第5章　函数重载与缺省参数

能使名字方便使用，是任何程序设计语言的一个重要特征。

当我们创建一个对象（即变量）时，要为这个存储区取一个名字。一个函数就是一个操作的名字。正是靠系统描述各种各样的名字，我们才能写出易于人们理解和修改的程序。这在很大程度上就像是写散文——目的是与读者交流。这里就产生了这样一个问题：如何把人类自然语言的有细微差别的概念映射到编程语言中。通常，自然语言中同一个词可以代表许多种不同的含义，这要依赖上下文来确定。这就是所谓的一词多义——该词被重载了。这点非常有用，特别是对于细微的差别。我们可以说“洗衣服，洗汽车”。如果非得说成“洗（洗衣服的洗）衣服，洗（洗汽车的洗）汽车”，那将是很愚蠢的，就好像听话的人对指定的动作毫无辨别能力一样。大多数人类语言都是有冗余的，所以即使漏掉了几个词，我们仍然可以知道话的意思。我们不需要单一的标识——而可以从上下文中理解它的含义。

然而大多数编程语言要求我们为每个函数设定一个唯一的标识符。如果我们想打印三种不同类型的数据：整型、字符型和实型，我们通常不得不用三个不同的函数名，如 print_int()、print_char()和print_float(),这些既增加了我们的编程工作量，也给读者理解程序增加了困难。

在C++中，还有另外一个原因需要对函数名重载：构造函数。因为构造函数的名字预先由类的名字确定，所以只能有一个构造函数名。但如果我们想用几种方法来创建一个对象时该怎么办呢？例如创建一个类，它可以用标准的方法初始化，也可以从文件中读取信息来初始化，我们就需要两个构造函数，一个不带参数（缺省构造函数），另一个带一个字符串作为参数，以表示用于初始化对象的文件的名字。所以函数重载的本质就是允许函数同名。在这种情况下，构造函数是以不同的参数类型被调用的。

重载不仅对构造函数来说是必须的，对其他函数也提供了很大的方便，包括非成员函数。另外，函数重载意味着，我们有两个库，它们都有一个同名的函数，只要它们的参数不同就不会发生冲突。我们将在这一章中详细讨论这些问题。

这一章的主题就是方便地使用函数名。函数重载允许多个函数同名，但还有另一种方法使函数调用更方便。如果我们想以不同的方法调用同一函数，该怎么办呢？当函数有一个长长的参数列表，而大多数参数每次调用都一样时，书写这样的函数调用会使人厌烦，程序可读性也差。C++中有一个很通用的作法叫缺省参数。缺省参数就是在用户调用一个函数时没有指定参数值而由编译器插入参数值的参数。这样 f(“hello”),f(“hi”,1)和f(“howdy”,2,‘c’)可以用来调用同一函数。它们也可能是调用三个已重载的函数，但当参数列表相同时，我们通常希望调用同一函数来完成相同的操作。

函数重载和缺省参数实际上并不复杂。当我们学习完本章的时候，我们就会明白什么时候用到它们，以及编译、连接时它们是怎样实现的。

5.1　范围分解

在第2章中我们介绍了名字范围分解的概念（有时我们用“修饰”这个更通用的术语）。在下面的代码中：

```
void f();
class x {void f();};
```

类x内的函数f（）不会与全局的f（）发生冲突，编译器用不同的内部名 f（）（全局）和x::f()（成员函数）来区分两个函数。在第2章中，我们建议在函数名前加类名的方法来命名函数，所以编译器使用的内部名字可能就是 _f和_x_f。函数名不仅与类名关系密切，而且还跟其他因素有关。

为什么要这样呢？假设我们重载了两个函数名：

```
void print(char);
void print(float);
```

无论这两个函数是某个类的成员函数还是全局函数都无关紧要。如果编译器只使用函数名字的范围，编译器并不能产生单一的内部标识符，这两种情况下都得用 _print结尾。重载函数虽然可以让我们有同名的函数，但这些函数的参数列表应该不一样。所以，为了让重载函数正确工作，编译器要用函数名来区分参数类型名。上面的两个在全局范围定义的函数，可能会产生类似于_print_char和_print_float的内部名。因为，为这样的名字分解规定一个统一的标准毫无意义，所以不同的编译器可能会产生不同的内部名（让编译器产生一个汇编语言代码后我们就可以看到这个内部名是个什么样子了）。当然，如果我们想为特定的编译器和连接器购买编译过的库的话，这就会引起错误。另外，编译器在用不同的方式来产生代码时也可能出现这样的问题。

