C++ 高級程序設計技術
可重用類的構造

陳勇浩　王建文　编译

陕西电子编辑部

73.87419
C279

陕西电子编辑部
地址：西安市西五路 16 副 5 号
邮编：710004
电话：791344
电挂：7795
陕西省内部图书准印证
陕新出批(1992 年)字第 326 号

内 容 提 要

C++是 C 语言为适应近代软件工程而发展的超集，它吸收了面向对象程序设计的思想，提供了许多易于通用程序设计和软件维护的机制。C++ 语言正以其特有的优点越来越受到人们的青睐。

本书是 C++的第二代丛书。以前有关 C++的许多书籍都着重讲述其语法知识及一般用法，而本书则通过许多实例生动地讲述了用 C++进行程序设计，尤其是进行类设计的策略和技巧，并且，提供了大量实用的可重用类。

全书共分三部分：第一部分简要介绍了 C++语言的重要特点；第二部分通过实例讲述了如何有效地利用 C++提供的新机制进行面向对象的程序设计；第三部分构造了许多非常实用的可重用类。

每章后附有习题，以帮助读者加深理解。书末列出了全部可重用类代码，以供选用。

本书适用于广大的软件设计人员，尤其有助于提高面向对象的程序设计能力。

目　录

此为试读，需要完整PDF请访问：www.ertongbook.com

第一章　软件危机

我们不打算详细地阐述软件危机，在此，仅以一个例子说明有关的一些问题。

1981 年以来，UNIX 系统支持 ed、ex、Vi 及 Jim 等四种编辑器。ed 是一种非常简单的编辑器；ex 是 ed 的增强型版本，提供了更为复杂的编辑操作；Vi 是从 ex 发展来的，是一个面向屏幕的编辑器。在这四种编辑器中，除了 jim 之外（jim 是后来开发的一种编辑器，其部分目的是要实践新的编辑思想），其它三个编辑器 ed、ex 及 vi 有许多操作都是相同的，它们的很多命令非常相似。

下面，我们考虑编辑器常做的一种操作：在一个文件中搜索特定正文模式，这种操作叫正则表达式搜索，每一个编辑器都提供了这种搜索能力。当然，各个编辑器依据其特定的目的而略有不同。

除了以上四种编辑器外，UNIX 的其它几个命令也提供了正则表达式搜索功能。例如，grep、fgrep、egrep 以及 awk 等命令。另外，命令处理程序 sh 也能进行文件名的匹配搜索。

仅仅一个 UNIX 系统就提供了如此众多的正则表达式搜索机制，这不能不使人感到吃惊。同时，也说明了问题之所在。UNIX 系统上的九种搜索机制彼此间都或多或少地有些不一样，或许对各种应用来说，这种不一样是必要的，然而，这给系统的终端用户带来了许多麻烦，他们为了选择适当的命令，需要了解九种不同的搜索机制。

1-1　库函数的失败

在意识到以上的问题之后，提出了一种改进方案，即建立一个库函数。所有的命令都使用这个库函数来完成各自的模式匹配操作，这样，从用户角度来看，就显得更为一致。这一方案最终导致设计了两个库函数：regcx 和 regcxp，它们用于正则表达式搜索。

然而，没有一个实用程序使用了 regcx 和 regcxp！目前，UNIX 系统上能完成复杂模式匹配操作的程序都采用了以下两种途径之一：

(1) 修改 cx 代码以满足特定应用的要求；

(2) 重新编写自己的模式匹配程序。

这两种途径的开发和维护费用都相当高，而且用户不易使用。

那么，为什么不使用库函数呢？原因有两个：大多数人都喜欢他们自己的方法，而不愿意去调用别人编好的库函数，此其一；其二，库函数不可能以一种非常通用的方式去满足各种应用情况。

下面是正规表达式搜索所需要的有关抽象描述：

(1) 正则表达式

　　所要搜索的正文；

(2) 正则表达式编译程序

它把正则表达式转换为正则表达式解释程序可以使用的有效形式;

(3) 正则表达式解释程序

它完成实际的搜索操作;

(4) 缓冲区

待搜索的正文体.

