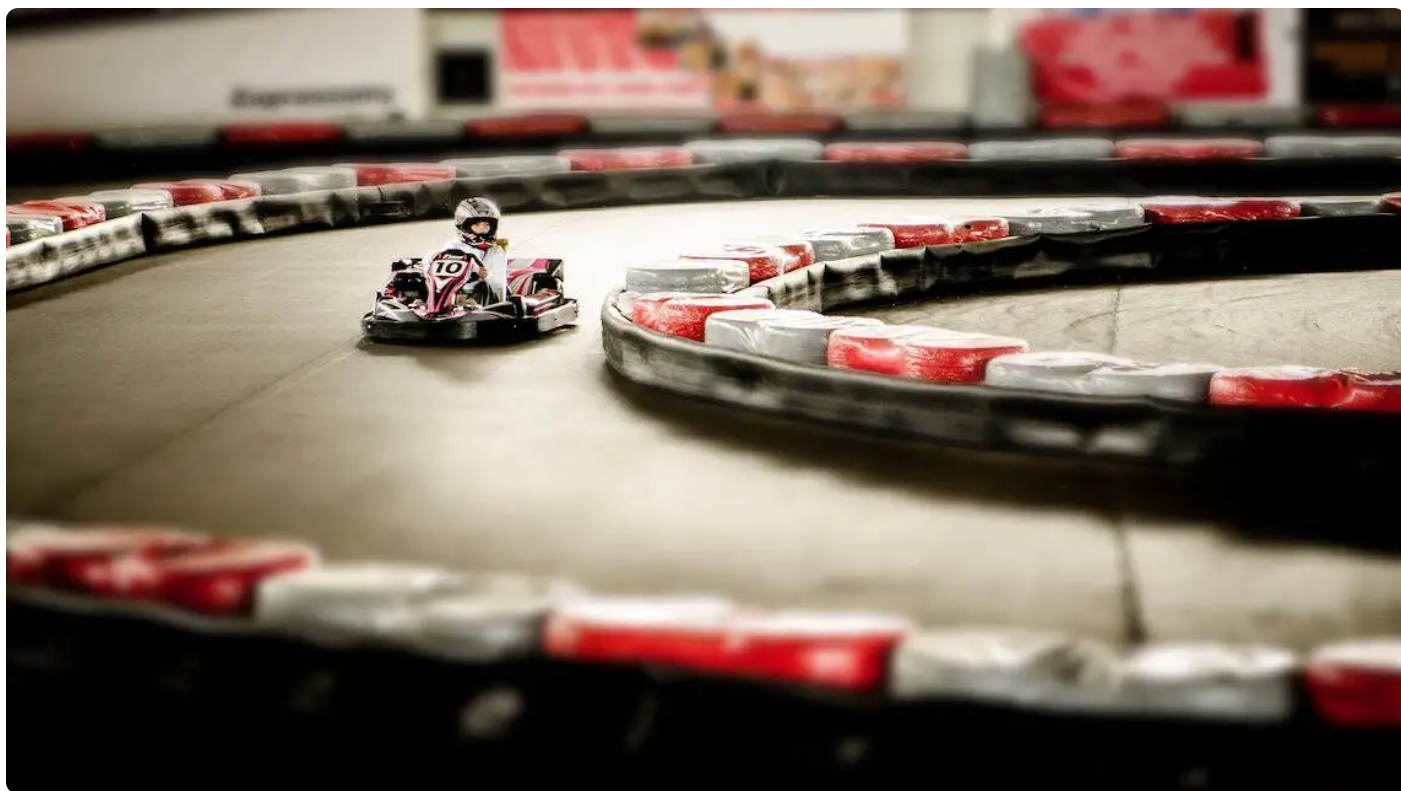


## 16 | Bit library（一）：如何利用新bit操作库释放编程生产力？

2023-02-27 卢誉声 来自北京

《现代C++20实战高手课》

课程介绍 >



讲述：卢誉声

时长 08:45 大小 8.00M



你好，我是卢誉声。

我们都知道，C++ 继承了 C 语言所有的底层操作能力，其中最重要的两个特性就是指针和位操作。对于指针，现代 C++ 标准已经通过智能指针提出了较好的解决方案。

但是在 C++20 之前，C++ 的位操作支持还是很“基础”的。它缺乏安全性，而且不够灵活。因此，我们就围绕 C++20 对位操作能力的扩展这个话题，讨论一下全新的 Bit library。

