TEMPLATES

Generic Programming

Florian Warg, Max Staff

June 29, 2017

MOTIVATION

- often you discover that your algorithm is generic
- you want to apply it to any type that conforms to a concept

```
int min(int x, int y)
{
    return (x < y) ? x : y;
}
```

APPROACH 1: FUNCTION OVERLOADING

```
int min(int x, int y) { return (x < y) ? x : y; }
char min(char x, char y) { return (x < y) ? x : y; }
```

- these are all the same
- there must be an overload for every type

APPROACH 2: TEMPLATES

- function should be defined for every comparable type
- the powerful template engine takes care of that

```
template<class T>
T min(T x, T y) { return (x < y) ? x : y; }
/* ... */
auto i = min(42, 1);
auto k = min(34.0, 23.1);
auto l = min(make_unique<int>(2), nullptr);
```

TEMPLATE SYNTAX

- template parameters can be values, but also types
- templates can be applied to types and functions

```
template<class T>
T min(T x, T y);

template<class T, std::size_t N>
struct array { T data[N]; };
```

TYPE DEDUCTION

- template parameters can / must be explicitly stated
- in C++14 function template parameters can also be deduced

```
auto i = min<int>(41, 1);
auto k = min<double>(34.0, 23.1);
auto l = min<double>(10, 0.0); // parameter needed!

auto x = min(23.0f, 23.01f);
array<int, 8> a; // parameters needed!
```

OVERLOAD RESOLUTION

- the compiler prefers concrete functions to generic ones
- template functions can be explicitly called

```
void fn(int x) { cout << "concrete\n"; }

template<class T>
void fn(T x) { cout << "templated\n"; }

fn(42); // "concrete"
fn(10.0); // "templated"
fn<int>(42); // "templated"
```

TEMPLATE SPECIALIZATION

- templates can be specialized
- you provide a special implementation for specific parameters

```
template<class T>
bool isInt(T t) { return false; }

template<>
bool isInt<int>(int t) { return true; }
```

TEMPLATE INSTANTIATION

- templates can only be instantiated when their expressions are well formed
- if a template cannot be instantiated, alternative templates will be considered
- this is called **Substitution Failure Is Not An Error** (SFINAE)
- if no template can be instantiated, a compiler error occurs

```
template<class T>
void call(T t) { t(); }

void f() { cout << "f called!"; }

call(f);
call([](){ cout << "lambda called!"; });
call(42); // 42() is ill formed! (Substitution failure)
```