C++17语言特性

结构体绑定

- 绑定了一个匿名的结构体，其中的u,v只是匿名结构体的成员变量
- 对绑定的修饰符，实际上是对匿名结构体的修饰

```
auto [u, v] = getStruct();
// 类似于下面的定义
auto e = getStruct(); aliasname u = e.i; aliasname v = e.s;

const auto& [u, v] = ms; // 引 用， 因 此u/v指 向ms.i/ms.s
alignas(16) auto [u, v] = ms; // 对 齐 匿 名 实 体， 而 不 是v，u按照16字节对齐
```

类中的结构体绑定

- 非静态数据成员必须在同一个类中定义（也就是 说，这些成员要么是全部直接来自于最终的类，要么是全部来自同一个父类）：

```
struct B {
    int a = 1;
    int b = 2; };
struct D1 : B { };
auto [x, y] = D1{}; // OK

struct D2 : B { int c = 3; };
auto [i, j, k] = D2{}; // 编 译 期ERROR
```

- 联合还不支持使用结构化绑定
- 我们不能用临时对象 (prvalue)初始化一个非 const引用：

```
auto& [a, b, c, d] = getArray(); // ERROR
```

内联变量

- 在头文件中以 inline的方式定义全局变量/对象

- 只要一个编译单元内没有重复的定义即可。此例中的定义即使被多个编译单元使用，也会指向同一个对象。

```cpp
class MyClass {
    inline static std::string msg{"OK"};    // OK（自C++17 起）
};
inline MyClass myGlobalObj; // 即 使 被 多 个CPP文 件 包 含 也OK
```

聚合体扩展

- 聚合体初始化时可以用一个子聚合体初始化来初始化类中来自基类的成员。

```cpp
struct Data{
std::string name;
double value;
};

struct MoreData:Data{
bool done;
};

MoreData x{{"test1", 6.778}, false}; // 自 从C++17起OK
MoreData y{"test", 6.778, false};
```

- C++17引入了一个新的类型特征 is_aggregate<> 来测试一个类型是否是聚合体
- 在 C++17之前，Derived不是聚合体。因此 Derived d1{}; 会调用 **Derived隐式定义的默认构造函数**，这个构造函数会调用基类 Base的构造函数。尽管基类的默认构造 函数是 private的，但在派生类的构造函数里调用它也是有效的，因为派生类被声明为友元类。 自从 C++17起，例子中的 Derived是一个聚合体，所以它没有隐式的默认构造函数（构造函数没有使 用 using声明继承）;基类有一个private的构造函数（见上一节）所以不能使用花括号来初始化

```cpp
struct Derived;

struct Base {
    friend struct Derived;
private:
    Base(){}
};

struct Derived:Base{};

```

```
int main() {
    Derived d1{}; // C++17后ERROR
    Derived d2; // OK(但是可能没有初始化)
}
```

- 假设**基类的构造函数被声明为私有构造函数或者派生类是从基类虚继承的**，那么就不能在派生类中声明继承构造函数。

强制省略拷贝或传递未实质化的对象

具名返回值优化(NRVO)

- RVO：如果函数**返回一个无名的临时对象**，该对象将被编译器移动或拷贝到目标中，在此时，编译器可以优化代码以减少对象的构造，即省略拷贝或移动。
- NRVO：命名返回值优化（指的是**省略了命名对象的拷贝这一事实**）；如果函数按值返回一个类的类型，并且返回语句的表达式是一个具有自动存储期限的非易失性对象的名称（即不是一个函数参数），那么**可以省略优化编译器所要进行的拷贝/移动**。如果是这样的话，返回值就会**直接构建在函数的返回值所要移动或拷贝的存储器**（即被拷贝省略优化掉的拷贝或移动的存储器）中。

```
// 原始代码
Data process(int i) {
    Data data;
    data.mem_var = i;
    return data;
}

// 编译器未做优化
Data process(Data &_hiddenArg, int i){
    Data data;
    data.Data::Data();              // constructor for data
    data.mem_var = i;
    _hiddenArg.Data::Data(data);    // copy constructor for data
    return;
    data.Data::~Data();             // destructor for data
}

// 编译器使用NRVO优化
Data process(Data &_hiddenArg, int i){
    _hiddenArg.Data::Data();
    _hiddenArg.mem_var = i;
    Return
}
```

强制省略临时变量拷贝的作用

- 自从 C++17起用临时变量初始化对象时省略拷贝变成了强制性的
- 可以返回不允许拷贝或移动的对象。
- 对于移动构造函数被显式删除的类，现在也可以返回临时对象来初始化新的对象：

```
class CopyOnly{
public:
    CopyOnly(){}
    CopyOnly(int){}
    CopyOnly(const CopyOnly&) = default;
    CopyOnly(CopyOnly&&) = delete;
};

CopyOnly ret() {return CopyOnly{};}

CopyOnly x = 42;
// 在 C++17之前 x的初始化是无效的，因为拷贝初始化（使用 =初始化）需要把 42转换为一个临时对象，然后要 用这个临时对象初始化 x原则上需要移动构造函数，尽管它可能不会被调用。
```

lambda表达式扩展

- 自从 C++17起，lambda表达式会尽可能的隐式声明 constexpr。
- 可以向lambda传递this的拷贝；lambda里捕获了 *this，所以传递进 lambda的是一份拷贝。因此，即使在 d被销毁之后使用捕获的对象也没有 问题

```
class Data {
private:
    std::string name;
public:
    Data(const std::string& s) : name(s) {
    }
    auto startThreadWithCopyOfThis() const {
        // 开启并返回新线程，新线程将在3秒后使用this：
        using namespace std::literals;
        std::thread t([*this] {
            std::this_thread::sleep_for(3s);
            std::cout << name << '\n';
        });
        return t;
    }
};

int main()
{
    std::thread t;
    {
        Data d{"c1"};
        t = d.startThreadWithCopyOfThis();
    }   // d不再有效
    t.join();
}
```

新属性和属性特性

[[nodiscard]]

- 鼓励编译器在**某个函数的返回值未被使用**时给出警告
- 注意如果成员函数被覆盖或者隐藏时基类中标记的属性不会被继承

[[maybe_unused]]

- 可以避免编译器在某个变量未被使用时发出警告。新的属性可以应用于类的 声明、使用 typedef或者 using定义的类型、一个变量、一个非静态数据成员、一个函数、一个枚举类型、一 个枚举值等场景。

[[fallthrough]]

- 以避免编译器在 switch语句中某一个标签缺少 break语句时发出警告

```
switch (place) {
    case 1:
        std::cout << "very ";
        [[fallthrough]];
    case 2:
        std::cout << "well\n";
        break;
    default:
        std::cout << "OK\n";
        break;
}
```

属性扩展

1. 属性现在可以用来标记命名空间。例如，你可以像下面这样弃用一个命名空间：

```
namespace [[deprecated]] DraftAPI {
    ...
}
```

   这也可以应用于内联的和匿名的命名空间。

2. 属性现在可以标记枚举子（枚举类型的值）。例如你可以像下面这样引入一个新的枚举值作为某个已有枚举值（并且现在已经被废弃）的替代：

```
enum class City { Berlin = 0,
                  NewYork = 1,
                  Mumbai = 2,
                  Bombay [[deprecated]] = Mumbai,
                  ... };
```