好，话不多说，就让我们从基本的 C++ 位操作开始讲起（课程配套代码可以从 [这里](#) 获取）。

### C++ 位操作技术与不足

C++ 提供的基础位操作技术与 C 语言一脉相承，主要通过位操作符对整数进行位操作。我对这些基本操作做了一个总结，你可以参考后面的表格回顾一下。

操作符	解释	示例
&	二元操作符，位与	1 & 2
	二元操作符，位或	1   2
^	二元操作符，按位异或	1 ^ 2
~	一元操作符，位非	~1
<<	二元操作符，左移	1 << 2
>>	二元操作符，右移	1 >> 2



相比 C 语言，C++ 一直为 C 的底层能力提供一些高层次的安全化包装，比如为了解决裸指针的各种安全缺陷，提出了各类智能指针。

基于这种思路，C++ 也通过标准库提供了 `bitset`，对二进制位串进行包装，可以在整数和 `bitset` 以及其字符串表示之间进行转换，并支持表格中的几个基础的位操作符。

但在 C++20 之前，C++ 的位操作支持还是很“基础”的，我们重点讨论几个比较明显的问题。

**首先，没有提供字节序的检测和转换能力。**基于位进行二进制数据解析的时候，最大的问题就是不同 CPU 的大小端设计（比如 x86 体系结构是小端字节序，arm 支持采用大小端字节序任选其一）。

这就导致在不同体系结构下，编写位操作可能产生各种兼容性问题。C++ 不仅没有提供编译目标架构的字节序检测能力，也没有提供不同字节序之间的转换能力，所以这些基础设施我们只能自行实现。

**其次，缺乏安全的强制类型转换手段。**一些场景，比如对浮点类型的数值进行位操作需要先将浮点类型转换成对应宽度的无符号整数，就经常需要将一些数据强制转换为另一个类型，而不

改变其二进制位的值。

C++ 仅提供了 `reinterpret_cast`，执行指针类型的强制转换，但没有提供值的强制类型转换能力。虽然我们可以通过 `memcpy` 等手段实现，但这样也不够安全——编译器无法检测到可能发生的任何问题。

**最后，是移位操作问题。**C++ 把移位的具体含义交给了具体实现。具体来说，就是移位操作分为“算术移位”和“逻辑移位”，算术移位需要考虑到有符号整数的符号问题，逻辑移位是直接补零，C++ 并没有定义有符号整数的具体实现方式，除了有符号整数的移位问题，C++ 和 C 一样，并没有提供循环移位的手段。

## C++20 位操作库

既然位操作的潜在问题这么多，C++ 是怎么解决的呢？

在 C++20，C++ 标准终于开始思考解决这些基本的位操作问题了。为此提供了位操作库，具体实现在 `<bit>` 头文件中，我们分别来看看。

### 字节序处理

C++20 提供了枚举类 `endian`（这早就该标准化了嘛☺），用来定义字节序的大小端。具体定义如下所示：

```
1 enum class endian {
2     little,
3     big,
4     native // = little/big
5 };
```

