Advanced C++

Philip Blakely

Laboratory for Scientific Computing, University of Cambridge

Part I

C++11/14/17

Outline

C++11

- C++11 was standardized in 2011. You may see references to C++0x in old documentation.
- It incorporates useful constructs from the Boost libraries.
- New container types.
- Lambda functions
- Features geared towards ease of coding and optimization.
- These lectures assume that you are a reasonably competent C++ user.
- Check what version of C++ your compiler defaults to.
- For `gcc` use the `--std=c++11` flag.

C++14

- C++14 was finalised in December 2014, and is largely a series of fixes and clarifications to the C++11 standard.
- Occasionally you may see reference to C++1y in old documentation written before the publication date was known.
- For `gcc` use the `--std=c++14` flag.
- From `gcc` version 6.1 onwards, C++14 is the default.

C++17

- C++17 was finalised in December 2017, and is largely a series of fixes and clarifications to the C++11 standard.
- Occasionally you may see reference to C++1z in old documentation written before the publication date was known.
- For `gcc` use the `--std=c++17` flag.
- `gcc-8` is feature complete for C++17, but C++14 remains the default.

References

- Bjarne's guide to differences between C++03 and C++11: `http://www.stroustrup.com/C++11FAQ.html`
- Bjarne Stroustrup, "The C++ Programming Language", 4th Edition, Addison Wesley
- Effective Modern C++, Scott Meyer, O'Reilly, 2014
- `http://en.cppreference.com/w/cpp` - has clear indication of which features are in C++03, C++11, C++14, C++17 (and C++20)
- See `https://gcc.gnu.org/projects/cxx-status.html` for list of features implemented in `gcc` for each standard.

Standards documents

- The ISO C++ Standards are published by the International Organization for Standardization.
- The certified standards documents are only available to buy, at 600 GBP each.
- However, see `https://en.cppreference.com/w/cpp/links` for links to the final working drafts, which are a very good approximation to the final version.
- (This is not dissimilar from the way you can make your papers freely accessible via `www.data.cam.ac.uk` without violating the publishing journal's terms and conditions.)

Examples

- Virtually every topic in this course has an associated example code in the `Examples` directory.
- Running `make all` will compile all tests.
- In some cases I have used the macro `AVOID_INTENTIONAL_ERRORS` to allow me to check for correct compilation.
- `make example_name` will compile without errors.
- If you run `make EXPOSE_ERRORS=1 example_name` then this may show various (expected) errors.
- If compiling the later C++14/17 features, you will need to specify: `CXX=/lsc/opt/gcc-8.2.0/bin/g++-8` as well (or some other compiler with similar support).

Makefile interlude

While writing this part I discovered that Makefiles allow target-specific variable definitions:

```
$(TARGETS): CXXSTD := -std=c++11
$(TARGETS14): CXXSTD := -std=c++14
$(TARGETS17): CXXSTD := -std=c++17
%: %.C
	$(CXX) $(CXXFLAGS) $(CXXSTD) $< -o $@
```

means that only targets defined in `$(TARGETS14)` will have the `-std=c++14` option, and similarly for C++17.

Detecting C++11

- If you write code that requires C++11 syntax, and someone accidentally compiles with a C++03 compiler, then they will have a large number of errors.
- A simple way to detect whether you have a C++11 compiler is:

```
#if __cplusplus < 201103L
#error This program requires a C++11 compiler.
#endif
```

  which will error out early on, at the pre-processing stage.
- The equivalent numbers for all versions are:
  - 199711L (C++98 or C++03)
  - 201103L (C++11)
  - 201402L (C++14)
  - 201703L (C++17)
- These are defined by the C++ standard, so are guaranted to work (or at least indicate that the compiler claims to be compliant with that standard).

