第 3 单元　素数产生器

素数（prime number，prime）又称质数，是在大于 1 的整数中，除了 1 和它本身外，不再有别的约数的数。素数产生器的功能是输出一个自然数区间中的所有素数。

3.1　问题描述与对象建模

3.1.1　对象建模

本例的意图是建立一个自然数区间，如图 3.1 所示的[11, 101]、[350, 5500]、[3, 1000]等区间内的素数序列（prime series）。把每一个正整数区间的素数序列作为一个对象，则对这个问题建模，就是考虑定义一个具有一般性的素数产生器——PrimeGenerator 类。这个类的行为是产生一个素数序列的函数 getPrimeSequence( )。这个类中不同对象的区别是每个对象的区间下限（lowerNaturalNumber）和区间上限（upperNaturalNumber）不同。于是，可以得到如图 3.2 所示的 PrimeGenerator 类初步模型及其声明代码。

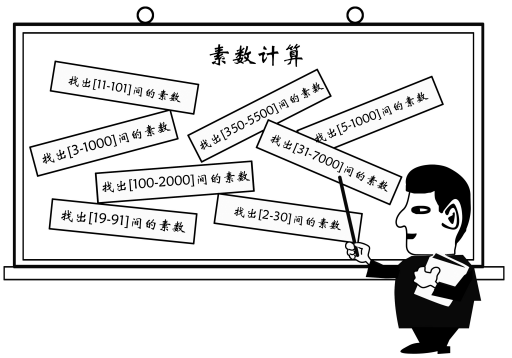

图 3.1　不同的求素数对象

【代码 3-1】 PrimeGenerator 类初步模型声明。

```
class PrimeGenerator {
   int lowerNaturalNumber;
   int upperNaturalNumber;
public:
   void getPrimeSequence ();
}
```

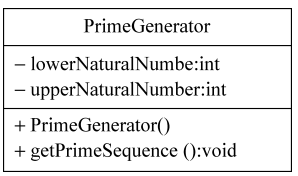

图 3.2　PrimeGenerator 类初步模型

3.1.2 getPrimeSequence( )函数的基本思路

getPrimeSequence( )函数的功能是给出 [lowerNaturalNumber,upperNaturalNumber] 区间内的素数序列。基本思路是，从 lowerNaturalNumber 到 upperNaturalNumber，逐一对每一个数进行测试，看其是否为素数。如果是，则输出该数（用不带回车的输出，以便显示出一个序列）；否则，继续对下一个数进行测试。

每次测试使用的代码相同，只是被测试的数据不同。也就是说，这样一个函数中的代码要不断重复执行，直到达到目的为止。这种程序结构称为重复结构，也称循环结构。

在实现 getPrimeSequence( )函数时有如下两种考虑。

（1）用 isPrime( )判定一个数是否为素数。

为了将 getPrimeSequence( )函数设计得比较简单，把测试一个数是否为素数的工作也用一个函数 isPrime( )进行。所以 getPrimeSequence( )函数就是重复地对区间内每个数用函数 isPrime( )进行测试。

isPrime( )函数用来对于某个自然数进行测试，看其是否为素数。其原型应当为：“bool isPrime(int number);”。

测试一个自然数是否为素数的基本方法是：把这个数 number 依次用 2~number/2 去除，只要有一个能整除，该数就不是素数。

所以，这两个函数都要采用重复结构。

（2）在 getPrimeSequence( )函数中直接判定一个数是否为素数。

下面分别来讨论。

3.2 使用 isPrime( )的 PrimeGenerator 类实现

C++有 3 种重复控制结构：while、do-while 和 for。无论哪种重复结构，都要包含如下用于控制重复过程的三部分内容：初始化部分、循环条件和修正部分。

在 2.4 节中，已经讨论过 while 结构和 do-while 结构的用法。下面讨论用 for 结构实现 getPrimeSequence( )和 isPrime( )函数的方法。

3.2.1 用 for 结构实现的 getPrimeSequence( )函数

如前所述，循环结构是通过初始化部分、循环条件和修正部分来控制循环过程的。while 结构和 do-while 结构将这 3 个部分分别放在不同位置，而 for 结构则把这 3 个部分放在一起，形成如下形式：

```
for (初始化部分; 循环条件; 修正部分) {
    循环体
}
```

这样，可以对循环过程的控制一目了然，特别适合用于循环次数可以预先确定的情况。所以也把 for 循环称为计数循环。

【代码 3-2】 采用 for 结构的 getPrimeSequence( )代码。

```
void PrimeGenerator::getPrimeSequence () {
   std::cout << lowerNaturalNumber << "到" << upperNaturalNumber << "之间的素数序列为：";
   for(int m = lwerNaturalNumber; m <= upperNaturalNumber; m ++)  //循环控制
        if (isPrime (m))
            std::cout << m << ",";
}
```

说明：

（1）for 结构也称计数型重复结构。当重复具有明显的计数特征时，采用 for 结构更为合适。

（2）在 C++中，表达式 m = m + 1 可以简化为 m += 1。“+ =”称为“赋值加”，是加和赋值的组合操作符。如 i + = 5，相当于 i = i + 5。除赋值加外，复合赋值操作符还有：“– =”“*=”“/=”等。复合赋值操作符的优先级别与赋值操作符相同。注意，任何由两个字符组成的操作符（如“= =”“>=”“<=”“!=”及复合赋值操作符等）作为一个整体，字符之间不能加空格。

