C++ Templates

SPECFEM++ Hackathon

Congyue Cui

When two functions are similar

```
int timesTwo(int x) {
    return x * 2;
}

double timesTwo(double x) {
    return x * 2;
}

int main() {
    std::cout << timesTwo(10) << std::endl;
    std::cout << timesTwo(10.5) << std::endl;
    return 0;
}
```

Merge them in one

```cpp
template <typename T>
T timesTwo(T x) {
    return x * 2;
}

int main() {
    std::cout << timesTwo(10) << std::endl;
    std::cout << timesTwo<int>(10) << std::endl;
    std::cout << timesTwo(10.5) << std::endl;
    std::cout << timesTwo<double>(10) << std::endl;
    return 0;
}
```

Merge them in one

There are still two functions created

```cpp
template <typename T>
T timesTwo(T x) {
    return x * 2;
}

int main() {
    std::cout << timesTwo(10) << std::endl;
    std::cout << timesTwo<int>(10) << std::endl;
    std::cout << timesTwo(10.5) << std::endl;
    std::cout << timesTwo<double>(10) << std::endl;
    return 0;
}
```

Declare specification explicitly

times_two.hpp (compiles to times_two.o)

```
template <typename T>
T timesTwo(T x) {
    return x * 2;
}

// commenting out these lines will result in error
template int timesTwo<int>(int);
template double timesTwo<double>(double);
```

main.cpp (links compiled file times_two.o)

```
template <typename T>
T timesTwo(T x);

int main() {
    std::cout << timesTwo(10) << std::endl;
    std::cout << timesTwo(10.5) << std::endl;
    return 0;
}
```

Implement in two functions

```
template <typename T>
void print(T x);

template<>
void print<int>(int x) {
    std::cout << "int: " << x << std::endl;
}

template<>
void print<double>(double x) {
    std::cout << "double: " << x << std::endl;
}
```

Optimize conditions

If condition at run time

```
void timesTwoAndOrPrint(bool timesTwo, int x) {
    if (timesTwo) {
        std::cout << x * 2 << std::endl;
    } else {
        std::cout << x << std::endl;
    }
}
```

Optimize conditions

If condition at compile time

```
template <bool timesTwo>
void timesTwoAndOrPrint(int x) {
    if constexpr (timesTwo) { // requires C++ 17
        std::cout << x * 2 << std::endl;
    } else {
        std::cout << x << std::endl;
    }
}
```

Optimize conditions

If condition at compile time

```
template <bool timesTwo>
void timesTwoAndOrPrint(int x) {
    if constexpr (timesTwo) { // requires C++ 17
        std::cout << x * 2 << std::endl;
    } else {
        std::cout << x << std::endl;
    }
}
```

Evaluates to

```
template <>
void timesTwoAndOrPrint<true>(int x) {
    std::cout << x * 2 << std::endl;
}

template <>
void timesTwoAndOrPrint<false>(int x) {
    std::cout << x << std::endl;
}
```

Specification with conditions

```cpp
template <bool timesTwo, int x, std::enable_if_t<timesTwo, int> = 0>
void timesTwoAndOrPrint() {
    std::cout << x * 2 << std::endl;
}
```

```cpp
template <bool timesTwo, int x, std::enable_if_t<!timesTwo, int> = 0>
void timesTwoAndOrPrint() {
    std::cout << x << std::endl;
}
```

Class templates

```
template <bool timesTwo, int x, typename T>
class TimesTwoAndOrPrint {
    T value;
public:
    TimesTwoAndOrPrint(T value) : value(value) {}
    void print() {
        if constexpr (timesTwo) { // requires C++ 17
            std::cout << value * 2 << std::endl;
        } else {
            std::cout << value << std::endl;
        }
    }

    static void printx() {
        if constexpr (timesTwo) { // requires C++ 17
            std::cout << x * 2 << std::endl;
        } else {
            std::cout << x << std::endl;
        }
    }
};
```

Partial specification (class only)

```
template <bool timesTwo, int x, typename T>
class TimesTwoAndOrPrint;

template <int x, typename T>
class TimesTwoAndOrPrint<true, x, T> {
    T value;
public:
    TimesTwoAndOrPrint(T value) : value(value) {}
    void print() {
        std::cout << value * 2 << std::endl;
    }

    static void printx() {
        std::cout << x * 2 << std::endl;
    }
};
```

Key Points

When to usage template

- When multiple functions shared similar implementation
- Make program more modular (replace if-else clause with separate functions)
- Reduce the use runtime condition (when the possible function parameter is a finite set)

Key Points

Syntax

- Function and class template
- Template parameters (typename, int, bool, etc.)
- Implicit / explicit template call
- Implicit / explicit template instantiation
- Meta programming with template (constexpr, enable_if_t)
- Partial specification for class template

Key Points

Don’t over-use it

- Reduced readability
- Longer compilation time
- Complex error message