第10章

函数的高级应用

本章承续第 6 章，介绍函数的重载、参数的默认值、模板函数等等让 C++ 的函数在使用上更方便的语法。此外，我们也讨论了几个非常实用的问题，包括如何取得随机数，递归函数的设计以及排序与搜索的相关算法。

10.1 函数的重载

当同样的函数名称被重复用于不同函数的声明和定义时，称为重载（overloading）。只要编译器能从参数的数据类型或数目清楚地区别要调用的是哪个函数，重载是一个很方便的程序写作方式，因为我们不必为功能类似的函数取不同的名称，在使用时也比较单纯。

C++ 能提供函数重载的原因，是因为它是一种对数据类型 (data type) 的检查和控制非常严格的语言。对于编译器而言，只要参数不相同，不管是数目不一样还是数据类型不同，都视为不同的函数。

例如：

```
int Max(int, int);
float Max(float, float);            // 参数的数据类型不同
```

以及

```
double Area(double x);
double Area(double x, double y);  // 参数的数目不一样
```

在程序中都可以因为调用时所使用的参数不同而调用正确的函数。

在下面的完整程序 Overload.cpp 中，我们以参数的数量不同，来区别所调用的是哪个函数。函数 Max() 有两个版本，分别为两个参数和三个参数，编译器可以正确区别。

范例程序 文件 Overload.cpp

```
// Overload.cpp
#include <iomanip>
using std::cin;
using std::cout;
using std::endl;
using std::setw;
// 函数的原型
float Max(float, float);
float Max(float, float, float);  // 参数的数目不一样
//------- 主程序 ---------------------------------
int main()
```

```
    {
     float a=5.0, b=6.0, c=7.0;
     cout << "Max(a, b)    的值是: " << Max(a, b) << endl;
     cout << "Max(a, b, c) 的值是: "
          << Max(a, b, c)  << endl;
     return 0;
    }
// --- 函数的定义 ----------------------------------
float Max(float x, float y)
  { return (x>=y) ? x : y;}
float Max(float x, float y, float z)
  {
   float Temp = x;
    if (x<y) Temp = y;
    if (Temp<z) Temp = z;
    return Temp;
  }
```

程序执行结果

```
Max(a, b)    的值是: 6
Max(a, b, c) 的值是: 7
```

10.2　参数的默认值

函数的参数可以指定默认值，称为默认参数 (default arguments)。例如，有一个函数 Area()，它的第二个参数的默认值为 12.0:

```
float Area(float Width, float Length = 12.0);
```

而且它的定义为两个参数的乘积:

```
float Area(float Width, float Length)
```

```
{ return Width*Length; }
```

如果调用时没有写出第二个参数的值:

```
A = Area(6.5);  // 只用一个参数调用
```

则会以默认值来设定第二个参数，亦即 A 的值为 6.5*12.0．如程序 Default.cpp 所示：

范例程序 文件 Default.cpp

```
// Default.cpp
// --- 声明函数 Area() --------------------
float Area(float Width, float Length = 12.0);
// --- 主程序 ----------------------------
int main()
 {
   float A;
   A = Area(6.5);  // 只用一个参数调用
   return 0;
 }
// --- 定义函数 Area() --------------------
float Area(float Width, float Length)
 { return Width*Length; }
```

讨 论

虽然在主程序中只用一个参数调用函数 Area( )，第二个参数自动会被默认值 12.0 补足，因此 A 的值是 6.5*12.0 = 78.

默认值只能在函数声明时指定或是函数定义放在声明位置的时候．但不可以同时在函数的声明式和定义式中指定．为了让编译器能正确区分，在参数列中有默认值的参数必须放置

于所有其它参数的后面，不可以混杂。此外，在调用时，所有使用默认值的参数之后也必须要使用默认值。例如，有一个函数的声明式为：

```
void Func (float, float x = 0,int n = 5,char = "r");
```