 复制代码

这里着重强调 `endian::native` 这一枚举值，它由编译器根据编译目标体系结构自动确定——这可太棒了！

我们结合实际例子来体会一下用法。

```
1 #include <iostream>
```

 复制代码

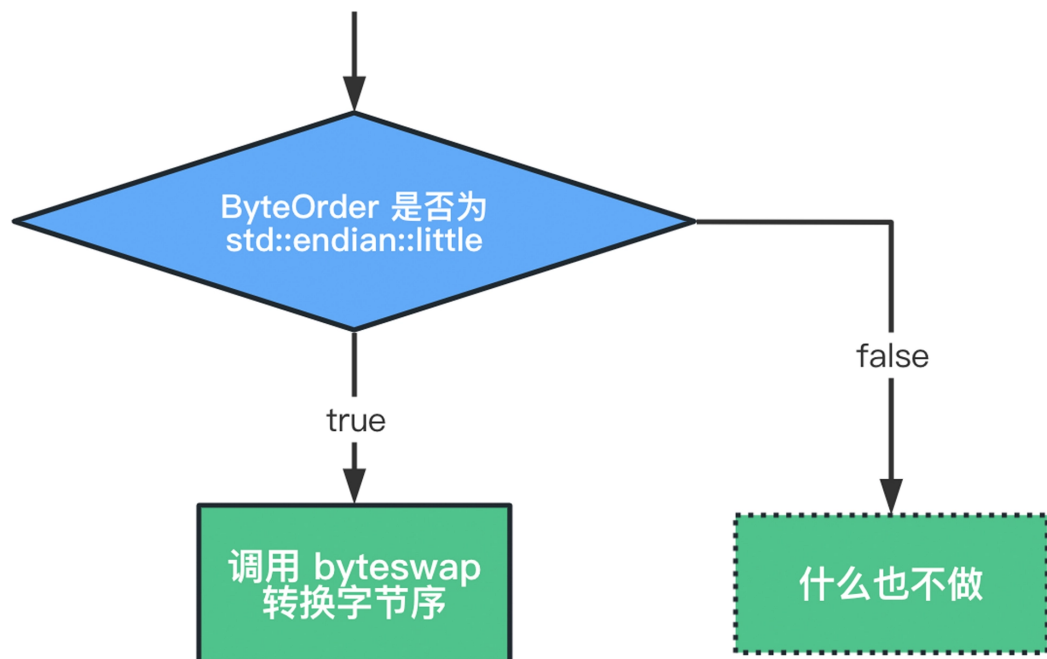
```

2 #include <bit>
3 #include <cstdint>
4 #include <concepts>
5
6 template <std::integral T>
7 T byteswap(T src) {
8     T dest = 0;
9
10    for (uint8_t* pSrcByte = reinterpret_cast<uint8_t*>(&src),
11         *pDestByte = reinterpret_cast<uint8_t*>(&dest) + sizeof(T) - 1;
12         pSrcByte != reinterpret_cast<uint8_t*>(&src) + sizeof(T);
13         ++pSrcByte, --pDestByte) {
14        *pDestByte = *pSrcByte;
15    }
16
17    return dest;
18 }
19
20 template <std::integral T, std::endian ByteOrder = std::endian::native>
21     requires (ByteOrder == std::endian::little)
22 T consumeBigEndian(T src) {
23     return byteswap(src);
24 }
25
26 template <std::integral T, std::endian ByteOrder = std::endian::native>
27     requires (ByteOrder == std::endian::big)
28 T consumeBigEndian(T src) {
29     return src;
30 }
31
32 int main() {
33     uint8_t networkBuffer[] {
34         1, 2
35     };
36
37     // 将从网络数据流中获取的值直接转换成uint16_t
38     uint16_t networkValue = *(reinterpret_cast<uint16_t*>(networkBuffer));
39     std::cout << "Original value: " << networkValue << std::endl;
40
41     // 无论如何都转换字节序
42     uint16_t swappedValue = byteswap(networkValue);
43     std::cout << "Swaped value: " << swappedValue << std::endl;
44
45     // 仅对字节序为大端的平台转换字节序
46     uint16_t checkedValue = consumeBigEndian(networkValue);
47     std::cout << "Checked value: " << checkedValue << std::endl;
48
49     return 0;
50 }

```

这段代码的作用是检测目标平台的字节序，并将一个大端数据转换成小端数据。如果目标平台本身字节序就是大端，那么什么都不会执行。

C++23 提供了 `byteswap` 帮助我们执行字节序转换，但 C++20 没有提供该特性，所以我们只能自己实现一个版本。



这段代码定义了两个版本的 `consumeBigEndian`，通过 `requires` 在编译时检测 `std::endian::native` 是否为 `std::endian::little`，如果是就会通过 `byteswap` 转换字节序，否则返回原始值。

## 强制类型转换

事实上，C 语言提供的强制类型转换有很多潜在的类型安全问题，为了避免大家使用 C 风格的强制类型转换，C++ 提供了 `static_cast`、`const_cast` 和 `reinterpret_cast`，为不同场景的类型转换服务。

