Modern C++ Standards

C++11 through C++17*

Dominique LaSalle
dominique@solidlake.com

Modern C++ Standards

Compiler Versions

- C++ 11: g++ >= 4.8.1, icc >= 15.0, MSVC >= 2015, clang >= 3.3
- C++ 14: g++ >= 5.0, icc >= 17.0*, MSVC >= 2015*, clang >= 3.4
- C++ 17: g++ >= 7.0 (-std=c++1z)
- Feature by feature listing for most compiles `http://en.cppreference.com/w/cpp/compiler_support`

C++ 11 (Major)

Key Features

- Smart pointer types (e.g., `std::unique_ptr` and `std::shared_ptr`).
- Threading.
- Range based for loop (foreach).
- Lambda functions.
- The `auto` keyword.
- The `override` keyword.
- The `noexcept` keyword.
- The `constexpr` keyword.

Smart pointers - `std::unique_ptr`

- Uses moves semantics (`rvalue` references) to ensure only one owner.
- Calls `delete` when owner is destructed.

```
{
  std::unique_ptr<int> myInt(new int[1]);
  ...
  std::unique_ptr<int> myNewInt(myInt);
  // myInt is no longer a valid pointer
  ...
} // myNewInt's pointer get's deleted when it leaves scope
```

- Useful for polymorphic class members.
- Fits most non-vector heap allocations.

Smart pointers - `std::shared_ptr`

- Uses reference counting to determine when to call `delete`.

```
{
  std::shared_ptr<int> myInt(new int[1]);
  ...
  std::shared_ptr<int> myNewInt(myInt);
  // both are valid pointers to the same memory
}
```

- Useful for when the lifetime of a heap object is not dependent upon another object.

Range based for loops.

- Syntactic sugar for traversing containers via iterators and dereferencing.

```
std::vector<float> myValues;
...
for (float const & value : myValues) {
  sum += value;
}
```

- Useful for avoiding iterators and counters.
- Makes life a lot better when traversing `std::map`.

Lambda functions.

- Create function pointers inline which capture current variables.

```
void lowerCase(std::string & str)
{
  std::transform(str.begin(), str.end(), [](char const c) {
    return std::tolower(c);
  });
}
```

- Useful for creating small function pointers with limited scope.

The `auto` keyword

- Determines variable type at compile time.

```
auto front = myContainer.begin();
auto back = myContainer.end()-1;
while (front < back) {
  if (*front > *back) {
    ...
  }
  ++front;
  --back;
}
```

- Useful for avoiding long but obvious type names.

The `override` keyword

- Requires that the method overrides a parent's method.

```
class Bar
{
  public:
    virtual void barDo();
};
class Foo : public Bar
{
  public:
    // will compile
    void barDo() override;
    // won't compile
    void barDo(int) override;
};
```

The `noexcept` keyword

- Tells the compiler the function/method will never throw an exception.
- Throwing an exception in a `noexcept` region calls `terminate()`.

```
void foo(double a) noexcept;

void foo(double const a) noexcept
{
  ...
}
```

The `constexpr` keyword

- Allows variables or non-void functions to be evaluated at compile time.
- Implies `const`.

```
constexpr int getIntPI()
{
  return 3;
}

int piBins[getIntPI()];
```

- Allows for function generated constants without a performance hit.
- Allows for constant variables to be defined in a class declaration.

Other nice things

- No longer need spaces between closing angle brackets in nested templates `std::vector<std::pair<int,float>>`.
- Adds `nullptr` which is convertible to all pointer types but not integers (unlike `NULL`).
- Adds initializer lists for class constructors (e.g., `std::vector<int> x = {1,2,3,4,5};`).

C++ 14 (Minor)

Key Features

- Expansion of `auto` usage (return type, lambdas, etc.).
- Expansion of `constexpr` usage.

C++ 17 (Major)

Key Features

- Adds initialization clause to `if` and `switch` statements.
- Adds *structured bindings* (e.g., multi-value returns).
- Adds classes for representing filesystem objects.
- Adds threaded algorithms (e.g., parallel exclusive prefix sum).
- Based on `C11` instead of `C99`.
- For more details see:
  `https://stackoverflow.com/questions/38060436/what-are-the-new-features-in-c17`

C++ 17

Initialization clause

- Allow for scoped initialization of variables inside of `if` and `select` statements.

```
if (int x = foo(); x) {
  printf("Error calling foo(): returned %d\n", x);
}
```

Structured Bindings

- Allow for multiple return values from functions.

```
std::tuple<int, std::string> foo()
{
  return std::make_tuple(5, "bar");
}
...
auto [code, msg] = foo();
// code is an int, and msg is a string
```

- Makes use of `auto` keyword.

Filesystem

- Provides high level filesystem operations as part of standard.

```
// list all files in my home directory
for (auto & dir : std::filesystem::directory_iterator("/home/domin
  printf("%s\n", dir.c_str());
}

// recursively copy whole directory tree
std::filesystem::copy("/home/dominique", "/tmp/home_bkup",
    std::filesystem::copy_options::recursive);
```

- Large set of functionality including checking permissions, file/directory existence, and symlinks.

Parallel Algorithms

- Provide common multi-threaded operations.

```
// parallel for loop
std::for_each(std::execution::par, vec.begin(), vec.size(),
    [](int& arg) { arg <<= 2 });

// parallel prefix sum
std::exclusive_scan(std::execution::par, vec.begin(),
    vec.end(), out.begin(), 0);

// parallel reduce -- defaults to summation
std::reduce(std::execution::par, vec.begin(), vec.end());
```