则下面的调用都是正确的

```
Func(1.5);
Func(3.2, 1.0);
Func(0.0, 1.0, 10, 'a');
```

但不可以有下列的调用方式：

```
Func(1.0, 2.0, 'a');          // 错误!
```

此外，固然我们可以同时使用函数的重载和参数指定默认值的功能，但是如果重载时只以参数的数量来区别，则容易发生无法分辨的错误。

例如，以下是两个具有默认参数的重载函数 Func()之原型：

```
int Func(int n, int m = 0, float x = 1.0);
int Func(int n, int m);
```

这两个函数无法用下列调用语句正确区分：

```
M = Func(2,3);          // 错误!
```

在编译时将无法通过。

10.3　模板函数

模板函数（Function Template）是用来自动产生许多具有共同程序算法的函数之模板。这些函数将具有共同的逻辑和操作步骤，但其输入和输出的数据类型可以不一样。

例如：

```
template <class T>
T Sum(T x, T y)
   {return x+y;}
```

相当于同时声明了许多名称都叫做 Sum() 的函数，但其输入和输出的数据类型未定，可以是 int, float, double 或 char，真正的函数定义要由编译器判定实际使用的数据类型后才能完成。在上式中，“**template**”和 “**class**”都是 C++ 的关键词。“**template**”表示我们即将声明一个模板函数，而 “**class**”则用来声明以下的模板函数中所要使用的数据类型的暂定名称。理论上，任何名称都可以用来作为暂定的数据类型名称，但习惯上都使用英文字母 T.

模板函数内可以使用不只一种暂定的数据类型。

例如：

```
template <class T1, class T2, class T3>
T1 Func(T1 x,T2 y, T3 z)
{
    // … 函数内容
}
```

就一次使用了三种不同的数据类型于输入和输出的部分。模板函数常常可以用来取代函数的重载，而且更加简洁。例如：

```
template<class T>
T Abs(T x) {return (x>0)?  x : -x;}
```

就相当于同时定义了

```
int Abs (int x) {return (x>0)?  x : -x;}
float Abs (float x) {return (x>0)?  x : -x;}
double Abs (double x) {return (x>0)?  x : -x;}
```

三个函数。由于模板的参数数据类型未定，因此不可以有默认值。

提示

声明（declaration）和定义（definition）

我们曾一再强调 C++ 的变量必须先声明才能使用。例如“int N = 0;”这个语句就同时完成了数据类型为 int 的变量 N 之声明、定义和初始化（默认值）。除了 extern 变

量以外，一般变量的声明都同时完成了定义。

以编译器的观点而言，声明和定义是不同的动作。声明只是用来提供“如何安排某个变量或函数的存储空间”的信息。而定义则是确实完成存储空间的大小和位置的安排。依此了解，函数的原型（prototype）只是函数的声明。如果我们把函数的定义放在函数原型的位置，则相当于同时执行了函数的声明和定义。

模板函数形同一群重载函数的定义。模板函数里面必须包含完整的函数内容，只是数据类型暂时还不确定而已。置放模板函数的规则和函数原型相同，都位于主程序之前。

10.4　随机数的取得

随机数（random numbers）是随机变化，无法预测的数值。在设计程序时，我们有时候会需要用到随机数。举例来说，在仿真测量讯号的时候，常常需要故意加上噪声，以获得较逼真的效果。又譬如研究概率相关的掷骰子问题，也需要产生随机数，以仿真掷骰子时随机的特性。近年很多新的最佳化方法，譬如单纯形(我们即将在第 24 章介绍)，仿真退火(simulated annealing)和遗传算法（genetic algorithms），更是依赖随机数来寻找最佳答案。

伪随机数产生器

C++ 在 <cstdlib> 内建了一个伪随机数产生器（pseudo-random number generator）。“伪”这个字用来形容它无法用来产生真正的随机数，只是一个近似的随机数产生器。真正的随机数具有无穷大的周期，只能以概率来预测其值的分布状况。

要得到伪随机数可以调用函数 `rand()`或`_lrand()`:

- 函数 `rand()`　可以产生 0 到 RAND_MAX 的整数。
- 函数 `_lrand()` 可以产生 0 到 LRAND_MAX 的整数。

RAND_MAX 和 LRAND_MAX 都定义在 <cstdlib> 里面，RAND_MAX 的值通常为 0x7FFFU (32767)，而 LRAND_MAX 的值通常为 0x7FFFFFFFU (2147483647)。

此外，<cstdlib> 里面还有一个专用的函数 `randomize()`用来给予这个伪随机数产生器一个临时的初始值 (称为种子，seed)。如果没有先调用这个函数，每一次执行的时候都会产生同一个序列的随机数。函数 `randomize()`的作用机制是读取计算机内部时钟的数值作为产生随机数的基础。因为每一次执行的时候时钟的数值都不相同，因此可以产生不重复的随

机数.

