Part V

C++11/14/17

Outline

Variadic macros

- Macros can now have a variable number of arguments in C++11:

```
#define PRINT(X, ...) printf(X "\n", __VA_ARGS__);

int main(void){
  PRINT("Error: %s", "Message");
  PRINT("Error: %s %d", "Message", 42);
}
```

- `__VA_ARGS__` is replaced by all the remaining parameters to the macro.
- This is not a good example of its use, and I am unable to think of one. This suggests you should not use it...
- See `Examples/variable_macros.C`

Variadic templates

- Templates can now have a variable number of template parameters

```
void print(){}

template<typename T, typename... Args>
void print(const T& t, Args... a){
  std::cout << t << std::endl;
  print(a...);
}
int main(void){
  print("Hello", 10.9, 11u);
}
```

- The ... syntax indicates that the parameter pack `Args` is expanded.
- The ... syntax can be used wherever a list of elements is required.

Variadic templates...

Example use of variadic templates to create custom version of `std::tuple`:

```
template<typename... Params> struct Tuple;

template<typename T, typename... OtherParams>
struct Tuple<T, OtherParams...> : Tuple<OtherParams...>{
  Tuple(T p, OtherParams... o) : Tuple<OtherParams...>(o...),
    param(p){
  }

  template<int N>
  typename std::enable_if<(N > 0), typename EltType<(N>0) ?
    N-1 : 0, OtherParams...>::type>::type get()const{
     return Tuple<OtherParams...>::template get<N-1>();
  }

  template<int N>
  typename std::enable_if<N == 0, T>::type
  get()const{
     return param;
  }
  T param;
};
```

Variadic templates...

- Extraction of the N'th type from a parameter pack as used in the previous slide:

```
template<int N, typename... Elts> struct EltType;

template<int N, typename T, typename... Elts>
struct EltType<N, T, Elts...>{
  typedef typename EltType<N-1, Elts...>::type type;
};

//! Recursion-ending specialisation
template<typename T, typename... Elts>
struct EltType<0, T, Elts...>{
  typedef T type;
};
```

- See `Examples/variadic_tuple.C`.
- Use `t.get<1>()` to access element 1 of the tuple.

Variadic templates...

- As a further example of the power of parameter packs:

```
template<typename F, typename... Args>
double fIncreased(F f, Args... x){
  return f((x+1)...);
}

std::cout << "f(4,5,6) = " << fIncreased(f, 3,4,5) << std::endl;
```

- The `(x+1)...` translates into `x+1` for each of `Args`.
- See `Examples/variadic_templates.C`

Folding expressions

- The following works from C++17 only
- When using template parameter packs, you may wish to apply an operation to combine all elements into one:

```
template<typename... T>
int sum(T... b){
  return (b + ...);
}
```

  produces a function that sums all values passed to it, and is known as a unary right fold.
- Most operators can be used here, and can have initial left or right operands:

```
template<typename... T>
int startMinusSum(int a, T... b){
  return (a - ... - b);
}
```

  and this is known as a binary left fold.

Folding expressions ctd

- However:

```
template<typename... T>
bool equal(T... b){
  return (b == ...);
}
```

  does not do what you want: `b1 == b2 == b3` is unlikely to be useful.

- In fact `gcc` produces an error with this case due to lack of parentheses.
- In this case, the answer is to write:

```
template<typename S, typename... T>
bool equal2(S a, T... b){
  return ( (a == b) && ... );
}
```

- See `Examples/folding_expr.C` for full code.

Empty folding expressions

- What happens if there are no arguments to a folding expression?
- For `&&` the answer is `true`
- For `||` the answer is `false`
- For `,` the answer is `void()`
- For example, an empty sum is undefined. The answer is not zero, as for non-numeric arguments, the identity element for addition may not be equivalent to zero.
- Similarly, an empty multiplication is not `1`.

Outline

Templated aliases

- In C++11, typedefs can be templated:

```
template<typename T>
using ListConstIter = typename std::list<T>::const_iterator;
```

- This provides a shorthand for a constant iterator over a list of elements of type `T`.

```
std::list<int> a{0,6,9,13,-14};
ListConstIter<int> b = a.begin();
```

- This particular example is probably better done with `auto`, but the principle stands.
- See `templated_typedef.C`.

