Object-Oriented Programming

C++ Templates

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Generic Programming

Generic Programming is a programming language paradigm where algorithms are written in terms of types to be specified later and then instantiated when needed for specific types of parameters

- One of many paradigms supported by C++, along with procedural programming (functions) and object-oriented programming

- Replaces the need for variant types (one variable can store many different types of data, but typically not at the same time)

- Replaces the need for much of the use of inheritance

C++ Templates

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

- A feature of the C++ programming language that supports *generic programming*
- Can write code that works with multiple data types without duplicating the code for each type.
- Templates can be applied to both functions and classes
- Code generated from templates can be heavily optimized
- Much of the power and flexibility of the C++ Standard Library comes from the use of C++ templates
- All te
com

Function Template

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

- `template` keyword
- Template parameters enclosed in angle brackets
- `typename` keyword used to declare the template parameter

Instantiating Function Templates

```
#include <string>

using namespace std::literals;

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {

    add(5,4);
    add(5.0, 4.0);
    add(5L, 4L);
    add<std::string>("Hello ", "World!");
    add("Hello"s, "World!"s);

    return 0;
}
```

- Compiler determines T from the types of parameters in the call

- *Instantiates*(i.e., generates) a separate function for each type

- Can also explicitly declare the template parameter value (typically a type)

- Can expand as much as needed to new types

Non-Type

```
template<int I>
int incr(int n) {
    return n + I;
}

int main() {

    incr<2>(4);
    incr<2>(5);
    incr<10>(4);

    return 0;
}
```

- Template parameters do not have to be types
- When a template is instantiated with specific types or values, the compiler generates a specialized version of the template code for those types or values

Class Templates

```cpp
#include <vector>
#include <stdexcept>

template <typename T>
class Stack {
public:

    // push
    void push(const T& element) {
        elements.push_back(element);
    }

    // pop
    void pop() {
        if (elements.empty())
            throw std::out_of_range("Empty stack");

        elements.pop_back();
    }

    // top
    T top() const {
        if (elements.empty())
            throw std::out_of_range("Empty stack");

        return elements.back();
    }

    // empty
    bool empty() const {
        return elements.empty();
    }

private:
    std::vector<T> elements;
};
```

```cpp
int main() {
    Stack<int> sizes;
    sizes(1);

    Stack<std::string> words;
    words.push("Hello");
    words.push("World");

    return 0;
}
```

Fibonacci

```
int fibonacci(int n) {

    if (n == 0)
        return 0;

    if (n == 1)
        return 1;

    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

```
template <int N>
struct Fibonacci {
    static constexpr int value =
        Fibonacci<N - 1>::value + Fibonacci<N - 2>::value;
};

template <>
struct Fibonacci<0> {
    static constexpr int value = 0;
};

template <>
struct Fibonacci<1> {
    static constexpr int value = 1;
};
```

C++ Template Fibonacci Sequence

```cpp
template <int N>
struct Fibonacci {
    static constexpr int value =
        Fibonacci<N - 1>::value + Fibonacci<N - 2>::value;
};

template <>
struct Fibonacci<0> {
    static constexpr int value = 0;
};

template <>
struct Fibonacci<1> {
    static constexpr int value = 1;
};
```

```cpp
int main() {

    static_assert(Fibonacci<0>::value == 0);
    static_assert(Fibonacci<1>::value == 1);
    static_assert(Fibonacci<2>::value == 1);
    static_assert(Fibonacci<3>::value == 2);
    static_assert(Fibonacci<4>::value == 3);
    static_assert(Fibonacci<5>::value == 5);
    static_assert(Fibonacci<6>::value == 8);
    static_assert(Fibonacci<7>::value == 13);

    return 0;
}
```

Multiple template parameters

```cpp
#include <string>

using namespace std::literals;

template <typename T1, typename T2>
auto add(T1 a, T2 b) {
    return a + b;
}

int main() {

    add(5,4);
    add(5.0, 4);
    add(5, 4.0);
    add(5.0, 4.0);
    add("Hello ", "World!"s);

    return 0;
}
```

Default template parameters

```cpp
#include <vector>
#include <deque>

template <typename T, typename Container = std::vector<T>>
class Stack {
    // ...
private:
    Container data;
};

int main() {

    Stack<int> s1;