有关函数重载我们就讲到这里，我们可以对不同的函数用同样的名字，只要函数的参数不同。编译器会通过分解这些名字、范围和参数来产生内部名以供连接器使用。

5.1.1 用返回值重载

读了上面的介绍，我们自然会问：“为什么只能通过范围和参数来重载，为什么不能通过返回值呢？”乍一听，似乎完全可行，同样将返回值分解为内部函数名，然后我们就可以用返回值重载了：

```
void f();
int f();
```

当编译器能从上下文中唯一确定函数的意思时，如int x = f();这当然没有问题。然而，在C中，我们总是可以调用一个函数但忽略它的返回值，在这种情况下，编译器如何知道调用哪个函数呢？更糟的是，读者怎么知道哪个函数会被调用呢？仅仅靠返回值来重载函数实在过于微妙了，所以在C++中禁止这样做。

5.1.2 安全类型连接

对名字的范围分解还可以带来一个额外的好处。这就是，在 C中，如果用户错误地声明了一个函数，或者更糟糕地，一个函数还没声明就调用了，而编译器则按函数被调用的方式去推断函数的声明。这是一个特别严重的问题。有时这种函数是正确的，但如果不正确，就会成为一个很难发现的错误。

在C++中，所有的函数在被使用前都必须事先声明，出现上述情况的机会大大减少了。编译器不会自动为我们添加函数声明，所以我们应该包含一个合适的头文件。然而，假如由于某种原因我们还是错误地声明了一个函数，可能是通过自己手工声明，或包含了一个错误的头文件（也许是一个过期的版本），名称分解会给我们提供一个安全网，也就是人们常说的安全连接。

请看下面的几个例子。在第一个文件中，函数定义是：

```
//: DEF.CPP -- Function definition
  void f(int) {}
```

在第二个文件中，函数没有声明就调用了。

```
//: USE.CPP -- Function misdeclaration
void f(char);
main() {
//! f(1);  //Causes a linker error
}
```

即使我们知道函数实际上应该是f(int),但编译器并不知道，因为它被告知 (通过一个明确的声明) 这个函数是f(char)。因此编译是成功的，在C中，连接也能成功，但在C++中却不行。因为编译器会分解这些名字，这个函数的定义变成了诸如 f_int之类的名字，而使用的函数则是f_char。当连接器试图引用f_char时，它只能找到f_int，所以它就会报告一条出错信息。这就是安全连接。虽然这种问题并不经常出现，但一旦出现就很难发现，尤其是在一个大项目中。这是利用C++编译器查找C语言程序中很隐蔽的错误的一个例子。

5.2 重载的例子

现在我们回过头来看看前面的例子，这里我们用重载函数来改写。如前所述，重载的一个很重要的应用是构造函数。我们可以在下面的stash类中看到这点。

```
//: STASH4.H -- Function overloading
#ifndef STASH4_H_
#define STASH4_H_

class stash {
  int size;  // Size of each space
  int quantity; // Number of storage spaces
  int next; // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  stash(int Size); // Zero quantity
  stash(int Size, int InitQuant);
  ~stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH4_H_
```

stash()的第一个构造函数与前面一样，但第二个带了一个 Quantity参数来指明分配内存的初始大小。在这个定义中，我们可以看到quantity的内部值与storage指针一起被置零。

```
//: STASH4.CPP -- Function overloading
#include "..\4\stash4.h"
```

```
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

stash::stash(int Size) {
  size = Size;
  quantity = 0;
  next = 0;
  storage = 0;
}

stash::stash(int Size, int InitQuant) {
  size = Size;
  quantity = 0;
  next = 0;
  storage = 0;
  inflate(InitQuant);
}

stash::~stash() {
  if(storage) {
    puts("freeing storage");
    free(storage);
  }
}

int stash::add(void* element) {
  if(next >= quantity) // Enough space left?
    inflate(100); // Add space for 100 elements
  // Copy element into storage,
  // starting at next empty space:
  memcpy(&(storage[next * size]),
         element, size);
  next++;
  return(next - 1); // Index number
}

void* stash::fetch(int index) {
  if(index >= next || index < 0)
    return 0;  // Not out of bounds?
  // Produce pointer to desired element:
  return &(storage[index * size]);
}

int stash::count() {
```

```
  return next; // Number of elements in stash
}

void stash::inflate(int increase) {
  void* v =
    realloc(storage, (quantity+increase)*size);
  assert(v);  // Was it successful?
  storage = (unsigned char*)v;
  quantity += increase;
}
```