这些类型转换符都有自己的约束与类型安全检测，迫使开发者在进行类型转换时，要先弄清楚自己真的需要哪种转换。但是，它们都解决不了一个常见的场景——将一个值转换成二进制位数相同的另一个值。

为此，C++20 通过 `<bit>` 头文件提供了 `bit_cast` 这个函数。后面是一段示例代码。

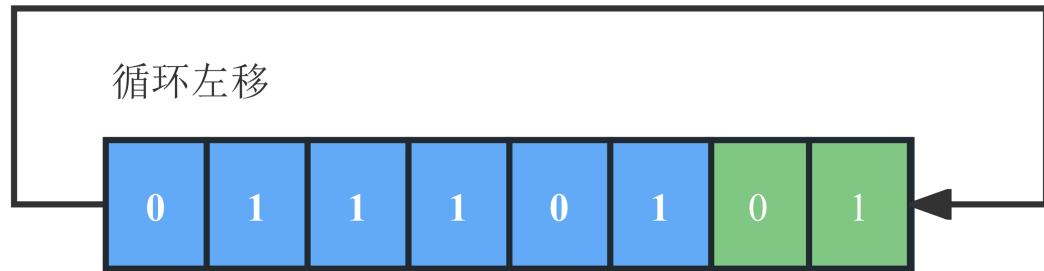
```
1 #include <iostream>
2 #include <bit>
3 #include <cstdint>
4
5 int main() {
6     float f32v = 784.513f;
7     uint32_t u32v = std::bit_cast<uint32_t>(f32v);
8
9     std::cout << "f32v: " << f32v << std::endl;
10    std::cout << "u32v: " << u32v << std::endl;
11
12    double f64v = 1123.11f;
13    uint64_t u64v = std::bit_cast<uint64_t>(f64v);
14
15    std::cout << "f64v: " << f64v << std::endl;
16    std::cout << "u64v: " << u64v << std::endl;
17
18    // 编译错误
19    uint16_t u64v = std::bit_cast<uint16_t>(f64v);
20
21    return 0;
22 }
```

在这段代码中，我们通过 `bit_cast` 将一个 `float` 类型变量转换为 `uint32_t`，并将一个 `uint64_t` 类型变量转换为 `double`，但是无法将 `float` 类型转换为 `uint16_t`。

这样一来，我们就多了一个类型转换工具，还可以确保特定的类型安全，降低滥用 C 类型转换带来的风险。

## 循环移位

循环移位是另一个常见需求，循环移位与普通移位差别在于移位后的补位规则。举个例子，下图是对 8 位的二进制串，分别循环左移与逻辑左移 2 位的结果。



逻辑左移



逻辑左移会对“由移动产生”的位直接补零，而循环左移则会将移出的位串，直接循环移动到新值的末尾，形成一个循环。

C++20 提供了用于循环左移的 `std::rotl`，还有用于循环右移的 `std::rotr`。后面是示例代码。

复制代码

```
1 #include <iostream>
2 #include <bit>
3 #include <bitset>
4 #include <cstdint>
5 #include <iomanip>
6
7 int main() {
8     uint8_t value = 0b01011101;
9     std::cout << std::setw(16) << std::left << "value" << " = " << std::bitset<
10     std::cout << std::setw(16) << std::left << "rotl" << " = " << std::bitset<8
11     std::cout << std::setw(16) << std::left << "left logical" << " = " << std::
12     std::cout << std::setw(16) << std::left << "rotr" << " = " << std::bitset<8
13     std::cout << std::setw(16) << std::left << "right logical" << " = " << std:
14
15     return 0;
16 }
```

在这段代码中，我们调用了 `rotl` 和 `rotr` 执行循环移位，同时调用了 `<<` 和 `>>`。我们可以从执行结果中看出不同的移位方式之间的差别。

```
value           = 01011101
rotl            = 01110101
left logical    = 01110100
rotr            = 01010111
right logical   = 00010111
```

