



# C++ Templates and Parameterized Return Types

Version	1.1, 9 January 2006
Language	C++
Platform	Platform Independent
Tested On	Windows XP, Microsoft Visual C++ (.Net, not 2003)
Author	Craig Mason-Jones (craig@lateral.co.za)
Licence	<a href="http://creativecommons.org/licenses/by-nc-sa/2.0/za/">http://creativecommons.org/licenses/by-nc-sa/2.0/za/</a>

## Summary

One can use template specialisation and partial-specialisation to provide different functionality based on the types passed to a templated `struct`, but partial-specialisation is not provided for functions (or methods). What does one do, though, when one wants to differentiate a function's behaviour based on whether it returns a `void` or an actual type? This discussion particularly relates to my efforts to wrap an embedded language.

## Problem

I wanted to code a number of methods for a class that would have the following syntax:

```
1 callN < return_type, parameter1, parameter2, ..., parameterN> (p1, p2, ..., pN)
```

I coded the method (for a single parameter) like this:

```
2 template<typename R, typename P1>
3 R MyStack::call1 (P1 p1) {
4     // Do whatever
5     R r;
6     *this >> r;
7     return r;
8 };
```

Here `MyStack` represents some sort of stack, where the operation of the function `callN` leaves the result on the stack, which I extract with the extraction operator. I can use the `call1(p1)` method with any return type as long as I have an extraction operator defined for that type.

However, sometimes the result of my calling code might not return a value (the calling code is calling an embedded language). In this case, there is nothing for me to extract from the stack.

## Possible Solutions

The first, and easy solution, is simply to define parallel functions called, say, `voidcallN(...)`, which don't extract a result. But I wanted an easier syntax for my users, so I decided to try a little template meta-programming. Ideally I would like to be able to specialize my template by adding:

```
9 void MyStack::call2<void> {
10     ..
11 };
```

Unfortunately, this is not acceptable C++ syntax, since partial-specialisation is not supported for functions. Of course I can wrap the function in a class:

```
12 template <typename R, typename P1>
13 struct MyCall {
14     R call(P1 p) { .. return r; }
15 };
16 template <typename P1>
17 struct MyCall<void, P1> {
18     void call(P1 p) { .. return; }
19 };
```

This works fine for simple calls, but I want my `MyStack` to support the templated function. If I write:

```
20 template<typename R, typename P1>
21 R MyStack::Call2 (P1 p) {
```

```

22     MyCall2<R,P1> c;
23     return c.call(p);
24 };

```

It still doesn't work for a `void` return type, since I'm trying to return a `void` in a `void` function.

So I thought to try some meta-programming:

```

25 template<typename R>
26 struct IsVoid { static const bool result = false; };
27 template <>
28 struct IsVoid<void> { static const bool result = true; };
29
30 template<typename R, typename P1>
31 R MyStack::call1 (P1 p) {
32     // Do whatever
33     if (IsVoid<R>::result) {
34         return;
35     } else {
36         R r; *this >> r; return r;
37     }
38 }

```

But this doesn't work because, although the code in line 36 will never be run, it can't be compiled as soon as the user writes `call1<void, int>(..)`, because line 36 becomes: `void r; *this>>r; return r;` and of course the compiler doesn't like that '`void r;`'. (or the returning of a value from a `void` function.)

I found this solution:

```

39 struct VoidPlaceholder { };
40
41 MyStack& MyStack::operator >> (VoidPlaceholder& vp) { /* Don't do anything */ return *this; }
42
43 template <typename R>
44 struct VoidAvoid { const static bool isvoid = false; typedef R ReturnT; };
45 template <>
46 struct VoidAvoid<void> { const static bool isvoid = true; typedef VoidPlaceholder ReturnT; };
47
48 template<typename R, typename P1>
49 VoidAvoid<R>::ReturnT call1(P1 p1) {
50     // Do the call
51     VoidAvoid<R>::ReturnT r;
52     *this >> r; return r;
53 };

```

It is less elegant than I would have wished, but it does work correctly, even though it involves some unnecessary calls. The insertion operator, though coded, will never be called, but without it, the compiler dislikes line 36 again.

## Conclusion

I've written and tested this sample on Microsoft Visual C++ .Net, but not .Net 2003, which is apparently a different compiler. The code is, I think, rather confusing when what I want to do is really simple, and could be managed very easy with the template partial-specialization for functions, but that's not C++ (yet). At the same time, I can't help thinking that a clever meta-language with some pre-processing could achieve this result very easily. Something like:

```
54 <% meta %>
55 template <typename R, typename P1>
56 R call1 (P1 p1) {
57     // Do my call
58     <% if R=='void' then %>
59         return
60     <% else %>
61         R r; *this >> r; return r;
62     <% end %>
63 };
64 <% end-meta %>
65 ...
66 int main() {
67     ...
68     Stack-><% meta-call call1<void, int> %>(10);
69 }
```

Writing a preprocessor to achieve this doesn't seem so difficult, but one would need to consider nested templates, and differentiating methods and functions and classes, which makes it a little more difficult...

## Other Compilers

I've only tried this under Microsoft Visual C++, where it works. I would be interested to hear whether it works under any other compilers, or not, and whether there is a more elegant solution with another compiler, or simply a more elegant solution entirely.