当我们用第一个构造函数时，没有内存分配给 storage,内存是在第一次调用 add()来增加一个对象时分配的，另外，执行add()时，当前的内存块不够用时也会分配内存。

下面的测试程序说明了这点，它检查第一个构造函数。

```
//: STSHTST4.CPP -- Function overloading
#include "..\4\stash4.h"
#include <stdio.h>
#include <assert.h>
#define BUFSIZE 80

main() {
  int i;
  FILE* file;
  char buf[BUFSIZE];
  char* cp;
  // ....
  stash intStash(sizeof(int));
  for(i = 0; i < 100; i++)
    intStash.add(&i);
  file = fopen("STSHTST4.CPP", "r");
  assert(file);
  // Holds 80-character strings:
  stash stringStash(sizeof(char) * BUFSIZE);
  while(fgets(buf, BUFSIZE, file))
    stringStash.add(buf);
  fclose(file);

  for(i = 0; i < intStash.count(); i++)
    printf("intStash.fetch(%d) = %d\n", i,
           *(int*)intStash.fetch(i));
  i = 0;
  while(
    (cp = (char*)stringStash.fetch(i++)) != 0)
    printf("stringStash.fetch(%d) = %s",
           i - 1, cp);
  putchar('\n');
}
```

我们可以修改这些代码，增加其他参数来调用第二个构造函数。这样我们可以选择 stash的初始大小。

5.3 缺省参数

比较上面两个stash（）构造函数，它们似乎并没有多大不同，对不对？事实上，第一个构造函数只不过是第二个的一个特例——它的初始大小为零。在这种情况下去创建和管理同一函数的两个不同版本实在是浪费精力。

C++中的缺省参数提供了一个补救的方法。缺省参数是在函数声明时就已给定的一个值，如果我们在调用函数时没有指定这一参数的值，编译器就会自动地插上这个值。在 stash的例子中，我们可以把：

```
stash(int  Size);// zero  quantity
stash(int  Size,int  Quantity);
```

用一个函数声明来代替

```
stash(int Size, int  Quantity=0);
```

这样，stash(int)定义就简化掉了——所需要的是一个单一的stash(int,int)定义。

现在这两个对象的定义

```
stash  A(100),B(100,0);
```

将会产生完全相同的结果。它们将调用同一个构造函数，但对于 A，它的第二个参数是由编译器在看到第一个参数是整型而且没有第二个参数时自动加上去的。编译器能看到缺省参数，所以它知道应该允许这样的调用，就好像它提供第二个参数一样，而这第二个参数值就是我们已经告诉编译器的缺省参数。

缺省参数同函数重载一样，给程序员提供了很多方便，它们都使我们可以在不同的场合使用同一名字。不同之处是，当我们不想亲手提供这些值时，由编译器提供一个缺省参数。上面的那个例子就是用缺省参数代替函数重载的一个很好的例子。用函数重载我们得把一个几乎同样含义、同样操作的函数写两遍甚至更多。当然，如果函数之间的行为差异较大，用缺省参数就不合适了。

在使用缺省参数时必须记住两条规则。第一，只有参数列表的后部参数才可是缺省的，也就是说，我们不可以在一个缺省参数后面又跟一个非缺省的参数。第二，一旦我们开始使用缺省参数，那么这个参数后面的所有参数都必须是缺省的。（这可以从第一条中导出。）

缺省参数只能放在函数声明中，通常在一个头文件中。编译器必须在使用该函数之前知道缺省值。有时人们为了阅读方便在函数定义处放上一些缺省的注释值。如：

```
void  fn(int x  /* =0*/ ) { //...
```

缺省参数可以让声明的参数没有标识符，这看上去很有趣。我们可以这样声明：

```
void  f(int  X, int = 0, float =1.1);
```