## 其他位运算

除了上述位操作支持，C++20 还提供了一些其他简单的位操作函数，我用表格进行了总结。

操作符	函数备注
<code>has_single_bit</code>	检查输入数字是否为 2 的幂 (也就是二进制位只有一个 1)
<code>bit_ceil</code>	返回不小于指定数值的最小的 2 的幂
<code>bit_floor</code>	返回不大于指定数值的最大的 2 的幂
<code>bit_width</code>	返回可以表示指定数值的最小二进制位数
<code>countl_zero</code>	从最高有效位 (自左向右) 开始计算连续 0 的数量
<code>countl_one</code>	从最高有效位 (自左向右) 开始计算连续 1 的数量
<code>countr_zero</code>	从最低有效位 (自右向左) 开始计算连续 0 的数量
<code>countr_one</code>	从最低有效位 (自右向左) 开始计算连续 1 的数量
<code>popcount</code>	计算无符号整数中位数为 1 的数量



我为你简单演示一下这些函数的用法，代码是后面这样。

复制代码

```
1 #include <iostream>
2 #include <bit>
3 #include <bitset>
4 #include <stdint>
```



```

5 #include <format>
6
7 // 测试has_single_bit
8 void testHasSingleBit() {
9     for (uint32_t value = 0; value < 8u; ++value) {
10         std::cout << std::format("has_single_bit({}): {}",
11             std::bitset<3>(value).to_string(),
12             std::has_single_bit(value)
13         ) << std::endl;
14     }
15 }
16
17 // 测试bit_ceil与bit_floor
18 void testCeilFloor() {
19     for (uint32_t value = 0; value < 8u; ++value) {
20         std::cout << std::format("ceil({}): {}",
21             std::bitset<4>(value).to_string(),
22             std::bitset<4>(std::ceil(value)).to_string()
23         ) << std::endl;
24     }
25
26     for (uint32_t value = 0; value < 8u; ++value) {
27         std::cout << std::format("ceil({}): {}",
28             std::bitset<4>(value).to_string(),
29             std::bitset<4>(std::floor(value)).to_string()
30         ) << std::endl;
31     }
32 }
33
34 // 利用constexpr和bit_width自动计算bitset的模板参数
35 template <std::uint64_t value>
36 std::bitset<std::bit_width(value)> wbitset() {
37     constexpr int bitWidth = std::bit_width(value);
38
39     return std::bitset<bitWidth>(value);
40 }
41
42 // 测试bit_width
43 void testBitWidth() {
44     constexpr uint32_t value = 97u;
45     constexpr uint32_t ceilValue = std::bit_ceil(value);
46     constexpr uint32_t floorValue = std::bit_floor(value);
47
48     std::cout << std::format("ceil({}): {}\nfloor({}): {}",
49         std::bitset<std::bit_width(value)>(value).to_string(),
50         std::bitset<std::bit_width(ceilValue)>(ceilValue).to_string(),
51         wbitset<value>().to_string(),
52         wbitset<floorValue>().to_string()
53     ) << std::endl;
54
55 }
56

```

```

57 // 测试countl_zero, countl_one, countr_zero, countr_one, popcount
58 void testCounts() {
59     for (const std::uint8_t value : { 0, 0b11111111, 0b11000000, 0b00000110 })
60         std::cout << std::format("countl_zero({}) = {}",
61                                 std::bitset<8>(value).to_string(),
62                                 std::countl_zero(value)
63                             ) << std::endl;
64 }
65
66 for (const std::uint8_t value : { 0, 0b11111111, 0b11000000, 0b00000110 })
67     std::cout << std::format("countl_one({}) = {}",
68                             std::bitset<8>(value).to_string(),
69                             std::countl_one(value)
70                         ) << std::endl;
71 }
72
73 for (const std::uint8_t value : { 0, 0b11111111, 0b11000000, 0b00000110 })
74     std::cout << std::format("countr_zero({}) = {}",
75                             std::bitset<8>(value).to_string(),
76                             std::countr_zero(value)
77                         ) << std::endl;
78 }
79
80 for (const std::uint8_t value : { 0, 0b11111111, 0b11000000, 0b00000111 })
81     std::cout << std::format("countr_one({}) = {}",
82                             std::bitset<8>(value).to_string(),
83                             std::countr_one(value)
84                         ) << std::endl;
85 }
86
87 for (const std::uint8_t value : { 0, 0b11111111, 0b11000000, 0b00000111 })
88     std::cout << std::format("popcount({}) = {}",
89                             std::bitset<8>(value).to_string(),
90                             std::popcount(value)
91                         ) << std::endl;
92 }
93 }
94
95 int main() {
96     testHasSingleBit();
97     testCeilFloor();
98     testBitWidth();
99     testCounts();
100
101     return 0;
102 }

