Namespaces and Templates in C++

Robot Vision Systems - Seminar 1

Mattias Tiger

Cognitive Robotics Group, Knowledge Processing Laboratory
Artificial Intelligence and Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

Linköping University

- Namespace
  - Declaration and usage
  - Using directive
  - Alias
- Template
  - Arguments/Parametrization
  - Specialization
  - Function and Class templates
  - Some examples

- Structure a program into logical units
- Manage variable/function shadowing
  - Allow functions and types to have the same name (within different namespaces)

```
int fn(int a) { return a; } // First fn() declaration

namespace foo {
    int fn(int a) { return a+1; } // Second fn() declaration

    namespace bar { // Nestled namespace
        int fn(int a) { return a+2; } // Third fn() declaration
    }
}

int main() {
    std::cout << fn(1) << "\n";
    std::cout << foo::fn(1) << "\n";
    std::cout << foo::bar::fn(1) << "\n";
    ...
```

- The **using** keyword.

```
namespace foo {
    int fn(int a) { return a; } // First fn() declaration
}

namespace bar {
    int fn(int a) { return a+1; } // Second fn() declaration
}

using namespace bar; // Remove the need to use bar:: when calling bar::fn()

int main() {
    std::cout << fn(1) << "\n"; // Second fn() (bar::fn()) is called
    ...
```

- Namespace **alias**.

```
namespace foo {
    int fn(int a) { return a+1; }
}

namespace bar = foo; // bar is now an alias of foo

int main() {
    std::cout << bar::fn(1) << "\n"; // foo::fn() is called
    ...
```

Can be useful for shortening deep nestled namespaces:

```
boost::numeric::ublas::vector<double> v; // Long
namespace ublas = boost::numeric::ublas; // Alias
ublas::vector<double> v; // Short
```

- Always use namespaces to encapsulate code in projects.
  - It is possible to declare the same namespace in multiple files.
- Be careful with the **using** keyword
  (Can create future conflicts).

```
using namespace foo;
using namespace bar;
```

One day both libraries (foo and bar) include the same function *fn* which is used by you a lot, and there are now errors everywhere...

- Write generic code (same functionality for different types).
- Perform compile-time calculations, code generation and optimizations.
  - Meta-Template Programming (MTP)

```
template <class T>
T max(T a, T b) { return a > b ? a : b; }

int main() {
    std::cout << max(1, 2) << "\n"; // Implicit type declaration
    std::cout << max<int>(1, 2) << "\n"; // Explicit type declaration

    std::cout << max(1.0, 2.5) << "\n";
    std::cout << max('a', 'b') << "\n";

    int v = max(1, 2.0); // Error, max(int, double) not defined!
    ...
```

The template type keywords **class** and **typename** are equivalent for normal use. When declaring template templates however only **class** can be used (beyond this presentation).

```
template <class T>
template <typename T>
```

Generic types and constants are allowed.

```
template <class T, int K>
struct MyStruct {
    T data[K];
};

int main() {
    MyStruct<double,55> doubleArray;

    doubleArray.data[2] = 3;
    ...
```

- Multiple template arguments

```
template <class T>
T max(T a, T b) {...} // Output type same as argument

template <class A, class B>
A max(A a, B b) {...} // Output type same as first argument
```

- Template Specialization

```
template <class A, class B>
A max(A a, B b) {...}

template <class B>
int max(int a, B b) {...} // Specialization of max()

template <>
int max(int a, double b) {...} // Full speicalization of max()
```

2D Point Class

```
template <class T>
class Point2D {
    T x,y;
public:
    Point2D(T xValue, T yValue) : x(xValue), y(yValue) {}
    T getX() { return x; } // Declaration and Definition
    T getY() { return y; } // ^^^^^^^^^^^^^^^^^^^^^^^^^^
    double abs(); // Declaration
};

template <class T>
double Point2D<T>::abs() { // Definition
    return sqrt(getX()*getX() + getY()*getY());
}
int main()
{
    Point2D<int> p(2, 3);
    std::cout << p.getX() << "\n";
    std::cout << p.abs() << "\n"; // Notice that abs() return double
    ...
```

Factorial

```
template <int N>
struct Factorial {
    static const int value = N * Factorial<N-1>::value; //Recursive
};

template <>
struct Factorial<0> {
    static const int value = 1; //End case
};

int main() {
    int x = Factorial<10>::value; // x = 3628800 at compile time
 }
```

Constexpr (C++11) can get the same result sometimes:

```
constexpr int factorial(int n)
{
    return n <= 1 ? 1 : (n * factorial(n-1));
}
```

Variadic Templates

Functions and templates with an arbitrarily number of arguments. (New in C++11)

```
template <class T>
T accumulate(T value) {
    return value;
}

template <class T, class... Args>
T accumulate(T first, Args... rest) {
    return first + accumulate(rest...);
}

int main() {
    int sum = accumulate(1, 2, 3, 7, 8, 9);

    std::string s1 = "a", s2 = " ", s3 = "sum", s4 = "!";
    std::string stringSum = accumulate(s1, s2, s3, s4);
    ...
```

Template Summary

- Compile time code generation and checks. (A time saver)
- Longer build time.
- Templates can be in header files only.
- Heavy templated C++ library code show little resemblance to the pure OOP subset of C++.
- Traditionally hard to debug, but progress has been made.

Further recommended reading (author of Armadillo):
http://conradsanderson.id.au/misc/sanderson_templates_lecture_uqcomp7305.pdf.