在C++中，在函数定义时，我们并不一定需要标识符，像：

```
void f(int X, int,float f)  {/*...*/}
```

在函数体中，x和f可以被引用，但中间的这个参数值则不行,因为它没有名字。这种调用还必须用一个占位符（placeholder）,有f(1)或f(1,2,3.0)。这种语法允许我们把一个参数当作占位符而不去用它。其目的在于我们以后可以修改函数定义而不需要修改所有的函数调用。当然，用一个有名字的参数也能达到同样的目的，但如果我们定义的这个参数在函数体内没有使用，

多数编译器会给出一条警告信息，并认为我们犯了一个逻辑错误。用这种没有名字的参数就可以防止这种警告产生。

更重要的是，如果我们开始用了一个函数参数，而后来发现不需要用它，我们可以高效地将它去掉而不会产生警告错误，而且不需要改动那些调用该函数以前版本的程序代码。

位向量类

这里我们进一步看一个操作符重载和缺省参数的例子。考虑一个高效存储真假标志集合的问题。如果我们有一批数据，这些数据可以用“ on ”或“ off ”来表示。用一个叫位向量的类来存储它们应该是很方便的。有时，位向量并不是作为应用程序的一个工具来使用，而是作为其他类的一部分。

当然对一组标志进行编码，最容易的方法就是每个标志占一个字节，请看下例：

```
//: FLAGS.CPP -- List of true/false flags
#include <stdio.h>
#include <string.h>
#include <assert.h>
#define FSIZE 100
#define TRUE 1
#define FALSE 0

class flags {
  unsigned char f[FSIZE];
public:
  flags();
  void set(int i);
  void clear(int i);
  int read(int i);
  int size();
};

flags::flags() {
  memset(f, FALSE, FSIZE);
}

void flags::set(int i) {
  assert(i >= 0 && i < FSIZE);
  f[i] = TRUE;
}

void flags::clear(int i) {
  assert(i >= 0 && i < FSIZE);
  f[i] = FALSE;
}

int flags::read(int i) {
```

```
  assert(i >= 0 && i < FSIZE);
  return f[i];
}

int flags::size() { return FSIZE; }

main() {
  flags fl;
  for(int i = 0; i < fl.size(); i++)
    if(i % 3 == 0) fl.set(i);
  for(int j = 0; j < fl.size(); j++)
    printf("fl.read(%d)= %d\n", j, fl.read(j));
}
```

然而，这样很浪费存储空间，因为我们用了八位来表示一个只要一位就可表示的标志。有时这种存储很重要，特别是我们想用这个类去建其他的类时。所以下面的 BitVector就用一位表示一个标志。函数重载出现在构造函数和bits()函数中。

```
//: BITVECT.H -- Bit Vector
#ifndef BITVECT_H_
#define BITVECT_H_

class BitVector {
  unsigned char* bytes;
  int Bits, numBytes;
public:
  BitVector(); // Default: 0 size
  // init points to an array of bytes
  // size is measured in bytes
  BitVector(unsigned char* init,
            int size = 8);
  // binary is a string of 1s and 0s
  BitVector(char* binary);
  ~BitVector();
  void set(int bit);
  void clear(int bit);
  int read(int bit);
  int bits(); // Number of bits in the vector
  void bits(int sz); // Set number of bits
  void print(const char* msg = "");
};
#endif // BITVECT_H_
```

第一个构造函数（缺省构造函数）产生了一个大小为零的 BitVector。我们不能在这个向量中设置任何位，因为它们根本就没有位。首先我们必须用重载过的 bits()函数增加这一矢量的大小。这个不带参数的版本将返回向量的当前大小，而 bits(int)会把向量的大小改成参数指定的大小。这样我们可以用同样的函数名来设置和得到向量的大小。注意对新的大小并没有限制

——我们可以增大它，也可以减少它。

第二个构造函数要用到一个无符号字符数组的指针，这也是一个原始字节数组，第二个参数告诉构造函数该数组总共有多少个字节，如果第一个参数是零而不是一个有效的指针，这个数组被初始化为零。如果我们没有给出第二个参数，其缺省值为8。