（3）m = m + 1 还有一种更简洁的表示形式：+ + m 或 m ++，“+ +”称为增量操作符或自增操作符。与增量操作符“+ +”对应的是减量操作符“– –”，或称自减操作符。

（4）这个重复（循环）结构的基本作用是对一个自然数区间中的每一个数都进行是否为素数的测试。这种思路称为穷举。在这个穷举过程中，每测试完一个自然数后，就通过i++这样的操作来找到下一个自然数。这种在一个值的基础上通过某种操作找后一个值的过程称为迭代。穷举和迭代是重复结构的两个基本用途，也是一切计算机算法的基础。

代码 3-2 中设计的 getPrimeSequence( )代码疏忽了一个问题：没有考虑用户给出的区间下限小于 2 的情况，也没有考虑给出的区间上下限反置的情况。下面的代码弥补了这一缺陷。

【代码 3-3】 考虑下限小于 2 时进一步完善的 getPrimeSequence ()代码。

```
void PrimeGenerator::getPrimeSequence(){
  std::cout << lowerNaturalNumber << "到" << upperNaturalNumber << "之间的素数序列为：");
  if ( lowerNaturalNumber > upperNaturalNumber){    //区间界限输入颠倒时交换
    int temp = lowerNaturalNumber;
    lowerNaturalNumbe = upperNaturalNumber;
    upperNaturalNumber = temp;
  }
 ┌--------------------------------┐
 ¦ for(int m = lwerNaturalNumber; m <= upperNaturalNumber; m ++)
 ¦   if (lowerNaturalNumber < 2){
 ¦     std::cout << 2 << ",";
 └-----continue;                                    //短路一次，循环中后面的部分
    )
    if (isPrime (m))
      std::cout << m << ",";
}
```

3.2.2 用 for 结构实现的 isPrime( )函数

isPrime( )函数是用从 2 到 number/2 之间的数，依次去除被检测的数 number。若有一个数能整除 number 就可以判定 number 不是素数。这个重复过程具有明显的计数特征，所以应采用 for 结构。

【代码 3-4】 采用 for 结构的 isPrime( )方法。

```
bool PrimeGenerator::isPrime (int number) {
    for (int m = 2;  m < number; m ++) {
        if (number % m == 0) {
             return false;        //发现 number 能被一个数整除，就断定它不是素数
        }
    }
    return true;                  //测试结束还没有被任何数整除，断定其为素数
}
```

3.2.3 完整的 PrimeGenerator 类及其测试

【代码 3-5】 PrimeGenerator 类界面声明。

```
//文件名：prime01.h
class PrimeGenerator {
   int lowerNaturalNumber;
   int upperNaturalNumber;
public:
   PrimeGenerator(int,int);
   void getPrimeSequence ();
   static bool isPrime(int);
};
```

【代码 3-6】 PrimeGenerator 类的实现。

```
#include "prime01.h"
#include <iostream>

PrimeGenerator::PrimeGenerator(int lNum,int uNum){
        lowerNaturalNumber = lNum;
        upperNaturalNumber= uNum;
}

void PrimeGenerator::getPrimeSequence () {
   std::cout << lowerNaturalNumber << "到" << upperNaturalNumber << "之间的素数序列为：";
   for(int m = lowerNaturalNumber;m <= upperNaturalNumber; m ++)( //循环控制
     if (lowerNaturalNumber < 2){
       std::cout << 2 << ",";
       continue;                  //短路一次，循环中后面的部分
     )
     if (isPrime(m))
```

```
            std::cout << m << ",";
   )
}

bool PrimeGenerator::isPrime (int number) {
    for (int m = 2;  m < number; m ++) {
        if (number % m == 0) {
            return false;         //发现 number 能被一个数整除，就断定它不是素数
       }
    }
    return true;                  //测试结束还没有被任何数整除，断定其为素数
}
```

【代码 3-7】 PrimeGenerator 类测试代码。

```
#include "prime01.h"
#include <iostream>

int main(){
   PrimeGenerator psg(3,1000);
   psg.getPrimeSequence();

   return 0;
}
```

测试结果如下。

```
3到1000之间的素数序列为：3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,7
3,79,83,89,97,101,103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,18
1,191,193,197,199,211,223,227,229,233,239,241,251,257,263,269,271,277,281,283,29
3,307,311,313,317,331,337,347,349,353,359,367,373,379,383,389,397,401,409,419,42
1,431,433,439,443,449,457,461,463,467,479,487,491,499,503,509,521,523,541,547,55
7,563,569,571,577,587,593,599,601,607,613,617,619,631,641,643,647,653,659,661,67
3,677,683,691,701,709,719,727,733,739,743,751,757,761,769,773,787,797,809,811,82
1,823,827,829,839,853,857,859,863,877,881,883,887,907,911,919,929,937,941,947,95
3,967,971,977,983,991,997,
```

3.3 不使用 isPrime( )的 PrimeGenerator 类实现

3.3.1 采用嵌套重复结构的 getPrimeSequence( )函数

若不使用 isPrime( )函数，则 getPrimeSequence( )函数将成为一个嵌套的重复结构。

【代码 3-8】 采用嵌套重复结构的 getPrimeSequence( )函数。

```
#include "prime02.h"
#include <iostream>

PrimeGenerator::PrimeGenerator(int lNum,int uNum){
        lowerNaturalNumber = lNum;
        upperNaturalNumber= uNum;
}
```

```
void PrimeGenerator::getPrimeSequence(){
   std::cout << lowerNaturalNumber << "到" << upperNaturalNumber << "之间的素数序列为：";
   for(int m = lowerNaturalNumber;m <= upperNaturalNumber; m ++){
      bool flag = true;
      if (lowerNaturalNumber < 2){
         std::cout << 2 << ",";
         continue;                          //短路一次，循环中后面的部分
    )
    for (int n = 2;  n < m; n ++) {
          if (m % n == 0) {
             flag = false;                  //能被一个数整除，就断定它不是素数
          break;
      }
   }
   if(flag == true)
      std::cout<< m <<",";
   }
}
```

