const Ts& ... ts)
{
  ((cout << ts << " "), ... );
}
```

New rules for auto deduction from braced-init-list

```
// error: not a single element
auto x1{ 1, 2, 3 };
// decltype(x2) is std::initializer_list<int>
auto x2 = { 1, 2, 3 };
// decltype(x3) is int, previously deduced to
// initializer_list<int>
auto x3{ 3 };
// decltype(x4) is double
auto x4{ 3.0 };
```

constexpr lambda

```
auto identity = [] (int n) constexpr { return n; };
static_assert(identity(123) == 123);

constexpr int addOne(int n) {
  return [n] { return n + 1; }();
}
static_assert(addOne(1) == 2);
```

UTF-8 Character Literals

```
char x = u8'x';
```

Lambda capture this by value

```
struct foo {
  foo() : _x{0} {}
  int _x;
  auto log_by_ref() {
    return [this]() { cout << _x << endl; };
  }
  auto log_by_val() {
    return [*this]() { cout<<_x<<endl;};
  }
};
struct foo f;
auto ref = f.log_by_ref();
auto val = f.log_by_val();
f._x = 1234;
ref();  val(); // both 1234
f._x = 4321;
ref(); // 4321
val(); // 1234
```

Inline variables

```
struct S { int x; };
inline S x1 = S{321};
```

Nested namespaces

```
namespace A::B::C {
  class foo;
}
```

Structured bindings

```
template<typename T>
pair<T, bool> racine(T d) {
  if (d<0) return pair(-1, false);
  return pair(sqrt(d), true);
}

auto [s, success] = racine(1998.0);
if (success) cout << s << endl;
```

Initializers in if and switch statements

```
if (auto res=m.insert({key,value}); res.second) {
  cout<<key<<"/"<<value<<" inserted"<<endl;
}
```

Removal of trigraphs

```
??= ??/ ??' ??( ??) ??! ??< ??> ??-
```

constexpr if

```
template <typename T> int compute(T x) {
  // no () around consexpr
  if constexpr (std::is_integral<T>::value) {
    return x * x;
  } else if constexpr (is_same<T, string>::value) {
    return x.size();
  } else if constexpr (is_base_of<foo, T>::value) {
    x.bar();
    return 0;
  }
  return 0;
}
```

Hexadecimal floating-point literals

```
cout << 0x10.1p0 << endl // 16.0625
  << 0X0.8p0 << endl     // 0.5
  << 0X50.8p5 << endl;   // 2576
```

Direct List Initialization of Enums

```
// underlying type must be fixed (char here)
enum class color : char { red, blue, green };
// must be non-narrowing, i.e 129 is an error
color c1 { 3 }, c2 { 88 };
```

[[fallthrough]] attribute

```
switch (i) {
case 1: cout<<"one"<<endl; // warning
case 2: cout<<"two"<<endl;
[[fallthrough]];
case 3 : cout<<"three"<<endl; // warning supressed
}
```

[[nodiscard]] attribute

Can be applied to a type (function with that return type will be marked as [[nodiscard]])

```
[[nodiscard]] int foo() { return 1; };
void bar() {
  foo(); // Warning
```

[[maybe_unused]] attribute

```
[[maybe_unused]] static void f() {} // No warning
[[maybe_unused]] int x = 42; // No warning
```

static_assert without message

```
static_assert(VERSION >= 2);
```