[CS 225] Advanced C/C++

Lecture 8: Review of templates

Sławomir "Swavek" Włodkowski
(Summer 2020)

Agenda

- Function templates
- Type alias templates
- Namespace variable templates

- Class templates will be covered in the next lecture.

Function templates

- Generic programming: “cookie-cutters” for functions.
- Support specialization
  - Full (explicit) specialization – customizing a base template for a full set of template arguments.
  - No partial specialization.

Function templates

- Support function template argument deduction.
  - Non-template perfect match functions are considered first.
  - Then compiler considers base templates.
  - Then compiler considers specializations of a selected base.

Function templates

- Support overloading like other functions.
  - Problem: conflicts of overloaded specializations (declarations’ order matters).
  - Solution: overload function and delegate their calls to class template specialization’s member functions.

Type alias templates

```
#include <iostream>

template <typename T>
struct remove_reference
{ using type = T; };

template <typename T>
struct remove_reference<T&>
{ using type = T; };

template <typename T>
using remove_reference_t =
    typename remove_reference<T>::type;

int main() {
   using MyType = int&;
   remove_reference_t<MyType> x{};
   std::cout << x << std::endl;
}
```

Keyword: `using`

Used for abstracting away information where does a type come from.

Type aliases defined with `typedef` cannot be templated!

Namespace variable templates

```
#include <iostream>

template <typename T, int TMul = 1>
const T pi = static_cast<T>(
    3.14159265358979323851 * TMul
);

int main()
{
    std::cout
        << pi<float> << "\n"
        << pi<int, 100> << std::endl;
}
```

Used for abstracting away information where does a value come from and for casting to a desired type.