Contents

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的指针，
> - 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;   // 为了能使用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的指针才有用。[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。