Non-capturing C++ lambdas can be converted to a pointer to function, but what about the calling convention?

devblogs.microsoft.com/oldnewthing/20150220-00

February 20, 2015

Raymond Chen

First, let’s look at how lambdas are implemented in C++.

It is similar in flavor to the way lambdas are implemented in C#, but the details are all different.

When the C++ compiler encounters a lambda expression, it generates a new anonymous class. Each captured variable becomes a member of that anonymous class, and the member is initialized from the variable in the outer scope. Finally, the anonymous class is given an `operator()` implementation whose parameter list is the parameter list of the lambda, whose body is the lambda body, and whose return value is the lambda return value.

I am simplifying here. You can read the C++ language specification for gory details. The purpose of this discussion is just to give a conceptual model for how lambdas work so we can get to answering the question. The language also provides for syntactic sugar to infer the lambda return type and capture variables implicitly. Let’s assume all the sugar has been applied so that everything is explicit.

Here’s a basic example:

```
void ContainingClass::SomeMethod()
{
 int i = 0, j = 1;
 auto f = [this, i, &j](int k) -> int
    { return this->calc(i + j + k); };
 ...
}
```

The compiler internally converts this to something like this:

```
void ContainingClass::SomeMethod()
{
 int i = 0, j = 1;

 // Autogenerated by the compiler
 class AnonymousClass$0
 {
 public:
  AnonymousClass$0(ContainingClass* this$, int i$, int& j$) :
   this$0(this$), i$0(i$), j$0(j$) { }
  int operator(int k) const
     { return this$0->calc(i$0 + j$0 + k); }
 private:
  ContainingClass* this$0; // this captured by value
  int i$0;                 // i captured by value
  int& j$0;                // j captured by reference
 };

 auto f = AnonymousClass$0(this, i, j);
 ...
}
```

We are closer to answering the original question. but we're not there yet.

As a special bonus: If there are no captured variables, then there is an additional conversion operator that can convert the lambda to a pointer to a nonmember function. This is possible only in the case of no captured variables because captured variables would require an `AnonymousClass$0` instance parameter, but there is nowhere to pass it.

Here's a lambda with no captured variables.

```
void ContainingClass::SomeMethod()
{
 auto f = [](int k) -> int { return calc(k + 42); };
 ...
}
```

The above code gets transformed to

```
void ContainingClass::SomeMethod()
{
 class AnonymousClass$0
 {
 public:
  AnonymousClass$0()  { }
  operator int (*)(int k) { return static_function; }
  int operator(int k) const { return calc(k + 42); }
 private:
  static int static_function(int k) { return calc(k + 42); }
 };

 auto f = AnonymousClass$0();
 ...
}
```

Okay, now we can get to the actual question: How can I specify the calling convention for this implicit conversion to a pointer to nonmember function?

(Note that calling conventions are not part of the C++ standard, so this question is necessarily a platform-specific question.)

The Visual C++ compiler automatically provides conversions for every calling convention. So with Visual C++, the transformed code actually looks like this:

```
void ContainingClass::SomeMethod()
{
 class AnonymousClass$0
 {
 public:
  AnonymousClass$0()  { }
  operator int (__cdecl *)(int k) { return cdecl_static_function; }
  operator int (__stdcall *)(int k) { return stdcall_static_function; }
  operator int (__fastcall *)(int k) { return fastcall_static_function; }
  int operator(int k) { return cdecl_static_function(k); }
 private:
  static int __cdecl cdecl_static_function(int k) { return calc(k + 42); }
  static int __stdcall stdcall_static_function(int k) { return calc(k + 42); }
  static int __fastcall fastcall_static_function(int k) { return calc(k + 42); }
 };

 auto f = AnonymousClass$0();
 ...
}
```

In other words, the compiler creates all the conversions, just in case. (The versions you don’t use will be removed by the linker.)

But only for noncapturing lambdas.

Raymond Chen

Follow