All You Need is C++11

Experimenting with the new C++11 features. In this blog I explain some of my adventures with C++11 encountered during my daily development work

Wednesday, July 25, 2012

Printing tuples

One of the power tools of C++11 standard library are **tuples**... a.k.a. `std::tuple<Args...>`. It was not possible to have such utility in C++98 because of the lack of *variadic templates*. Indeed the std::tuple object heavily relies on this feature which has been introduced with the C++11 standard.

Tuples are collections composed of heterogeneous objects of pre-arranged dimensions. A tuple can be considered a generalization of a struct's member variables. It's use is very similar to the `std::pair` class which was available since C++98, however while pairs can only contains 2 generic elements, a tuple can be of undefined size.

The standard way of using tuples in C++11 is the following:

```
auto t1 = std::make_tuple( 10, 3.4f, "awesomeness" );
std::tuple<const int&, int> t2{ std::cref(std::get<0>(t1)), 20 };
auto t3 = std::tie( std::get<0>(t1), std::get<1>(t2) );
std::get<0>(t3) = 2;
assert( (std::get<0>(t1) == 2) && (std::get<0>(t2) == 2) && (std::get<0>(t3) == 2) );
```

gistfile1.cpp hosted with ♥ by **GitHub** **view raw**

Tuple `t1` is constructed using the `std::make_tuple` function is an utility which easy the construction of tuples without worrying about the type of the single elements which is instead inferred thanks to the template mechanism. Another way to build a tuple is shown for tuple `t2` for which we use the tuple class constructor. It is worth noting that we use an *initializer list* (`{ }`) which is again one of the new feature of the C++11 standard (which we will cover one day). When building this tuple, the type of the first element is a *const reference* to the first element of the tuple `t1`. At last, tuple `t3` is constructed using the std::tie function which creates a tuple of lvalue references. Therefore by writing the first element of tuple `t3` we indeed propagate the value to both **t1** and **t2**. And the assert in the last line of the code snippet is satisfied.

When working with tuples, it is sometimes useful, for debugging purposes for example, to print their values to an output stream. This can be done by overloading the **<<**operator, however, because the access to the tuple's elements is strongly typed, we need some metaprogramming magic in order to be able to print each tuple element. The problem is the following, accessing an element of the tuple is only possible via the `std::get` method which takes a constant teamplate parameter representing the index of the the value we want to access. Because this value needs to be a constant expression (otherwise the compiler would not be able to determine the return value of the function) we cannot simply iterate through the elements of a the tuple using a loop iterator. What instead we need to do is use recursion:

```
// Define a type which holds an unsigned integer value
template<std::size_t> struct int_{};

template <class Tuple, size_t Pos>
std::ostream& print_tuple(std::ostream& out, const Tuple& t, int_<Pos> ) {
  out << std::get< std::tuple_size<Tuple>::value-Pos >(t) << ',';
  return print_tuple(out, t, int_<Pos-1>());
}

template <class Tuple>
std::ostream& print_tuple(std::ostream& out, const Tuple& t, int_<1> ) {
  return out << std::get<std::tuple_size<Tuple>::value-1>(t);
}

template <class... Args>
ostream& operator<<(ostream& out, const std::tuple<Args...>& t) {
  out << '(';
  print_tuple(out, t, int_<sizeof...(Args)>());
  return out << ')';
}
```

gistfile1.cpp hosted with ♥ by **GitHub** **view raw**

This method is generic in the sense that it can be used to print any tuple, the output obtained by printing the tuples `t1`, `t2` and `t3` at the exit of the program is:

```
std::cout << t1 << std::endl; // => (2,3.4f,"awesomeness")
std::cout << t2 << std::endl; // => (2,20)
std::cout << t3 << std::endl; // => (2,20)
```

gistfile1.cpp hosted with ♥ by **GitHub** **view raw**

Easy, isn't it? :)