Outline

NULL pointer

- In C++03 you would use `NULL` or `0` to represent an undefined pointer.
- In C++11 you should use `nullptr` instead.
- This provides for better readability and for distinctness in overloading to accept either an integer or a pointer.
- See `Examples/null.C`

enum classes

- In C++03 we had `enums`:

```
enum Colour{Red, Green, Blue};
```

- which defines a type `Colour` at global scope that implicitly converts to an `int`:

```
Colour b = Blue;
int r = Red;
```

- In C++11 we have a strongly typed `enum class`:

```
enum class ProperColour : char
{ Cyan, Magenta, Yellow, Black };
```

- which defines a type `ProperColour` with its own scope that uses a `char` to store its value, but does not implicitly convert to an `char`.

enum classes ctd

- For example, the following are OK:

```
ProperColour c = ProperColour::Cyan;
char y = (char)ProperColour::Yellow;
```

- The following are not OK:

```
char c = Cyan;
char c2 = ProperColour::Cyan;
ProperColour m = Magenta;
```

- See `Examples/enum.C`

Template closing brackets

- A problem you may not have realised you had:

```
std::vector<std::pair<int,int>> a;
```

  is invalid in C++03.
- In C++03 `>>` is interpreted as a right-shift operator, following the "maximal munch" principle.
- In C++03 you need a space between the two `>` brackets.
- From C++11 the above syntax is valid.
- In some (fairly contrived) cases, this may cause code to give different results under C++11 and C++03.
- See `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1757.html`
- See `Examples/maxMunch.C`.

Outline

Constant values

- Template parameters (for example) must have their values known at compile-time.
- In C++03 we are restricted to using `static const int` members and using recursive templates if we want to do any complex calculations.
- In C++11 we can declare a function to be a `constexpr`, specifying that it can be evaluated at compile-time:

```
constexpr int triang(int n){
  return (n>1) ? n + triang(n-1) : 1;
}
```

- We can then use it as a template parameter (`Examples/constexpr.C`):

```
template<int D>
struct Vector{
  double m_data[D];
};

Vector<triang(5)> a;
```

Constant values

- In C++11 there are restrictions to using `constexpr`:
  - It must consist of a single `return` statement.
  - It must not contain any local variables.
  - It must not have side-effects, e.g. modifying a global variable.
- These are somewhat relaxed at C++14.
- A `constexpr` function can be used to initialize any `static const` member.

Constant values - non-integral

- Note that non-integral `static const` members are now permitted in C++11, and can be initialized inside the class:

```
constexpr double expon(double n){
  return exp(n);
}
template<int D>
struct Vector{
  static constexpr double m_val = expon(D);
};
```

- See `Examples/constexprfloat.C`

constexpr functions

- In C++11 `constexpr` functions were very restricted in their form, and essentially had to be a series of arithmetic expressions.
- In C++14 `constexpr` functions can be any function that does not contain:
  - `goto`
  - `try`-block
  - Uninitialized variables or static variables.

  and does not require the evaluation of
  - undefined behaviour,
  - lambda expressions,
  - exception handling (`catch`/`throw`)
  - `new`, `delete`, `dynamic_cast`, and similar.

  These could be in the function, but not be part of the execution path followed on evaluation.
- For the full list see `https://isocpp.org/files/papers/N3652.html`

constexpr functions

- Therefore, from C++14 we can have:

```
constexpr int factorial(int n){
  int f = 1;
  for(int x=1 ; x <= n ; x++){
    f *= x;
  }
  return f;
}
```

- See `constexpr_14.C` for full example.
- This includes an example of signed char overflow that I think should fail to compile (since undefined behaviour should not be permitted), although `gcc-8` and `clang-6` allow it.

Outline

Uniform initialization

- If we want to initialize members of a class at construction time, we can use:

```
struct A{
  double x;
  double y;
};

struct B{
  B() : a{1.0, 3.1}, c{4.2}
  {
  }

  A a;
  double c;
};
```

- where previously we would have had to initialize the elements of `a` within the body of `B()`, or via a constructor for `A`.
- This brings improvements for initializing `const` member data.

Uniform initialization

- Further, we can initialize elements of a newly allocated array:

```
A* data = new A[2]{{1.0, 9.8}, {3.2, 9.1}};
```

  although this is not very well self-documenting.
- See `Examples/uniform_init.C` for full details.

