Perfect forwarding and universal references i C++

November 03, 2014 at 05:52 Tags C & C++ , Articles

One of the new features in C++11 aimed at increased code efficiency is the `emplace` family of methoc containers. `std::vector`, for example, has an `emplace_back` method to parallel `push_back`, and `emplace` to parallel `insert`.

Here's a short demonstration of the benefits these new methods bring:

```
class MyKlass {
public:
  MyKlass(int ii_, float ff_) {...}

private:
  {...}
};

some function {
  std::vector<MyKlass> v;

  v.push_back(MyKlass(2, 3.14f));
  v.emplace_back(2, 3.14f);
}
```

If you trace the execution of the constructors and destructor of `MyKlass`, you'll see something like the following for the `push_back` call:

- Constructor for a temporary `MyKlass` object
- Move constructor (if one was defined for `MyKlass`, otherwise a copy constructor) for the object actually allocated inside the vector
- Destructor for the temporary

This is quite a lot of work. Much of it isn't required though, since the object passed to `push_back` is obviously an rvalue that ceases to exist after the statement is completed; there's no reason to create destroy a temporary - why not just construct the object inside the vector directly?

This is exactly what `emplace_back` does. For the `v.emplace_back(2, 3.14f)` call above, all you see i single constructor invocation. This is the object constructed *inside* the vector. No temporaries are needed.

`emplace_back` accomplishes this by invoking the constructor of `MyKlass` on its own and forwarding its arguments to the constructor. This feat is made possible by two new features in C++11: variadic templates and perfect forwarding. In this article I want to explain how perfect forwarding works and h to use it.

The perfect forwarding problem

Let `func(E1, E2, ..., En)` be an arbitrary function call with generic parameters `E1, E2, ..., En`. \
like to write a function `wrapper` such that `wrapper(E1, E2, ..., En)` is equivalent to `func(E1, E2, ..., En)`. In other words, we'd like to define a function with generic parameters tha
forwards its parameters *perfectly* to some other function.

To have something concrete to relate this definition to, think of the `emplace_back` method discussed above. `vector<T>::emplace_back` forwards its parameters to a constructor of `T`, without actually knov
how `T` looks like.

Next, I'm going to show a few examples of how we might approach this in pre-11 C++. For simplicity'
sake, I'll put variadic templates aside; let's assume all we need to forward is two arguments.

The first approach that comes to mind is:

```
template <typename T1, typename T2>
void wrapper(T1 e1, T2 e2) {
    func(e1, e2);
}
```

This will obviously not work if `func` accepts its parameters by reference, since `wrapper` introduces a value passing step. If `func` modifies its by-reference parameter, it won't be visible in the caller of `wrapper` (only the copy created by `wrapper` itself will be affected).

OK, then, we can make `wrapper` accept its parameters by reference. This should not interfere with `func`'s taking parameters by value, because the call to `func` within `wrapper` will create the requi
copy.

```
template <typename T1, typename T2>
void wrapper(T1& e1, T2& e2) {
    func(e1, e2);
}
```

This has another problem, though. Rvalues cannot be bound to function parameters that are referen
so the following completely reasonable calls will now fail:

```
wrapper(42, 3.14f);                  // error: invalid initialization of
                                     //        non-const reference from
                                     //        an rvalue

wrapper(i, foo_returning_float());   // same error
```

And no, making those reference parameters `const` won't cut it either, because `func` may legitimately
want to accept non-`const` reference parameters.

What remains is the brute-force approach taken by some libraries: define overloads for both `const` a
non-`const` references:

```
template <typename T1, typename T2>
void wrapper(T1& e1, T2& e2)                { func(e1, e2); }

template <typename T1, typename T2>
```

```
void wrapper(const T1& e1, T2& e2)          { func(e1, e2); }

template <typename T1, typename T2>
void wrapper(T1& e1, const T2& e2)          { func(e1, e2); }

template <typename T1, typename T2>
void wrapper(const T1& e1, const T2& e2)    { func(e1, e2); }
```

Exponential explosion. You can imagine how much fun this becomes when we want to cover some reasonable amount of function parameters. To make things worse, C++11 adds rvalue references to mix (which we'd also want to forward correctly), and this clearly isn't a scalable solution.

Reference collapsing and special type deduction for rvalues

To explain how C++11 solves the perfect forwarding problem, we have to first understand two new ru that were added to the language.

Reference collapsing is the easier one to explain, so let's start with it. Taking a reference to a referen is illegal in C++. However, it can sometimes arise in the context of templates and type deduction:

```
template <typename T>
void baz(T t) {
  T& k = t;
}
```

What happens if we call this function as follows:

```
int ii = 4;
baz<int&>(ii);
```

In the template instantiation, `T` is explicitly set to `int&`. So what is the type of `k` inside? What the compiler "sees" is `int& &` - while this isn't something the user is allowed to write in code, the compile simply infers a single reference from this. In fact, prior to C++11 this wasn't standardized, but many compilers accepted such code anyway because these cases occasionally arise in template metaprogramming. With the addition of rvalue references in C++11, it became important to define wh happens when various reference types augment (e.g. what does `int&& &` mean?).

The result is the *reference collapsing* rule. The rule is very simple. `&`always wins. So `& &` is `&`, and s are `&& &` and `& &&`. The only case where `&&` emerges from collapsing is `&& &&`. You can think of it as logical-OR, with `&` being 1 and `&&` being 0.