提示

伪随机数产生器的初始值还可以用函数 srand( ) 来给予. 使用 srand( ) 时, 必须自行给予一个 int 参数. 例如

```
srand(time(0));
```

的效果就和“randomize( );”一样, 取用计算机内部时钟的数值作为参数. 但是这个时候必须在程序开头处加入前置处理指令“#include <ctime>”.

产生指定范围内的整数随机数

为了获得一个介于 0 到 N-1 之间的整数随机数 (integer random number), 有两种做法:

```
rand()%N
random(N)
```

其中函数 random()的内部事实上进行的是“_lrand()%N”的运算, 因此两者都是通过余数运算“%”来产生指定范围内的整数随机数.

一般而言, 如果给定 N, 我们需要的是介于 1 和 N 之间的整数随机数. 因此, 我们可以把上述函数调用稍作变动,并包装成以下两个 inline 函数以方便使用 (这两个函数功能相同):

```
inline int RandI(int N) {return random(N)+1;}
inline int RandI2(int N){return 1 + rand()%N;}
```

在以下的范例程序 TestDice.cpp 中, 我们以 inline 函数 RandI()为基础, 写成一个函数 TestDice() 以仿真掷骰子的现象 (骰子的值为整数 1~6).

首先, 我们通过程序检查 RAND_MAX 和 LRAND_MAX 的大小, 接着, 仿真连掷 20 次

骰子的过程．最后，统计连掷 6000 次骰子的结果，以查看各点数出现的次数是否均匀，验证伪随机数产生器的性能．

范例程序　文件 TestDice.cpp

```
// TestDice.cpp
#include <iomanip>
#include <cstdlib>
using namespace std;
// -- 定义 inline 函数 RandI() 以产生 1~N 之间的随机数 --
inline int RandI(int N) {return random(N)+1;}
//------- 声明函数 TestDice() ----------------------
void TestDice();
const int TestNum = 6000;
//-------  主程序 ---------------------------------
int main()
{
  cout << setiosflags(ios::right)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << std::setprecision(4);
  cout << "RAND_MAX  (0x7FFFU) 的值是 :"
       << setw(7) << RAND_MAX  << endl;
  cout << "LRAND_MAX (0x7FFFFFFFU) 的值是 :"
       << setw(7) << LRAND_MAX << endl;
  TestDice(); return 0;
}
//------- 定义函数 TestDice() ---------
void TestDice()
{
  int Freq[6], Face, i;
  for (i=0; i<6; i++) Freq[i]=0;     // (1) 初始化
  randomize();
  // (2) 连掷 20 次
  cout << "连掷 20 次的结果: " << endl;
```

```
    for (i=1; i<=20; i++)
      {
        cout << setw(5) << RandI(6);
        if (i%5 == 0) cout << endl;
      }
    cout << endl;
    // (3) 统计连掷 TestNum 次的结果
    for (int Roll=0; Roll< TestNum; Roll++)
      {
        Face = RandI(6);
        Freq[Face-1]++;
      }
    cout << "   点数    次数   "      << endl;
    cout << "------------------ " << endl;
    for (i=0; i<6; i++)
      cout << setw(5)  << (i+1)
           << setw(10) << Freq[i]  << endl;
    cout << "------------------ "  << endl;
}
```

