Exercise Session 11

January 25, 2013

1. Write a different program for each of the following point using the type traits, available in the header file type_traits .

   (a) Statically check if a template parameter of a class is **int** or **double**; rise an error otherwise.

   (b) Declare a function template that prints the variable passed as argument. Statically check if the latter is not a raw pointer.

   (c) Declare **unsigned int** i, then declare a **float** or an **int** if i is signed or unsigned, respectively, without knowing the type of i explicitly. Check the result.

2. Write a program that perform a dot product of two arrays using the template metaprogramming technique. Using the following hints:

   - Use the container std :: array to store the two vectors.
   - Declare a class templetized on the index, using a std :: size_t (you need to include <cstdef>) to declare the template parameter. size_t is the type used to address arrays, in this way you avoid errors.
   - In the class introduce a static method, called apply, which perform one operation of the dot product and calls again apply with the index decreased by one. Beware, this method should be **inline** .
   - Specialize the method apply for the case of zero index.
   - Overload a **operator** $*$ that implements the dot product using the class just defined. It is better if it returns a constexpr.

Solution

1. The programs which use the traits are reported below

   (a) An implementation is the following:

```
#include <type_traits>

template <typename T>
struct B
{
    B ()
    {
        static_assert( std::is_same<T, int >::value
                       ||
                       std::is_same<T, double >::value ,
                       "Bad value"  );
    }
```

```
};

int main()
{
    // Syntax correct
    B<double> b;

    // Syntax wrong
    B<char> a;

    return 0;
}
```

First of all we need to include the <type_traits> to have the access to the methods to handle the types. This functionality is introdued from the C++11. We can access to different traits and, in particular for our case, to the class std::is_same. The latter takes in input two template parameters needed for the comparison, we can access to its member value to determine if they are equal or not. This operation is done statically. After the evaluation of both the std::is_same we have two boolean linked with the || operator. Also in this case the operation is done statically, since the values are known. Finally we can use the static_assert to rise a compilation error if the condition, specified in the first argument, is **false**. The static_assert is checked only during the compilation and not at run time.
Compiling the program we obtain

```
main.cpp:7:7: error: static_assert failed "Bad value"
    { static_assert( std::is_same<T,int>::value ||
std::is_same<T,double>::value, "Bad value" ); }
      ^              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
main.cpp:15:13: note: in instantiation of member function 'B<char>::B' requested
here
    B<char> a;
            ^
1 error generated.
```

The template parameter **char** is not allowed by the class.

(b) An implementation is the following:

```
#include <type_traits>
#include <iostream>

template <typename T >
void printValue ( const T& t )
{
    static_assert ( !std::is_pointer<T>::value, "Bad value" );
    std::cout << t;
}

int main()
{
    int a = 9;
    printValue(a);

    int * b = new int(7);
    printValue(b);

    return 0;
}
```

Since the input parameter of printValue is a-priori unknown we have to check if it is a raw pointer or not. We use the class std::is_pointer to check if the type, specified as the first template parameter, is a pointer or not. Beware, this class does not work as you expect with std :: shared_ptr but it works only with raw pointers.

(c) An implementation is the following:

```
#include <type_traits>

int main()
{
    unsigned int i;

    const bool check = std::is_signed< decltype(i) >::value;

    typedef std::conditional< check, float, int >::type new_type;

    static_assert ( !std::is_same< new_type, float >::value, "Bad value" );

    return 0;
}
```

We need a method that can do a conditional chose depending on a boolean. In particular the return value have to be a type. The most easy way is to use std :: conditional , which takes three template paramters: the boolean as first then the two possible types. In our case if check is **true** then new_type is a **float** otherwise is a **int**. To deduce the type of the variable i we use the function decltype then, using the trait std :: is_signed , we can understand if i is a signed or unsigned type.

2. To construct the dot product using the template metaprogramming the key ingredients are templates and **static** and **inline** functions. In this way the compiler can perform a lot of optimization. First of all we need a class, templetized on the index of the array, that perform the operations. Its implementation is reported below.

```
#ifndef HH_METADOT_HH
#define HH_METADOT_HH
#include <array>
#include <cstddef>

/*! An example of template metaprogramming.

  It implements the dot product of two std::arrays.

  std::size_t returns the correct type used to dimension array. Since
  it can be implementation dependent (32 or 64 bits) it is better to
  use it instead of just unsigned int.

  The class can be further generalised.

*/
template<std::size_t M>
struct metaDot
{
  template<std::size_t N,typename T>
  static T apply(std::array<T,N>const & a,std::array<T,N> const & b)
  {
    return a[M-1]*b[M-1] + metaDot<M-1>::apply(a,b);
  }
};

//! Specialization for the first element.
template<>
struct metaDot<1>
{
  template<std::size_t N,typename T>
  static T apply(std::array<T,N>const & a,std::array<T,N> const & b)
  {
    return a[0]*b[0];
  }
};
```

```cpp
template<std::size_t N,typename T>
constexpr T operator * (std::array<T,N>const & a,std::array<T,N> const & b)
{
  return metaDot<N>::apply(a,b);
}

#endif
```

As the comment says the class is templetized on the `std::size_t`, since it returns the correct type used to dimension array. This type it can be implementation dependent (32 or 64 bits) it is better to use it instead of just unsigned int. The class `metaDot` contains the static method `apply` which is templetize on the same template variable of the `std::array`. Notice that we statically check that the two arrays have the same dimension. The method simply perform a single operation of the dot product and then call again `apply` with the index decreased by one. This strategy lead us to perform the dot product recursively. The method is static, so we do not need an object of type `metaDot` for each of the index. Moreover the implementation is done inside the class so the inlining is automatic. Finally we need a stopping criteria, so we implement a specialization of `apply` for the first index. The free function **operator** ∗ mask the call of the method `apply` by using the normal ∗ operator. The return value is a `constexpr T` which can be more usefull since all the variables are known at compile time.
The use of the dot product is reported below.

```cpp
#include "metadot.hpp"
#include <iostream>
int main(){
  using std::array;
  array<double,5> a={1.,1.,1.,1.,1.};
  array<double,5> b={1.,2.,3.,4.,5.};
  std::cout<<"a*b="<<metaDot<5>::apply(a,b)<<std::endl;
  std::cout<<"a*b="<<a*b<<std::endl;

}
```