

Contents

I 基本语言特性	1
1 结构化绑定	3
1.1 细说结构化绑定	4
1.2 结构化绑定的适用场景	7
1.2.1 结构体和类	7
1.2.2 原生数组	8
1.2.3 <code>std::pair</code> , <code>std::tuple</code> 和 <code>std::array</code>	8
1.3 为结构化绑定提供 Tuple-Like API	10
1.4 后记	17
2 带初始化的 <code>if</code> 和 <code>switch</code> 语句	19
2.1 带初始化的 <code>if</code> 语句	19
2.2 带初始化的 <code>switch</code> 语句	21
2.3 后记	21
3 内联变量	23
3.1 内联变量产生的动机	23
3.2 使用内联变量	25
3.3 <code>constexpr static</code> 成员现在隐含 <code>inline</code>	26
3.4 内联变量和 <code>thread_local</code>	27
3.5 后记	29
4 聚合体扩展	31
4.1 扩展聚合体初始化的动机	31
4.2 使用聚合体扩展	32
4.3 聚合体的定义	34
4.4 向后的不兼容性	34
4.5 后记	35

5	强制省略拷贝或传递未实质化的对象	37
5.1	强制省略临时变量拷贝的动机	37
5.2	强制省略临时变量拷贝的好处	39
5.3	更明确的值类型体系	40
5.3.1	值类型体系	40
5.3.2	自从C++17起的值类型体系	42
5.4	未实质化的返回值传递	43
5.5	后记	43
6	lambda 表达式扩展	45
6.1	constexpr lambda	45
6.1.1	使用constexpr lambda	47
6.2	向lambda传递this的拷贝	48
6.3	以常量引用捕获	50
6.4	后记	50
7	新属性和属性特性	51
7.1	[[nodiscard]] 属性	51
7.2	[[maybe_unused]] 属性	53
7.3	[[fallthrough]] 属性	53
7.4	通用的属性扩展	54
7.5	后记	55
8	其他语言特性	57
8.1	嵌套命名空间	57
8.2	有定义的表达式求值顺序	57
8.3	更宽松的用整型初始化枚举值的规则	60
8.4	修正auto类型的列表初始化	61
8.5	十六进制浮点数字面量	62
8.6	UTF-8 字符字面量	63
8.7	异常声明作为类型的一部分	63
8.8	单参数static_assert	67
8.9	预处理条件__has_include	67
8.10	后记	68
II	模板特性	69
9	类模板参数推导	71
9.1	使用类模板参数推导	71

9.1.1	默认以拷贝方式推导	73
9.1.2	推导 <code>lambda</code> 的类型	73
9.1.3	没有类模板部分参数推导	75
9.1.4	使用类模板参数推导代替快捷函数	76
9.2	推导指引	77
9.2.1	使用推导指引强制类型退化	78
9.2.2	非模板推导指引	79
9.2.3	推导指引与构造函数冲突	79
9.2.4	显式推导指引	80
9.2.5	聚合体的推导指引	81
9.2.6	标准推导指引	81
9.3	后记	86
10	编译期 <code>if</code> 语句	87
10.1	编译期 <code>if</code> 语句的动机	88
10.2	使用编译期 <code>if</code> 语句	89
10.2.1	编译期 <code>if</code> 的注意事项	90
10.2.2	其他编译期 <code>if</code> 的示例	92
10.3	带初始化的编译期 <code>if</code> 语句	94
10.4	在模板之外使用编译期 <code>if</code>	95
10.5	后记	96
11	折叠表达式	97
11.1	折叠表达式的动机	98
11.2	使用折叠表达式	98
11.2.1	处理空参数包	99
11.2.2	支持的运算符	102
11.2.3	使用折叠表达式处理类型	106
11.3	后记	107
12	处理字符串字面量模板参数	109
12.1	在模板中使用字符串	109
12.2	后记	110
13	占位符类型作为模板参数	111
13.1	使用 <code>auto</code> 模板参数	111
13.1.1	字符和字符串模板参数	112
13.1.2	定义元编程常量	113
13.2	使用 <code>auto</code> 作为变量模板的参数	114
13.3	使用 <code>decltype(auto)</code> 模板参数	116

