上一篇文章中，我从C语言开始说起，聊了聊面向过程式的编程范式，相信从代码的角度你对这类型的语言已经有了一些理解。作为一门高级语言，C语言绝对是编程语言历史发展中的一个重要里程碑，但随着认知的升级，面向过程的C语言已经无法满足更高层次的编程的需要。于是，C++出现了。

C++语言

1980年，AT&T贝尔实验室的**Bjarne Stroustrup**创建的C++语言横空出世，它既可以全面兼容C语言，又巧妙揉和了一些面向对象的编程理念。现在来看，不得不佩服Stroustrup的魄力。在这里，我也向你推荐一本书，书名是《C++语言的设计和演化》。

这本书系统介绍了C++诞生的背景以及初衷，书的作者就是Stroustrup本人，所以你可以非常详细地从语言创建者的角度了解他的设计思路和创新之旅。当然，就是在今天，C++这门语言也还有很多争议，这里我不细说。如果你感兴趣的话，可以看看我几年前在酷壳上发表的文章《C++的坑真的多吗？》。

从语言角度来说，实际上早期C++的许多工作是对C的强化和净化，并把完全兼容C作为强制性要求（这也是C++复杂晦涩的原因，这点Java就干得比C++彻底得多）。在C89、C99这两个C语言的标准中，有许多改进正是从C++中所引进的。

可见，C++对C语言的贡献非常之大。是的，C++很大程度就是来解决C语言中的各种问题和各种不方便。比如：

- 用引用来解决指针的问题。
- 用namespace来解决名字空间冲突的问题。
- 通过try-catch来解决检查返回值编程的问题。
- 用class来解决对象的创建、复制、销毁的问题，从而可以达到在结构体嵌套时可以深度复制的内存安全问题。
- 通过重载操作符来达到操作上的泛型。（比如，消除上一篇文章中提到的比较函数cmpFn，再比如用>>操作符消除printf()的数据类型不够泛型的问题。）
- 通过模板template和虚函数的多态以及运行时识别来达到更高层次的泛型和多态。
- 用RAII、智能指针的方式，解决了C语言中因为需要释放资源而出现的那些非常ugly也很容易出错的代码的问题。
- 用STL解决了C语言中算法和数据结构的N多种坑。

C++泛型编程

C++是支持编程范式最多的一门语言，它虽然解决了很多C语言的问题，但我个人觉得它最大的意义是解决了C语言泛型编程的问题。因为，我们可以看到一些C++的标准规格说明书里面，有一半以上都在说明STL

的标准规格应该是什么样的，这说明泛型编程是C++重点中的重点。

理想情况下，算法应是和数据结构以及类型无关的，各种特殊的数据类型理应做好自己分内的工作。算法只关心一个标准的实现。**而对于泛型的抽象，我们需要回答的问题是，如果我们的数据类型符合通用算法，那么对数据类型的最小需求是什么？**

那C++是如何有效解决程序泛型问题的呢？我认为有三点。

第一，它通过类的方式来解决。

- 类里面会有构造函数、析构函数表示这个类的分配和释放。
- 还有它的拷贝构造函数，表示了对内存的复制。
- 还有重载操作符，像我们要去比较大于、等于、不等于。

这样可以让一个用户自定义的数据类型和内建的那些数据类型就很一致了。

第二，通过模板达到类型和算法的妥协。

- 模板有点像DSL，模板的特化会根据使用者的类型在编译时期生成那个模板的代码。
- 模板可以通过一个虚拟类型来做类型绑定，这样不会导致类型转换时的问题。

模板很好地取代了C时代的宏定义带来的问题。

第三，通过虚函数和运行时类型识别。

- 虚函数带来的多态在语义上可以让“同一类”的类型进行泛型。
- 运行时类型识别技术可以做到在泛型时对具体类型的特殊处理。

这样一来，就可以写出基于抽象接口的泛型。

拥有了这些C++引入的技术，我们就可以做到C语言很难做到的泛型编程了。

正如前面说过的，一个良好的泛型编程需要解决如下几个泛型编程的问题：

1. 算法的泛型；
2. 类型的泛型；
3. 数据结构（数据容器）的泛型。

C++泛型编程的示例 - Search函数

就像前面的`search()`函数，里面的 `for(int i=0; i<len; i++)` 这样的遍历方式，只能适用于**顺序型的数据结构**的方式迭代，如：`array`、`set`、`queue`、`list`和`link`等。并不适用于**非顺序型的数据结构**。

