ECE326
PROGRAMMING LANGUAGES

Lecture 20 : C++ Template Metaprogramming

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

Template Metaprogramming

- Part of a language that generates code
  - More powerful and complex than generics
- Allows compile-time polymorphism
  - Executing code at compile-time!
- Template instantiation
  - By substituting *template* parameters
  - Generates type-safe code
  - Each instantiation generates new code
    - Abuse can result in very large executable size

C++ Template

- Syntax

```
template<template parameter, …>
function or class definition
```

- **typename** (or class)
  - Indicates the following identifier is a type

```
template<class T>     // T is a type
T min(T a, T b) {
    return a < b ? a : b;
}

template<typename T> // T is a type
class Foo {
    T var;
};
```

Template Instantiation

- When template is used, code is generated by compiler
  - Performs type-safe template argument substitution

```
template<typename T> // T is a type
T min(T a, T b) {
    return a < b ? a : b;
}
```

min<int>

```
// template instantiated
int min(int a, int b) {
    return a < b ? a : b;
}
```

```
// in main
std::cout << min<int>(4, 9); // prints 4

// for template function only, you can omit <int>
std::cout << min(3, 7); // template parameter inferred
```

C++ Template

- Type must support all used operations in template
  - E.g. operator< must be supported by type T
- No trait bounds
  - Must read template body to see what type T must do
  - Compiler error for templates is very hard to read

```cpp
template<typename T>
T min(T a, T b) {
    // template assumes T supports the less than operator
    return a < b ? a : b;
}
```

Code Organization

- Review
  - #include *copy-pastes* content of file into current file
- Convention
  - Put class definition and function *declaration* in header file (.h)
  - Put function and method *definition* in source file (.cpp)
- Template
  - Put all template definition in header file
    - Unless you do not want other source file to have access to it

Template Parameter

- Template parameter can be *anything*
  - Does not have to be a type
    - E.g. can be int or float
- Template arguments must be *compile-time* values
  - Types or constants

```
template<typename T, int X>
void normalize(T array[], int size) {
    for (int i = 0; i < size; i++) {
        array[i] /= (T)X; // X is a compile-time constant int
    }
}
```

Multiple Parameters

- Template functions can have multiple type parameters
  - Instantiation may require disambiguation

```
template<typename T, typename F>
T convert(F v) {
    return (T)v;
}

double foo(double d) {
    int i = convert<int>(d);   // from double to int
    char c = convert<char>(i); // from int to char

    /* instantiate convert<double, char> */
    double(*func)(char) = convert; // function pointer
    return func(c) + 1.2;
}
```

Template Specialization

- Specialization for a particular template argument
- Benefit
  - Code does not make it to executable if not used!

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

template<> // this syntax signifies specialization
const char * min(const char * a, const char *b) {
     // use strcmp to compare c-strings
    return strcmp(a, b) < 0 ? a : b;
}
```

Template Class

- You have used many in ECE297
  - std::vector
  - std::priority_queue
  - std::unordered_map
- Unlike template functions, type cannot be inferred.
  - Must explicitly specify template arguments

```
std::vector<int> v = {1, 2, 3};
// auto keyword will type inference, otherwise compiler-error
for (auto it = v.begin(); it != v.end(); it++) {
    std::cout << *it << std::endl;
}
```

Template Class

```
template<class T> // same as typename T
class Stack {
public:
    Stack(unsigned cnt=8);
    ~Stack() { delete [] array; }
    bool push(const T &); // push element to top of stack
    bool pop();           // remove current top of stack
    T * top();            // return current top of stack
    int empty() const { return end == -1; }
    int full() const { return end == count - 1; }
private:
    int count;  // capacity of stack
    int end;   // points to top of stack
    T * array;
} ;
```

Member Functions

- Can be defined outside of class template
  - But must be defined together, preferably in same header file

```
template<class T>
Stack<T>::Stack(unsigned cnt) : count((int)cnt), end(-1),
    array(new T[cnt]) {}

template<class T>
bool Stack<T>::push(const T & item) {
    if (full()) return false;
    array[++end] = item;
    return true;
}
```

Member Functions

