

Contents

I 基本语言特性	1
1 结构化绑定	2
1.1 细说结构化绑定	3
1.2 结构化绑定的适用场景	6
1.2.1 结构体和类	7
1.2.2 原生数组	7
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	9
1.4 后记	17
2 带初始化的 <code>if</code> 和 <code>switch</code> 语句	18
2.1 带初始化的 <code>if</code> 语句	18
2.2 带初始化的 <code>switch</code> 语句	20
2.3 后记	20
3 内联变量	21
3.1 内联变量产生的动机	21
3.2 使用内联变量	23
3.3 <code>constexpr static</code> 成员现在隐含 <code>inline</code>	24
3.4 内联变量和 <code>thread_local</code>	25
3.5 后记	27
4 聚合体扩展	28
4.1 扩展聚合体初始化的动机	28
4.2 使用聚合体扩展	29
4.3 聚合体的定义	30
4.4 向后的不兼容性	31
4.5 后记	32
5 强制省略拷贝或传递未实质化的对象	33
5.1 强制省略临时变量拷贝的动机	33
5.2 强制省略临时变量拷贝的好处	34
5.3 更明确的值类型体系	36
5.3.1 值类型体系	36
5.3.2 自从 C++17 起的值类型体系	38
5.4 未实质化的返回值传递	39
5.5 后记	40

6	lambda 表达式扩展	41
6.1	constexpr lambda	41
6.1.1	使用 constexpr lambda	43
6.2	向 lambda 传递 this 的拷贝	44
6.3	以常量引用捕获	46
6.4	后记	46
7	新属性和属性特性	47
7.1	[[nodiscard]] 属性	47
7.2	[[maybe_unused]] 属性	49
7.3	[[fallthrough]] 属性	49
7.4	通用的属性扩展	50
7.5	后记	51
8	其他语言特性	52
8.1	嵌套命名空间	52
8.2	有定义的表达式求值顺序	52
8.3	更宽松的用整型初始化枚举值的规则	55
8.4	修正 auto 类型的列表初始化	56
8.5	十六进制浮点数字面量	57
8.6	UTF-8 字符字面量	58
8.7	异常声明作为类型的一部分	59
8.8	单参数 static_assert	62
8.9	预处理条件 __has_include	62
8.10	后记	63
II	模板特性	65
9	类模板参数推导	66
9.1	使用类模板参数推导	66
9.1.1	默认以拷贝方式推导	68
9.1.2	推导 lambda 的类型	68
9.1.3	没有类模板部分参数推导	70
9.1.4	使用类模板参数推导代替快捷函数	71
9.2	推导指引	73
9.2.1	使用推导指引强制类型退化	73
9.2.2	非模板推导指引	74
9.2.3	推导指引与构造函数冲突	75
9.2.4	显式推导指引	75
9.2.5	聚合体的推导指引	76

9.2.6 标准推导指引	77
9.3 后记	82
10 编译期 <code>if</code> 语句	83
10.1 编译期 <code>if</code> 语句的动机	83
10.2 使用编译期 <code>if</code> 语句	85
10.3 编译期 <code>if</code> 的注意事项	86
10.3.1 其他编译期 <code>if</code> 的示例	88
10.4 带初始化的编译期 <code>if</code> 语句	91
10.5 在模板之外使用编译期 <code>if</code>	91
10.6 后记	92
11 折叠表达式	93
11.1 折叠表达式的动机	93
11.2 使用折叠表达式	94
11.2.1 处理空参数包	95
11.2.2 支持的运算符	98
11.2.3 使用折叠表达式处理类型	102
11.3 后记	103
12 处理字符串字面量模板参数	104
12.1 在模板中使用字符串	104
12.2 后记	105
13 占位符类型作为模板参数（例如 <code>auto</code>）	106
13.1 使用 <code>auto</code> 模板参数	106
13.1.1 字符和字符串模板参数	107
13.1.2 定义元编程常量	108
13.2 使用 <code>auto</code> 作为变量模板的参数	109
13.3 使用 <code>decltype(auto)</code> 模板参数	110
13.4 后记	111
14 扩展的 <code>using</code> 声明	112
14.1 使用变长的 <code>using</code> 声明	112
14.2 使用变长 <code>using</code> 声明继承构造函数	113
14.3 后记	114
III 新的标准库组件	115