【代码 3-9】 修改的 PrimeGenerator 类声明。

```
//文件名：prime02.h
class PrimeGenerator{
   int lowerNaturalNumber;
   int upperNaturalNumber;
public:
   PrimeGenerator(int,int);
   void getPrimeSequence ();
   //bool isPrime(int);                   //注释掉该行
};
```

测试代码和结果与代码 3-7 同。

说明：

（1）在代码 3-8 中，为了测试一个数是否为素数，采用了一个标记变量 flag。流程一旦进入外 for 循环中，就会定义一个 flag，并将其初始化为 true。在内 for 循环中，一旦发现被测试数不是素数，便将 flag 置 false，并用 break 跳出内循环；否则会一直到对被测试数进行完全测试后才退出内循环。在内循环外，首先检测 flag 有无改变。若无改变，则打印被测试数，然后跳到外循环的增量处取下一个数测试；若有变化，则直接跳到外循环的增量处取下一个数测试。

（2）在这个函数中，变量 *m* 定义在 for 循环体之前（属初始化部分），其作用域为函数作用域。*n* 和 flag 都定义在内 for 循环体前、外循环之内，具有语句作用域。

3.3.2 重复结构中的 continue 语句和 break 语句

在代码 3-3 中使用了 continue 语句，在代码 3-8 中使用了 break 语句。这两个语句的功能如图 3.3 所示。

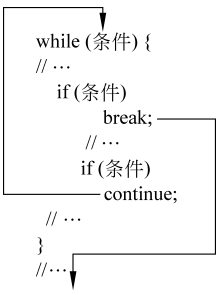

图 3.3 continue 与 break 的功能

continue 语句的作用是短路循环体中后面的语句，转入下一轮的判断、循环。break 语句的作用是结束当前的循环。这两种操作都是在一定条件下才需要执行，所以在循环体中这两个语句常与 if-else 结构相配合。

注意：

（1）break 只对循环和 switch-case 结构有效。

（2）当结构嵌套时，break 语句只对当前一层循环或 switch- case 结构有效。

3.4 数　　组

前面的程序所生成的素数序列是一过性的，生成一个，输出一个，想使用其中某一个或几个素数，再也找不到了。

或许有人会问，那用简单变量不是可以存储吗？是的。用简单变量是可以存储。例如用 prime0、prime1、prime2、prime3…存储一个素数序列。但是这种方法中的每一个变量从名字上并不反映它们之间的整体关系，名字上虽然都使用了 prime 和序号，但仅仅是对人而言，是人辨识的结果，程序本身还无法做到这一点。因此，用简单变量存储一个素数序列并没有多大实用价值。C++提供的数组，则可以做到这些。

3.4.1 数组及其定义

1. 数组的特点

数组（array）是系统定义的一种构造数据类型，它有如下特点。

（1）数组用一个名字组织多个同类型数据。这个名字称为数组名，组成数组的数据称为数组元素。

（2）数组元素可以是原子的（基本类型的），也可以是组合的、对象的，但必须类型相同。“类型相同”意味着每个元素都具有相同的存储空间，并可以进行同样的运算。这个类型称为数组的基类型。

（3）数组中的数据元素在逻辑上和物理上都是顺序的，在逻辑上，每个元素要顺序编号（从 0 开始），这个顺序号称为下标，所以数组元素被称为下标变量；在物理上，下标元素在内存中也是按照逻辑顺序依次存放的。

例如，要存储 30 以内的 9 个素数{2,3,7,11,13,17,19,23,29}，可以给它们起一个名字 prime30，依次将它们存进变量 prime30[0]、prime30[1]、prime30[2] …、prime30[8]中。这 9 个变量称为数组 prime30 的 9 个分量，它们之间不仅有逻辑上的顺序关系，而且如图 3.4 所示，在物理上它们在内存中依次相邻地存放。由于它们用括在方括号中的下标相互区分，因此称为数组 prime30 的下标变量。

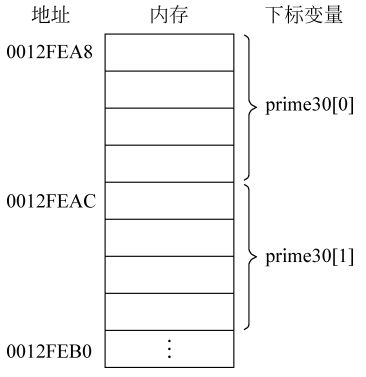

图 3.4　prime30 的内存示意图

注意，任何数组的下标变量的下标都是从 0 开始。

2. C++数组定义的一般规则

与 C++中的任何标识符一样，数组名必须先定义才可以使用。数组的定义格式如下。

```
类型  数组名［整型常量表达式］;
```

例如，声明语句

```
int prime30[9];
```

告诉编译器要为数组 prime30 连续地开辟可以存放 9 个 int 类型数据的空间。从图 3.4 中可以看出，下标变量与地址的关系。在一个数组中，各个下标变量占用的存储空间相同，都是一个数组基类型的大小。当基类型为 int 时，每个下标变量都占用 4B 的存储空间，于是下标增 1，下标变量的地址增 4。

