```
    template <typename T>
    const T& as() const;             // member template accessor

    // ...

};  // Size of MyAny is 40; alignment of MyAny is alignof(char*), e.g., 8.
```

We could, in addition, use the **`alignas`** attribute to ensure that the minimal alignment of `d_buffer` is at least 8 (or even 16):

```
// ...
alignas(8) char d_buffer[39];  // small buffer aligned to, at least, 8
// ...
```

We chose the size of `d_buffer` in the example above to be 39 for two reasons. First, we decided that we want 32-byte types to fit into the buffer, meaning that the size of `d_buffer` should be at least 32. Combined with the use of **`char`** for the `d_onHeapFlag`, which is guaranteed to have the size of 1, we require that **`sizeof`**`(MyAny) >= 33`. Second, we want to ensure that no space is wasted on padding. On platforms where **`alignof`**`(MyAny)` is 8, which will be the case for many 64-bit platforms, **`sizeof`**`(MyAny)` would be 40, which we choose to achieve by increasing the useful capacity to 39 instead of having the compiler add unused padding.

The templated constructor of `MyAny` can then decide, potentially at compile time, whether to store the given object `x` in the internal small buffer storage or on the heap, depending on `x`’s size and alignment:

```
template <typename T>
MyAny::MyAny(const T& x)
{
    if (sizeof(x) <= ~~39~~ && alignof(T) <= alignof(char*))
    {
        // Store x in place in the small buffer.
        new(d_buffer) T(x);
        d_onHeapFlag = false;
    }
    else
    {
        // Store x on the heap and its address in the buffer.
        d_buf_p = reinterpret_cast<char*>(new T(x));
        d_onHeapFlag = true;
    }
}
```