程序执行结果

```
RAND_MAX  (0x7FFFU) 的值是：32767
LRAND_MAX (0x7FFFFFFFU) 的值是：2147483647
连掷 20 次的结果：
    1    1    4    3    4
    2    6    2    5    1
    4    6    1    4    5
    2    2    3    1    4

  点数    次数
------------------
    1      1002
    2      1027
    3      1005
```

```
4       980
5       972
6      1014
------------------
```

讨 论

由仿真连掷 6000 次骰子的统计结果，可以知道各点数都出现 1000 次左右，因此，这个 C++ 伪随机数产生器的性能基本上可以接受。

随机获得一个介于 0 到 1 之间的浮点数

为了随机获得一个介于 0 到 1 之间的浮点数。只要以 rand() 产生的整数除以 RAND_MAX，或以 _lrand() 产生的整数除以 LRAND_MAX 就可以。

我们可以把这个计算过程包装成以下三个 inline 函数以方便使用 (这三个函数的功能相同)：

```
inline double Rand() {return double(rand())/RAND_MAX;}
inline double Rand2(){return double(_lrand())/LRAND_MAX;}
inline double Rand3()
   {return double(random(LRAND_MAX)+1)/LRAND_MAX;}
```

在以下的范例程序中，我们以 inline 函数 Rand()为基础，写成一个函数 TestRand()以仿真随机产生介于 0 到 1 之间的浮点数。

首先，我们通过程序仿真连续随机产生 20 个浮点数的过程。接着，统计连续产生 6000 次的结果，以查看各个范围内出现的次数是否均匀(每个范围间隔 0.1)。最后，将 6000 次的结果平均 (理想值为中间值 0.5)，以验证伪随机数产生器的性能。

范例程序 文件 TestRand.cpp

```
// TestRand.cpp
#include <iomanip>
#include <cstdlib>
using namespace std;
// 定义 inline 函数 Rand() 产生 0 ~ 1 之间的随机数
inline double Rand()
      {return double(rand())/RAND_MAX;}
//------- 声明函数 TestRand() ---------
void TestRand();
const int TestNum = 6000;
//------- 主程序 ----------------------
int main()
{
  cout << setiosflags(ios::right)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << std::setprecision(4);
  TestRand(); return 0;
}
//------- 定义函数 TestRand() ---------
void TestRand()
{
  int Freq[10];
  double Sum, Temp;
  // (1) 初始化
  randomize();      // 也可写成 srand(time(0));
  for (int i=0; i<10; i++) Freq[i]=0;
  // (2) 连试 20 次 Rand()
  cout << "连试 20 次 Rand(): " << endl;
  for (int i=1; i<=20; i++)
    {
      cout << setw(10) << Rand();
      if (i%5 == 0) cout << endl;
    }
```

```
    cout << endl;
    // (3) 统计连试 TestNum 次的结果
    Sum = 0.0;
    for (int Roll=0; Roll< TestNum; Roll++)
    {
      Temp = Rand();
      for (int i=1; i<=10; i++)
        if ((Temp<=i*0.1) && (Temp>(i-1)*0.1))
          Freq[i-1]++;
      Sum += Temp;
    }
    cout << "   范围                    次数   " << endl;
    cout << "---------------------------------- " << endl;
    for (int i=1; i<=10; i++)
       cout << ((i-1)*0.1) << " < Rand() <= "
            << (i*0.1)
            << setw(6)    << Freq[i-1] << endl;
    cout << "---------------------------------- " << endl;
    cout << "平均: " << Sum/double(TestNum) << endl;
    return;
}
```

程序执行结果

```
连试 20 次 Rand():
   0.1886    0.3008    0.6612    0.6320    0.0216
   0.1375    0.7226    0.3215    0.0080    0.6875
   0.7928    0.5995    0.7400    0.4369    0.8018
   0.9930    0.6493    0.0373    0.8551    0.6647

          范围                    次数
---------------------------------
0.0000 < Rand() <= 0.1000   605
0.1000 < Rand() <= 0.2000   601
0.2000 < Rand() <= 0.3000   560
```

```
0.3000 < Rand() <= 0.4000   600
0.4000 < Rand() <= 0.5000   602
0.5000 < Rand() <= 0.6000   648
0.6000 < Rand() <= 0.7000   550
0.7000 < Rand() <= 0.8000   643
0.8000 < Rand() <= 0.9000   608
0.9000 < Rand() <= 1.0000   583
---------------------------------
平均: 0.5014
```