说明：

（1）定义数组时，方括号内的整型常数表达式用于表示数组的大小，即数组中元素的个数。

（2）一个数组在程序中只能定义一次，不可以重复定义。

（3）数组的大小是数组可以存储的数据的数量。由于数组下标变量都是从 0 开始，因此下标变量的最大下标是(数组大小 – 1)。

3. C++11 的泛化常量表达式

在 C++程序中，有些地方要求必须使用常量表达式，如在数组定义中，数组的大小必须用常量表达式指定。例如下面的代码是合法的。

```
int a[2 + 3];
```

但是，在 C++中，常量表达式的概念是比较窄的，它要求这种表达式能在编译器和运行时都得到相同的结果。例如不允许含有函数调用，因为在编译时没有办法知道一个函数是不是常量。因此，对于函数

```
int getTwo(){return 2};
```

下面的代码是非法的。

```
int b[getTwo() + 3];              //非法
```

C++11 引入一个关键字 constexpr，程序员可以用其对一个函数的行为加以限制，保证一个函数是一个编译器常量。因此，在 C++11 中，下面的代码是合法的。

```
constexpr int getTwo() {return 2};
int b[getTwo() + 3];              //合法
```

注意：使用 constexpr 关键字修饰函数有 3 个要点。

（1）这个函数的返回值类型不能是 void。

（2）在函数体中不能声明变量或新类型。

（3）函数体内只能包含声明语句、空语句和单个 return 语句，且 return 语句中的表达式也必须是常量表达式。

3.4.2 数组的初始化规则

数组初始化就是在定义数组的同时决定数组元素的值。C++关于数组初始化有如下几项规则。

1. 数组初始化的基本形式

数组初始化的基本形式是将一组初始化表达式按元素顺序依次写在一对花括号内。

```
int prime30[9] = {2,3,7,11,13,17,19,23,29};
```

2. 数组初始化的默认形式

（1）数组初始化时，可以省略数组大小由编译器根据初始值的个数自动确定。如上述使用过的声明语句：

```
 int prime30[] = {2,3,7,11,13,17,19,23,29};
```

（2）允许省略为 0 的初始化值。如语句

```
int a[] = {1,2,0,3,0};
```

可以写成

```
int a[] = {1,2,,3,};
```

注意，中间的逗号不能缺少。

（3）当最后的几个元素初始化值为 0 时，可以只写出前面的数列，但数组体积不可省略。如语句

```
int a[8] = {1,2,3,0,0,0,0,0};
```

可以写成

```
int a[8] = {1,2,3};
```

或

```
int a[8] = {1,2,3, , };
```

或

```
int a[] = {1,2,3,,,,,};
```

但不能写成

```
int a[] = {1,2,3};
```

（4）C++11 允许初始化数组时省略赋值号。例如：

```
int a[3] {1,2,3};
```

（5）C++11 允许花括号内为空，这意味着将把所有元素都初始化为 0。例如：

```
unsigned int a[30] = {};          //将所有元素初始化为 0
unsigned int b[50] {};            //将所有元素初始化为 0
float c[80] {};                   //将所有元素初始化为 0.0
```

3. 数组初始化的约束

（1）数组初始化只能在数组定义时进行，否则就是非法的。例如：

```
int a[5];
a = {1,2,3,4,5}.                  //非法
```

（2）不可以用一个数组名去初始化另一个数组或者用一个数组给另一个数组赋值，例如：

```
int a[5] {1,2,3,4,5};
int b[5] = a;                     //非法
```

（3）列表初始化不支持窄化转换，例如：

```
long a[3] = {1,2,3};              //不合法，因为数组 a 被定义为 long 而初始化列表为 int
```

3.4.3 用数组存储素数序列

在程序设计中，若存在一组同类型并且有顺序关系的数据，使用数组存储它们，可以带来许多方便。这时，可以用一个名字命名它们，并用下标标识每个元素在这个数据序列中的顺序位置。这类程序中常常会采用重复结构。这种程序一般具有如下结构。

（1）根据数列的类型和大小，声明一个数组。

（2）用重复结构将数列元素依次存入数组。

（3）用下标引用数组中的某些元素。

对于素数产生器来说，只要将输出到标准输出对象 cout 的语句改为输出到数组，再增加一个输出成员函数。

【代码 3-10】 用数组存储素数序列的 getPrimeSequence( )函数。

```
void PrimeGenerator::getPrimeSequence(){
   //std::cout << lowerNaturalNumber << "到" << upperNaturalNumber << "之间的素数序列为：";
   for(int m = lowerNaturalNumber,i = 0; m <= upperNaturalNumber; m ++){
      bool flag = true;
      if (lowerNaturalNumber < 2){
         prime[i++] = 2;
         continue;                //短路一次，循环中后面的部分
```

```
    )
    for (int n = 2;  n < m; n ++) {
        if (m % n == 0) {
           flag = false;                //能被一个数整除，就断定它不是素数
           break;
        }
    }
    if(flag == true)
       //std::cout<< m <<",";
       prime[i++] = m;
    }
}
```

说明：在这个代码中，使用了两个作为顺序的变量 m 和 i。m 既用来代表被判断的整数序列中的一个数，也是这个数的位置。i 代表存放这个序列中的素数的序列，也是存放这个素数序列的数组的下标。

