```
#include <cstring>      // std::memcpy
#include <type_traits>  // std::is_pod, std::is_trivially_copyable
#include <vector>       // std::vector

template <typename T>
std::vector<char> byteCopy1(const T& obj)  // object buffer
{
    static_assert(std::is_pod<T>::value, "T must be POD");  // Too restrictive!

    std::vector<char> result('\0', sizeof(obj));
    std::memcpy(result.data(), &obj, sizeof(obj));
    return result;  // newly allocated buffer containing object representation
}
```

In the example above, we have required that template type argument, `T`, be a POD type. The only constraint on `T` that would affect the validity of this function template, however, is that `T` must be trivially copyable. Hence, there is a substantial variety of types that one might expect to work with this `byteCopy1` function but which, for no legitimate reason, do not compile:

```
struct B { int x; };      // The base class is a POD.
struct D : B { int y; };  // derived class is trivial but not standard layout

void test1()
{
    B b = {};
    D d = {};

    std::vector<char> vb = byteCopy1(b);  // OK
    std::vector<char> vd = byteCopy1(d);  // Error, D is not a POD.
}
```

In the example above, even though `d` is not of POD type, there's nothing in the implementation of function template `byteCopy1` — other than the **`static_assert`** — that would prevent it from working on `d` too.

A more general solution (e.g., `byteCopy2`) would be for the contract to provide, again via a **`static_assert`**, just the minimal set of type constraints required, thereby enabling the function template to work with a much wider variety of argument types:

```
template <typename T>
std::vector<char> byteCopy2(const T& obj)
{
    static_assert(std::is_trivially_copyable<T>::value,  // appropriately
                  "T must be trivially copyable");       // restrictive

    // ...                    (same as in byteCopy1 above)
}
```