如哈希表`hash table`，二叉树`binary tree`、图`graph`等这样数据不是按顺序存放的数据结构（数据容器）。所以，如果找不到一种**泛型的数据结构的操作方式（如遍历、查找、增加、删除、修改....）**，那么，任何的算法或是程序都不可能做到真正意义上的泛型。

除了`search()`函数的“遍历操作”之外，还有`search`函数的返回值，是一个整型的索引下标。这个整型的下标对于“顺序型的数据结构”是没有问题的，但是对于“非顺序的数据结构”，在语义上都存在问题。

比如，如果我要在一个`hash table`中查找一个`key`，返回什么呢？一定不是返回“索引下标”，因为在`hash table`这样的数据结构中，数据的存放位置不是顺序的，而且还会因为容量不够的问题被重新hash后改变，所以返回数组下标是没有意义的。

对此，我们要把这个事做得泛型和通用一些。如果找到，返回找到的这个元素的一个指针（地址）会更靠谱一些。

所以，为了解决泛型的问题，我们需要动用以下几个C++的技术。

1. 使用模板技术来抽象类型，这样可以写出类型无关的数据结构（数据容器）。
2. 使用一个迭代器来遍历或是操作数据结构内的元素。

我们来看一下C++版的`search()`函数是什么样的。

先重温一下C语言版的代码：

```
int search(void* a, size_t size, void* target,
    size_t elem_size, int(*cmpFn)(void*, void*) )
{
    for(int i=0; i<size; i++) {
        if ( cmpFn (a + elem_size * i, target) == 0 ) {
            return i;
        }
    }
    return -1;
}
```

我们再来看一下C++泛型版的代码：

```
template<typename T, typename Iter>
Iter search(Iter pStart, Iter pEnd, T target)
{
    for(Iter p = pStart; p != pEnd; p++) {
        if ( *p == target )
            return p;
    }
    return NULL;
}
```

在C++的泛型版本中，我们可以看到：

- 使用`typename T`抽象了数据结构中存储数据的类型。
- 使用`typename Iter`，这是不同的数据结构需要自己实现的“迭代器”，这样也就抽象掉了不同类型的数据结构。
- 然后，我们对数据容器的遍历使用了`Iter`中的`++`方法，这是数据容器需要重载的操作符，这样通过操作符重载也就泛型掉了遍历。
- 在函数的入参上使用了`pStart`和`pEnd`来表示遍历的起止。
- 使用`*Iter`来取得这个“指针”的内容。这也是通过重载 `*` 取值操作符来达到的泛型。

当然，你可能会问，为什么我们不用标准接口`Iter.Next()`取代`++`， 用`Iter.GetValue()`来取代`*`，而是通过重载操作符？这样做是为了兼容原有C语言的编程习惯。

说明一下，所谓的`Iter`，在实际代码中，就是像`vector<int>::iterator`或`map<int, string>::iterator`这样的东西。这是由相应的数据容器来实现和提供的。

注：下面是`C++ STL`中的`find()`函数的代码。

```
template<class InputIterator, class T>
  InputIterator find (InputIterator first, InputIterator last, const T& val)
{
  while (first!=last) {
    if (*first==val) return first;
    ++first;
  }
  return last;
}
```

C++泛型编程示例 - Sum 函数

也许你觉得到这一步，我们的泛型设计就完成了。其实，还远远不够。`search`函数只是一个开始，我们还有很多别的算法会让问题变得更为复杂。

我们再来看一个`sum()`函数。

先看C语言版：

```
long sum(int *a, size_t size) {
    long result = 0;
    for(int i=0; i<size; i++) {
        result += a[i];
    }
    return result;
}
```

再看一下C++泛型的版本：

```
template<typename T, typename Iter>
T sum(Iter pStart, Iter pEnd) {
    T result = 0;
    for(Iter p=pStart; p!=pEnd; p++) {
        result += *p;
    }
    return result;
}
```

你看到了什么样的问题？这个代码中最大的问题就是 `T result = 0;` 这条语句：

- 那个`0`假设了类型是`int`；
- 那个`T`假设了`Iter`中出来的类型是`T`。

这样的假设是有问题的，如果类型不一样，就会导致转型的问题，这会带来非常buggy的代码。那么，我们怎么解决呢？

C++泛型编程的重要技术 - 迭代器

我们知道`Iter`在实际调用者那会是一个具体的像`vector<int>::iterator`这样的东西。在这个声明中，`int`已经被传入`Iter`中了。所以，定义`result`的`T`应该可以从`Iter`中来。这样就可以保证类型是一样的，而且不会有被转型的问题。