3.4.4 sizeof 操作符

下面考虑如何设计一个函数，将 prime 数组的元素一一输出出来。一般来说，对于一个数组元素的输出，最好、最直接的方法是采用 for 结构。但是，使用 for 需要知道数组 prime 的大小作为循环终值，以保证读数组元素过程中不会越界。但是，设计这个函数时，可能还不知道其终值。为了求得 prime 的大小，可以使用 sizeof 操作符。计算公式为：

sizeof(prime)/sizeof(int)

操作符 sizeof 用于计算一个数据或类型的字节宽度。这里，sizeof(prime)计算得到数组 prime 占用的字节数，sizeof(int) 计算得到 int 类型占用的字节数。二者之商为数组 prime 中的元素个数。

【代码 3-11】 素数序列输出函数 dispPrimeSequence( )。

```
void dispPrimeSequence( ){
  std::cout << lowerNaturalNumber << "到" << upperNaturalNumber << "之间的素数序列为：\n";
  for(int i = 0; i < sizeof(prime)/sizeof(int) && primeprime[i] != 0; i++)
    std::cout<< prime[i] <<",";
)
```

说明：

（1）由于人们还没有找到一个自然数区间中的素数的个数的分布规律，因此用于存放该区间的素数序列的数组一般会比较大。这样，素数序列会占用该数组的前面部分，后面的元素会设置成为 0。也就是说，输出只执行该数组中前面不为 0 的部分。

（2）增加了成员函数 dispPrimeSequence( )，还需要在类 PrimeGenerator 中增加相应的成员。

3.4.5 C++11 中基于容器的 for 循环

针对尚不知道大小的 prime 这类问题，使用 C++11 提供的基于容器的 for 循环将会非常

方便。基于容器的 for 循环是对一个线性容器中的元素一一进行遍历，而不需要循环控制变量，也就不需要循环变量的初值和终值。下面介绍其两种用法。

1. 在知道容器名时的基于容器的 for 循环格式

```
for (元素类型 &元素 : 容器名) {
      操作元素的语句
}
```

【代码 3-12】 使用基于容器 for 循环的函数 dispPrimeSequence( )。

```
void dispPrimeSequence( ){
  std::cout << lowerNaturalNumber << "到" << upperNaturalNumber << "之间的素数序列为: \n";
  for(int &x: prime)
    std::cout<< x <<",";
)
```

说明：

（1）这里 x 是一个桩变量，仅代表容器 prime 中的一个元素，其类型为 int。当这个 for 执行时，它会代表第一个元素，以后逐一移向下一个元素，知道元素结束。所以，其名字是无所谓的。

（2）在一般情况下，容器元素可以是简单变量，也可以是对象。并且可以对元素进行操作，改变其值。如果只简单地引用元素的值，不改变元素的值，则可以不使用元素的引用，即省略&操作符。

2. 在只知道容器内元素值而不知道容器名时基于容器的 for 循环格式

```
for (元素类型 元素 : {元素列表}) {
      操作元素的语句
}
```

【代码 3-13】 使输出一组已知的素数序列代码。

```
  for(int x: {2,3,7,11,13,17,19,23,29})
    std::cout<< x <<",";
)
```

3.4.6 数组 prime 的声明

上面的两个函数都设计好了。但是，还有一个关键问题没有解决，这就是数组 prime 的声明问题。为了声明数组 prime，需要解决两个问题：一个是声明在什么位置，另一个是如何确定数组的大小。

1. 数组 prime 声明语句的位置

数组 prime 的特殊性在于它是一个被多个成员函数共享的数组。在一般情况下，为函数

共享的数组，有两种设计考虑。

（1）将数组 prime 声明为 PrimeGenerator 的一个数据成员。在面向对象的程序中，由于一个类的数据成员可以被类的成员函数共享，因此将数组 prime 声明为类 PrimeGenerator 的一个成员比较合乎逻辑。但是，由于数组 prime 的大小在随对象而变化，用一个变量作为数组大小，又为 C++所不允许。这就破坏了类定义的抽象性。

（2）将数组 prime 声明在主函数 main()之内，这时，函数 getPrimeSequence 和 dispPrimeSequence 的头部要增加一个数组参数。

【代码 3-14】 增加了参数的函数 getPrimeSequence 和 dispPrimeSequence 的头部。

```
void getPrimeSequence(int prime[]){
…
}
void dispPrimeSequence(int prime[]){
…
}
```

【代码 3-15】 在主函数中声明数组的代码。

```
#include "prime02.h"

int main(){
  int prime0[1000]{};          //声明数组 prime0，并将全部元素初始化为 0
  PrimeGenerator psg0(3,1000);
  psg0.getPrimeSequence(prime0);
  psg0.dispPrimeSequence(prime0);

  int prime1[20]{};            //声明数组 prime1，并将全部元素初始化为 0
  PrimeGenerator psg1(3,1000);
  Psg1.getPrimeSequence(prime1);
  Psg1.dispPrimeSequence(prime1);

  return 0;
}
```

（3）将数组 prime 声明为外部的（在所有函数——包括主函数 main()之外），供各函数共享。这时，函数 getPrimeSequence 和 dispPrimeSequence 如代码 3-10 和代码 3-11（或代码 3-12）。

