lambda

语法

> [俘获] <模板形参>(可选) (形参) 说明符(可选) 异常说明 attr -> ret requires(可选){ 函数体 }
>
> --- C++20 完整声明
>
> [俘获] (形参) -> ret { 函数体 }
>
> [俘获] (形参) { 函数体 }
>
> [俘获] { 函数体 }

一般情况下使用`[capture list] (parameter list) -> return type { function body }`即可。

传递参数

```
sort(words.begin(), words.end(), [](string &a, string &b) { return a < b; });
```

捕获列表

```
// 获取一个迭代器，指向第一个满足大小>=sz的元素
auto wc = find_if(words.begin(), words.end(), [sz](string &a) { return a.size() >= sz; });
```

值捕获

```
void fcn1()
{
    size_t v1 = 42;
    auto f = [v1]() { return v1; };
    v1 = 0;
    auto v0 = f();      // v0为42
}
```

值捕获是在`lambda`中传递一份拷贝，对其修改不会更改外部的变量

引用捕获

```
void fcn2()
{
    size_t v1 = 42;
    auto f = [&v1]() { return v1; };
    v1 = 0;
    auto v0 = f();      // v0为0
}
```

引用捕获是向`lambda`传递一份引用，对其修改可以影响外部的变量

建议

尽量保持`lambda`的变量捕获的简单化

隐式捕获

```
// sz为隐式捕获，值传递
auto wc = find_if(words.begin(), words.end(), [=](string &a) { return a.size() >= sz; });

// sz为隐式捕获，引用传递
auto wc = find_if(words.begin(), words.end(), [&](string &a) { return a.size() >= sz; });
```

混合捕获

混合使用**隐式**和**显式**捕获

```
// os为隐式捕获，引用传递；c是显式捕获，值传递
for_each(words.begin(), words.end(), [&, c](string &s) { os << s << c; });

// os为显式捕获，引用传递；c是隐式捕获，值传递
for_each(words.begin(), words.end(), [=, &os](string &s) { os << s << c; });
```

当混合使用**隐式**捕获和**显式**捕获时，捕获列表中的第一个元素必须是一个`&`或`=`。此符号指定默认的捕获方式为引用或值

当混合使用**隐式**捕获和**显式**捕获时，**显式**捕获的变量必须使用和**隐式**捕获不同的方式，即

隐式捕获	显式捕获
&	=
=	&

表 10.1：lambda 捕获列表	
[]	空捕获列表。lambda 不能使用所在函数中的变量。一个 lambda 只有捕获变量后才能使用它们
[*names*]	*names* 是一个逗号分隔的名字列表，这些名字都是 lambda 所在函数的局部变量。默认情况下，捕获列表中的变量都被拷贝。名字前如果使用了 &，则采用引用捕获方式
[&]	隐式捕获列表，采用引用捕获方式。lambda 体中所使用的来自所在函数的实体都采用引用方式使用
[=]	隐式捕获列表，采用值捕获方式。lambda 体将拷贝所使用的来自所在函数的实体的值
[&, *identifier_list*]	*identifier_list* 是一个逗号分隔的列表，包含 0 个或多个来自所在函数的变量。这些变量采用值捕获方式，而任何隐式捕获的变量都采用引用方式捕获。*identifier_list* 中的名字前面不能使用 &
[=, *identifier_list*]	*identifier_list* 中的变量都采用引用方式捕获，而任何隐式捕获的变量都采用值方式捕获。*identifier_list* 中的名字不能包括 this，且这些名字之前必须使用 &

可变lambda

使用`mutable`修饰可以在`lambda`中修改变量的值从而影响到外部的变量

```
void fcn3()
{
    size_t v1 = 42;
    auto f = [v1]() mutable { return v1; };
    v1 = 0;
    auto v0 = f();      // v0为0
}
```

指定lambda的返回类型

如果一个`lambda`体中包含`return`之外任何的任何语句，则编译器会假定此`lambda`返回`void`

```
// 正确
transform(vi.begin(), vi.end(), vi.begin(), [](int i) { return i < 0 ? -i : i; });

// 错误：不能推到lambda的返回类型
transform(vi.begin(), vi.end(), vi.begin(),
          [](int i) { if(i < 0) return -i; else return i; });

// 更正
transform(vi.begin(), vi.end(), vi.begin(),
          [](int i) -> int { if(i < 0) return -i; else return i; });
```

友元

友元允许类外访问类的私有成员

友元全局函数

```
class Testclass {
    ...
    friend void foo();
private:
    int _a;
    ...
};

void foo()
{
    Testclass t;
    t._a = 100;
}
```

- 友元函数不是类的成员，不受权限影响
- 友元函数可以访问类的所有成员
- 定义在类内部时，需要带类相关的参数，且是隐式内联的

友元类

```
class Friendclass;
class Testclass {
    ...
    firend class Friendclass;   // Friendclass类作为Testclass类的友元类
    ...
private:
    int _a;
};
```

```
class Friendclass {
    ...
    void foo(Testclass &t)
    {
        t._a = 1010;
    }
    ...
}
```

如果一个类指定了其的友元类，则友元类的成员函数可以访问此类的所有成员

友元成员函数

```
class Testclass;
class Friendclass {
    ...
    void foo(Testclass &t);
    ...
}

class Testclass {
    ...
    firend void Friendclass::foo(Testclass &t);
    ...
private:
    int _a;
};

void Friendclass::foo(Testclass &t)
{
    t._a = 1010;
}
```

将一个类的成员函数作为另一个类的友元，此成员函数可以访问另一个类的所有成员

总结

缺点

破坏了封装性

应用

- 常见于框架设计
- 运算符重载

运算符重载

C++的运算符

名称	运算符
算数	+、-、/、%、++、-- 等
位操作	&、|、~ 等
逻辑	&&、||、! 等
比较	==、<、>、<=、>=、!= 等
赋值	=、+=、-=、*=、/=、%= 等
其他	[]、()、->、,、new、delete、new[]、delete[]、->*、sizeof 等

其中不能被重载的为：

- `?:` 三目运算符
- `.` 成员访问运算符
- `::` 域运算符
- `sizeof` 运算符
- `.*` 成员指针访问运算符运算符

注意事项

- 运算符重载尽量不要改变运算符原来的语义
- 运算符重载不改变运算符的优先级和操作数的个数
- 本质上是调用重载函数

operator =

- 如果没有实现 `=` 的运算符重载，则编译器会提供一个默认的 `=` 运算符重载,其功能类似于 `memcpy`
- 一般情况如果自己不提供 `=` 的运算符重载,则使用 `=delete` 禁掉

operator ++ 与 opeator --

- 前 `++` 返回对象的引用
- 后 `++` 加一个参数类型，但是不会使用，所以没有形参
- 后 `++` 返回的是原来的值

operator new 与 operator delete

new

- 内部不能调用 `new`，会递归调用，导致栈溢出

- 编译器会自动进行指针的数据类型转换
- 对于对象，编译器会自动调用构造函数

delete

- 编译器自动调用析构

new 和 delete 运算符重载为成员函数

- 默认的重载函数是静态成员
- 重载函数中没有`this`指针

operator new[] 与 operator delete[]

new[]

- 如果`new`的是对象数组，则传入的大小会多4个字节，用于存放数组元素的个数
- 编译器会自动转换指针到对象的位置

delete[]

- 编译器会自动转换指针到对内存的首地址

operator * 与 operator ->

- 通常用于智能指针
- 智能指针是为了达到只申请内存不管释放的目的,申请后内存能够自动释放.