所以，我们需要精心地实现一个“迭代器”。下面是一个“精简版”的迭代器（我没有把`C++ STL`代码里的迭代器列出来，是因为代码太多太复杂，我这里只是为了说明问题）。

```
template <class T>
class container {
public:
    class iterator {
    public:
        typedef iterator self_type;
        typedef T   value_type;
        typedef T*  pointer;
        typedef T&     reference;

        reference operator*();
        pointer operator->();
        bool operator==(const self_type& rhs);
        bool operator!=(const self_type& rhs);
        self_type operator++() { self_type i = *this; ptr_++; return i; }
        self_type operator++(int junk) { ptr_++; return *this; }
        ...
        ...
    private:
        pointer _ptr;
    };

    iterator begin();
    iterator end();
    ...
    ...
};
```

上面的代码是我写的一个迭代器（这个迭代器在语义上是没有问题的），我没有把所有的代码列出来，而把它的一些基本思路列了出来。这里我说明一下几个关键点。

- 首先，一个迭代器需要和一个容器在一起，因为里面是对这个容器的具体的代码实现。
- 它需要重载一些操作符，比如：取值操作`*`、成员操作`->`、比较操作`==`和`!=`，还有遍历操作`++`，等等。
- 然后，还要`typedef`一些类型，比如`value_type`，告诉我们容器内的数据的实际类型是什么样子。
- 还有一些，如`begin()`和`end()`的基本操作。
- 我们还可以看到其中有一个`pointer _ptr`的内部指针来指向当前的数据（注意，`pointer`就是`T*`）。

好了，有了这个迭代器后，我们还要解决`T result = 0`后面的这个`0`的问题。这个事，算法没有办法搞定，最好由用户传入。于是出现了下面最终泛型的`sum()`版函数。

```
template <class Iter>
typename Iter::value_type
sum(Iter start, Iter end, T init) {
    typename Iter::value_type result = init;
    while (start != end) {
        result = result + *start;
        start++;
    }
    return result;
}
```

我们可以看到`typename Iter::value_type result = init`这条语句是关键。我们解决了所有的问题。

我们如下使用：

```
container<int> c;
container<int>::iterator it = c.begin();
sum(c.begin(), c.end(), 0);
```

这就是整个STL的泛型方法，其中包括：

- 泛型的数据容器；
- 泛型数据容器的迭代器；
- 然后泛型的算法就很容易写了。

需要更多的抽象

更为复杂的需求

但是，还能不能做到更为泛型呢？比如：如果我们有这样的一个数据结构`Employee`，里面有`vacation`就是休假多少天，以及工资。

```
struct Employee {
    string name;
    string id;
    int vacation;
    double salary;
};
```

现在我想计算员工的总薪水，或是总休假天数。

```
vector<Employee> staff;
//total salary or total vacation days?
sum(staff.begin(), staff.end(), 0);
```

我们的`sum`完全不知道怎么搞了，因为要累加的是`Employee`类中的不同字段，即便我们的`Employee`中重载了+操作，也不知道要加哪个字段。

另外，我们可能还会有：求平均值`average`，求最小值`min`，求最大值`max`，求中位数`mean`等等。你会发现，算法写出来基本上都是一样的，只是其中的“累加”操作变成了另外一个操作。就这个例子而言，就是，我想计算员工薪水里面最高的，和休假最少的，或者我想计算全部员工的总共休假多少天。那么面对这么多的需求，我们是否可以泛型一些呢？怎样解决这些问题呢？

更高维度的抽象

要解决这个问题，我希望我的这个算法只管遍历，具体要干什么，那是业务逻辑，由外面的调用方来定义我就好了，和我无关。这样一来，代码的重用度就更高了。

下面是一个抽象度更高的版本，这个版本再叫`sum`就不太合适了。这个版本应该是`reduce`——用于把一个数组`reduce`成一个值。

```
template<class Iter, class T, class Op>
T reduce (Iter start, Iter end, T init, Op op) {
    T result = init;
    while ( start != end ) {
        result = op( result, *start );
        start++;
    }
    return result;
}
```

上面的代码中，我们需要传一个函数进来。在STL中，它是个函数对象，我们还是这套算法，但是`result`不是像前面那样去加，是把整个迭代器值给你一个`operation`，然后由它来做。我把这个方法又拿出去了，所以就会变成这个样子。

在C++ STL中，与我的这个reduce函数对应的函数名叫 accumulate()，其实际代码有两个版本。

第一个版本就是上面的版本，只不过是用for语句而不是while。

```
template<class InputIt, class T>
T accumulate(InputIt first, InputIt last, T init)
{
    for (; first != last; ++first) {
        init = init + *first;
    }
    return init;
}
```

