C++ Templates: Session 3

Anupama Chandrasekhar

Jayashree Venkatesh

SFINAE

- Recall the process of overload resolution:
  - When the set of candidates for a call includes function templates, the compiler has to determine what template arguments should be used for that candidate.
  - Then substitute those arguments in the function parameter list and its return type.
  - This, however, could lead to substitutions that lead to errors. The language, though, has decided that candidates with these substitution problems are simply ignore => **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror (*sfee-nay*)

SFINAE: Example

```
// number of elements in a raw array
template<typename T, unsigned N>
std::size_t len(T(&)[N])
{
    return N;
}

//number of elements for a type having size_type
template <typename T>
typename T::size_type len(T const& t)
{
    return t.size();
}

int main()
{
    int a[10];
    cout << len(a) << '\n'; // len() for array matches
    cout << len("tmp") << '\n'; // len() for array matches
    cout << len(vector<int>{1,2,4}); // len for classes with size_type
}
```

SFINAE: Example

- In the previous example, when passing a raw pointer neither of the templates match without a failure, as a result the compiler will complain that no matching `len()` function is found.

```
int* p;
cout << len(p) << '\n';
```

- However, if passing an object which has `size_type` but not `size()`, we get a different error

```
allocator<int> x;
cout << len(x);
```

How to use SFINAE in your code?

- Let’s say we want to make sure that we catch the error in the ”`allocator`” example at the substitution phase, i.e `len()` should choose only those class types that have `size()` and `size_type`. There is a common pattern or idiom that is used to deal with this:
  - Specify return type with trailing return type syntax ->
  - Define a return type using `decltype` and comma operator
  - Formulate all expressions that must be valid at the building of a comma operator converted to `void` incase the comma operator is overloaded for the specific type.
  - Define an object of the real return type at the end of the comma operator.

`std::enable_if<>`

- Since C++ 11, STL provides a helper template `std::enable_if<>` to ignore templates under some compile-time conditions. E.g.

```
#include <type_traits>
template <typename T>
typename std::enable_if<(sizeof(T) > 4)>::typename
foo()
{

}
```

- std::enable_if is a type trait that evaluates a given compile-time expression :
  - If `true`, its member `type` yields a type
  - If `false`, the member `type` is not defined

std::enable_if_t<>

- Since C++14, all type traits yielding a type have a corresponding alias template that allows you to skip some boilerplate “typename” and “type” in this case.

```
#include <type_traits>
template <typename T>
std::enable_if_t<(sizeof(T) > 4)>
foo()
{

}
```

CRTP: Curiously Recurring Template Pattern

- Refers to a class of techniques that consists of passing a derived class as a template argument to one of the base classes.

```
template <typename T>
class BaseClass
{

};

class Derived : public Base<Derived>
{

};
```

CRTP: Why is this useful?

- Static polymorphism of a different kind. By passing the derived class as a template parameter of the base class, the base class can customize it's own behavior without using virtual functions. **Catch:** objects have to be resolvable at compile time.
- The idea is to "inject" the real type of the derived class into the base.
- In real world this techniques is seen in:
  - Clang Front End (`RecursiveASTVisitor` class)
  - LLVM (`HeuristicBase` class)
  - Boost (ofcourse)
  - Microsoft Active Template Library

References for further reading

- https://eli.thegreenplace.net/2011/05/17/the-curiously-recurring-template-pattern-in-c
- https://www.fluentcpp.com/2017/05/12/curiously-recurring-template-pattern/
- https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

Type Deduction Basics

- How does a template obtain the template argument “T”:
  - It may be supplied directly

```
g<float>(12);
g<const float>(12);
```

  - Infer it from the function argument, if possible **(TYPE INFERENCE)**

```
template <typename T>
void g2(size_t k = sizeof(T));
```

```
template <typename T>
void g3(remove_reference_t<T> x); //Is T &?
```

Type Deduction Basics

- Infer it from the function argument, if possible **(TYPE INFERENCE)**

```
template <typename T>
void g4(T x, T y);

g4(1, 0.1);
```

- However if a default type argument is provided, that will be inferred.

```
template <typename T = double>
void g5(T x, T y);

g5(1, 0.1);
```

Type Inference

- Type inference depends on the following:
  - Type of the function parameter, contrast it with type of the function argument.
  - Conversions are not a part of type inference
- Problem Statement: Given the function below, use the function argument and relate it to the function parameter to determine T.

```
template <typename T>
void g6(<func_parameter> x);
```