Templated variables

- As well as templated functions, classes, and types, C++14 also allows templated variables.
- This may seem odd at first; surely the value of a variable defines its type, and templating it is worthless?
- However:

```
template<typename T> T epsilon = 0;
template<> float epsilon<float> = 1e-6;
template<> double epsilon<double> = 1e-12;
```

  could be useful, instead of using something dependent on `std::numeric_limits`
- See `Examples/templated_variables.C`
- Other examples include a `Matrix<T>`-type with an identity defined for a number of types `T`.

Part VI

C++14 specific

Numeric literals

- Binary literals may now be specified as:

```
int answer = 0b101010; // answer = 42
```

- Groups of digits can be separated using ':

```
int billion = 1'000'000'000;
```

  This is for readability purposes only (and may not play well with automatic highlighting in your text-editor).

- See `Examples/literals.C`

Attributes

- Functions, variables, types, and other C++ constructs are permitted to have attributes; extra information not inherent in their definition.
- Many compilers support their own attributes, typically providing hints that may improve performance.
- As of C++14, the only attributes allowed are:
  - `[[noreturn]]` For functions that never return
  - `[[deprecated("reason")]]` (where "reason" may be omitted) - provides a compile-time warning that a function is deprecated
- For example, this may allow the compiler to make certain optimizations.

```
[[noreturn]]
void abort(const std::string& msg, const std::string& file, int
    line){
  std::cout << "Abort: " << file << ":" << line << std::endl;
  std::cout << "Due to: " << msg << std::endl;
  exit(1);
}
```

Attributes ctd

- Or, for functions you want to discourage yourself or others from using:

```
[[deprecated("Use the more general iterator form.")]]
void sort(const std::vector<int>& a)
{
}
```

will print a message at compile time, if the function is used:

```
void sort(std::vector<int>&) is deprecated:
Use the more general iterator form
```

- See `Examples/attributes.C` for full code.

Other attributes

- Compilers may also support their own attributes.
- Those supported by `gcc` include (in any C++ version):
  - `[[gnu::aligned(32)]]` to align `x` on a memory address a multiple of 32 bytes.
  - `int myTmp [[gnu::unused]];` to suppress a compiler warning that a variable is unused.

Return type deduction

- The `auto` keyword in C++11 only applied to variables and to functions with `decltype`.
- C++14 allows the return type of any function to be deduced automatically:

```
template<typename T, typename S>
auto product(T t, S s){
  return t * s;
}
```

- If used within a header file, the function definition must be seen before it is used (to deduce the return type).

Return type deduction ctd

- Also, since C++ parsing is top-to-bottom, recursive functions must be arranged carefully:

```
auto factorial(int i){
  if(i<= 1){
    // Return type deduced to be 'int' here.
    return 1;
  }
  else{
    return factorial(i-1) * i;
  }
}
```

works, but reversing the `if` statement fails because the return type must be deduced first.

- See `Examples/return_type_deduction.C` for details.
- I suggest that `auto` is used sparingly, and only to avoid long typenames, or typenames deduced from template-constructs.
- In theory almost everything could have `auto`, but that way Python/JavaScript/lack of clarity lies...

auto lambda parameters

- In C++11 lambda function parameters had to have an explicit type; now they can have `auto` type:

```
int s = 5;
std::for_each(data.begin(), data.end(),
              [s](auto x){return x + s;});
```

  so that we do not need to explicitly find the `value_type` of `data`.
- Also, lambda functions without local capture can be converted to C-style function pointers:

```
auto f = [](auto x){return x + 5;};
int (*add5)(int) = f;
float (*add5f)(float) = f;
```

- giving two concrete function pointers that add 5 to either an integer or a float.
- See `Examples/lambda_14.C` for full example.

Member and aggregate initialization

- If a class/struct is initialized using an initializer list, then values defined in the class are used if the aggregate does not contain enough:

```
struct X{ int a; int b; int c = 9; };
X x = {2, 3};
```

- The above will fail to compile in C++11 (`c` cannot be initialized), but will succeed in C++14 (`c` $= 9$).
- Of course, `X y;` will succeed in any C++ version.

Part VII

C++17 specific

Outline

Preprocessor

- Trigraphs are no longer allowed. For example, ??= was equivalent to # in earlier C++ standards. This was for very old small keyboards without certain characters.
- The `__has_include` expression is supported to test whether a particular header file is available within the header search path:

```
#if __has_include(<qt4/Qt/qconfig.h>)
#define HAVE_QT
#endif
```

- I would suggest that a proper appreciation of `autoconf`, GNUMake, CMake, and similar tools would be of more use.

