C++17模板特性

类模板参数推导

- 自从 C++17起必须指明类模板参数的限制被放宽了。通过使用类模板参数推导 (class template argument deduction)(CTAD)，只要编译器能根据初始值推导出所有模板参数，那么就可以不指明参数。
- 类模板参数推导过程中会首先尝试以拷贝的方式初始化
- 注意，不像函数模板，类模板不能只指明一部分模板参数，然后指望编译器去推导剩余的部分参数。甚至使 用 <>指明空模板参数列表也是不允许的。

推导指引强制类型退化

- 重载推导规则的一个非常重要的用途就是确保模板参数 T在推导时发生退化
- 传递一个字符串字面量"hello"，传递的类型将是 const char(&)[6]，因此T被推导为 char[6]

```
template<typename T>
struct C {
    C(const T&){}
};

C x{"hello"}; // T被 推 导 为char[6]

// 推导指引
template<typename T> C(T) -> C<T>;

// 推导指引以值传递参数因此"hello"的类型 T会退化为 const char*。
```

- 推导指引并不一定是模板，也不一定应用于构造函数。例如，为下面的结构体添加的推导指引也是有效的

```
template<typename T>
struct S {
    T val;
};

S(const char*) -> S<std::string>;   // 把S<字符串字面量>映射为S<std::string>
```

- 推导指引会和类的构造函数产生竞争。类模板参数推导时会根据重载情况选择最佳匹配的构造函数/推导指 引。如果一个构造函数和一个推导指引匹配优先级相同，那么将会优先使用推导指引
- 如果用拷贝初始化（使用 =）将会忽略这一条推导指引。这意味着下面的初始化是无效的：

```
template<typename T>
struct S {
    T val;
};

explicit S(const char*) -> S<std::string>;
```

如果用拷贝初始化（使用 =）将会忽略这一条推导指引。这意味着下面的初

```
S s1 = {"hello"};       // ERROR（推导指引被忽略，因此是无效的）
```

```
template<typename T>
struct Ptr
{
    Ptr(T) { std::cout << "Ptr(T)\n"; }
    template<typename U>
    Ptr(U) { std::cout << "Ptr(U)\n"; }
};

template<typename T>
explicit Ptr(T) -> Ptr<T*>;
```

上面的代码会产生如下结果：

```
Ptr p1{42};     // 根据推导指引推导出Ptr<int*>
Ptr p2 = 42;    // 根据构造函数推导出Ptr<int>
```

编译期if语句

- 编译器可以计算编译期的条件表达式来在编译期决定使用一个if语句 的 then的部分还是 else的部分。其余部分的代码将会被丢弃，这意味着它们甚至不会被生成。
- 事实上，模板编译的第一个阶段（定义期间）将会检查语法和所有与模板无关的名称是否 有效。**所有的 static_asserts也必须有效，即使所在的分支没有被编译。**

```
#include<string>
template  std::string asString(T x) {
    if constexpr(std::is_same_v) { return x;} // 如果T不能自动转换为string该语句将无效
    else if constexpr(std::is_arithmetic_v ) { return std::to_string(x); }// 如果T不是数字类型该语句将无效
    else { return std::string(x);} // 如果不能转换为string该语句将无效
}
```

- 完美返回泛型值

```
#include <functional>
#include <type_traits>

template<typename Callable, typename... Args>
decltype(auto) call(Callable op, Args&&... args) {
    if constexpr (std::is_void_v<std::invoke_result_t<Callbale, Args...>>) {
        op(std::forward<Args>(args)...);
        return;
    }
    else {
        decltype(auto) ret{op(std::forward<Args>(args)...)};
        return ret;
    }
}
```

折叠表达式

- 一元左折叠 (⋯ op args) 将会展开为：((arg1 op arg2) op arg3) op ⋯
- 一元右折叠 (args op ⋯) 将会展开为：arg1 op (arg2 op ⋯ (argN-1 op argN)) 括号是必须的，然而，括号和省略号 (⋯) 之间并不需要用空格分隔。
- 任何情况下，从左向右求值都是符合直觉的。因此，更推荐使用左折叠的语法：

- **二元左折叠**

  (*value* **op** ... **op** *args*)

  将会展开为：*(((value* **op** *arg1)* **op** *arg2)* **op** *arg3)* **op** ...

- **二元右折叠**

  (*args* **op** ... **op** *value*)

  将会展开为：*arg1* **op** *(arg2* **op** ... *(argN* **op** *value))*

省略号两侧的 *op* 必须相同。

例如，下面的定义在进行加法时允许传递一个空参数包：

```
template<typename... T>
auto foldSum (T... s) {
    return (0 + ... + s);    // 即使sizeof...(s)==0也能工作
}
```

```
template<typename First, typename... Args>
void print (const First& firstarg, const Args&... args) {
    std::cout << firstarg;
    (std::cout << ... << [] (const auto& arg) -> decltype(auto) {
                            std::cout << ' ';
                            return arg;
                        }(args)) << '\n';
}
```

不过，一个更简单的实现 `print()` 的方法是使用一个 lambda 输出空格和参数，然后在一元折叠表达式里使用它：

```
template<typename First, typename... Args>
void print(First first, const Args&... args) {
    std::cout << first;
    auto outWithSpace = [] (const auto& arg) {
                            std::cout << ' ' << arg;
                        };
    (... , outWithSpace(args));
    std::cout << '\n';
}
```

如果 foo() 函数返回的类型重载了逗号运算符，那么代码行为可能会改变。为了保证这种情况下代码依然安全，你需要把返回值转换为 void：

```
template<typename... Types>
void callFoo(const Types&&... args)
{
    ...
    (... , (void)foo(std::forward<Types>(args))); // 调用foo(arg1)，foo(arg2)，...
}
```