```
#include <cassert>   // standard C assert macro
#include <string>    // std::string
#include <my_any.h>  // MyAny
void f()
{
    MyAny obj = 10;               // can be initialized with values of any type
    assert(obj.as<int>() == 10);  // Inner data can be retrieved at run time.

    obj = std::string{"hello"};   // can be reassigned from a value of any type
    assert(obj.as<std::string>() == "hello");
}
```

A straightforward implementation of `MyAny` would be to allocate an appropriately sized block of dynamic memory each time a value of a new type is assigned. Such a naive implementation would force memory allocations even though the vast majority of values assigned in practice are small (e.g., **fundamental types**), most of which would fit within the space that would otherwise be occupied by just the pointer needed to refer to dynamic memory. As a practical optimization, we might instead consider reserving a small buffer within the **footprint** of the `MyAny` object to hold the value provided (1) it will fit and (2) the buffer is sufficiently aligned. The natural implementation of this type, typically having a **`union`** of a **`char`** array and a **`char`** pointer as a data member, will naturally result in the alignment requirement of at least that of the **`char*`**, e.g., 4 on a 32-bit platform and 8 on a 64-bit one:

```
// my_any.h:

class MyAny  // ~~nontemplate class~~
{
    union {
        char* d_buf_p;       // pointer to dynamic memory if needed
        char  d_buffer[39];  // small buffer

    };  // Size of union is 39; alignment of union is alignof(char*).

    char d_onHeapFlag;            // Boolean (discriminator) for union (above)

public:
    template <typename T>
    MyAny(const T& x);              // member template constructor

    template <typename T>
    MyAny& operator=(const T& rhs);  // member template assignment operator
```