Subject: Nested template question
Posted by koldo on Sun, 09 Jun 2019 14:16:28 GMT
View Forum Message <> Reply to Message

Hi all

I wanted to overload a function to get a zero (it is a sample):
```
void test() {
 double val      = GetAZero<double>();
 std::complex<float>  valc1 = GetAZero<std::complex<float>>();
 std::complex<double> valc1 = GetAZero<std::complex<double>>();
}
```
This code works, but it is not nice:
```
template <class T> T    GetAZero() {return 0;}

template <>     std::complex<float> GetAZero() {return std::complex<float> (0, 0);}
template <>     std::complex<double> GetAZero() {return std::complex<double>(0, 0);}
```

This code is more compact, but it does not work:
```
template <class T>    T    GetAZero() {return 0;}

template <class std::complex<T>> std::complex<T> GetAZero() {return std::complex<T> (0, 0);}
```
Is there an adequate way to do this?

Subject: Re: Nested template question
Posted by Novo on Mon, 10 Jun 2019 18:42:55 GMT
View Forum Message <> Reply to Message

```
struct AZero {
 template <typename T>
 operator T() const {return 0;}

 template <typename T>
 operator std::complex<T>() const {return std::complex<T>(0, 0);}
};

double val    = AZero();
std::complex<float>  valc1 = AZero();
std::complex<double>  valc2 = AZero();
```
Checked with Clang.

What you are trying to do is

```
template <typename T>  T GetAZero() {return 0;}
template <typename T> std::complex<T> GetAZero<class std::complex<T>>() {return
std::complex<T> (0, 0);}
```
It won't compile because this is partial function specialization, which is allowed only for classes.

The way I implemented this is also partial function specialization, but for some reason it compiles

Subject: Re: Nested template question
Posted by koldo on Mon, 10 Jun 2019 19:40:17 GMT
View Forum Message <> Reply to Message

Thank you Novo

Nice hack... the problem is that my code is already inside a class, so GetAZero() is really a class function
To solve this, the inner class should have to point the outer class, maybe passing a pointer in the inner class constructor, but then the hack is getting more complex.

Subject: Re: Nested template question
Posted by Novo on Mon, 10 Jun 2019 21:48:30 GMT
View Forum Message <> Reply to Message

I don't really get what you need to do.
If it is just setting any class to zero, as in your example, then you do not need method GetAZero(). Class AZero will work out of the box.
If you are trying to specialize your template method for all possible variants of std::complex<T>, then this is called partial template specialization, and it works only for classes. You need to create a dummy class and partially specialize it.

```
template <typename T>
struct dummy {
 T DoIt() const { return T(); }
};

template <typename T>
struct dummy<std::complex<T>> {
 using PT = std::complex<T>;
 PT DoIt() const { return PT(0, 0); }
};

struct Boo {
 Boo() {
  double val    = GetAZero<double>();
  std::complex<float>  valc1 = GetAZero<std::complex<float>>();
  std::complex<double>  valc2 = GetAZero<std::complex<double>>();
 }

 template <class T>
 T GetAZero() { return dummy<T>().DoIt(); }
```

};

Subject: Re: Nested template question
Posted by koldo on Tue, 11 Jun 2019 07:59:39 GMT
View Forum Message <> Reply to Message

Thank you Novo.

What I mean is that in my case dummy structs should have to access Boo struct. DoIt() function could have an argument pointing to Boo. However I see that in general the solution is more complex than the problem. Thank you!

Subject: Re: Nested template question
Posted by slashupp on Mon, 12 Aug 2019 07:56:18 GMT
View Forum Message <> Reply to Message

Don't know if I understand the problem correctly,
but I came up with the following:

```
#include <iostream>
#include <complex>

template<typename...P> void say(P...p){ (std::cout << ... << p); }

template<typename T> T mkZero(T &t) { t=T(0); return t; }
template<typename T, typename...V> T mkVal(T &t, V...v) { t=T(v...); return t; }

int main()
{
    int i;
    float f;
    double d;
    std::complex<int> ci;
    std::complex<float> cf;
    std::complex<double> cd;

    mkVal(i, 11);
    mkVal(f, 22.22);
    mkVal(d, 33);
    mkVal(ci, 44, 44);
    mkVal(cf, 5.5, 55.55);
    mkVal(cd, 66, 66.6);
```

```
    say("valued: i=", i, ", f=", f, ", d=", d, ", ci=", ci, ", cf=", cf, ", cd=", cd, "\n");

    mkZero(i);
    mkZero(f);
    mkZero(d);
    mkZero(ci);
    mkZero(cf);
    mkZero(cd);

    say("zeroed: i=", i, ", f=", f, ", d=", d, ", ci=", ci, ", cf=", cf, ", cd=", cd, "\n");

    return 0;
}
```

Output:

```
valued: i=11, f=22.22, d=33, ci=(44,44), cf=(5.5,55.55), cd=(66,66.6)
zeroed: i=0, f=0, d=0, ci=(0,0), cf=(0,0), cd=(0,0)
```