13.4 后记	116
14 扩展的 using 声明	117
14.1 使用变长的 using 声明	117
14.2 使用变长 using 声明继承构造函数	118
14.3 后记	119
III 新的标准库组件	121
15 std::optional<>	123
15.1 使用 std::optional<>	123
15.1.1 可选的返回值	123
15.1.2 可选的参数和数据成员	125
15.2 std::optional<> 类型和操作	126
15.2.1 std::optional<> 类型	126
15.2.2 std::optional<> 的操作	126
15.3 特殊情况	133
15.3.1 bool 类型或原生指针的可选对象	133
15.3.2 可选对象的可选对象	133
15.4 后记	134
16 std::variant<>	135
16.1 std::variant<> 的动机	135
16.2 使用 std::variant<>	136
16.3 std::variant<> 类型和操作	138
16.3.1 std::variant<> 类型	138
16.3.2 std::variant<> 的操作	138
16.3.3 访问器	142
16.3.4 异常造成的无值	146
16.4 使用 std::variant 实现多态的异质集合	147
16.4.1 使用 std::variant 实现几何对象	147
16.4.2 使用 std::variant 实现其他异质集合	150
16.4.3 比较多态的 variant	151
16.5 std::variant<> 的特殊情况	152
16.5.1 同时有 bool 和 std::string 选项	152
16.6 后记	152

17	<code>std::any</code>	153
17.1	使用 <code>std::any</code>	153
17.2	<code>std::any</code> 类型和操作	156
17.2.1	Any 类型	156
17.2.2	Any 操作	156
17.3	后记	159
18	<code>std::byte</code>	161
18.1	使用 <code>std::byte</code>	161
18.2	<code>std::byte</code> 类型和操作	162
18.2.1	<code>std::byte</code> 类型	163
18.2.2	<code>std::byte</code> 操作	163
18.3	后记	166
19	字符串视图	167
19.1	和 <code>std::string</code> 的不同之处	167
19.2	使用字符串视图	168
19.3	使用字符串视图作为参数	168
19.3.1	字符串视图有害的一面	170
19.4	字符串视图类型和操作	173
19.4.1	字符串视图的具体类型	173
19.4.2	字符串视图的操作	174
19.4.3	其他类型对字符串视图的支持	177
19.5	在 API 中使用字符串视图	177
19.5.1	使用字符串视图代替 <code>string</code>	178
19.6	后记	179
20	文件系统库	181
20.1	基本的示例	181
20.1.1	打印文件系统路径类的属性	181
20.1.2	用 <code>switch</code> 语句处理不同的文件系统类型	184
20.1.3	创建不同类型的文件	185
20.1.4	使用并行算法处理文件系统	190
20.2	原则和术语	190
20.2.1	通用的可移植的分隔符	190
20.2.2	命名空间	190
20.2.3	文件系统路径	191
20.2.4	正规化	192
20.2.5	成员函数 VS 独立函数	192

20.2.6	错误处理	193
20.2.7	文件类型	195
20.3	路径操作	196
20.3.1	创建路径	196
20.3.2	检查路径	197
20.3.3	路径 I/O 和转换	199
20.3.4	本地和通用格式的转换	203
20.3.5	修改路径	204
20.3.6	比较路径	206
20.3.7	其他路径操作	207
20.4	文件系统操作	208
20.4.1	文件属性	208
20.4.2	文件状态	211
20.4.3	权限	213
20.4.4	修改文件系统	215
20.4.5	符号链接和依赖文件系统的路径转换	218
20.4.6	其他文件系统操作	220
20.5	遍历目录	221
20.5.1	目录项	222
20.6	后记	223

IV 已有标准库的拓展和修改 227

21	类型特征扩展	229
21.1	类型特征后缀 <code>_v</code>	229
21.2	新的类型特征	230
21.3	后记	235
22	并行 STL 算法	237
22.1	使用并行算法	238
22.1.1	使用并行 <code>for_each()</code>	238
22.1.2	使用并行 <code>sort()</code>	241
22.2	执行策略	243
22.3	异常处理	243
22.4	不使用并行算法的好处	244
22.5	并行算法概述	244
22.6	并行编程的新算法的动机	245
22.6.1	<code>reduce()</code>	245

