第十章 基于泛型的程序设计

10.1 泛型概述

一个程序实体能对多种类型的数据进行操作或描述的特性称为类属或**泛型**（Generics）。

基于具有类属性的程序实体进行程序设计的技术称为：**泛型程序设计**

具有类属性的程序实体通常有：

- 类属函数：一个能对不同类型的数据完成相同操作的函数。
- 类属类：一个成员类型可变、但操作不变的类

10.2 模板

10.2.1 函数模板

1. 用通用指针类型参数实现类属函数

   用通用指针实现类属函数面临的问题：

   - 比较麻烦，需要大量的指针操作。
   - 容易出错，编译程序无法进行类型检查。

2. 用函数模板实现类属函数

   函数模板是指带有类型参数的函数定义，格式如下：

   ```
   template <class T1, class T2, ...>
   <返回值类型> <函数名>(<参数表>)
   {   ......
   }
   ```

   T1、T2等是函数模板的参数，它们的取值是某个类型，class也可以写成typename。
   函数的<返回值类型> 、<参数表>中的参数类型以及函数体中的局部变量的类型可以是：T1、T2等。

3. 函数模板的实例化

   函数模板定义了一系列重载的函数。

   要使用函数模板所定义的函数，首先必须要对函数模板进行实例化：给模板参数提供一个具体的类型，从而生成具体的函数。

   函数模板的实例化通常是隐式的：

   - 由编译程序根据函数调用的实参类型自动地把函数模板实例化为具体的函数。
   - 这种确定函数模板实例的过程叫做**模板实参推导**（template argument deduction）。

   有时，编译程序无法根据调用时的实参类型来确定所调用的模板实例函数。

   解决办法：显式类型转换、**显式实例化**

   带非类型参数的函数模板在使用时需要显式实例化

10.2.2. 类模板

如果一个类定义中用到的类型可变、但操作不变，则该类称为类属类。

在C++中，类属类用类模板实现。

类模板的格式为：

```
template <class T1,class T2,...> //class也可以写成typename
class <类名>
{   <类成员说明>
}
```

T1、T2等为类模板的类型参数，在类成员的说明中可以用T1、T2等来作为它们的类型。

类模板的实例化

类模板定义了若干个类，在使用这些类之前需要对类模板进行实例化。类模板的实例化需要在程序中**显式**地指出

一个类模板的不同实例之间不共享该类模板中的静态成员

带非类型参数的类模板

10.2.3 模板的复用

模板也属于一种多态，称为参数化多态：一段带有类型参数的代码，给该参数提供不同的类型就能得到多个不同的代码，即，一段代码有多种解释。

模板的复用是通过对模板进行实例化（用一个具体的类型去替代模板的类型参数）来实现的：

- 由于实例化是在编译时刻进行的，它一定要见到相应的源代码，因此，模板属于源代码复用。
- 一个模板有很多的实例，是否实例化模板的某个实例，这要由使用情况来决定。

模板的定义和实现都放在头文件中，使用者通过包含这个头文件，把模板的源代码全包含进来，以备实例化所需。

重复实例的处理

在由多模块构成的程序中，由于每个模块是单独编译的，因此，会导致一个模板的某个实例存在于多个模块的编译结果中：

- 相同的函数模板实例
- 相同的类模板成员函数实例

相同代码的存在会造成目标代码庞大

解决：

- 由开发环境来解决：编译第二个模块的时候不生成这个实例。（代价大！）
- 由链接程序来解决：舍弃一个相同的实例。

10.3 基于STL的编程

10.3.1 STL概述

除了从C标准库保留下来的一些功能外，C++还提供了一个基于模板实现的标准模板库（Standard Template Library，简称STL）：

- 实现了一些面向序列数据的表示及常用的操作。
- 支持了一种抽象的编程模式，该模式隐藏了一些低级的程序元素，如数组、链表、循环等。

STL包含：

- 容器：用于存储序列化的数据元素，如：向量、队列、栈等。
- 算法：用于实现对容器中的数据元素进行一些常用的操作，如：查找、排序、统计、求和等一系列算术运算。
- 迭代器：实现了抽象的指针功能，它们指向容器中的数据元素，用于对容器中的数据元素进行遍历和访问。是容器和算法之间的桥梁：传给算法的不是容器，而是指向容器中元素的迭代器，这样使得算法与容器保持独立，从而提高算法的通用性。