正则表达式匹配的困难在于缓冲区的多样性. 每一种应用都要执行其特定的任务, 都需要以一种最有效的方式来实现其缓冲区. 因此, 在实际应用中, 没有两个缓冲区的实现是完全一样的. 问题的关键在于如何确定缓冲区与正则表达式解释程序之间的界面, 以便每一应用都能提供完成正则表达式搜索所需要的功能.

1-2 面向对象的方法

从技术上讲, 库函数使程序员不能对所需要的东西进行控制. 函数接口是有关数据类型及其操作的规定, 而没有提供对抽象及其界面的描述. 因此, 当你的数据类型或操作模式与库函数规定的稍微有一点不同时, 你便不能使用它们.

考虑正则表达式库函数的问题, 这里真正需要的是一个基缓冲区, 它描述了正则表达式解释程序所需要的缓冲区界面, 但并不限制其它编辑器或工具提供对抽象缓冲区自己的实现, 各种编辑器应能够根据自己特定的需要来实现相应的缓冲区.

面向对象的程序设计语言正是为解决这类问题而提出的, 这些语言具有以下四个方面的能力:

(1) 对事物提供完全抽象的描述;

(2) 通过组合或例化现有抽象而建造新的抽象, 以便重用;

(3) 对抽象与其环境间的交互进行描述;

(4) 以模块化的方式实现抽象.

C++是一种非常成功的语言, 它把面向对象程序设计的新思想与 C 语言完美地结合起来. 事实上, 在 C++提出之前, 面向对象的方法并没有被广泛接受, 因为其它面向对象的程序设计语言都要求放弃或者重新实现现有的大量代码, 而且这些语言在性能上也存在很大不足, C++克服了所有这些缺点.

用 C++进行程序设计, 需要牢记以下几个问题:

(1) 程序要处理哪些不同类型的实体;

(2) 实体之间有什么联系;

(3) 是否能够利用别人已经做过的工作;

(4) 怎样使代码更加通用.

同时, 还需要注意以下问题:

(1) 怎样尽可能地提高性能;

(2) 怎样以最小的代价解决这个问题.

1-3 面向重用的设计

目前，软件重用还只能在源代码级进行．那么，面向对象的程序设计语言是如何对此提供支持的呢?

C++是在C语言的基础上发展起来的，许多C++编译程序都要产生C语言代码，因此，也可以用C语言进行面向对象的程序设计．然而，C++提供了一整套机制，使得能够更加容易地建造对象．在C++中，对类的一个说明就是定义该类对象内部包含的数据和对数据进行操作的函数，以及顾客使用这些函数和数据的权限。

类的界面完成了对类的封装．

封装的一个结果是C++的源代码比C源代码更易于重用，因为在C++中，不必为了找到所需要的函数而查遍大量的源代码，类定义准确地告诉了你它能做什么，不能做什么．由于类封装了一个完整的工作单元，所以可以把类作为一个整体进行删除或修改，以适应新的需要．

封装还有另一个结果．因为类能控制顾客所能使用的东西，所以，类的界面就是顾客所知道的有关类的全部，它描述了一个抽象；而类的实现者可以按任何合适的方式修改类的内容，包括重写全部实现．

由于面向对象的语言允许我们建造抽象并例化它们，因此，面向对象的设计就集中于如何建造通用解以及怎样例化这些通用解以满足特定要求．不管某一特定对象能否满足其他用户的要求，类所具有的通用形式总能为解决其他人的问题而提供基础．

优秀的面向对象设计人员着力于工具的建造，以便其他程序员能直接利用这些工具来解决他们自己的问题．本书将构造大量可重用的软件工具，基于这些工具，你只要编写很少的代码，便能解决大量相当复杂的问题．

思 考 与 练 习

1．使软件重用成为可能的关键因素是什么？在你所使用的语言中，哪些机制使软件重用变得非常困难?