22.7 后记	253
23 新的STL算法详解	255
23.1 <code>std::for_each_n()</code>	255
23.2 新的数学STL算法	257
23.2.1 <code>std::reduce()</code>	257
23.2.2 <code>std::transform_reduce()</code>	259
23.2.3 <code>std::inclusive_scan()</code> 和 <code>std::exclusive_scan()</code>	262
23.2.4 <code>std::transform_inclusive_scan()</code> 和 <code>std::transform_exclusive_scan()</code>	264
23.3 后记	266
24 子串和子序列搜索器	267
24.1 使用子串搜索器	267
24.1.1 通过 <code>search()</code> 使用搜索器	267
24.1.2 直接使用搜索器	269
24.2 使用泛型子序列搜索器	269
24.3 使用搜索器谓词	270
24.4 后记	271
25 其他工具函数和算法	273
25.1 <code>size()</code> , <code>empty()</code> , <code>data()</code>	273
25.1.1 泛型 <code>size()</code> 函数	273
25.1.2 泛型 <code>empty()</code> 函数	275
25.1.3 泛型 <code>data()</code> 函数	275
25.2 <code>as_const()</code>	276
25.2.1 以常量引用捕获	276
25.3 <code>clamp()</code>	276
25.4 <code>sample()</code>	278
25.5 后记	281
26 容器和字符串扩展	283
26.1 节点句柄	283
26.1.1 修改 key	283
26.1.2 在容器之间移动节点句柄	284
26.1.3 合并容器	286
26.2 <code>emplace</code> 改进	287
26.2.1 <code>emplace</code> 函数的返回类型	287
26.2.2 <code>map</code> 的 <code>try_emplace()</code> 和 <code>insert_or_assign()</code>	287
26.2.3 <code>try_emplace()</code>	287

26.2.4	<code>insert_or_assign()</code>	289
26.3	对不完全类型的容器支持	289
26.4	<code>string</code> 改进	291
26.5	后记	292
27	多线程和并发	293
27.1	补充的互斥量和锁	293
27.1.1	<code>std::scoped_lock</code>	293
27.1.2	<code>std::shared_mutex</code>	294
27.2	原子类型的 <code>is_always_lock_free</code>	295
27.3	<code>cache</code> 行大小	296
27.4	后记	297
28	标准库的其他微小特性和修改	299
28.1	<code>std::uncaught_exceptions()</code>	299
28.2	共享指针改进	301
28.2.1	对原生 C 数组的共享指针的特殊处理	301
28.2.2	共享指针的 <code>reinterpret_pointer_cast</code>	301
28.2.3	共享指针的 <code>weak_type</code>	302
28.2.4	共享指针的 <code>weak_from_this</code>	302
28.3	数学扩展	303
28.3.1	最大公约数和最小公倍数	303
28.3.2	<code>std::hypot()</code> 的三参数重载	304
28.3.3	数学的特殊函数	304
28.3.4	<code>chrono</code> 扩展	304
28.3.5	<code>constexpr</code> 扩展和修正	306
28.3.6	<code>noexcept</code> 扩展和修正	307
28.3.7	后记	308
V	专家的工具	309
29	多态内存资源 (PMR)	311
29.1	使用标准内存资源	311
29.1.1	示例	311
29.1.2	标准内存资源	316
29.1.3	详解标准内存资源	318
29.2	定义自定义内存资源	323
29.2.1	内存资源的等价性	326
29.3	为自定义类型提供内存资源支持	328