10.3.2 容器

容器是由同类型元素构成的、长度可变的元素序列。
容器是用类模板来实现的，模板的参数是容器中元素的类型。
STL中包含了很多种容器，虽然这些容器提供了一些相同的操作，但由于它们采用了不同的内部实现方法，因此，不同的容器往往适合于不同的应用场合。

STL的主要容器

- vector<元素类型>：用于需要快速定位（访问）任意位置上的元素以及主要在元素序列的尾部增加/删除元素的场合。
  在头文件vector中定义，用**动态数组**实现。
- list<元素类型>：用于经常在元素序列中任意位置上插入/删除元素的场合。
  在头文件list中定义，用**双向链表**实现。
- deque<元素类型>：用于主要在元素序列的两端增加/删除元素以及需要快速定位（访问）任意位置上的元素的场合。
  在头文件deque中定义，用**分段的连续空间结构**实现。
- stack<元素类型>：用于仅在元素序列的尾部增加/删除元素的场合。
  在头文件stack中定义，基于deque、list或vector来实现。
- queue<元素类型>：用于仅在元素序列的尾部增加、头部删除元素的场合。
  在头文件queue中定义，基于deque或list来实现。
- priority_queue<元素类型>：它与queue的操作类似，不同之处在于：每次增加/删除元素之后，它将对元素位置进行调整，使得头部元素总是最大的。也就是说，每次删除操作总是把最大（优先级最高）的元素去掉。
  在头文件queue中定义，基于deque或vector来实现。
- map<关键字类型，值类型>和multimap<关键字类型，值类型>：用于需要根据关键字来访问元素的场合。容器中每个元素属于一个pair结构类型，该结构有两个成员：first和second，关键字对应first，值对应second，元素是根据其关键字排序的。
  - 对于map，不同元素的关键字不能相同；对于multimap，不同元素的关键字可以相同。
  - 它们在头文件map中定义，常常用某种二叉树来实现。

- set<元素类型>和multiset<元素类型>：它们分别是map和multimap的特例，每个元素只有关键字而没有值，或者说，关键字与值合一了。
  在头文件set中定义。
- basic_string<字符类型>：与vector类似，不同之处在于其元素为字符类型，并提供了一系列与字符串相关的操作。
  string和wstring分别是它的两个实例：
  - basic_string
  - basic_string<wchar_t>

  在头文件string中定义。

注意：如果容器的元素类型是一个类，则针对该类可能需要：

- 自定义拷贝构造函数和赋值操作符重载函数：对容器进行的一些操作中可能会创建新的元素（对象的拷贝构造）或进行元素间的赋值（对象赋值）。
- 重载小于操作符（<）：对容器进行的一些操作中可能要对元素进行“小于”比较运算。

10.3.3 迭代器

迭代器（iterator）实现了抽象的指针（智能指针）功能，它们指向容器中的元素，用于对容器中的元素进行访问和遍历。

在STL中，迭代器是作为类模板来实现的（在头文件iterator中定义），它们可分为以下几种类型：

- 输出迭代器（output iterator，记为：OutIt）
  只能修改它所指向的容器元素
  间接访问（*）
  ++
- 输入迭代器（input iterator，记为：InIt）
  只能读取它所指向的容器元素
  间接访问（*）和元素成员间接访问（->）
  ++、==、!=。
- 前向迭代器（forward iterator，记为：FwdIt）
  可以读取/修改它所指向的容器元素
  元素间接访问（*）和元素成员间接访问（->）
  ++、==、!=
- 双向迭代器（bidirectional iterator，记为：BidIt）
  可以读取/修改它所指向的容器元素
  元素间接访问（*）和元素成员间接访问（->）
  ++、--、==、!=操作
- 随机访问迭代器（random-access iterator，记为：RanIt）
  可以读取/修改它所指向的容器元素
  元素间接访问（*）、元素成员间接访问（->）和下标访问元素（[]）
  ++、--、+、-、+=、-=、==、!=、<、>、<=、>=
  注意：指向数组元素的普通指针可以看成是随机访问迭代器