15	<code>std::variant<></code>	115
15.1	<code>std::variant<></code> 的动机	115
15.2	使用 <code>std::variant<></code>	115
15.3	<code>std::variant<></code> 类型和操作	115
15.3.1	<code>std::variant<></code> 类型	115
15.3.2	<code>std::variant<></code> 操作	115
15.3.3	访问器	115
16	<code>std::byte</code>	116
17	文件系统库	117
17.1	基本的例子	117
17.1.1	打印文件系统路径类的属性	117
17.1.2	用 <code>switch</code> 语句处理不同的文件系统类型	117
17.1.3	创建不同类型的文件	117
17.1.4	使用并行算法处理文件系统	117
17.2	原则和术语	117
17.2.1	通用的可移植性路径分隔符	117
17.2.2	命名空间	117
17.2.3	文件系统路径	117
IV	已有标准库的拓展和修改	118
18	类型 <code>trait</code> 扩展	119
18.1	类型 <code>trait</code> 后缀 <code>_v</code>	119
18.2	新的类型 <code>trait</code>	119
19	并行 STL 算法	120
20	子串和子序列搜索器	121
20.1	使用子串搜索器	121
20.1.1	通过 <code>search()</code> 使用搜索器	121
20.1.2	直接使用搜索器	121
21	其他工具函数和算法	122
21.1	<code>size()</code> , <code>empty()</code> , <code>data()</code>	122
21.1.1	泛型 <code>size()</code> 函数	122
21.1.2	泛型 <code>empty()</code> 函数	122
21.1.3	泛型 <code>data()</code> 函数	122
21.2	<code>as_const()</code>	122
21.2.1	以常量引用捕获	122

22 标准库的其他微小的特性和修改	123
22.1 <code>std::uncaught_exceptions()</code>	123
22.2 共享指针改进	123
22.2.1 对原始 C 数组的共享指针的特殊处理	123
22.2.2 共享指针的 <code>reinterpret_pointer_cast</code>	123
22.2.3 共享指针的 <code>weak_type</code>	123
22.2.4 共享指针的 <code>weak_from_this</code>	123
22.3 数学扩展	123
22.3.1 最大公约数和最小公倍数	123
22.3.2 <code>std::hypot()</code> 的三参数重载	123
22.3.3 数学领域的特殊函数	123
22.3.4 <code>chrono</code> 扩展	123
22.3.5 <code>constexpr</code> 扩展和修正	123
22.3.6 <code>noexcept</code> 扩展和修正	123
22.3.7 后记	123
 V 专家的工具	 124
23 使用 <code>new</code> 和 <code>delete</code> 管理超对齐数据	125
23.1 使用带有对齐的 <code>new</code> 运算符	125
23.2 实现内存对齐分配的 <code>new()</code> 运算符	125
23.2.1 在 C++17 之前实现对齐的内存分配	125
23.2.2 实现类型特化的 <code>new()</code> 运算符	125
23.3 实现全局的 <code>new()</code> 运算符	125
23.4 追踪所有 <code>::new</code> 调用	125
23.5 后记	125
 24 编写泛型代码的改进	 126
24.1 <code>std::invoke<>()</code>	126
24.2 <code>std::bool_constant<></code>	126
 VI 一些通用的提示	 127

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。

14 扩展的 using 声明

using 声明扩展之后可以支持逗号分隔的名称，也可以支持参数包。

例如，你现在可以这么写：

```
class Base {
public:
    void a();
    void b();
    void c();
};

class Derived : private Base {
public:
    using Base::a, Base::b, Base::c;
};
```

