83 Small String Optimization in C++

小字符串优化，简称SSO。
字符串往往会和性能差联想到一起，本节主要关注C++标准库如何优化小字符串。

我们都知道创建一个标准字符串会导致内存分配，所以很多人会尽量减少字符串的使用。但是标准库中，小字符串（也就是不是很长的字符串），它们不需要堆分配，我们可以只分配一小块基于栈的缓冲区而不是堆分配的。如果你有一个不超过15个字符的字符串（VS2019），它不会在堆上分配内存，换句话说对于很小的字符串你就不要使用`const char*`或视图微观管理、优化你的代码了。

```
basic_string(_In_reads_(_Count) const _Elem* const _Ptr, _CRT_GUARDOVERFL
    : _Mypair(_Zero_then_variadic_args_t()) {
    auto&& _Alproxy = _GET_PROXY_ALLOCATOR(_Alty, _Getal());
    _Container_proxy_ptr<_Alty> _Proxy(_Alproxy, _Mypair._Myval2);
    _Tidy_init();
    assign(_Ptr, _Count);
    _Proxy._Release();
}
```

> 重要的的主要函数是assign，它接受一个指针参数，也就是const char*，我们的字符串

```
_CONSTEXPR20 basic_string& assign(
    _In_reads_(_Count) const _Elem* const _Ptr, _CRT_GUARDOVERFLOW const
      size_type _Count) {
    // assign [_Ptr, _Ptr + _Count)
    if (_Count <= _Mypair._Myval2._Myres) {
        _ASAN_STRING_MODIFY(static_cast<difference_type>(_Count -
          _Mypair._Myval2._Mysize));
        _Elem* const _Old_ptr   = _Mypair._Myval2._Myptr();
        _Mypair._Myval2._Mysize = _Count;
        _Traits::move(_Old_ptr, _Ptr, _Count);
        _Traits::assign(_Old_ptr[_Count], _Elem());
        return *this;
    }

    return _Reallocate_for(
        _Count,
        [](_Elem* const _New_ptr, const size_type _Count, const _Elem*
          const _Ptr) {
            _Traits::copy(_New_ptr, _Ptr, _Count);
            _Traits::assign(_New_ptr[_Count], _Elem());
        },
        _Ptr);
}
```

> 这里有一个if语句，如果满足则简单地将我们的字符串移到这个内存缓冲区中，就没有分配了；如果没通过if，则会Reallocate，调用new操作符导致堆分配。

来查一下这个Myres是多少：

```
_Traits::copy(_My_data._Bx._Buf, _Unfan
  1);
_Al.deallocate(_Ptr, _My_data._Myres +
_My_data._Myres = _BUF_SIZE - 1;
_ASAN_STRING_CREATE(*this);
```

> 可以看到它被设置为缓冲区大小-1

```
private:
    static constexpr auto _BUF_SIZE   = _Scary_val::_BUF_SIZE;
    static constexpr auto _ALLOC_MASK = _Scary_val::_ALLOC_MASK;

    // When doing _String_val operations by memcpy, we are touching:
```

> 这个缓冲区大小_BUF_SIZE等于_Scary_val::_BUF_SIZE

```
// length of internal buffer, [1, 16]:
static constexpr size_type _BUF_SIZE = 16 / sizeof(value_type) < 1 ? 1 :
  16 / sizeof(value_type);
// roundup mask for allocated buffers, [0, 15]:
static constexpr size_type _ALLOC_MASK = sizeof(value_type) <= 1 ? 15
```

> 而它又等于一个constexpr（常量表达式），这个例子中它是16，减去1后是15

C++

```
#include <iostream>
#include <string>

void* operator new(size_t size)  // 操作符重载
{
    std::cout << "Allocating " << size << " bytes\n";
    return malloc(size);
}

int main()
{
    std::string name = "Cherno";  // 显然小于15个字符
    std::cin.get();
}
```

如果F5运行，会发现依然打印分配了8字节内存，这其实是VS的string类的问题，基本只在Debug模式下发生，因此切换到Release模式运行就可以看到想要的结果了：

```cpp
void* operator new(size_t size)
{
    std::cout << "Allocating " << size << " bytes\n";
    return malloc(size);
}

int main()
{
    std::string name = "Cherno";
    std::cin.get();
}
```

```
C:\Dev\Small String\Release\Small String.exe
```

```cpp
std::string name = "Cherno Small St"; // 15个字符，没有分配
std::string name = "Cherno Small St "; // 16个字符，立即产生32个字节的堆上内存分配
```

```cpp
std::string name = "Cherno Small St ";
std::cin.get();
```

```
C:\Dev\Small String\Release\Small String.exe
Allocating 32 bytes
```

这是一种优化，效率更高，更快。

那Debug模式下发生了什么导致了分配呢?

```cpp
#if _ITERATOR_DEBUG_LEVEL == 0
_INLINE_VAR constexpr _Fake_allocator _Fake_alloc{};
#define _GET_PROXY_ALLOCATOR(_Alty, _Al) _Fake_alloc // TRANSITION, VSO-1284799, should be
  _Fake_allocator{}
template <class _Alloc>
using _Container_proxy_ptr = _Fake_proxy_ptr_impl;
#else // _ITERATOR_DEBUG_LEVEL == 0
#define _GET_PROXY_ALLOCATOR(_Alty, _Al) static_cast<_Rebind_alloc_t<_Alty, _Container_proxy>>(
template <class _Alloc>
using _Container_proxy_ptr = _Container_proxy_ptr12<_Rebind_alloc_t<_Alloc, _Container_proxy>>;
#endif // _ITERATOR_DEBUG_LEVEL == 0
```

> 如果调试等级为0，也就是Release模式下，只会调用Fake_allocator，而它其实什么都不做：
>
> ```cpp
> struct _Fake_allocator {};
> ```
>
> 如果是Debug模式就会因这个proxy_ptr造成分配。