    Stack<int, std::deque<int>> s2;

    return 0;
}
```

Constrained Overloading

```cpp
#include <type_traits>
#include <vector>
#include <stdexcept>

template <typename T>
class Stack {
public:

    void push(T element) requires std::is_scalar_v<T> {
        elements.push_back(element);
    }

    void push(const T& element) requires (!std::is_scalar_v<T>) {
        elements.push_back(element);
    }

    void pop() {
        if (elements.empty())
            throw std::out_of_range("Empty stack");
        elements.pop_back();
    }

    const T& top() const {
        if (elements.empty())
            throw std::out_of_range("Empty stack");
        return elements.back();
    }

    bool empty() const { return elements.empty(); }

private:
    std::vector<T> elements;
};

int main() {
    Stack<int> si;     si.push(42);
    Stack<std::string> ss;  ss.push("hello");
}
```

Template Disadvantages

- Compiler needs access to function or method definitions to instantiate, leading to definition in the include file

- Instantiation for each type may lead to *code bloat*

- Code may become difficult to understand

- Error messages when types with the wrong interface are used

C++ Concepts

```cpp
template<Sortable Cont>
void sort1(Cont& container);

template<typename Cont>
    requires Sortable<Cont>()
void sort2(Cont& cont);

list<int> lst = ...;
sort1(lst);
```

- First introduced as "C++0x Concepts" in C++11, but were removed
- Concepts Lite were merged into C++20

C++ Concepts and Errors

```
template<Sortable Cont>
void sort1(Cont& container);

template<typename Cont>
    requires Sortable<Cont>()
void sort2(Cont& cont);

list<int> lst = ...;
sort1(lst);
```

```
error: no matching function for call to ‘sort(list<int>&)’
     sort(l);
          ^
  note: candidate is:
  note: template<Sortable T> void sort(T)
```

Extension Points

- Lambda functions passed to std::function fields/data members

- Direct inheritance from a Base class

- Inheritance from Handler used by a Base class

Observations

- Lambda functions with capture are very flexible and quick to implement in code

- Lambda functions passed to function pointer parameters are very fast

- However, lambda functions, when passed to std::function (that allows a non-empty capture), are very slow

C++ Templates

- Using C++ Templates with lambda functions is very fast

- The compiler can inline lambda functions in templates

- No use of slow std::function

- How `std::sort()` works

- How about as extension points?

Template Function Extension

```cpp
#include <vector>

template <typename Iterator, typename Comparison>
void sort(Iterator begin, Iterator end, Comparison compare) {
    // ...
    if (compare(*begin, *std::next(begin)))
        ;
    // ...
}

int main() {

    std::vector<int> v{ 2, 1, 5, 3, 4 };

    sort(v.begin(), v.end(), [](int n1, int n2)->bool { return n1 < n2; });

    return 0;
}
```

Template Class Extension

```cpp
template <typename compare>
class Process {
public:
    void process() {
        // ...
        if (compare(value1, value2))
            ;
        // ...
    }
private:
    int value1;
    int value2;
};

int main() {

    auto compare = [](int n1, int n2)->bool { return n1 < n2; };

    Process<decltype(compare)> processor;

    return 0;
}
```

Template Class Extension

```cpp
#include <string>

template <typename compare, typename compute>
class Process {
public:
    void process() {
        // ...
        if (compare(compute(value1), value2))
            ;
        // ...
    }
private:
    int value1;
    int value2;
};

int main(int argc, char* argv[]) {

    int adjust = std::stoi(argv[1]);

    auto lessthan = [](int n1, int n2)->bool { return n1 < n2; };
    auto addAdjust = [adjust](int n)->int { return n + adjust; };

    Process<decltype(lessthan), decltype(addAdjust)> processor;

    return 0;
}
```

Speed of Parameter Types

- Choices:

  Function pointer

  `std::function`

  Template parameter

- Clang Comparison
- GCC Comparison

Extension Points

- Lambda functions to std::function fields/data members

- Direct inheritance from a Base class

- Inheritance from Handler used by a Base class

- C++Templates for classes with lambda functions

C++ Templates as Extensions Points

- Lots of flexibility
- Allows the use of the capture
- Code is (potentially) inlined
- All the advantages of lambdas to std::function, without the speed disadvantage
- One Drawback: Have to declare in the include file (*.hpp*) as the compiler needs access to the definitions

Extension Points

- Lambda functions to std::function fields/data members

- Direct inheritance from a Base class

- Inheritance from Handler used by a Base class

- C++Templates for classes with lambda functions