2. 数组 prime 的大小问题

在本例中，给定的自然数区间大小会根据要求不断变化，而且人们还没有找到自然数区间中素数数目的规律。此外，C++只允许用编译时的常量来声明数组大小。因此，数组 prime 声明时的大小只能使用一个估计的整数。通常可以定义为数据序列段的上界。但是，这样每生成一个对象都需要修改数组 prime 的大小，否则不是太浪费内存空间，就是有时会形成数组越界。因此，在客户端主函数中进行数组声明是比较合适的选择。这样，既能符合 C++的语法规则，又有一定灵活性。

3. 本例的完美解决

数组是程序设计语言中的传统机制。它可以方便地组织同类有顺序关系的一组数据。但是，它也有许多限制。因此现代 C++已经在开始淡化数组，推荐使用更为方便的 vector。

vector 不需要预先定义大小，再配以基于容器的 for，将为程序设计带来极大方便。关于 vector 将在第 3 篇中介绍。那时，才会找到本例的完美解决方案。

3.5 string 类型

string 称为字符串类型。与 int、float、double、char 等 C++内置的基本数据类型不同，string 不是基本类型，而是系统定义在文件<string>中的一个关于字符串类型的名字。

除了作为关键字的名字，C++要求程序中使用的其他任何名字都必须先定义，才可以使用。但是，一般说来，程序员并不知道也不需要知道它是如何定义的，为此只要把定义它的头文件（对于 string，头文件是<string>）包含在这个程序中就可以了。包含命令为“#include”。如使用命令“#include <string>”就可以在编译预处理时将定义 string 的文件<string>包含进来，当然，string 的定义也就包含进来了。

作为一个应用类，string 提供了丰富的成员函数。有关细节将在第 8.4 节介绍。

3.6 知 识 链 接

3.6.1 C++操作符

1．C++操作符分类

C++从 C 语言中继承了丰富的操作符（详见附录 B）。这些操作符可以从不同的角度进行分类。

（1）按照操作功能，可以分为如下类型。

① 算术操作符，如+(正)、–（负）、+、–、*、/、%等。

② 关系与判等操作符，如>、>=、==、<=、<、!=。

③ 逻辑操作符，如&&（与）、||（或）、!（非）。

④ 赋值操作符，如=。

⑤ 成员操作符，如.(直接成员操作符)。

⑥ 函数调用操作符，如（）。

⑦ 插入、提取操作符，如<<、>>。

⑧ 其他，如*、&等。

（2）按照操作对象的数量，可以分为如下类型。

① 一元操作符，如+(正)、–（负）、!、*、&等。

② 二元操作符，如+、–、*、/、%、>、>=、==、<=、<、!=、&&、|、=等。

③ 三元操作符，只有条件操作符?:。

2．C++操作符与数据类型

数据类型决定了数据可以施加的操作的类型，即每一种操作符都有其适合的数据类型。因此，编译器进行数据类型检测时，就包括了判断哪种操作符可以应用到哪一种数据。

表 3.1 列出了一些常用操作符与常用数据类型之间的搭配关系。

表 3.1　常用操作符与常用数据类型之间的搭配关系

操作符 \ 数据类型	bool	char	int	double	string
=	√	√	√	√	√
>、>=、==、<、<=、!=	√	√	√	√	√
>>、<<	√	√	√	√	√
+、+=		√	√	√	√（连接）
–、*、/、++、--、+=、–=、*=、/=		√	√	√	
%		√	√		

说明：这些搭配关系是 C++预定义的。由于类也是类型，是程序员自己定义的类型，为了将这些预定义的操作符应用到程序员自己定义的类型上，C++提供了一种操作符重载的机制。有关内容会在后面介绍。

3. “++”的前缀形式与后缀形式

增量操作符有两种形式：前缀形式和后缀形式。

【代码 3-16】 i ++与++ i 的比较。

```
#include <iostream>

int main(){
    int i1 = 1, i2 = 1;
    std::cout << "++ i1 = " << ++ i1  << std::endl;
    std::cout << "i2 ++ = " << i2 ++ <<  std::endl;
    //std::cout << "i2 ++ = 5" << i2 ++ = 5 << std::endl;
    return 0; return 0;
}
```

测试结果如下。

```
++ i1 = 2
i2 ++ = 1
```

但若去掉注释，将出现如下错误。

```
[Error] invalid operands of types 'int' and '<unresolved overloaded function type>' to binary
'operator<<'
```

说明：前缀增量操作符是先增量后使用，如++ i 执行的过程是 i = i + 1，即最后返回的是 i。后缀增量操作符是先使用后增量，如 i++执行的过程是：先产生一个临时变量记录 i

的值，再执行+1，最后返回的是临时变量的值，而最后相当于一个表达式 i + 1。

3.6.2 左值表达式与右值表达式

左值（L-value）和右值（R-value）是两个关于表达式属性的基本概念。在程序设计的早期，它们只是一个基于内置赋值运算符需求的概念。作为赋值操作符左操作数的表达式称为左值，作为赋值操作符右操作数的表达式称为右值。例如对于含有 a、b、c 这 3 个变量的表达式 a = b + c，以赋值操作符为界，左侧是一个变量 a，右侧是两个变量相加的表达式 b + c。显然，只有变量才能放在赋值操作符的左侧，不能把一个常量和运算式放到赋值操作符的左侧。所以 L 和 R 被解释为：left 和 right。