各容器的迭代器类型

不同的容器所提供的迭代器类型会有所不同：

- 对于vector、deque以及basic_string容器类，与它们关联的迭代器类型为随机访问迭代器（RanIt）

- 对于list、map/multimap以及set/multiset容器类，与它们关联的迭代器类型为双向迭代器（BidIt）。
- queue、stack和priority_queue容器类，不支持迭代器！

可通过容器类的成员函数begin和end等获得容器的首尾迭代器。

迭代器之间的相容关系：在需要箭头左边迭代器的地方可以用箭头右边的迭代器去替代

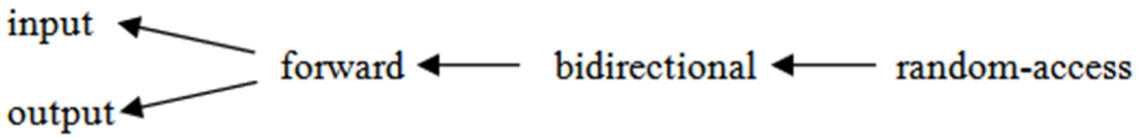

图 9-1 迭代器之间的相容关系

除了前面五种基本迭代器外，STL还提供了一些迭代器的适配器，便于一些特殊的操作，如：

- 反向迭代器（reverse iterator）：用于对容器元素从尾到头进行反向遍历：++操作是往容器首部移动，--操作是往容器尾部移动。可以通过容器类的成员函数rbegin和rend可以获得容器的尾和首元素的反向迭代器。
- 插入迭代器（insert iterator）：用于在容器中指定位置插入元素，其中包括：
  back_insert_iterator（用于在尾部插入元素）
  front_insert_iterator（用于在首部插入元素）
  insert_iterator（用于在任意指定位置插入元素）
  它们可以分别通过全局函数back_inserter、front_inserter和inserter来获得，函数的参数为容器。

10.3.4 算法

一系列通用的对容器中元素进行操作的全局函数

STL算法实现了对序列元素的一些常见操作，主要包括：调序算法、编辑算法、查找算法、算术算法、集合算法、堆算法、元素遍历算法

算法是用函数模板实现的，除了算术算法在头文件numeric中定义外，其它算法都在头文件algorithm中定义。

大部分STL算法都是遍历指定容器中某范围内的元素，对满足条件的元素执行某项操作。

算法的内部实现往往隐含着循环操作，但这对使用者是隐藏的：使用者只需要提供：容器的迭代器、操作条件以及可能的自定义操作，而算法的控制逻辑则由算法内部实现，这体现了一种抽象的编程模式。

算法与容器之间的关系

在STL中，不是把容器传给算法，而是把容器的某些迭代器传给它们，在算法中通过迭代器来访问和遍历相应容器中的元素。
这样做的好处是提高了算法的通用性：
不管什么容器，只要它的迭代器与算法所需要的迭代器相容，就可以传给算法，这样就使得算法不依赖于具体的容器。
需要注意的是：有些容器（stack、queue以及priority_queue）不支持迭代器，因此不能用STL中的算法来操作它们！

算法接受的迭代器类型

一个算法能接收的迭代器的类型是通过算法模板参数的名字来体现的

算法的操作范围

用算法对容器中的元素进行操作时，大都需要用两个迭代器来指出要操作的元素的范围

有些算法可以有多个操作范围，这时，除第一个范围外，其它范围一般不需要指定最后一个元素位置，它由第一个范围中元素的个数决定

对于一些带有目标范围的操作，目标范围中已有元素的个数一般不能小于源范围规定的元素个数，如果操作需要在目标范围内增加元素，则目标范围的迭代器要用某个插入迭代器

由于指向数组元素的普通指针可以看成是随机访问迭代器，因此可以作为操作范围来用

算法的自定义操作条件

有些算法可以让使用者提供一个函数或函数对象来作为自定义操作条件（或称为谓词），其参数类型为相应容器的元素类型，返回值类型为bool。

自定义操作条件可分为：

- Pred：一元"谓词"，需要一个元素作为参数
- BinPred：二元"谓词"，需要两个元素作为参数

有些算法可以让使用者提供一个函数或函数对象作为自定义操作，其参数和返回值类型由相应的算法决定

自定义操作可分为：

- Op或Fun：一元操作，需要一个参数
- BinOp或BinFun：二元操作，需要两个参数

10.4 函数式程序设计概述

命令式程序设计范式：针对一个目标，需要给出达到目标的操作步骤和状态变化，即要对"如何做"进行详细描述。它们与冯诺依曼体系结构一致，是使用较广泛的程序设计范式，适合于解决大部分的实际应用问题。