29.3.1 定义 PMR 类型	328
29.3.2 使用 PMR 类型	330
29.3.3 处理不同的类型	331
29.4 后记	332
30 使用 new 和 delete 管理超对齐数据	333
30.1 使用带有对齐的 new 运算符	333
30.1.1 不同的动态/堆内存竞争	333
30.1.2 使用 new 表达式传递对齐	333
30.2 实现内存对齐分配的 new() 运算符	333
30.2.1 在 C++17 之前实现对齐的内存分配	333
30.2.2 实现类型特化的 new() 运算符	333
30.3 实现全局的 new() 运算符	333
30.4 追踪所有 ::new 调用	333
30.5 后记	333
31 std::to_chars() 和 std::from_chars()	335
31.1 字符序列和数字值之间的底层转换的动机	335
31.2 使用示例	335
31.2.1 from_chars	335
31.2.2 to_chars()	335
32 std::launder()	337
32.1 std::launder() 的动机	337
32.2 launder() 如何解决问题	340
32.3 为什么/什么时候 launder() 不生效	341
32.4 后记	342
33 编写泛型代码的改进	339
33.1 std::invoke<>()	339
33.2 std::bool_constant<>	341
33.3 std::void_t<>	342
33.4 后记	343
VI 一些通用的提示	345
34 总体性的 C++17 设置	347
34.1 __cplusplus 的值	347
34.2 与 C11 的兼容性	347
34.3 处理信号处理器	347

34.4 向前运行保证	348
34.5 后记	348
35 废弃和移除的特性	349
35.1 废弃和移除的核心语言特性	349
35.1.1 <code>throw</code> 声明	349
35.1.2 关键字 <code>register</code>	349
35.1.3 禁止 <code>bool</code> 类型的 <code>++</code>	350
35.1.4 三字符	350
35.1.5 <code>static constexpr</code> 成员的定义/重复声明	350
35.2 废弃和移除的库特性	350
35.2.1 <code>auto_ptr</code>	350
35.2.2 算法 <code>random_shuffle()</code>	350
35.2.3 <code>unary_function</code> 和 <code>binary_function</code>	351
35.2.4 <code>ptr_fun()</code> 、 <code>mem_fun()</code> 、绑定器	351
35.2.5 <code>std::function<></code> 的分配器支持	351
35.2.6 废弃的 IO 流别名	351
35.2.7 废弃的库特性	351
35.3 后记	352

Part I

基本语言特性

这一部分介绍了 C++17 中新的核心语言特性，但不包括那些专为泛型编程（即 `template`）设计的特性。这些新增的特性对于应用程序员的日常编程非常有用，因此每一个使用 C++17 的 C++ 程序员都应该了解它们。

专为模板编程设计的新的核心语言特性在 [Part II](#) 中介绍。

Part II

模板特性

这一部分介绍了 C++17 为泛型编程（即 `template`）提供的新的语言特性。

我们首先从类模板参数推导开始，这一特性只影响模板的使用。之后的章节会介绍为编写泛型代码（函数模板，类模板，泛型库等）的程序员提供的新特性。

Part III

新的标准库组件

这一部分介绍 C++17 中新的标准库组件。

Part IV

已有标准库的拓展和修改

这一部分介绍 C++17 对已有标准库组件的拓展和修改。

Part V

专家的工具

这一部分介绍了普通应用程序员通常不需要知道的新的语言特性和库。它主要包括了为编写基础库和语言特性的程序员准备的用来解决特殊问题语言特性（例如修改了堆内存的管理方式）。

Chapter 32

std::launder()

有一个叫 `std::launder()` 的新的库函数，就我了解和看到的，它是一个解决核心问题的方法，然而，它不能真的产生效果。

32.1 std::launder() 的动机

根据当前的标准，下面的代码会导致未定义行为：

```
struct X {
    const int n;
    double d;
};
X* p = new X{7, 8.8};
new (p) X{42, 9.9};    // 请求把一个新的值放进p处
int i = p->n;          // 未定义行为（i可能是7也可能是42）
auto d = p->d;         // 也是未定义行为（d可能是8.8也可能是9.9）
```

原因是在当前的内存模型中，C++ 标准中的 `[basic.life]` 这一节中，粗略的讲到：

如果，...，一个新的对象在一个已经被原本对象占据的位置处创建，

- 一个指向原本的对象指针，
- 一个引用原本对象的引用，
- 原本对象的名称

将会自动指向新的对象...如果：

- 原本对象的类型没有 `const` 修饰，并且如果是类类型的话还要不包含 `const` 修饰的或者引用类型的非静态数据成员。
- ...