我们可能以为我们可以用BitVector b(0)这样的声明来产生一个8个字节的BitVector 对象，并把它们初始化为零。如果没有第三个构造函数，情况确实如此。第三个构造函数取 char* 作为它的唯一的参数。参数0既可以用于第二个构造函数（第二个参数缺省）也可用于第三个构造函数。编译器无法知道应该选用哪一个，所以我们会得到一个含义不清的错误。为了正确地产生这样一个BitVector对象，我们必须将零强制转换成一个适当的指针：BitVector b((unsigned char *)0)。这的确有些麻烦，所以我们可以选用BitVector b产生一个空向量，然后把它们扩展到适当的大小，b.bits(64),这就得到8个字节的向量。

编译器必须把char*和unsigned char*当作两个数据类型，这点很重要，否则BitVector(unsigned char*,int)在第二个参数缺省时就和BitVector(char*)一模一样了，编译器无法确定调用哪个函数。

注意print()函数有一个char*型的缺省参数。如果我们知道编译器怎么处理字符串常量，那么这个函数可能让我们感到有点奇怪。编译器在我们每次调用这个函数时都产生一个缺省的字符串吗？答案是否定的。它是在一个特定的保留区内产生一个单一的字符串作为静态全局数据，然后把这个字符串的地址作为缺省值传递给函数的。

一个位串

BitVector的第三个构造函数引用了一个指向字符串的指针，这个字符串代表了一个位串。这样就给用户提供了方便，因为它允许向量的初值可以用自然形式的 0110010来表示。对象产生时会匹配这个串的长度，根据串值来设置或清除每个位。

其他函数还有set()、clear()和read(),这些函数都很重要。它们都用感兴趣的位数作为参数。print()函数打印一条消息，它的缺省参数是空字符串，然后是 BitVector的比特位模型，又一次使用 0和1。

当实现BitVector类时会遇到两个问题。第一个是如果我们需要的位数并不恰好是 8的倍数（或机器的字长），我们必须取最接近的字节数。第二个是在选择某个当前位时要注意。比方，用一个字节数组产生了一个BitVector对象时，数组中的每个字节必须从左读到右，以使我们调用print()函数时出现我们预期的样子。

下面是一组成员函数的定义：

```
//: BITVECT.CPP -- BitVector Implementation
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include "..\4\bitvect.h"
#include <limits.h> //CHAR_BIT = # bits in char
// A byte with the high bit set:
const unsigned char highbit =
  1 << (CHAR_BIT - 1);

BitVector::BitVector() {
  numBytes = 0;
```

```
  Bits = 0;
  bytes = 0;
}
// Notice default args are not duplicated:
BitVector::BitVector(unsigned char* init,
                     int size) {
  numBytes = size;
  Bits = numBytes * CHAR_BIT;
  bytes = (unsigned char*)calloc(numBytes, 1);
  assert(bytes);
  if(init == 0) return; // Default to all 0
  // Translate from bytes into bit sequence:
  for(int index = 0; index<numBytes; index++)
    for(int offset = 0;
        offset < CHAR_BIT; offset++)
      if(init[index] & (highbit >> offset))
        set(index * CHAR_BIT + offset);
}

BitVector::BitVector(char* binary) {
  Bits = strlen(binary);
  numBytes =  Bits / CHAR_BIT;
  // If there's a remainder, add 1 byte:
  if(Bits % CHAR_BIT) numBytes++;
  bytes = (unsigned char*)calloc(numBytes, 1);
  assert(bytes);
  for(int i = 0; i < Bits; i++)
    if(binary[i] == '1') set(i);
}

BitVector::~BitVector() {
  free(bytes);
}

void BitVector::set(int bit) {
  assert(bit >= 0 && bit < Bits);
  int index = bit / CHAR_BIT;
  int offset = bit % CHAR_BIT;
  unsigned char mask = (1 << offset);
  bytes[index] |= mask;
}

int BitVector::read(int bit) {
  assert(bit >= 0 && bit < Bits);
  int index = bit / CHAR_BIT;
  int offset = bit % CHAR_BIT;
```

```
  unsigned char mask = (1 << offset);
  return bytes[index] & mask;
}

void BitVector::clear(int bit) {
  assert(bit >= 0 && bit < Bits);
  int index = bit / CHAR_BIT;
  int offset = bit % CHAR_BIT;
  unsigned char mask = ~(1 << offset);
  bytes[index] &= mask;
}

int BitVector::bits() { return Bits; }

void BitVector::bits(int size) {
  int oldsize = Bits;
  Bits = size;
  numBytes =  Bits / CHAR_BIT;
  // If there's a remainder, add 1 byte:
  if(Bits % CHAR_BIT) numBytes++;
  void* v = realloc(bytes, numBytes);
  assert(v);
  bytes = (unsigned char*)v;
  for(int i = oldsize; i < Bits; i++)
    clear(i); // Erase additional bits
}

void BitVector::print(const char* msg) {
  puts(msg);
  for(int i = 0; i < Bits; i++){
    if(read(i)) putchar('1');
      else putchar('0');
    // Format into byte blocks:
    if((i + 1) % CHAR_BIT == 0) putchar(' ');
  }
  putchar('\n');
}
```

