Exercise 4 Type Traits

1. (Other Type Traits, Investigation)

This exercise entails determining which type traits functions to use for the following functionality. Having discovered the appropriate functions then create some code with two specific types to show how it works:

a) Is a type either a signed or an unsigned arithmetic type?
b) Make a given integral type signed/unsigned.
c) Obtain the number of dimensions of an array type.
d) Remove/ add a pointer from or to a given type.
e) Remove/add reference from or to a given type.

2. (Pointers and non-Pointers)

This exercise consists of calling some functions from the *Primary type category*.

Answer the following questions:

a) Write a function to determine if a type is a pointer, null pointer, *lvalue* reference or *rvalue* reference.
b) Determine if a type is a member function pointer or if it is a pointer to a non-static member object.
c) Is a shared pointer a pointer type? Is it a pointer type when converted to a raw pointer?

Typical code is:

```
template <typename T>
    void IsPointer(const T& t)
{ // First example of type_traits;check if t is a pointer

    // Return type is std::true_type or std::false_type
    if (std::is_pointer<T>::value)
    {
       std::cout << "This is a pointer type argument\n";
    }
    else
    {
       std::cout << "_not_ a pointer type argument\n";
    }
}
```

3. (Simple switchable *Bridge* Functionality)

We create a template function that supports both pointers and reference types. If it is a pointer it is dereferenced and then printed while if it is not a pointer type and if it is a scalar reference type then it is printed directly. Use the `is_pointer()` function in conjunction with `std::true_type` and `std::false_type` to determine which implementation to call.

4. Give the top two advantages of *type traits*:

a) Creating type-independent code.
b) "Compile-time" *reflection*.
c) It is a replacement for subtype polymorphism.
d) It is used to add properties to C++ types.