DigiPen
INSTITUTE OF TECHNOLOGY
SINGAPORE

[CS 225] Advanced C/C++

Lecture 10: Perfect forwarding

Sławomir "Swavek" Włodkowski
(Summer 2020)

Agenda

- Understanding universal references
- Template argument deduction
  - Reference collapsing
  - Reversing collapsed references
- Perfect forwarding
- STL
  - `std::swap`
  - `std::move`
  - `std::remove_reference`
  - `std::forward`

std::swap

```
template <typename T>
void swap(T& t1, T& t2)
{
    T temp{std::move(t1)};
    t1 = std::move(t2);
    t2 = std::move(temp);
}
```

std::move

```
int&& move(int& t)
{
    return static_cast<int&&>(t);
}


int&& move(int&& t)
{
    return static_cast<int&&>(t);
}
```

`std::move`

Since the implementation of `std::move()` looks the same for `int&` and `int&&`, we should replace it with a single function template.

The problem is should the argument be `T&` or `T&&`?

Universal references

Forwarding references, also known as *universal references*, are l-value reference or r-value reference types before applying parameter type deduction rules to **collapse** to either of them.

Universal references

The syntax is very similar to r-value references, but:

- The type must be deduced.
- There must be no qualifiers or “decorations”, just `T&&`.
- There must be no deduction from an initializer list (`auto`).

```
template <typename T>
T&& f(T&&)
{
    // Do something.
}
```

```
auto&& x = f();
```

***Consider**: `T&&` (`T` is deduced) is a forwarding reference. A type `const T&&` is an r-value reference, regardless if `T` is deduced or not.*

Reference collapsing rules

References are not objects!

You cannot create:

- an array of references,
- a pointer to a reference,
- a reference to a reference.

Then, given `T` set to `int&&`, what does `T&` mean?

Reference collapsing rules

<T>	Ref		T&&
A&	&	→	A&
A&	&&	→	A&
A&&	&	→	A&
A&&	&&	→	A&&
A	&	→	A&
A	&&	→	A&&

```
template <typename T>
void f(T&&)
{
    // T = int&&
    // T&& = int&& && = int&&
}



f<int&&>(123 /* int&& */);
```

Reversing collapsed references

T&&		T
A&	→	A&
A&&	→	A

```
template <typename T>
void f(T&&)
{
    // T&& = int&
    // T = int&
}

int a = 1;
f(a /* int& */);
```

Perfect forwarding

All references given a name become l-values, such a function's formal parameters.

Challenge:

How to create a function `f1` that calls another function `f2` without loosing r-value nature of r-value parameters?

```
void f1(? param)
{
    f2(?(param));
}
```

Perfect forwarding

Perfect forwarding is a technique of passing:

- l-value parameters as l-values,
- r-value parameters as r-values,

perfectly (without loss of information) to an inner call.

Perfect forwarding

Examples

- A naïve implementation of a vector container with a factory-like function that uses perfect forwarding.
- A wrapper class (i.e. a unique pointer) that encapsulates a target object so well that it internally calls its constructor with parameter values passed from the outside of the wrapper class.

std::forward

```
template <typename T>
T&& forward(remove_reference_t<T>& t)
{
  return static_cast<T&&>(t);
}
```

***Consider**: `remove_reference_t` makes parameter deduction impossible, forcing a programmer to specify a template parameter explicitly (which is required to tell the function if the value has been lvalue or rvalue.*

std::remove_reference

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

std::move

```
template <typename T>
remove_reference_t<T>&& move(T&& t)
{
  return static_cast<
    remove_reference_t<T>&&
  >(t);
}
```

***Consider**: `remove_reference_t` lets the function result type avoid reference collapsing, thus always resulting in rvalue (xvalue) result.*