10.5 递归函数

递归 (recursion) 在数学和信息科学都是一个很基本的概念。在数学上，递归函数的定义式使用相同的函数来定义自己，正如同程序语言的递归函数 (recursive function) 调用自己一样。递归函数不能一直调用自己，否则程序无法结束，因此递归函数必须要有一个终结条件 (termination condition) 让函数不再调用自己 (对于数学函数而言，则表示不再由自己定义)。非常多的数据处理程序都可以转化为递归程序。

阶乘

最为人熟知的递归函数就是正整数的阶乘 (factorial)。 n 的阶乘，写成 n！的定义式为

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1$$

$$= \begin{cases} 1, & \text{如果 } n = 0 \\ n \cdot (n-1)!, & \text{如果 } n \geq 1 \end{cases}$$

可以进一步写成递归函数 `Factorial()`，列于文件 Factorial.cpp 中。

范例程序　文件 Factorial.cpp

```
// Factorial.cpp
#include <iomanip>
#include <cstdlib>
using std::cin;
using std::cout;
using std::endl;
// --- 声明递归函数 Factorial() --------------------
int Factorial(int);
// ---- 主程序 -------------------------------------
int main()
{
   int N;
   cout << "请输入一个 12 以下的正整数: ";
   cin >> N;
   if ( N < 0 )
      cout << "错误! 您输入了负数." << endl;
   else
      cout << N << " ! = " << Factorial( N ) << endl;
   return 0;
}
// --- 定义递归函数 Factorial() ---------------------
int Factorial(int N)
{
   if ( N <= 1 )
     return 1;
   else
     return N * Factorial(N-1);    // 调用自己!
}
```

程序执行结果

```
请输入一个 12 以下的正整数: 10
10 ! = 3628800
```

讨 论

(1)所有的递归函数都可以改写成循环的形式。例如 Factorial() 可以写成

```
int Factorial2( int N )
{
   int Temp=1;
     for (int i=1; i<=N; i++)
     Temp *= i;
   return Temp;
}
```

计算结果完全相同。

(2)由于 int 数据类型通常为 32 bits，一个 int 数据所能表达的最大正整数为 2,147,483,647，而 13 的阶乘就已经是 6,227,020,800，因此，这个程序所能输入的最大正整数为 12。如果要计算更大的整数，可以将数据类型 int 改为 double。此外，递归函数的处理需要用到堆栈（stack，详见 6.1 节的说明），因此能够递归调用的次数受到可用的堆栈大小的限制。

最大公约数

最大公约数 (Greatest Common Divider，简写为 GCD) 的计算，也可以写成递归函数。正整数 M 和 N 的最大公约数，写成 GCD(M,N)，的定义式为

$$GCD(M,N) = \begin{cases} N, & \text{如果N整除M} \\ GCD(N, M\%N), & \text{如果N无法整除M} \end{cases}$$

可以使用 C++写成递归函数 GCD()，列于文件 GCD.cpp 中。

范例程序　文件 GCD.cpp

```
// GCD.cpp
#include <iomanip>
#include <cstdlib>
using namespace std;
// --- 声明函数 GCD() ---------------------------
int GCD(int, int);
// ---- 主程序 ---------------------------------
int main()
{
   int Num1, Num2;
   cout << "请输入第一个正整数 (共两个): " << endl;
   cin  >> Num1;
   cout << "请输入第二个正整数 (共两个): " << endl;;
   cin  >> Num2;
   cout << Num1 << " 和 " << Num2 << " 的最大公约数是 "
        << GCD(Num1, Num2);
}
// --- 定义函数 GCD() --------------------------
int GCD(int M, int N )
{
   if ( (M%N) == 0 )
     return N;
   else
     return GCD(N, M%N);
}
```

程序执行结果

```
请输入第一个正整数 (共两个):
96
请输入第二个正整数 (共两个):
36
96 和 36 的最大公约数是 12
```