C++11 Topics

(+) Initializer lists extended to complex types, e.g. vector:

```
vector<string> v = { "xyzzy", "plugh", "abracadabra" };
```

(+) Type inference - “auto” keyword and “decltype”

SEE itr2.cpp sample code.

decltype() can be used to determine the type of another variable — useful if you have an “auto” variable and later want to declare a variable of the same type:

```
auto x = some_complex_expression;

….

decltype(x) y = some_other_complex_expression_possibly_involving_x;

for (auto i = 0; i < document_vector.size(); i++) {
        /* loop body */
}
```

(+) Range-based for loop

```
int my_array[5] = {1, 2, 3, 4, 5};
// double the value of each element in my_array:
for (int &x : my_array) {
    x *= 2;
}
// similar but also using type inference for array
elements
for (auto &x : my_array) {
    x *= 2;
}
```

(+) Lambda functions and expressions

```
[](int x, int y) -> int { return x + y; }
```

Variable “total” is captured by reference:

```
vector<int> some_list{ 1, 2, 3, 4, 5 };
```

```
int total = 0;
for_each(begin(some_list), end(some_list), [&total](int
x) {
    total += x;
});
```

Note the for_each loop — this was defined in C++98.

(+) Alternative function syntax

```
template<class Lhs, class Rhs>
  decltype(lhs+rhs) adding_func(const Lhs &lhs, const
Rhs &rhs) {return lhs + rhs;} //Not legal C++11
```

Not legal because “lhs" and “rhs" are undefined before function is parsed.

```

template<class Lhs, class Rhs>
  auto adding_func(const Lhs &lhs, const Rhs &rhs) ->
decltype(lhs+rhs) {return lhs + rhs;}
```

Can use this syntax generally, e.g.

```
struct SomeStruct  {
    auto func_name(int x, int y) -> int;
};

auto SomeStruct::func_name(int x, int y) -> int {
    return x + y;
}
```

(+) Multithreading and Thread-local storage

SEE pw.cpp and pw_th.cpp code samples

Can use thread_local to create thread-local storage.

(+) Tuples - an example of variadic templates

Templates can have a variable number of template variables, which allows for the creation of tuples:

```
typedef tuple <int, double, long &, const char *>
test_tuple;
long lengthy = 12;
test_tuple proof (18, 6.5, lengthy, "Ciao!");

lengthy = get<0>(proof);  // Assign to 'lengthy' the
value 18.
get<3>(proof) = " Beautiful!";  // Modify the tuple’s
fourth element.
```

(+) Regular Expressions

If you know what they are, you’re glad they’re in C++11…

```
const char *reg_esp = "[ ,.\\t\\n;:]";  // List of
separator characters.

// this can be done using raw string literals:
// const char *reg_esp = R"([ ,.\t\n;:])";

std::regex rgx(reg_esp); // 'regex' is an instance of
the template class
                         // 'basic_regex' with argument
of type 'char'.
std::cmatch match; // 'cmatch' is an instance of the
template class
                   // 'match_results' with argument of
type 'const char *'.
const char *target = "Unseen University - Ankh-
Morpork";

// Identifies all words of 'target' separated by
characters of 'reg_esp'.
if (std::regex_search(target, match, rgx)) {
    // If words separated by specified characters are
present.

    const size_t n = match.size();
    for (size_t a = 0; a < n; a++) {
        std::string str (match[a].first,
match[a].second);
        std::cout << str << "\n";
    }
}
```

(+) Smart Pointers

We already talked about unique_ptr — it takes care of deleting dynamic objects for you.

There is also a shared_ptr which is similar, except it allows multiple pointers to the same object and only deletes the object when there are no longer any pointers referencing it.

```
shared_ptr<int> p1(new int(5));
shared_ptr<int> p2 = p1; //Both now own the memory.

p1.reset(); //Memory still exists, due to p2.
p2.reset(); //Deletes the memory, since no one else
owns the memory.
```

(+) Random Number Generation

Much better random number support. There are three "generators":

linear_congruential_engine
subtract_with_carry_engine
mersenne_twister_engine (** this is a good one **)

and numerous distributions (e.g. gamma, normal, uniform, lognormal, …).

```
#include <random>
#include <functional>
using namespace std;

uniform_int_distribution<int> distribution(0, 99);
mt19937 engine; // Mersenne twister MT19937
auto generator = bind(distribution, engine);
int random = generator(); // Generate a uniform
integral variate between 0 and 99.
int random2 = distribution(engine); // Generate another
sample directly using the distribution and the engine
```

```
objects.
```

(+) Exception specification

The use of exception lists in a function declaration has been deprecated, e.g. don’t do the following (under C++11):

```
void someFunc() throw (DivideByZeroEx, BadFileEx)
```

It’s still okay to say a function will throw no exceptions:

```
void someFunc() throw ()
```

but this can also be done with the new “noexcept” keyword:

```
void someFunc() noexcept
```