第一个构造函数很简单，就是把所有的变量赋零。第二个构造函数分配内存并初始化位数。接下来用了一个小技巧。外层的 for循环指示字节数组的下标，内层 for循环每次指示这个字节的一位，然而这一位是自左向右用 init[index]&(0x80>>offset)计算出来的。注意这是按位进行与运算的，而且16进制的0x80（最高位为1，其他位为零）右移offset位，产生一个屏蔽码。如果结果不为零，那么在这一位上一定是 1，这个set()函数被用来设置BitVector 内部位。注意描述字节位时应从左到右，只有这样用 print()函数显示的结果看上去才是有意义的。

第三个构造函数把一个二进制0、1序列的字符串转换成一个BitVector。位数就取字符串的

长度。但字符串的长度可能并不正好是8的整数倍，所以字节数numBytes先将位数除以8，然后根据余数是否为0来调整。这种情况下，扫描位是在源串中从左到右进行的，这与第二个构造函数不同。

set()、clear()和read()三个函数形式都差不多，开始三行完全一样：assert()检查传入的参数是否合法，然后产生指向字节数组的索引和指向被选字节的偏移。 Set()和read()用同样的方法产生屏蔽字节：将1移位到所要的位置。但set()是用选定的字节与屏蔽字节相“或”来将该位置1，而read()是用选定的字节与屏蔽字节相“与”来获得该位的状态。clear()是将1移位到指定位来产生屏蔽字节的，然后将选定的字所有的位求“反”(用~)，再与屏蔽字节相“与”，这样只有指定的位被置为零。

注意set()、read()和clear()可以写得更紧凑些，如clear()可以这样来写：

```
bytes[bit/CHAR_BIT]&=~(1<<(bit % CHAR_BIT));
```

这样写可以提高效率，但肯定降低了可读性。

两个重载的bits()函数在行为上差别很大。第一个仅仅是一个存取函数（一种向没有访问权限的人提供私有成员数据的函数），告知数组中共有多少位。第二个函数用它的参数来计算所需的字节数，然后用realloc()函数重新分配内存（如果bytes为零，它将分配新内存），并对新增的位置零。注意，如果我们要求的位数与原有的位数相等，这个函数仍有可能重新分配内存(这取决于realloc()函数的实现)。但这不会破坏任何东西。

print()函数显示msg字串，标准的C库函数puts()已经加了一个新行，所以对缺省参数将输出一个新行。然后它用read()读取每一位的值以确定显示什么字符。为了阅读方便，在每读完8位后它显示一个空格。由于第二个BitVector构造函数是读取字节数组的方式， print()函数将会用熟悉的形式显示结果。

下面的程序通过检验BitVector的所有函数来测试BitVector类。

```
//: BVTEST.CPP -- Testing the BitVector class
#include "..\4\bitvect.h"

main() {
  unsigned char b[] = {
    0x0f, 0xff, 0xf0,
    0xAA, 0x78, 0x11
  };
  BitVector bv1(b, sizeof b / sizeof *b),
    bv2("10010100111100101010001010010010101");
  bv1.print("bv1 before modification");
  for(int i = 36; i < bv1.bits(); i++)
    bv1.clear(i);
  bv1.print("bv1 after modification");
  bv2.print("bv2 before modification");
  for(int j=bv2.bits()-10; j<bv2.bits(); j++)
    bv2.clear(j);
  bv2.set(30);
  bv2.print("bv2 after modification");
  bv2.bits(bv2.bits() / 2);
  bv2.print("bv2 cut in half");
```

```
    bv2.bits(bv2.bits() + 10);
    bv2.print("bv2 grown by 10");
    BitVector bv3((unsigned char*)0);
}
```