随着程序设计的广泛应用，对于表达式的理解也随之得到了深化，人们逐渐赋予了左值性和右值性越来越多的意义。于是，对其解释就从基于内置赋值运算符需求升华到了表达式求值过程中数据实体的性质上。有些表达式执行结束后得到的是依然存在的持久对象；而有些表达式在执行中虽然会形成一些临时实体，但在表达式执行结束后，这些临时实体也就随之被撤销了。例如下面的函数。

```
int returnRvalue(int a,int b){
  return a + b;
}
```

当用表达式“c = returnRvalue(2,3)”调用时，该函数会先生成一个临时实体，并在该值返回后再撤销。这种现象的影响和意义远大于基于内置赋值运算符需求的意义。于是，人们又把实体分为两类：永久实体和临时实体。只有永久对象可以作为左值，临时对象只能作为右值。遂之将“L”和“R”重新解释为：location（位置）和 read（读）。因为只有永久实体才可以进行取地址操作，临时实体则不可以。基于这一点，人们给出了一个简洁的左值判别方法：看能不能对表达式取地址。如果能，则为左值；否则为右值。例如，“a > b ? a : b”就是一个左值表达式，因为它执行后，存在的不是 a 就是 b，因此可以进行取地址操作，所以它是左值表达式。而表达式 a + b 就只能作为右值表达式，不能进行取地址操作。下面给出一些例子来进行说明。

```
int a = 10;
int b = 20;
int *pFlag = &a;
string str1 = "hello ";
string str2 = "world";
```

基于上述定义，下面给出哪些是左值、哪些是右值的实例。

（1）a 和 b 都是持久对象（可以对其取地址），是左值。

（2）a + b 是临时对象（不可以对其取地址），是右值。

（3）a ++是先为持久对象 a 创建一个副本，再使持久对象 a 的值加 1，最后返回那份副本，而那份副本是临时对象（不可以对其取地址），是右值。

（4）++a 相当于 a = a + 1，返回的是持久对象 a 增 1 后的值（可以对其取地址），是左值。

（5）pFlag 和*pFlag 都是持久对象（可以对其取地址），是左值。

（6）10、20 和"hello"都是字面量——纯常量（不可以对其取地址），是右值。

（7）string("hello")是临时对象（不可以对其取地址），是右值。

（8）str1 是持久对象（可以对其取地址），是左值。

（9）str1 + str2 是调用了+操作符，而+操作符返回的是一个 string（不可以对其取地址），故其为右值。

然而，随之又出现了一些问题。例如，一个变量用 const 修饰，值就不可被改变。为此，又把左值分为可修改的左值和不可修改的左值。只有可修改的左值才可以用于赋值表达式的左边。

3.6.3 具有副作用的表达式与序列点

1. 表达式的副作用

表达式是 C++程序中关于数据的一种存在与表示形式，即表达式的本质是求值。但是，如果表达式在求值的过程中对环境进行了修改，这就产生了副作用（side effect）。最常见的副作用就是表达式在求值的过程中对变量的值进行了修改。例如：

```
int a = 3, b = 1;
a = 5;                //修改了 a 的值
b = a = 6
```

这里，表达式 a = 5 的本质作用是求值，即应当得到这个表达式的值 5，但是求这个值的同时产生了一个副作用——把变量 a 的值修改了。同样，表达式 b = a = 6 被解释为 b = (a = 6)，即表达式 a = 6 的求值结果为 6。但是，在这个求值过程中，却产生了两个副作用：将变量 a 的值由 5 修改为了 6，把 b 的值也修改为了 6。在后面将会看到，赋值表达式的副作用在某些情况下会给计算带来莫名其妙的结果。

2. 未定义的子表达式求值顺序

前面介绍的优先级、结合性和强制结组，是影响表达式求值顺序的最普遍规则。但是，对于一个表达式中的子表达式的求值顺序，C++是没有定义的。例如，在表达式 int a = f(x) + g(y)中，C++没有定义是先进行 f(x)的求值，还是先进行 g(y)的求值。

再如，在表达式(a = a + b) / (a = a – b)中，C++也没有定义是先进行 a = a + b 的求值，还是先进行 a = a – b 的求值。于是，会有下述两种情况出现。

（1）若先对 a = a + b 求值，并且变量 a 与 b 的值均为 5，则可以得到 10 / 5，即 2。

（2）若先对 a = a – b 求值，并且变量 a 与 b 的值均为 5，则可以得到 5 / 0，将出现异常。

3. 副作用的完成时间与序列点

要想规避未定义行为 + 副作用引起的麻烦，不能仅以编译器来检查，因为它是实现定义行为的；也不能依靠测试发现，因为它不是逻辑错误。唯一能做的是规范操作符本身的行为。为了做到这一点，就要明确一个操作符所有的副作用在何时完成：有些操作符的副

作用是在主操作的同时完成的，如=、前缀++、前缀– –；而有些操作符的副作用是在该操作符的主作用完成之后完成的，如后缀的++和– –。但是，这些之后完成的副作用要之后到什么时刻呢？标准没有具体说明。不过程序语言通常都规定了副作用完成的最晚实现时刻，称为序列点（sequence point）、序点、时间点、顺序点及执行点。C++要求在序列点上，该点之前所有运算的副作用都应该结束，并且后继运算的副作用还没有发生。