第二章　C++指南

通常认为 C++是 C 语言的扩充，它提供了许多易于通用程序设计和软件维护的机制．本章简述 C++的这些新特点．

2-1　参数原型

C++对 C 语言的第一个扩充是引入了参数原型．所谓参数原型，是指在说明一个函数的时候，函数参数的类型也一起说明．这样做使得编译程序能够检查传递的参数个数和类型是否与期望的一致，以及使得链接程序在连接时能够检查函数说明与函数调用的一致性，从而实现类型安全链接．

以下是 C 和 C++函数定义的比较:

C version

```
main(argc, argv)
  int argc;
  const char  * argv[ ]
{
     //...
}
```

C++ version

```
main(int argc,
     const char  * argv[ ])
{
     //...
}
```

函数说明也使用了参数原型．下面是 printf( )函数的说明:

C version

```
extern int
printf( );
```

C++ version

```
extern int
printf(const char  *, . . . );
```

C 语言版本只是简单说明 printf( )是一个函数；而 C++还指出了 printf( )的第一个参数是字符常指针，printf( )不能修改它，随后是一串未说明的参数，其类型预先不知道．

C++中说明函数指针时也需要同时说明参数:

C version

```
extern int  ( * pfi)( );
```

C++ version

```
extern int( * pfi)(int);
```

ANSI C 标准采用了 C++的参数原型，但二者之间有差别．在 ANSI C 中:

```
extern int f( );
```

意指"函数 f，它返回一整数值，参数没有说明"；而在 C++中，则是指"函数 f 是一个无参函数，它返回一整数值"．ANSI C 标准建议避免以上这种用法，它仅仅是为了与现有 C 代码的兼容才被保留下来．一个更好的说明是:

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

C++增强了函数调用的类型检查。在C++中，不允许调用一个尚未说明的函数。

2-2 类型安全链接

C++利用函数原型来实现类型安全链接。当调用一个C++函数时，C++把函数名变换成一个包含有参数类型信息的名字：

C++ Code	Output

extern int
CurrentTime(void);
//...
i=CurrentTime( );
	extern int
	__11CurrentTime_Fv( );
	//...
	i=__11CurrentTime_Fv( );

如果实际定义的函数需要一个整型参数，那么输出代码中的函数名就会与上面的不一样，这种情况下，链接程序认为函数CurrentTime(void)没有定义而作错误处理。

2-3 函数名重载

参数说明使得C++函数名可以被重用，因为参数类型可用于决定具体使用哪一个函数。这样，可以定义多个名字都为printf( )的函数，每个函数依据其参数类型来完成相应的功能。

```
void print(int i)
{
    printf("%d", i);
}

void print(float f)
{
    printf("%f", f);
}

void print(mystruct& s)
{
    // . . . your interpretation here. . .
}
// . . .
```

由于函数说明包含了参数类型信息，C++能够正确地处理函数调用，即使函数的定义和调用出现在不同源文件中，也不会发生错误。

此为试读，需要完整PDF请访问：www.ertongbook.com

2-4 引用(reference)

C++的另一特点是引用，引用是一个对象的别名．引用有两种用法：在函数参数原型中使用，类似于 Pascal 中的 var 参数；用在 C++代码中，创建别名．在 C 语言中，通过使用指针类型的参数也可以实现引用．

C version

```
typedef struct mystruct_s
        mystruct_s;

int f(mystruct_s *);
// ...

mystruct_s mystruct;
i = f(&mystruct);

void
f(mystruct_s *s)
{
    (*s).field = value;
}
```

C++ version

```
// no typedef needed

int f(mystruct_s&);
// ...

mystruct_s mystruct;
i = f(mystruct);

void
f(mystruct_s& s)
{
    s.field = value;
}
```

二者的差别在于：在 C 中，必须显式地取得参数的地址，并显式地通过这个地址去访问参数；而在 C++中，这些操作都是自动完成的．另外，在 C++的说明中，不需要保留字 struct，因为 C++的结构标识名和类型名属于同一名空间．