- 过程式程序设计
- 面向对象程序设计

声明式程序设计范式：只需要给出目标的定义，不需要对如何达到目标（包括操作步骤和状态变化）进行描述，即只需要对"做什么"进行描述。有良好的数学理论支持，易于保证程序的正确性，并且，设计出的程序比较精炼和具有潜在的并行性。

- 函数式程序设计
- 逻辑式程序设计

10.4.1 什么是函数式程序设计

把程序组织成一组数学函数，计算过程体现为基于一系列函数应用（把函数作用于数据）的表达式求值。

函数也被作为值（数据）来看待，即函数的参数和返回值也可以是函数。

基于的理论是递归函数理论和lambda演算。

函数式程序设计的基本特征

- "纯"函数：以相同的参数调用一个函数总得到相同的值。（引用透明）除了产生计算结果，不会改变其他任何东西。（无副作用）
- 没有状态：计算体现为数据之间的映射，它不改变已有数据，而是产生新的数据。（无赋值操作）
- 函数也是值：函数的参数和返回值都可以是函数，可由已有函数生成新的函数。（高阶函数）
- 递归是主要的控制结构：重复操作采用函数的递归调用来实现，而不采用迭代（循环）。
- 表达式的惰性（延迟）求值（Lazy evaluation）：一个表达式只有需要用到它的值的时候才会去计算它。

- 潜在的并行性：由于程序没有状态以及函数的引用透明和无副作用等特点，因此一些操作可以并行执行。

函数式编程语言

纯函数式编程语言：Common Lisp, Scheme, Clojure, Racket, Erlang, OCaml, Haskell, F#

对函数式编程提供支持的语言：Perl, PHP, C# 3.0, Java 8, Python, C++(11)

在C++中，函数式编程主要通过STL来实现。

10.4.2 函数式程序设计中的常用操作

（1）递归：实现重复操作，不采用迭代方式（循环），而是采用递归。

由于函数递归调用效率低、递归调用深度受栈空间的限制，因此，函数式编程常采用尾递归，即递归调用是函数的最后一步操作。

注意：不是函数体中只有一次递归调用就是尾递归

尾递归调用便于编译程序优化：

- 由于递归调用后不再做其它事，从而不会再使用当前栈空间的内容，因此，递归调用时不必额外分配栈空间，可重用当前的栈空间。
- 可以自动转成迭代。

（2）过滤/映射/规约

1. 过滤（Filter）：把一个集合中满足某条件的元素选出来，构成一个新的集合
2. 映射（Map）：对一个集合中的每个元素分别进行某个操作，结果放到一个新集合中
3. 规约（Reduce）：对一个集合中的所有元素连续进行某个操作，最后得到一个值

（3）部分（偏）函数应用和柯里化

1. 部分（偏）函数应用：

   对于一个多参数的函数，在某些应用场景下，它的一些参数往往取固定的值

   部分函数应用是指：对一个多参数的函数，只给它的某些参数提供值，从而生成一个新函数，该新函数不包含原函数中已提供值的参数。

   部分函数应用可缩小一个函数的适用范围，提高函数的针对性

2. 柯里化（Currying）：

   柯里化是指把一个多参数的函数变换成一系列单参数的函数，它们分别接收原函数的第一个参数、第二个参数、……。

   柯里化的意义：

   - 数学上：对单参数函数的研究模型可以用到多参数函数上。
   - 对程序设计：不必把一个多参数的函数所需要的参数同时提供给它，可以逐步提供

   利用函数的柯里化，也可以缩小一个函数的适用范围，提高函数的针对性

部分函数应用和柯里化的区别

- 部分函数应用得到的是一个参数个数较少的函数，对该函数的调用将得到一个具体的值；而柯里化得到的则是一个由一系列单参数的函数所构成的函数链，对柯里化后的函数调用将得到函数链上的下一个函数，对函数链上最后一个函数的调用才会得到具体的值。
- 部分函数应用可以按任意次序绑定原函数的参数值，而柯里化只能依次绑定原函数的参数值。