Templates

Jonathan Windle

University of East Anglia

J.Windle@uea.ac.uk

May 30, 2017

Overview I

Intro

- Code synthesized for required classes by expanding template.
- Synthesized code is then compiled as well.
- Advantages: No impact on runtime performance, easy for the compiler to optimise code.
- Disadvantages: Need to know what synthesized code is needed at compile time. Larger executable.

Template Functions

- When the function is invoked, the compiler:
  - Deduces the values of the templates parameter.
  - Determines whether it exists already or not.
  - If not, a new instance is generated by expanding the template.
  - Compiles the new instance of the template

```
template <typename T>
void exchange(T& x, T& y){
        T tmp = x;
        x = y;
        y = tmp;
}
```

- Exchanging two integers would result a synthesized function:

```
// New identifier synthesized by compiler
void _exchange_int(int& x, int& y) {
        int tmp = x;
        x = y;
        y = tmp;
}
```

Type Safety

- The compiler performs type checking on synthesized code.
- If trying to swap and integer and a double, there will be an error at compile time.
- Opportunities for problems to be detected:
  - When the template itself is compiled
  - When the compiled detects a use of the template.
  - When the instance of the template is compiled.

Explicit Template Arguments

- Sometimes types cannot be deduced from the invocation.
- E.g Return type of a function:

```
template <typname R, typename X, typename Y>
R ratio(X& x, Y& y) {
        return x/static_cast<R>(y);
}

int main(int argc, char* argv[]) {
        int i = 87;
        short s = 42;

// R set explicitly, X & Y deduced implicitly.
        cout << ratio<int>(i,s) << endl; // 2

        cout << ratio<double>(i,s) << endl;// 2.07134

        return 0;
}
```

- Deduced template parameters must be right most in the list.

Nontype Template Parameters

- They do not have to be typenames:

```
template <int N>
void repeat(const String& s) {
        for (int i = 0; i < N; i++) {
                cout << s << endl;
        }
}

int main(int argc, char* argv[]) {
// Template argument given explicitly
        repeat<3>("Hello world");

        int n = 4;
// Local variable cannot be used as a non-type argument
        repeat<n>("Won't work");
        return 0;
}
```

- Template instantiated at compile time, so arguments must be known at compile time.

Array length can be deduced

```
template <typename T, int N>
int arrayLength(const T(&array)[N]) { // N part of array
    type
        return N;
}
```

- This works because the compiler deduces that the parameter must be type T and size N.

Class Templates

- Used in a very similar way to methods.
- Example definition:

```
template <typename T> class Array {
private:
        T* array;
        int N;
public:
// Method for overloaded operator declared
        T& operator[](int i);
        Array(const int size){
                array = new T[size];
                N = size;
        }
};
// Generic so must be defined as a template.
template <typename T>
T& Array<T>::operator[](int i) {
        return array[i];
}
```

Ranged for loop I

- To work a collection must have:
  - A `begin()` method that returns the iterator to the first element.
  - An `end()` method that returns an iterator for the element just past the end of the collection.
- Iterator is an object that:
  - Overloads the * operator to return a reference to an element.
  - Overloads the $++$ operator to move the iterator onto the next element.
  - Overloads the relational operators, e.g. $==$, $!=$, ¡, ¿ etc...
- Pointers behave like iterators.
  - We can use a pointer for an iterator
  - Example isn't very safe...

Ranged for loop II

```
template <typename T> class Array {
private:
        T* array;
        int N;
public:
        T* begin() { return array; }

        T* end() { return array+N; }

        int length() const { return N; }

        T& operator[](int i) { return array[i]; }

        Array(const int size) {
                array = new T[size];
                N = size;
        }
}
```

The End