C++的主要序列点有如下几个。

（1）函数调用时，实际参数求值完毕，函数被实际函数调用前。

（2）操作符&&、||、？：中的？和逗号操作符的第一个运算对象计算之后。

（3）完整表达式（full expression）操作结束的时间点是序列点。完整表达式不是子表达式，而子表达式就是表达式中的表达式。下面的表达式是完整表达式。

① 初始化表达式。

② 表达式语句中的表达式。

③ 选择语句（如 if 或 switch）的控制表达式。

④ while 或 do 语句中的控制表达式。

⑤ for 语句头部的 3 个表达式。

⑥ return 语句中的表达式。

（4）完整的声明结束时。

（5）库函数即将返回之时。

4. 序列点对于求值顺序的影响

有了序列点概念，编译器在决定一个可能含有序列点的表达式求值的计算顺序时，要先考虑序列点，再根据优先级别和结合性决定。规则如下。

（1）对于一个序列点，要先对其左侧的表达式求值。

（2）当一个表达式中有多个序列点时，先对哪个序列点进行考虑，还是未定义行为。

3.6.4 表达式类型的推断与获取：auto 与 decltype

C++是一种强类型语言，它要求每个表达式都有特定的类型，并且要求声明变量或对象时，必须指定其类型。但是有时并不需要一定要明确是什么类型，也有时需要用一个已有表达式的类型去定义一个变量。为此，C++11 提供了两个关键字 auto 和 decltype，来要求编译器自动推断表达式和获取一个表达式的类型。

1. auto

（1）auto 语义的变化。在 C++11 之前，auto 关键字用来指定变量的存储期属性。在 C++11 中，它成了一个类型的占位符，其基本功能是通知编译器去根据初始化代码推断所声明变量的真实类型。　　例如：

```
auto i = 42;                //i 是 int 类型
auto l = 42LL;              //l 是 long long 类型
auto emp = new Employee();  //emp 是 Employee 类型
```

```
auto s("hello");                    //s 的类型为 string
```

需要说明，在 C++14 中，auto 的作用进一步扩大，其中一个重要的作用是函数返回类型推导。这意味着，程序员写函数时，不一定非要写一个具体的返回类型，只要写一个 auto 就可以了。这种机制实际上是将类型安全机制交由编译器承担。

（2）auto 关键字与基于范围的 for 循环经常会配合，使用在不明确容器类型的情况下。

【代码 3-17】 基于容器的 for 与 auto 配合应用示例。

```
#include <iostream>
using namespace std;

int main(){
   int a[10] = { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };

   for( auto x : a ) {              //用 auto 关键字推断元素类型
      cout << x << " ";
   }
   cout << endl;

   for( auto &x : a ) {             //使用引用以修改元素的值
       x -= 10;
   }
   cout << endl;

   for(auto &x : a ) {
      cout << x  << " ";
   }
  return 0;
}
```

输出如下。

```
11 12 13 14 15 16 17 18 19 20
1 2 3 4 5 6 7 8 9 10
```

2. decltype

decltype 是一个操作符，它可以获取一个表达式的类型，格式如下：

```
decltype (表达式);
```

decltype 的用途是用获取的类型，来声明一个变量或对象。例如在下面的语句中

```
decltype(f()) sum = x;              //sum 的类型就是函数 f 的返回类型
```

编译器并不实际调用函数 f( )，仅使用当调用发生时 f( )的返回值类型作为 sum 的类型。即编译器为 sum 指定的类型就是假如 f( )被调用将会返回的那个类型。所以，decltype 关键字的作用是推导表达式的类型。

说明：decltype 的推导结果会与给定的表达式的特征有关。因此，decltype 在返回类型

前，先要对所操作的表达式进行分类推导。粗略地说，主要规则有如下几条。

（1）如果所操作的表达式是标识符表达式（id-expression，即指向一个局部变量、命名空间作用域变量、静态成员变量及函数参数）或类成员访问表达式，则返回的就是这些标识符被声明的类型，记作 T。

（2）如果所操作的表达式是一个左值表达式（不包括标识符表达式和类成员访问表达式），则返回的就是这些表达式类型的引用，记作 T&。

（3）如果所操作的表达式是一个右值表达式（字面量或临时对象），则返回所操作表达式的实际类型，记作 T。

（4）有些情况下返回 T&&（右值引用）。将在 9.4 节介绍。

例如，对于下面的声明

```
const int& foo();
const int bar();
int i;
struct A {double x ;};
const A* a = new A();
```

有如下一些推导结果：

```
decltype(foo()) x1;       //类型为 const int&，规则（1）
decltype(foo()) x2;       //类型为 int，规则（1）
decltype(a -> x) x4;      //类型为 double，规则（1）
decltype(a -> x) x5;      //类型为 const double&，规则（2）
decltype(i) x3;           //类型为 int，规则（1）
decltype((i)) x6;         //错误：推导出(i)的类型为 int &，欲将 b 定义为 int &，但无初始化
```

可以看出，一个标识符表达式或类成员访问表达式加上括号后，就不再是标识符表达式或类成员访问表达式。

C++11 校验表如下。

```
if（一个操作数为 long double 类型）
   另一个操作数转换为 long double 类型;
else if（一个操作数为 double 类型）
   另一个操作数转换为 double 类型;
else if（一个操作数为 float 类型）
   另一个操作数转换为 float 类型;
else if（一个操作数为 float 类型）
   另一个操作数转换为 float 类型;
else   //执行整型提升
{
   if （两个数都是有符号或都是无符号）
        将级别低的转换为级别高的类型;
   if （一个是有符号低级别，另一个是无符号高级别）
        将级别低的有符号操作数转换为无符号操作数所属类型;
   else if （有符号类型可表示无符号类型所有可能取值）
        将无符号操作数转换为有符号操作数所属类型;
   else
```