The other addition to C++11 relevant to this article is special type deduction rules for rvalue reference some cases [1]. Given a function template like:

```
template <class T>
void func(T&& t) {
}
```

Don't let `T&&` fool you here - `t` is not an rvalue reference [2]. When it appears in a type-deducing context, `T&&` acquires a special meaning. When `func` is instantiated, `T` depends on whether the

argument passed to `func` is an lvalue or an rvalue. If it's an lvalue of type `U`, `T` is deduced to `U&`. If it'
rvalue, `T` is deduced to `U`:

```
func(4);            // 4 is an rvalue: T deduced to int

double d = 3.14;
func(d);            // d is an lvalue; T deduced to double&

float f() {...}
func(f());          // f() is an rvalue; T deduced to float

int bar(int i) {
  func(i);          // i is an lvalue; T deduced to int&
}
```

This rule may seem unusual and strange. That's because it is. However, it starts making sense wher
realize it was designed to solve the perfect forwarding problem.

Solving perfect forwarding with std::forward

Let's get back to our original `wrapper` template. Here's how it should be written in C++11:

```
template <typename T1, typename T2>
void wrapper(T1&& e1, T2&& e2) {
    func(forward<T1>(e1), forward<T2>(e2));
}
```

And this is `forward` [3]:

```
template<class T>
T&& forward(typename std::remove_reference<T>::type& t) noexcept {
  return static_cast<T&&>(t);
}
```

Let's say we call:

```
int ii ...;
float ff ...;
wrapper(ii, ff);
```

Examining the first argument (since the second is handled similarly): `ii` is an lvalue, so `T1` is deduce
to `int&` following the special deduction rules. We get the call `func(forward<int&>(e1), ...)`.
Therefore, `forward` is instantiated with `int&` and we get this version of it:

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

Now it's time to apply the reference collapsing rule:

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

In other words, the argument is passed on by reference to `func`, as needed for lvalues.

The other case to handle is:

```
wrapper(42, 3.14f);
```

Here the arguments are rvalues, so `T1` is deduced to `int`. We get the call `func(forward<int>(e1), ...)`. Therefore, `forward` is instantiated with `int` and we get this versi of it:

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

The by-reference argument is casted to an rvalue reference, which is what we wanted from `forward`.

One can see `forward` as a pretty wrapper around `static_cast<T&&>(t)` when `T` can be deduced to either `U&` or `U&&`, depending on the kind of argument to the wrapper (lvalue or rvalue). Now we get `wrapper` as a single template that handles all kinds of forwarding cleanly.

The `forward` template exists in C++11, in the `<utility>` header, as `std::forward`.

Another thing I want to mention is the use of `std::remove_reference<T>`. In fact, it you think about it, `forward` could do without it. Reference collapsing does the job already, so `std::remove_reference<T>` is superfluous. It's there to turn the `T& t` into a non-deducing context (according to the C++ standard, section 14.8.2.5), thus forcing us to explicitly specify the template parameter when calling `std::forward`.

Universal references

In his talks, blog posts and book, Scott Myers gave the name "universal references" to rvalues that appear in a type-deducing context. Whether this is a useful mnemonic or not depends on the behold personally, when I first read the relevant chapters of the new "Effective C++", I found myself seriousl confused on this topic. Only later, when I understood the underlying mechanisms (reference collapsi and special deduction rules), the topic became somewhat clearer.

The trap is that saying "universal references" [4] is surely more succinct and nice than "rvalue referei in a type deducing context", but once you want to really understand some piece of code (rather than cargo-culting boilerplate), you'll find that avoiding the full definition is impossible.

Examples of using perfect forwarding

Perfect forwarding is extremely useful, because it enables a kind of higher order programming. *Highe order functions* are functions that may take other functions as arguments or return them. Without per forwarding, higher order functions are cumbersome because there is no convenient way to forward arguments to wrapped functions. And by "functions" here I mean classes as well, whose constructors are still functions.

In the beginning of the article I mentioned the `emplace_back` method of containers. Another good examples is `make_unique`, which I described in the previous article:

```
template<typename T, typename... Args>
unique_ptr<T> make_unique(Args&&... args)
{
    return unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

There, I pleaded to ignore the strange `&&` syntax and focus on the variadic template packs, but now there's no trouble fully understanding the code. It goes without saying that perfect forwarding and variadic templates very often go hand in hand, because we generally don't know how many argumer the functions or constructors we pass around accept.

For a significantly more complex use of perfect forwarding, you may also want to take a look at `std::bind`.

Links to resources

Here are some of the resources I found helpful while preparing this article:

1. The 4th edition of "The C++ Programming Language" by Bjarne Stroustrup
2. The new "Effective Modern C++" by Scott Myers has an extensive discussion of "universal references". In fact, more than 1/5th of the book is dedicated to the topic (gasp!).
3. Technical paper n1385: "The forwarding problem: Arguments".
4. Thomas Becker's C++ Rvalue references explained is extremely well-written and useful.

---

[1] There rules apply in other situations, like `auto` and `decltype`. Here I'm only presenting the templa case.

[2] I think it's unfortunate that the C++ commitee didn't pick a different syntax for this case and overloaded the meaning of `&&` instead. I realize it seems like a relatively uncommon use, for whic it'd be a shame to change the language syntax (a thing the commitee tries to avoid as much as possible), but IMHO the situation is too confusing now. Even Scott Myers admitted in a talk and s comments on his blog that after 3 years this material is still "sinking in". And Bjarne Stroustrup ha mistake in the 4th edition of "The C++ Programming Language" when describing `std::forward` - forgetting to explicitly provide a template argument when calling it. This stuff is complex!

[3] This is a simplified version of `std::forward` from the C++11 standard library. That one has an additional overload explicitly for rvalues, the goal of which I'm still trying to decipher. Let me know you have an idea.

[4] "Forwarding references" is another name I've heard used elsewhere.