链滴

《C4P》读书笔记回顾（一）：重载

作者：StephenZhang
原文链接：https://ld246.com/article/1601201127645
来源网站：链滴

Preface

每次打算动笔写一点关于C++的文章的时候，总是想着要把整个系列文章写得多么高大上；但事实上高大上的描述不能很容易地将要表达的内容传递给受众。

该系列是对笔者两年前阅读《C++ Primer Plus（6th Edition）》（C4P）和《C++ Primer（5th Edition）》（C3P）的回忆式总结，因此更多的将会聚焦在语言的层面，去表达我自身对这些语法或者法糖的理解。

原定计划是在2020 Q3-Q4更新一部分《CLRS》的读书笔记，这个计划会继续执行；C++相关的内是额外的。因为笔者由于工作原因暂时不太经常接触到C++的编写，因此这些文章更多的是帮助自己至于遗忘。

C++虽然学起来很难，但是当你深入后，你会觉得，这里是别有一番洞天的，希望你能从笔者笔下感到C++的乐趣！

函数的重载

C++对于函数重载的定义是：在**相同作用域**中，函数名相同、参数列表不同的函数互相称为重载函数。

注意，定义中强调了两个重要的条件：

- 相同作用域
- 相同的函数名，不同的参数列表

其中，“不同的参数列表”可以是不同的参数类型，也可以是不同的参数顺序，或二者兼而有之。

那么，这样的两个函数是无法构成重载关系的：

```
int func(string name, uint8_t uid);
```

```
void func(string name, uint8_t uid);    // 不构成重载
```

只有返回值不同的两个函数（或多个）不构成重载关系，但是满足重载关系的函数可以拥有不同的返值，例如：

```
long long calc(char* name, uint8_t uid);
bool calc(uint8_t uid, string name);        // 构成重载
```

为什么会这样呢？我们随便给它们一个定义，然后看看编译出的.o文件中是什么样的。

使用GCC编译后，我们可以选择使用nm命令查看，应该可以查看到类似的两行：

```
0000000000000010 T _Z4calchNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
0000000000000000 T _Z4calcPch
```

注意在相同的前缀_Z4后面，是我们的函数名，而在函数名后面，则是参数列表的类型——编译器由文件中的符号生成编译器可以识别的符号的过程称为Symbol mangling。这一步中，不同形参列表函数生成了不同的符号，从而可以实现函数重载，而在Mangling的过程中，可以看到并没有返回值型的参与。BTW，不同编译器Mangling的规则时不同的，这也就是为什么不同编译器产生的库文件能混用的原因。

运算符的重载

什么是运算符重载?

运算符的重载是OOP中一个重要的组成部分；同时，如果在不使用模板（Template）的情况下，运符重载可以部分实现泛型编程（伪）。

让我们从一个示例开始：

Miss Mary即将和Mr. Stark举行婚礼了，他们决定把各自的婚前财产合并，成为小家庭的启动资金。设二者都是People类[1]的一个实例。

```
struct People
{
    string name;        // 姓名
    int32_t finance;    // 资金

    // Constructor
    People(string n, int32_t f): name(n), finance(f) {}
};

People Mary = People("mary", 10000);
People Stark = People("stark, 10000);    // 二人各有10000$的婚前资产
```

婚礼举行后，他们组成了一个家庭stark_and_mary，是类Family的实例。

```
struct Family
{
    vector<string> members;        // 家庭成员列表
    int32_t all_finance;        // 共同财产
};
Family stark_and_mary;
```

现在他们的小家庭中空空如也，怎么把数据转移到stark_and_mary呢？如果采用面向过程的方法，大概会写出如下代码：

```
stark_and_mary.members.push_back(Stark.name);
stark_and_mary.members.push_back(Mary.name);
stark_and_mary.all_finance = Stark.finance + Mary.finance;
```

如果我们把Family定义为People的和，婚姻可以视为加法，这样会发生什么呢？

```
Family operator+(const People& husband, const People& wife)
{
    Family f;
    f.members.push_back(husband.name);
    f.members.push_back(wife.name);
    f.all_finance = husband.finance + wife.finance;

    return f;
}
stark_and_mary = Stark + Mary;
```

观察上面的代码，是不是感觉好一点呢？这里的代码将+视为一个普通函数进行了重载，从而将每一新人结婚的过程抽象成了通用的操作过程。

运算符重载的过程和函数重载十分类似，只不过把函数名换成了关键字operator和相应的运算符；和数重载不同的是，运算符重载不能改变运算符在用于基本类型时的语义，也不能创造新的运算符，更能带有默认值参数。

为什么要使用运算符重载?

其实不言自明，使用重载可以提高代码的抽象程度，避免过多的代码冗余等等……

运算符重载的几种情况

运算符的参数以及返回值

一般运算符在进行重载时，并不限制其参数类型和返回值类型，但是对于参数的个数和顺序还是有要的。一般二元运算符接受两个参数，一元运算符接受一个参数。以+为例，它左右两边的操作数都是的参数，带有一个返回值；或者[]，带有一个参数，一个返回值。最为特殊的是()，它的参数类型和个没有限制，是否带有返回值也不加限制，事实上，它的存在，是仿函数（functor）[2]的基础。

如果二元运算符以类成员函数的形式重载，那么可以省去代表其左侧操作数的参数，而以隐式的this针代替。Mr. Stark在婚前不小心丢了手机，于是他花了299$购买了iPhone 11，我们为类People添一个重载的减法运算符：

```
struct People
{
    string name;       // 姓名
    int32_t finance;   // 资金

    // Constructor
    People(string n, int32_t f): name(n), finance(f) {}

    // operator reloading
```

```
    People& operator-(int32_t money)
    {
        this.finance -= money;
        return *this;
    }
};
Stark = Stark - 299;    // Mr.Stark购买了手机
Stark = Stark.operator-(299); // 与上一条语句等价
```

为清晰起见，我在上面的代码中明确写出了this指针，实际上可以省去不写。上面两种调用形式是等的，只要记住，运算符重载和函数重载具有很高的相似性，运算符可以看作函数。

但是这样又会引发一个新的问题，即如果People对象作为减数，出现在-的右侧怎么办呢？记住，运符可以看做函数！若是直接调用上面的代码，会引发编译器报错，参数类型无法匹配（因为顺序和类都不一样嘛）。此时我们只能重写一份-的重载，使之适应新的运算要求。

不可以被重载的运算符

只有几种，它们是：sizeof、::（作用域运算符）、.（取成员运算符）、?:（三目运算符）、.*（成员针运算符）[3]。

只允许在类内被重载的运算符

它们分别是：[]、()（函数索引运算符）、->、=。但是，一元运算符如++和--，以及所有带=的复合算符推荐以成员函数的形式重载；其他二元运算符则建议使用友元函数[4]的形式重载。

1. 在C++中，struct和class只有默认访问权限的区别，为方便起见，不再很细致的区分“结构体”或“类”的称呼
2. 仿函数，即重载了()的类，这样的类可以被当做函数调用，在行为上也和普通函数类似
3. .*似乎很少见，如果要访问的对象成员是一个指针的话，可能*(obj.ptr)是更通用的形式
4. 关于友元，还是放在日后完成吧，本篇已够长了……