在 C++17 之前，你需要使用 3 个 using 声明分别进行声明。

14.1 使用变长的 using 声明

逗号分隔的 using 声明允许你用泛型代码从可变数量的所有基类中派生同一种运算。

这项技术的一个很酷的应用是创建一个重载的 lambda 的集合。通过如下定义：

tmpl/overload.hpp

```
// 继承所有基类里的函数调用运算符
template<typename... Ts>
struct overload : Ts...
{
    using Ts::operator()...;
};

// 基类的类型从传入的参数中推导
template<typename... Ts>
overload(Ts...) -> overload<Ts...>;
```

你可以像下面这样重载两个 lambda：

```
auto twice = overload {
    [](std::string& s) { s += s; },
    [](auto& v) { v *= 2; }
};
```

这里，我们创建了一个 **overload** 类型的对象，并且提供了推导指引 来根据 lambda 的类型推导出 **overload** 的基类的类型。并且我们使用了聚合体初始化 来调用每个 lambda 生成的闭包类型的拷贝构造函数来初始化基类子对象。

上例中的using声明使得overload类型可以同时访问所有子类中的函数调用运算符。如果没有这个using声明，两个基类会产生同一个成员函数operator()的重载，这将会导致歧义。⁴

最后，如果你传递一个字符串参数将会调用第一个重载，其他类型（操作符*=有效的类型）将会调用第二个重载：

```
int i = 42;
twice(i);
std::cout << "i: " << i << '\n';    // 打印出: 84
std::string s = "hi";
twice(s);
std::cout << "s: " << s << '\n';    // 打印出: hihi
```

这项技术的另一个应用是std::variant访问器。

14.2 使用变长using声明继承构造函数

除了逐个声明继承构造函数之外，现在还支持如下的方式：你可以声明一个可变参数类模板Multi，让它继承每一个参数类型的基类：

tmpl/using2.hpp

```
template<typename T>
class Base {
    T value{};
public:
    Base() {
        ...
    }
    Base(T v) : value{v} {
        ...
    }
    ...
};

template<typename... Types>
class Multi : private Base<Types>...
{
public:
    // 继承所有构造函数:
    using Base<Types>::Base...;
    ...
};
```

有了所有基类构造函数的using声明，你可以继承每个类型对应的构造函数。

现在，当使用不同类型声明Multi<>时：

```
using MultiISB = Multi<int, std::string, bool>;
```

⁴clang 和 Visual C++ 都不会把不同基类中不同类型的同名函数当作歧义处理，所以这个例子中其实不需要using。然而，这段代码如果没有using声明的将不具备可移植性。

你可以使用每一个相应的构造函数来声明对象：

```
MultiISB m1 = 42;
MultiISB m2 = std::string("hello");
MultiISB m3 = true;
```

根据新的语言规则，每一个初始化会调用匹配基类的相应构造函数和所有其他基类的默认构造函数。因此：

```
MultiISB m2 = std::string("hello");
```

会调用 `Base<int>` 的默认构造函数，`Base<std::string>` 的字符串构造函数，`Base<bool>` 的默认构造函数。

原则上讲，你也可以通过如下声明来支持 `Multi<>` 进行赋值操作：

```
template<typename... Types>
class Multi : private Base<Types>...
{
    ...
    // 派生所有赋值运算符
    using Base<Types>::operator=...;
};
```

14.3 后记

逗号分隔的 `using` 声明列表由 Robert Haberlach 在 <https://wg21.link/p0195r0> 中首次提出。最终被接受的提案由 Robert Haberlach 和 Richard Smith 发表于 <https://wg21.link/p0195r2>。

关于继承构造函数有一些核心的问题。最终修复这些问题的提案由 Richard Smith 发表于 <https://wg21.link/n4429>。

还有一个由 Vicente J. Botet Escriba 提出的提案。除了 `lambda` 之外，它还支持重载普通函数、成员函数来实现泛型的 `overload` 函数。然而，这个提议并没有进入 C++17 标准。详情请见 <https://wg21.link/p0051r1>。

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）。

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。