¹ 这个行为并不是新的。它在 C++03 中就被指明，目的是为了允许几项编译器优化（包括使用虚函数时的相似优化）。

¹译者注：因为英语喜欢把if放在后边，所以此处的第二个“如果”是前边的结果的条件。

按照标准中的说法，当对象中有常量或者引用类型的成员时，我们必须保证每次访问内存时都使用 `placement new` 返回的值：

```
struct X {
    const int n;
    double d;
};
X* p = new X{7, 8.8};
p = new (p) X{42, 9.9}; // 注意：把placement new的返回值赋给p
int i = p->n;           // OK, i现在保证是42
auto d = p->d;           // OK, d现在保证是9.9
```

不幸的是，这个规则很少有人知道或者用到。更糟的是，在实践中，有时候并不能这么简单的使用 `placement new` 的返回值。你可能需要额外的对象，而且当前的迭代器接口也不支持它。

使用返回值可能会导致开销的一个例子是存储的位置已经有成员存在。`std::optional<>`和 `std::variant<>`就是这种情况。

这里有一个简化的例子实现了类似于 `std::optional` 的类：

```
template<typename T>
class optional
{
private:
    T payload;
public:
    optional(const T& t) : payload(t) {}

    template<typename... Args>
    void emplace(Args&&... args) {
        payload.~T();
        ::new (&payload) T(std::forward<Args>(args)...); // *
    }

    const T& operator*() const & {
        return payload; // OOPS: 返回没有重新初始化的payload
    }
};
```

如果这里 `T` 是一个带有常量或者引用成员的结构体：

```
struct X {
    const int _i;
    X(int i) : _i(i) {}
    friend std::ostream& operator<< (std::ostream& os, const X& x) {
        return os << x._i;
    }
};
```

那么下面的代码将导致未定义行为：

```
optional<X> opStr{42};
optStr.emplace(77);
std::cout << *optStr;    // 未定义行为（可能是42也可能是77）
```

这是因为输出操作之前调用了 `operator*`，后者返回 `payload`，而 `placement new`（在 `emplace()` 调用中）在 `payload` 处放置了一个新的值却没有使用返回值。

在一个类似这样的类中，你需要添加一个额外的指针成员来存储 `placement new` 的返回值，并在需要时使用它：

```
template<typename T>
class optional
{
private:
    T payload;
    T* p;    // 为了使用placement new的返回值
public:
    optional(const T& t) : payload(t) {
        p = &payload;
    }

    template<typename... Args>
    void emplace(Args&&... args) {
        payload.~T();
        p = ::new (&payload) T(std::forward<Args>(args)...);
    }

    const T& operator*() const & {
        return *p;    // 这里不要使用payload!
    }
};
```

基于分配器的容器例如 `std::vector` 等也有类似的问题。因为它们在内部通过分配器使用 `placement new`。例如，一个类似于 `vector` 的类的粗略实现如下：

```
template<typename T, typename A = std::allocator<T>>
class vector
{
public:
    typedef typename std::allocator_traits<A> ATR;
    typedef typename ATR::pointer pointer;
private:
    A _alloc;    // 当前分配器
    pointer _elems;    // 元素的数组
    size_t _size;    // 元素的数量
    size_t _capa;    // 容量
public:
    void push_back(const T& t) {
        if (_capa == _size) {
            reserve((_capa+1)*2);
        }
    }
};
```

```

    }
    ATR::construct(_alloc, _elems+_size, t);    // 调用placement new
    ++_size;
}

T& operator[] (size_t i) {
    return _elems[i];    // 对于被替换的有常量成员的元素将是未定义行为
};

```

再一次，注意 `ATR::construct()` 并没有返回调用 placement new 的返回值。因此，我们不能使用这个返回值来代替 `_elems`。

注意只有 C++11 之后这才会导致问题。在 C++11 之前，使用有常量成员的元素既不可能也没有正式的支持，因为元素必须能拷贝构造并且可赋值（尽管基于节点的容器例如链表对有常量成员的元素能完美工作）。然而，C++11 引入了移动语义之后，就可以支持带有常量成员的元素了，例如上边的类 `X`，然后也导致了上述的未定义行为。