第二个版本，更为抽象，因为需要传入一个“二元操作函数”——BinaryOperation op来做accumulate。accumulate的语义比sum更抽象了。

```
template<class InputIt, class T, class BinaryOperation>
T accumulate(InputIt first, InputIt last, T init,
             BinaryOperation op)
{
    for (; first != last; ++first) {
        init = op(init, *first);
    }
    return init;
}
```

来看看我们在使用中是什么样子的：

```
double sum_salaries =
  reduce( staff.begin(), staff.end(), 0.0,
          [](double s, Employee e)
            {return s + e.salary;}  );

double max_salary =
  reduce( staff.begin(), staff.end(), 0.0,
          [](double s, Employee e)
            {return s > e.salary? s: e.salary; } );
```

注意：我这里用了C++的lambda表达式。

你可以很清楚地看到，reduce这个函数就更通用了，具体要干什么样的事情呢？放在匿名函数里面，它会定义我，我只做一个reduce。更抽象地来说，我就把一个数组，一个集合，变成一个值。怎么变成一个值呢？由这个函数来决定。

Reduce 函数

我们来看看如何使用reduce和其它函数完成一个更为复杂的功能。

下面这个示例中，我先定义了一个函数对象counter。这个函数对象需要一个Cond的函数对象，它是个条件判断函数，如果满足条件，则加1，否则加0。

```
template<class T, class Cond>
struct counter {
    size_t operator()(size_t c, T t) const {
        return c + (Cond(t) ? 1 : 0);
    }
};
```

然后，我用上面的counter函数对象和reduce函数共同来打造一个counter_if算法（当条件满足的时候我就记个数，也就是统计满足某个条件的个数），我们可以看到，就是一行代码的事。

```
template<class Iter, class Cond>
size_t count_if(Iter begin, Iter end, Cond c){
    return reduce(begin, end, 0,
                  counter<Iter::value_type, Cond>(c));
}
```

至于是什么样的条件，这个属于业务逻辑，不是我的流程控制，所以，这应该交给使用方。

于是，当我需要统计薪资超过1万元的员工的数量时，一行代码就完成了。

```
size_t cnt = count_if(staff.begin(), staff.end(),
                   [](Employee e){ return e.salary > 10000; });
```

Reduce时可以只对结构体中的某些值做Reduce，比如说只对 `salary>10000` 的人做，只选出这个里面的值，它用Reduce就可以达到这步，只要传不同的方式给它，你就可以又造出一个新的东西出来。

说着说着，就到了函数式编程。函数式编程里面，我们可以用很多的像`reduce`这样的函数来完成更多的像STL里面的`count_if()`这样的有具体意义的函数。关于函数式编程，我们会在后面继续具体聊。

小结

在这篇文章中，我们聊到C++语言是如何通过泛型来解决C语言遇到的问题的，其实这里面主要就是泛型编程和函数式编程的基本方法相关的细节，虽然解决编程语言中类型带来的问题可能有多种方式，不一定就是C++这种方式。

而我之所以从C/C++开始，目的只是因为C/C++都是比较偏底层的编程语言。从底层的原理上，我们可以更透彻地了解，从C到C++的演进这一过程中带来的编程方式的变化。这可以让你看到，在静态类型语言方面解决泛型编程的一些技术和方法，从而感受到其中的奥妙和原理。

因为形式是多样的，但是原理是相通的。所以，这个过程会非常有助于你更深刻地了解后面会谈到的更多的编程范式。

以下是《编程范式游记》系列文章的目录，方便你了解这一系列内容的全貌。**这一系列文章中代码量很大，很难用音频体现出来，所以没有录制音频，还望谅解。**

- 编程范式游记（1）- 起源
- 编程范式游记（2）- 泛型编程
- 编程范式游记（3）- 类型系统和泛型的本质
- 编程范式游记（4）- 函数式编程
- 编程范式游记（5）- 修饰器模式
- 编程范式游记（6）- 面向对象编程
- 编程范式游记（7）- 基于原型的编程范式
- 编程范式游记（8）- Go 语言的委托模式
- 编程范式游记（9）- 编程的本质
- 编程范式游记（10）- 逻辑编程范式
- 编程范式游记（11）- 程序世界里的编程范式

极客时间
左耳朵耗子
全年独家专栏《左耳听风》
每邀请一位好友订阅
你可获得36元现金返现
获取海报
陈皓
资深技术专家
骨灰级程序员

戳此获取你的专属海报