对象bv1、bv2、bv3显示了三种不同的BitVector类和它的构造函数。set()和clear()函数也被检验（read()在print()内检验）。在程序的尾部， bv2被减少了一半然后又增大，用以说明将BitVector的尾部置零的一种方法。

我们应该知道在标准的C++库中包含着bits和bitstring类，这些类向位向量提供了一个更完全（也更标准）的实现。

5.4 小结

函数重载和缺省参数都为调用函数提供了方便。有时为弄清到底哪个函数会被调用，也让人迷惑不清。比如在BitVector类中，下式就似乎对两个bits()函数都可以调用：

```
int bits(int sz=-1);
```

如果调用它时不带参数，函数就会用缺省的值 -1，它认为我们想知道当前的位数。这种使用似乎同前面的一样，但事实上存在着明显的不同，至少让我们感觉不舒服。

在bits（）内部我们得按参数的值作一个判断，如果我们必须去找缺省值而不是作为一个普通值，根据这一点，我们就可以形成两个不同的函数。一个是在一般情况下，一个是在缺省情况下。我们也可以把它分割成两个不同的函数体，然后让编译器去选择执行哪一个，这可以提高一点效率，因为不需要传递额外的代码，由条件决定的额外代码也不会被执行。如果我们需要反复调用这个函数，这种效率的少许提高就会表现得很明显。

在这种情况下，用缺省参数我们确实会丢失某些东西。首先，缺省值不能作他用，如本例中-1。现在，我们不能区分一个负数是一个意外还是一个缺省情况。第二，由于在单一参数时只有一个返回值，所以编译器就会丢失很多重载函数时可以得到的有用信息。比如，我们定义：

```
int i=bv1.set(10);
```

编译器会接受它但不再告诉我们其他东西，但作为类的设计者，我们可能认为是一个错误。

再看看用户遇到的问题。当用户读我们的头文件时，哪种设计更容易理解呢？缺省值 -1意味着什么？没有人告诉他们。而用两个分开的函数则非常清楚，因为一个带有一个参数但不返回任何值，而另一个则不带参数但返回一个值。即使没有有关的文档，也很容易猜测这两个函数完成什么功能。

我们不能把缺省参数作为一个标志去决定执行函数的哪一块，这是基本原则。在这种情况下，只要能够，就应该把函数分解成两个或多个重载的函数。缺省参数应该是能把它当作变通值来处理的值，只不过这个值出现的可能比其他值要大，所以用户可以忽略它或只在需要改变缺省值时才去用它。

缺省参数的引用是为了使函数调用更容易，特别是当这些函数的许多参数都有特定值时。它不仅使书写函数调用更容易，而且阅读也更方便，尤其当用户是在制定参数过程中，把那些最不可能调整的缺省参数放在参数表的最后面时。

缺省参数的一个重要应用是在开始定义函数时用了一组参数，而使用了一段时间后发现要增加一些参数。现在我们只要把这些新增参数都作为缺省的参数，就可以保证所有使用这一函

数的代码不会遇到麻烦。

5.5 练习

1) 创建一个message类，其构造函数带有一个 char*型的缺省参数。创建一个私有成员char*，并假定构造函数可以传递一个静态引用串：简单将指针参数赋给内部指针。创建两个重载的成员函数print();一个不带参数，而只是显示存储在对象中的消息，另一个带有 char*参数，它将显示该字符串加上对象内部消息。比较这种方法和使用构造函数的方法，看哪种方法更合理？

2) 测定您的编译器是怎样产生汇编输出代码的，并尝试着减小名字分解表。

3) 用缺省参数修改STASH4.H和STASH4.CPP中的构造函数，创建两个不同的stash对象来测试构造函数。

4) 比较flags类与BitVector类的执行速度。为了保证不会与效率弄混，把 set()、clear()和read()中的index、offset和mask定义合并成一个单一的声明来完成适当的操作（测试这个新的代码以确保代码正确）。

5) 修改FLAGS.CPP以使它可以动态地为标志分配内存，传给构造函数的参数是空间 存储的大小，其缺省值为100。保证在析构函数中清除这些存储空间。