`std::launder()` 被引入就是为了解决这些问题。然而，正如我之前所说的一样，事实上使用 `std::launder()` 完全不能解决 `vector` 的问题。

32.2 `launder()` 如何解决问题

C++ 标准委员会的核心工作组决定通过引入 `std::launder()` 来解决这个问题（见<https://wg21.link/cwg1776>）：如果你有一个因为底层内存被替换而导致访问它变成未定义行为的指针：

```

struct X {
    const int n;
    double d;
};
X* p = new X{7, 8.8};
new (p) X{42, 9.9};    // 请求把一个新的值放进p处
int i = p->n;           // 未定义行为（i可能是7也可能是42）
auto d = p->d;          // 也是未定义行为（d可能是8.8也可能是9.9）

```

任何时候你都可以调用 `std::launder()` 来确保底层内存被重新求值：

```

int i = std::launder(p)->n;    // OK, i是42
auto d = std::launder(p)->d;    // OK, d是9.9

```

注意 `launder()` 并不能解决使用 `p` 时的问题，它只是解决了使用它的那些表达式的问题：

```

int i2 = p->n;    // 仍然是未定义行为

```

任何时候你想访问替换之后的值都必须使用 `std::launder()`。

这可以在如下类似于 `optional` 的类中工作：

```

template<typename T>
class optional
{
private:

```

```

    T payload;
public:
    optional(const T& t) : payload(t) {
    }

    template<typename... Args>
    void emplace(Args&&... args) {
        payload.~T();
        ::new (&payload) T(std::forward<Args>(args)...);    // *
    }

    const T& operator*() const & {
        return *(std::launder(&payload));    // OK
    }
};

```

注意我们必须确保每一次对 `payload` 的访问都要像这里的 `operator*` 中一样经过 `std::launder()` 的“粉刷 (whitewashing)”。

32.3 为什么/什么时候 `launder()` 不生效

然而，对于像 `vector` 这种基于分配器的容器，之前的解决方案并没有效果。这是因为如果我们尝试类似这样做：

```

template<typename T, typename A = std::allocator<T>>
class vector
{
public:
    typedef typename std::allocator_traits<A> ATR;
    typedef typename ATR::pointer pointer;
private:
    A _alloc;    // 当前分配器
    pointer _elems; // 元素的数组
    size_t _size;    // 元素的数量
    size_t _capa;    // 容量
public:
    void push_back(const T& t) {
        if (_capa == _size) {
            reserve((_capa+1)*2);
        }
        ATR::construct(_alloc, _elems+_size, t);    // 调用placement new
        ++_size;
    }

    T& operator[] (size_t i) {
        return std::launder(_elems)[i]; // OOPS: 仍然是未定义行为
    }
}

```

```
...  
};
```

在 `operator[]` 中的 `launder()` 并没有作用，因为 `pointer` 可能是个智能指针（即是类类型），而对于它们 `launder()` 没有作用。²

如果尝试：

```
std::launder(this)->_elems[i];
```

也没有用，因为 `launder()` 只对生命周期已经结束的对象指针才有用。³

因此，`std::launder()` 并不能有助于解决基于分配器的容器中元素含有常量/引用成员导致未定义行为的问题。看起来一个通用的核心修复是很必要的（参见我的文章<https://wg21.link/p0532>）。

32.4 后记

`std::launder()` 作为国家机构对 C++14 的一个注释引入（见<https://wg21.link/n3903>，它最早作为核心工作组的 issue 1776 进行讨论（见<https://wg21.link/cwg1776>）。之后它由 Richard Smith 和 Hubert Tong 在<https://wg21.link/n4303>中首次提出。最终被接受的提案由 Richard Smith 发表于<https://wg21.link/p0137r1>。

²感谢 Jonathan Wakely 指出这一点。

³感谢 Richard Smith 指出这一点。

Part VI

一些通用的提示

这一部分介绍了一些有关 C++17 的通用的提示，例如对 C 语言和废弃特性的兼容性更改。