- `func_parameter` may be:
  - Passed by value, Pointer or reference type, forwarding reference

Type Inference: Case 1 (Pass by Value)

```
template <typename T>
void g6(<func_parameter> x);
```

```
template <typename T>
void g6(T x);
```

```
int k = 12;
int& kl = k;
int const k = 12;
int const & kcl = 12;
```

g(x)	T
42	int
k	int
kl	int
kc	int
kcl	int

Rules for Type Inference
Rule 1: Ignore the reference part
Rule 2: Ignore cv qualifiers (like `const`)

Type Inference: Case 2 (Reference or Pointer)

```
template <typename T>
void g6(T& x);

template <typename T>
void g6(const T& x);
```

```
template <typename T>
void g6(<func_parameter> x);
```

```
int k = 12;
int& kl = k;
int const k = 12;
int const & kcl = 12;
```

g(x)	T &	T const &
42	~~int~~	int
k	int	int
kl	int	int
kc	int const	int
kcl	int const	int

Rules for Type Inference
Rule 1: Ignore the reference part
Rule 2: Pattern match function argument with function parameter to find T

Introduction to Rvalues and Lvalues

- One of the most prominent (and confusing) features of C++11 is *move semantics.* You can use it to optimize copying and assignment by moving ("Stealing") internal resources from a source object to destination object instead of copying. To understand these we need to distinguish between expressions that are lvalues and rvalues, because rvalues indicate objects that are eligible for move operation.
- **Lvalues** expression : yes you can take its address. **`t&`**
- **Rvalues** are typically temporaries and you can't take their values. **`t&&`**

Type Inference: Case 3 (Forwarding Reference)

```
template <typename T>
void g6(T&& x)
```

g(x)	T &&
42	int
k	int &
kl	int &
kc	int const &
kcl	int const &

```
template <typename T>
void g6(<func_parameter> x);
```

```
int k = 12;
int& kl = k;
int const k = 12;
int const & kcl = 12;
```

- **Rule 1:** If the expr is an lvalue, both T and func_parameter are deduced to be lvalue references. This is the only case where T is deduced to be a reference.
- **Rule 2:** If expression is an rvalue, Case 2 rules apply.

Forwarding References

- Suppose you want to write generic code that forwards the basic property of passed arguments:
  - Modifiable objects should be forwarded so they can still be modified.
  - Constant objects should be forwarded as read-only.
  - Movable objects should be forwarded as movable.

Perfect Forwarding

- We want to write a wrapper function that passes the arguments correctly:

```
void g (X&)
{
    cout << "g() for a variable\n";
}

void g(X const&)
{
    cout << "g() for a constant\n";
}

void g(X&&)
{
    cout << "g() for a movable object\n";
}
```

```
template <typename T>
void f(T t)
{
    g(t);
}
```

- `g` accepts arguments by reference, the `f` accepts arguments by value, so the changes made by `g` will not be seen by the caller.

Perfect Forwarding

- If `f` accepts arguments by reference

```
template <typename T>
void f(T& t)
{
    g(t);
}
```

- Different Problem: If `f` accepts parameters by reference, r-values cannot be bound

```
f(20);
```

- And making the parameters `const` wont help, because we might want to pass non-`const` parameters.

Brute Force Solution

```
// f() forwards argument to g()
void f(X& x)
{
    g(x);
}

void f(X const& cx)
{
    g(cx);
}

void f(X&& val)
{
    g(std::move(val));
}
```

Better Solution: Forwarding References

- Use forwarding reference

```
// f() forwards argument to g()
template<typename T>
void f(T&& t)
{
    g(forward<T>(t));
}
```

Template Debugging

- How do you know if the compiler is deducing the type you want to it to? Use a template that is declared, not defined and force a compiler error.

```
template <typename T>
class TD;

template <typename T>
void f(T& param)
{

    TD<T> ttype;
    TD<decltype(param)> paramType;
}

int main()
{
    int x = 22;
    const int& rx = x;
    f(rx);
}
```

Summary Slide

Strategy Name	Description
Tag Dispatch	Use type tags to dispatch calls from to different overloads
SFINAE	Choose the right overload using the SFINAE property
CRTP	Static polymorphism by passing the derived class as an argument to the base class
Debugging Using dummy template	Use a dummy template declaration to make the compiler throw an error that contains the deduced type.