Minor clean-ups

- The operator `++` on a boolean type no longer exists.
- The `register` keyword is now removed; you can use it as a variable name. In C it indicates that a variable should be put in a CPU register. In practice it now makes little difference to performance in C anyway.

static_assert

The following minor modification of `static_assert` is now supported:

```
template<int D>
class A{
  static_assert(D >= 0);
};
```

I.e. without the human-readable message required in C++11.

Hexadecimal floating point literals

For some purposes, it is useful to specify floating point numbers w.r.t. base 2:

```
const double quarter = 0x1.0p-2;
double three_eighths = 0x0.cp-1;
```

The first should be obvious; the second expands in binary as $(\frac{1}{2} + \frac{1}{4}) \times 2^{-1}$ since `0xc = 1100b`.

```
std::cout << std::hexfloat <<
    std::numeric_limits<double>::epsilon()
```

displays `0x1p-52` since double-precision has machine epsilon 2^{-52}.

byte type

- In order to allow clearer distinction between numbers for arithmetic or text (`char`, `unsigned char`) and pure memory storage, C++17 introduces `std::byte`.
- It is defined as:

```
enum class byte : unsigned char {};
```

- and can be used as:

```
std::byte a{0x49};
std::vector<std::byte> v(10, a);
std::cout << "v[3]=" << std::hex << (int)v[3] << std::endl;
```

- Note the casting required not to print as a text character.

Outline

Attributes

- C++17 introduces further attributes:
- `[[fallthrough]]` - suppress potential compiler warning when `case` statements allow fall-through.

```
int input;
switch(input){
  case 1:
  case 3:
    std::cout << "Input is less than 4" << std::endl;
    [[fallthrough]];
  case 5:
    std::cout << "Input is odd" << std::endl;
    break;
}
```

- Without the attribute, this could cause the compiler to produce a warning or error.

attributes

- `[[nodiscard]]` - Causes compiler to give warning if a value returned from a function is ignored.
- `[[maybe_unused]]` - Marks a variable or function parameter as unused, which allows compiler to suppress a warning about unused variables.

```
template<typename T>
[[nodiscard]] bool sendMsg([[maybe_unused]] const T* src,
    [[maybe_unused]] size_t n){
  return false;
}
```

- where the **sendMsg** function returns success or failure error code.
- We could leave the variables un-named, but what if we want to document them (e.g. with Doxygen)?
- See `Examples/attributes_17.C` for full code.
- Also note that attributes can be applied to namespaces and enumerators, but none are given in the standard.

attributes

- Multiple attributes in the same namespace can be specified as:

```
int myTmp [[ using gnu: unused, aligned(32) ]];
```

instead of

```
int myTmp [[ gnu::unused, gnu::aligned(32) ]];
```

Construction of aggregates

- C++ has always allowed aggregate initialization:

```
struct S{int s; double f;};
S a{42, 3.142};
```

- However, only from C++17 is initialization of base-classes using this approach allowed:

```
struct Name : S{ std::string n; };
Name n{ {10, 2.3}, "Ford"};
```

- The first element corresponds to the initialization of the base-class `S`. The second corresponds to the element `n`.
- Multiple base-classes are supported, in order:

```
struct Nested : S, Name { char a; };
Nested p{ {10, 3.2}, { {9, 1.2}, "Frankie" }, 'b' };
```

- Note that virtual base-classes are not allowed to be initialized in this way.
- See `Examples/aggregate_init.C` for full code.

inline variables

- Now allowed to have multiple definitions of `extern` variables, so long as they have `inline`, and there is a definition of the variable in each translation unit in which it is used.
- Similar to inline functions

Outline

New versions of if and switch

- In some cases, you may have an **if** or **switch** statement that depends on a variable whose value is not needed outside the test:

```
auto iter = myMap.find(10);
if( iter != myMap.end() ){
  return iter->second;
}
else{
  std::cout << "Key 10 not found" << std::endl;
}
```

- This can now be rewritten:

```
if( auto iter = myMap.find(10); iter != myMap.end() ){
  return iter->second;
}
else{
  std::cout << "Key 10 not found" << std::endl;
}
```

New versions of if and switch

- The advantage is that `iter` does not leak into the surrounding scope. It is not needed outside of the `if` scope.
- This could be more important if initialization of `iter` required some form of resource allocation that should be released after the `if`.
- Similarly, `switch( init ; testvalue )` exists.
- See `Examples/if_17.C` for full code.