Applied C++11

2013-01-25 Fri

Outline

C++11

- C++11 (formerly known as C++0x) is the most recent version of the standard of the C++
- Approved by ISO(International Organization for Standardization) on 12 August 2011
- Includes several additions to the core language and extends the C++ standard library

Core language usability enhancements:

- Primary purpose is to make the language easier to use
- Improve type safety
- Minimize code repetition
- Make erroneous code less likely

Auto Type Inference

- The compiler can infer the type of a variable at the point of declaration:

```
int x = 5;
auto x = 4;
```

- Very helpful when working with templates and iterators:

```
vector<int> vec;
auto itr = vec.iterator();
// instead of vector<int>::iterator itr
```

Study Case

```
template <typename BuiltType, typename Builder>
void
makeAndProcessObject (const Builder& builder)
{
    BuiltType val = builder.makeObject();
    // do stuff with val
}
```

two necessary template parameters

- type of the "builder" object
- type of the object being built

```
MyObjBuilder builder;
makeAndProcessObject<MyObj>( builder );
```

Study Case (cont.)

- By using 'auto' no longer need to write the specific type of the built object

```
template <typename Builder>
void
makeAndProcessObject (const Builder& builder)
{
    auto val = builder.makeObject();
    // do stuff with val
}
```

```
MyObjBuilder builder;
makeAndProcessObject( builder );
```

Lambda Functions

- Lambda Function: is a function defined, and possibly called, without being bound to an identifier
- you can write inline in your source code
- convenient to pass as an argument to a higher-order function
- allow to create entirely new ways of writing programs

Basic Lambda Syntax

```
#include <iostream>
using namespace std;
int main()
{
    [] () { cout << "Hello, World!"; }();
}
```

- "[]": capture specifier, specifies for the compiler creation of a lambda function
- "()": argument list
- What about return values?

More on the Lambda Syntax

- If there is no return statement, uses "void" as default
- If you have a simple return expression, the compiler will deduce the type of the return value

```
// compiler knows this returns an integer
[] () { return 1; }
```

- specify the return value explicitly

```
// now we're telling the compiler what we want
[] () -> int { return 1; }
```

Example: Sorting a list

- We create a list of cities.

```
// Define a class for a city
class City {
    public:
        int population;
        int foundation_year;
        int surface_area;
};

int main(){
    std::list<City> cities;
    // Fill the list with data
}
```

- To sort a set of objects, we have to define a comparison function.

Example: Sorting a list (cont.)

- In the STL, comparison functions have two arguments, and return true if the first argument goes before the second.
- To sort by population, we can define a compare function:

```
bool compare(City a, City b) {
    return a.population < b.population;
}
...
cities.sort(compare);
```

- Alternatively, we can use a lambda function:

```
cities.sort([](City a, City b)->bool
    {return a.pupulation < b.population;});
```

Multithreading

- C++11 threads are more convenient than directly using Posix threads.

```
#include <thread>

int main() {
...
    // Declaring threads
    std::thread t;

    // Assign a function
    t = std::thread(function, arguments);

    // Finalizing
    t.join();
...
}
```

- Compile with the flag `-pthread`

Multithreading example: Dot product

```
// Definition of a class to store the values
class PartialDP {
    public:
        double dotproduct;
        double *x;
        double *y;
        int first;  // First index of array
        int length; // Count of elements

        PartialDP() : dotproduct(0){}
        set(double *xin, double *yin, int f, int l){
            x = xin; y = yin; first = f; length = l;
        }

        operator () () {
            for (int i = first; i < first + length; i++) {
                dotproduct += x[i] + y[i];
            }
        }
};
```

Multithreading example (cont.)

```
#include <thread>
const int size = 1000, nthreads = 4;
int main(){
    std::thread t[nthreads];
    double x[size], y[size];
    int subsize = size / nthreads;  // Assume no remainder
    PartialDP dp[nthreads];
    for (int i = 0; i < nthreads; i++){
        dp[i].set(x, y, i*subsize, subsize);
        t[i] = std::thread(dp[i]);
    }
    // Compute
    for (int i = 0; i < nthreads; i++) t[i].join();

    // Accumulate the product
    double result = 0;
    for (int i = 0; i < nthreads; i++)
        result += dp[i].dotproduct;
}
```

Compiling C++11

- With gcc, C++11 features can be compiled by using g++ `-std=c++11 <args>`