引用以一种用户透明的方式，使得参数传递更为有效．例如，下面两个说明对用户来说是一样的：

By-value version

```
int f(const mystruct_s);
```

By-reference version

```
int f(const mystruct_s&);
```

在调用处，同一个对象作为参数传递给 f，两种情况下，f 都不能修改对象．

从实现角度看，传值方式和引用方式有两个重要差别：引用方式传递的是指向对象的指针，而传值方式需要将整个对象拷贝到堆栈上，在函数返回时再将它们删除；更为重要的是传值方式需要调用构造函数和析构函数来创建和销毁临时对象．稍后将讨论构造函数和析构函数．

使用引用的另一个场合是在初始化时，用于给对象建立别名：

```
mystruct_s& s_ref = s;
```

对 s_ref 的任何操作都等价于对 s 的操作，反之亦然．因为它们仅仅是同一对象的不同名字而已．

2-5 类(class)

类是 C++最重要的特点。类类似于 C 语言的结构，但提供了另外三个属性：保护、成员函数和继承。

在讨论类时，需要谈及三种人：类的设计者、类用户以及库的设计者。类的设计者设计实现类；类用户不加修改地使用这些类作为构造自己软件的工具；而库的设计者则可能以现有类作为基础来设计新类。

在面象对象的程序设计中，常常希望确保对象内部的一致性。在 C 语言中，要做到这一点很不容易，因为类的设计者需要依靠用户的合作。C 语言没有提供一种机制来把实现细节隐蔽起来，因此，用户可以随便修改它们。这将导致两个严重的后果：首先，用户在修改一个对象时，易造成对象内部的不一致；其次，由于用户的介入，类的设计者很难再再设计，以改善类的性能。

在此需要一种方法，一种能确保对象内部保持一致的方法。首先，要控制用户所能修改的东西。C++的类能把成员说明为私有的(private)，从而实现这种控制。

```
class SimpleString {
  private:
    char *__string;
    int __length;
};
```

用户不能直接存取或修改__length 和__string 成员。

在具有了保护对象成员的能力之后，应该提供一种受限的方式来访问这些成员。C++利用成员函数来实现这一点。成员函数与一个对象类型相联系，它们可以修改该对象的所有成员，不管该成员是不是私有的。

为了让用户能够存取 SimpleString 对象的内容，增加一个成员函数 string( ):

```
class SimpleString {
    char *__string;
    int __length;
  public:
    const char *string( );
};
```

其中，private: 和 public: 修饰符可以插入到任何成员的前面，类的设计者可以按任意合适的顺序对成员进行说明。在省缺方式下，成员都是私有的，因此，可以省略前面例子中的 private: 保留字。__string 和__length 仍是私有成员，但 string( )函数是公有的，用户可以调用它。

```
SimpleString str;
printf(str.string( ));
```

成员函数 string( )实现如下:

```
const char *
```

```
SimpleString::string( )
{
    return __string;
};
```

函数名前的类名及双冒号说明这是一个成员函数的定义。需要注意的是，类的成员名对成员函数来说，都是可见。

由于成员函数 string( )返回一个字符常指针，用户不能通过该指针修改 SimpleString 对象。有时，需要修改对象，为此定义另一个成员函数:

```
class SimpleString {
    char * __string;
    int __length;
 public:
    const char * string( );
    void setString(const char * );
} ;

void
SimpleString::setString(const char * s)
{
    __string = new char[strlen(s)+1];
    strcpy(__string, s);
    __length = strlen(__string);
}

user code:

SimpleString str;
str.setString("fred");
```

2-6 常成员函数

如果对象说明成 const 型的，C++怎么知道哪一个成员函数可以对它进行操作呢？由于 SimpleString::string( )函数不改变对象内容，所以它能对 const 型的对象进行操作。一个函数如果能够安全地对 const 型的对象进行操作，就把这个函数叫做常成员函数。对 SimpleSting 类的定义稍作修改，便可让 C++知道这种用户的合法性。

```
class SimpleString {
    char * __string;
    int __length;
 public:
    const char * string( ) const;
    void setString(const char * );
} ;

const char *
SimpleString::string( )
const
{
```

```
    return __string ? __string :" ";
}

user code:

const SimpleString str = "fred";  // we'll get to this
str.setString("fred");     // illegal - nonconst member function
str.string( );             // okay - const member function
```