```

这些都是简单的函数示例。下图是运行时输出，供你参考。

```
ceil(0011): 0011
ceil(0100): 0100
ceil(0101): 0101
ceil(0110): 0110
ceil(0111): 0111
ceil(1100001): 10000000
floor(1100001): 1000000
countl_zero(00000000) = 8
countl_zero(11111111) = 0
countl_zero(11000000) = 0
countl_zero(00000110) = 5
countl_one(00000000) = 0
countl_one(11111111) = 8
countl_one(11000000) = 2
countl_one(00000110) = 0
countr_zero(00000000) = 8
countr_zero(11111111) = 0
countr_zero(11000000) = 6
countr_zero(00000110) = 1
countr_one(00000000) = 0
countr_one(11111111) = 8
countr_one(11000000) = 0
countr_one(00000111) = 3
popcount(00000000) = 0
popcount(11111111) = 8
popcount(11000000) = 2
popcount(00000111) = 3
```

其中，36-41 行的 `wbitset` 函数是一个比较巧妙的实现，我们一起做个分析。

`bitset` 类型可以帮助我们快速将一个整数转换为二进制串，并将其转换为字符串。但是 `bitset` 需要通过模板参数指定位串的位数。因此，这种情况下我们需要自己来计算 `bitset` 的位数到底是多少——这太麻烦了。

为了解决这个问题，我们使用 `std::bit_width` 计算表示模板参数 `value` 的最小位数。由于 `bit_width` 是 `constexpr` 函数，因此，如果它的参数是编译时常量，那么就能直接用在模板参数里！这样就能确定 `bitset` 的最小位数了。

## 总结

C++20 作为一个里程碑式的重要标准，终于开始解决基本的位操作问题，这其中包括：

1. 字节序的检测和转换能力。
2. 补充完善的安全的强制类型转换工具。
3. 增强的移位操作。

这些新工具为我们实现位操作提供了更加完备的支持。同时，也为实现序列化和反序列化提供了标准化支持。


那么，新的位操作库到底是怎么帮助我们释放编程生产力的呢（特别是在序列化和反序列化方面）？下一讲，我将为你娓娓道来...

## 课后思考

C++20 位操作库提供的函数，其实我们也能自己实现，请你思考 `bit_width` 和 `bit_floor` 这两个函数如何实现，可以把你能想到的最简洁的实现方式贴出来。

欢迎将你的方案与大家一起分享。我们一同交流。下一讲见！

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

[上一篇](#) 15 | Formatting实战：如何构建一个数据流处理实例？

[下一篇](#) 17 | Bit library（二）：如何利用新bit操作库释放编程生产力？



peter

2023-02-28 来自北京

请教老师几个问题：

**Q1:** 浮点數位操作的转换是隐式转换吗？

C++20之前的浮点數位操作，需要先转换为整数，这个转换是隐式转换吗？还是需要写代码实现？（好久不用C++，有点忘记了）

**Q2:** 移位怎么区分算术移位和逻辑移位？

文中提到，C++20之前的移位操作，移位的具体含义交给了具体实现。那么，具体怎么区分是算术移位还是逻辑移位？

**Q3:** C++20支持序列化和反序列化吗？

文中提到“这些新工具为我们实现位操作提供了更加完备的支持。同时，也为实现序列化和反序列化提供了标准化支持”，那么，C++20已经支持序列化和反序列化了吗？（C++20之前的版本支持吗？）

作者回复: 1. 对浮点数进行移位操作需要显示转换，而且不能`static_cast`，需要先`reinterpret_cast`变量指针到整型的指针，移位后再转换回去，非常麻烦。

2. 区分只能看编译器是如何定义的。

3. C++20并不是支持序列化/反序列化，而是在C++20中，标准库提供了大端小端判断、交换字节序之类的基础工具支持，我们可以基于标准库更容易构建序列化和反序列化库，C++20之前没有相关的任何支持。