```
template<class T>
bool Stack<T>::pop() {
    if(empty()) return false;
    --end;
    return true;
}

// general rule of thumb:
// return a pointer if you want to indicate error with nullptr
// return by reference (T &) if no error is possible, OR
// if you plan to crash the program upon error
template<class T>
T * Stack<T>::top() {
    if (empty()) return nullptr;
    return &array[end];
}
```

typedef

- Creates a new name for a type

```
typedef old_type_name new_type_name_1, new_type_name_2, …;
```

- Common use
  - Indicate different usage
    - E.g. size_t instead of unsigned long to confer context
  - Reduce length of type names, or improve appeal

```
/* this is bad – hides the fact that it's a pointer */
typedef char * cstring;
typedef Stack<double> DoubleStack;
typedef vector<int> IntVector;
typedef int (* compare_f)(int, int);
```

Sample Use

```
DoubleStack ds = DoubleStack();
double * dp, d = 1.1;
cout << "Pushing elements onto stack" << endl;
while (ds.push(d)) {
    cout << d << ' '; d += 0.7;
}
cout << endl << "stack full" << endl
     << endl << "Popping elements from stack" << endl ;
while ((dp = ds.top())) {
    cout << *dp << ' '; ds.pop();
}
cout << endl << "stack empty" << endl ;
```

```
Pushing elements onto stack
1.1 1.8 2.5 3.2 3.9 4.6 5.3 6
stack full
```

```
Popping elements from stack
6 5.3 4.6 3.9 3.2 2.5 1.8 1.1
stack empty
```

Default Template Parameter

- Template parameters can take defaults too

```
template<class T=int, int C=16>
class Stack {
    int end;
    T array[C]; // static array size
public:
    Stack() : end(-1) {}

    bool push(const T &);
    bool pop();
    T * top();

    int empty() const { return end == -1; }
    int full() const { return end == C - 1; }
};
```

Template in Template

- Templates can use other templates

```
template<class T=char, size_t C=8>
class Stack {
    std::vector<T> array;
public:
    Stack() {}

    bool push(const T &);
    bool pop(); T * top();

    int empty() const { return array.size() == 0; }
    int full() const { return array.size() >= C; }
};
```

Template as Template Argument

- Templates arguments can take parameterized types

```
Stack<vector<char>> st = Stack<vector<char>>();
std::vector<char> * vp, v = { 'a' };

cout << "Pushing elements onto stack" << endl;
while (st.push(v)) {
    cout << v << ' '; v[0] += 2;
}

cout << "Popping elements from stack" << endl;
while ((vp = st.top())) {
    cout << *vp << ' '; st.pop();
}
```

Partial Specialization

- Example: Map template class

```
template<class K, class V, int S>
class Map {
    struct Pair { K key; V value; } array[S];
public:
    …
    V * find(const K & k) {
        int i;
        for (i = 0; i < S; i++) {
            if (array[i].key == k)
                return &array[i].value;
        }
        return nullptr;
    }
};
```

Partial Specialization

- Specialize only some of the template parameters

```
template<class V, int S>      // do not specify K
class Map<int, V, S> {        // K is fixed to int
    V values[S];
    bool valid[S];
public:
    …
    V * find(int k) {
        if (k < 0 || k >= S)
            return nullptr;
        if (!valid[k])
            return nullptr;
        return &values[k];
    }
};
```

Template Aliases

- Similar to typedef, but also works for partial templates
- Not the same as partial specialization
  - Only gives a different name to templates

```
template<class K, class V, int S>
class Map { … };

template<class K, int S>
using StringMap = Map<K, std::string, S>;
```

Recursive Template

- Template using a different instantiation of itself
  - Can be used to generate compile-time expression

```
template<int F>
struct Factorial {
    enum { value = F * Factorial<F - 1>::value };
};

template<> // base case template specialization
struct Factorial<1> {
    enum { value = 1 };
};

// e.g. Factorial<3>::value = 3 * Factorial<2>::value
//      Factorial<2>::value = 2 * Factorial<1>::value
//      Factorial<1>::value = 1
```