C++不允许常成员函数修改对象内容。

2-7 运算符重载

如果对对象的每一个操作都需要按前述的方式调用成员函数的话，程序的可读性就会大大下降。而用通常的运算符及其语法书写的代码，如：a+b，读起来要容易得多，因为人们很熟悉这种形式。C++提供了扩展运算符的方法，以便它们能够处理新的对象类型。这种方法就是运算符重载。例如，C++中可以定义串的赋值运算：

```
class SimpleString {
    char *_string;
    int _length;
  public:
    const char *string();
    void operator=(const char *);
} ;

void
SimpleString::operator=(const char *s)
{
    if (s) {
        _length = strlen(s);
        _string = new char[_length+1];
        strcpy(_string, s);
    }
    else {
        _length = 0;
        _string = 0;
    }
}

user code:

SimpeString str;
str = "fred";
```

在C++中，可以对同一个运算符给出多种形式的定义，它们分别适用于不同类型的对象。就C++而言，这些重载的运算符与其它函数完全一样。前面定义的SimpleString不能进行对象之间的赋值，下面的定义则允许把一个串或者一个SimpleString对象赋给另一个SimpleString对象。

```
class SimpleString {
    // . . .
```

```
  public:
    const char * string( );
    void operator=(const char * );
    void operator=(const SimpleString&);
};
void
SimpleString::operator=(const char * s)
{
    delete __string;
    if (s) {
        __length = strlen(s);
        __string = new char[__length+1];
        strcpy(__string, s);
    }
    else {
        __length = 0;
        __string = 0;
    }
}
void
SimpleString::operator=(const SimpleString& s)
{
    delete __string;
    if (s.__string) {
        __length = strlen(s.__string);
        __string = new char[__length+1];
        strcpy(__string, s.__string);
    }
    else {
        __length = 0;
        __string = 0;
    }
}
user code:
SimpleString str1, str2;
str1 = "fred";
str2 = str1;
```

注意，在运算符"="的定义中，使用了 SimpleString 引用，这样做保证了在执行赋值的过程中，不会有临时对象生成。

上面对运算符"="的定义存在一个问题。C 语言中，赋值是一种运算，它要求赋值运算符右边表达式的值能够赋给左边的变量。因此，以下赋值是不正确的：

```
SimpleString s1, s2;
s1 = s2 = "fred";
```

因为我们定义运算符"="返回空值。C++按以下方式解释上面的赋值：

```
s1.operator=(s2.operator=("fred"))
```

C++不能处理：

然而，也不能让运算符"="返回一个 SimpleString 对象，因为这需要创建一个临时对象。正确的做法是:

```
class SimpleString {
    // ...
  public:
    const char *string();
    SimpleString& operator=(const char *);
    SimpleString& operator=(const SimpleString&);
} ;

SimpleString&
SimpleString::operator=(const char *s)
{
    delete _string;
    if (s) {
        _length = strlen(s);
        _string = new char[_length+1];
        strcpy(_string, s);
    }
    else {
        _length = 0;
        _string = 0;
    }
    return *this;
}

SimpleString&
SimpleString::operator=(const SimpleString& s)
{
    delete _string;
    if (s._string) {
        _length = strlen(s._string);
        _string = new char[_length+1];
        strcpy(_string, s._string);
    }
    else {
        _length = 0;
        _string = 0;
    }
    return *this;
}

user code:

SimpeString str1, str2;
str2 = str1 = "fred";
```

运算符"="返回 * this，表示赋值操作的结果返回值就是赋值操作的左值.

运算符重载是 C++一个非常有用的机制，但也容易引起混淆，应当适当地使用它。

2-8 构造函数(constructor)

保持对象内部的一致性意味着在创建对象时，就应能对它们作适当的初始化。C++提供了构造函数来完成这种初始化。当一个对象被说明或者在堆中创建一个对象时，都要调用构造函数。

以下说明

此为试读，需要完整PDF请访问：www.ertongbook.com