Initializer lists

- In C++03 initializing a `std::vector` of values was annoying:

```
std::vector<int> a(4);
a[0] = 1; a[1] = 2; a[2] = 3; a[3] = 5;
```

  (Even initializing from `int a2[4] = {1,2,3,5}` is awkward.)

- There is an easier way in C++11:

```
std::vector<int> a{1, 2, 3, 5};
```

- This works for `std::list` as well.
- Similarly, for a `std::map`:

```
std::map<int,double> b{ {1, M_PI}, {2, M_E}, {6, 9.80665} };
```

  which is using uniform initialization for individual `std::pair` elements, and an initializer list overall.

- See `Examples/init-list.C`

Initializer list constructors

- How can you use this syntax in your own constructors?
- We would like to have:

```
Matrix rotate{{cos(theta), -sin(theta)},
              {+sin(theta), cos(theta)}};
```

- C++11 provides a special type which is passed to constructors: `std::initializer_list`, requiring the header `#include <initializer_list>`
- This acts as a generic container, which can be iterated over, with the bare minimum of `begin()`, `end()`, `size()`

Initializer list constructors

For a 2×2 matrix class:

```
Matrix::Matrix(std::initializer_list<
                          std::pair<double, double>> args){
  size_t i=0;
  for(auto it = args.begin() ; it != args.end() ; ++it, ++i){
    data[i][0] = (*it).first;
    data[i][1] = (*it).second;
  }
}
```

```
Matrix m{{0, 1}, {3, 4}};
```

(`auto` will be introduced shortly.)

- For each pair of `doubles` in the provided list, we set the elements of `data[2][2]`.
- See `Examples/init-list.C`
- This could be extended to generic-sized matrices via an `initializer_list<initializer_list<double>>`
- See `Examples/init-list-general.C`

Outline

Automatic type declarations

- Consider the following C++03 code:

```
int countPassengers(const std::list<const Vehicle*>& vehicles){
  int numPass = 0;
  for(std::list<const Vehicle*>::const_iterator
      it = vehicles.begin(); it != vehicles.end(); ++it){
    numPass += it->numPass();
  }
  return numPass;
}
```

- The type of `it` is complicated to type, and adds length to the code line without adding useful information.
- In C++11 we can type:

```
for(auto it = vehicles.begin() ;
      it != vehicles.end() ; ++it ){
```

- The `auto` keyword declares the variable `it` to be the exact type on the right of the equality.
- (For an even shorter approach see later slides.)

Automatic type declarations ctd

- The `std::max` function takes the form:

```
template<typename T> max(const T& t1, const T& t2){ ... }
```

  which causes an error if you try to call `std::max(1, 2.0)`.
- The solution is to use `std::max<double>(1, 2.0)`.
- Consider trying to write your own version:

```
template<typename T1, typename T2>
TYPE max(const T1& t1, const T2& t2){
  return (t1 > t2) ? t1 : t2;
}
```

- The problem is: what goes in place of the `TYPE`?
- It can't be `T1` or `T2` themselves.
- We can use `typename std::common_type<T1, T2>::type`
- This is defined to be the resulting type of `(true) ? t1 : t2`.
- (Strictly it's `(true) ? declval<T1>() : declval<T2>()`)
- See `Examples/max.C`

Automatic type declarations ctd

- A similar problem to the above is trying to write:

```
template<typename T, typename S>
?? operator*(const std::vector<T>& v, const S& s)
```

  i.e. multiplication of a `vector` class with elements of type `T` by a scalar of type `S`.

- What if `T` = `int` and `S` = `double`?
- You could start using `std::common_type` but this may not work if you have your own types.
- For example, consider a `vector` containing elements of type `Matrix` that all need to be multiplied by a scalar.
- `std::common_type` is not defined for this.

Automatic type declarations ctd

- The solution is `auto` combined with `decltype`:

```
template<typename T, typename S>
auto operator*(const std::vector<T>& v, const S& s) ->
std::vector<decltype(T{} * S{})> { ... }
```

- This syntax declares the result to be a `std::vector` of the type that would result from the product of scalars of types `T` and `S`.
- Equivalently, use `decltype(v[0] * s)`; the function parameters can be used, hence this has to come at the end of the line.
- See `Examples/vector.C`