# C++动态数组的实现与重用

陈凤祥,李汪根
(安徽师范大学 数学计算机科学学院,安徽 芜湖 241000)

**摘 要**:数组是应用程序中经常要用到的一种数据结构。为解决C++定义后的数组不能改变其大小的情况,根据软件重用的思想,给出了用函数模板、动态数组类和数组类模板生成C++动态数组的方法,并对这些方法在代码重用方面进行了比较。文中主要给出了动态生成二维数组的函数模板、动态数组类和数组类模板的定义,以及部分成员函数和运算符重载函数的实现代码或算法。以动态生成二维整型数组,进行矩阵的加法运算为例,给出了使用方法。应用文中所给的方法,能满足应用程序中动态定义一维和二维数组的需要。

**关键词**:软件重用;动态数组;函数模板;数组类;数组类模板
**中图分类号**:TP311 **文献标识码**:A **文章编号**:1673-629X(2010)02-0079-04

# The Implementation and Reuse of C++ Dynamic Arrays

CHEN Feng-xiang, LI Wang-gen
(School of Mathematics and Computer Science, Anhui Normal University, Wuhu 241000, China)

**Abstract**: Arrays are important data structures used to proggramming, the C++ array size is fixed after declaring. According to the idea of software reuse, these methods to generate C++ dynamic array were put forward and these methods based on function templates, array classes and array templates, meanwhile these methods were compared in code reuse. These definitions of function templates, array classes and array templates were proposed, also some code of member functions and operator overloading were implemented. Further more addition was implemented between matrixs, so dynamic array was come true to make use of these methods.

**Key words**: software reuse; dynamic array; function templates; array class; array templates

## 0 引 言

C++中声明一个数组时其每一维的大小不能是变量,但是很多情况下,在程序运行之前,并不能够确切地知道数组中会有多少个元素,这就给程序设计带来了一些不便。目前已有一些实现C++动态数组的方法,如用指针的方法实现动态数组[1,2]。这种方法简单,容易实现,但创建数组时只能针对一种数据类型,不利于代码的重用。也有用数组类模板实现C++动态数组的,这种方法可以根据需要实例化为元素类型不同的数组[3]。但文中所给方法只针对一维数组。事实上,在程序设计中,会遇到不少二维数组的应用。笔者依据功能抽象、参数抽象的思想[4],应用C++提供的动态内存分配技术,对动态生成数组及代码重用作了一些研究与实现。以动态生成一维数组和二维数组为例,分别加以阐述。

**收稿日期**:2009-06-02;**修回日期**:2009-09-17
**基金项目**:安徽省自然科学基金(070412043)
**作者简介**:陈凤祥(1956-),女,讲师,研究方向为软件工程;李汪根,博士,副教授,研究领域为智能计算和生物信息学。

## 1 利用函数模板实现动态生成数组

通常可将完成一个特定功能的一段代码写成一个函数,以供重用。代码级重用是迄今为止研究最深入、应用最广泛的重用技术[4]。因此可以将动态生成数组的代码,写成可供调用的函数,实现代码重用。但因数据类型的不同,必须要有与之相匹配的重载函数,从而造成很多重复劳动,也很容易出错,使维护的开销增大。C++模板能够定义一个函数或类的系列,该系列可作用在不同类型的信息上,比如可以使用函数模板来创建一个把相同算法用于不同数据类型的函数集[5]。使用函数模板可以减少重复劳动,减少维护和调试的开销。

### 1.1 动态生成一维数组的函数模板

在函数模板中,将数据类型用一个参数来替代,使其可以自动适应数据类型的变化。这里的数据类型既可以是基本数据类型,也可以是类的对象。能动态生成一维数组的函数模板定义如下:

```
template < class T >
T * CreateArray( T m, int n )
```

```
{ T * i_pointer;
    i_pointer = new T[n];
    return i_pointer; }
```

调用该函数模板时,只须将 $m$ 声明为所需的数据类型即可。若已定义类 point,要动态生成有 $n$ 个 point 类对象的一维数组的方法为:

```
point a, * p;
p = CreateArray(a, n);
```

删除对象数组的操作为:delete [] $p$;

### 1.2 动态生成二维数组的函数模板

能动态生成二维数组的函数模板可以定义如下:

```
template < class T >
T * * Create2Array(T m, int row, int col)
{ T * * const pArray = new T * [row];
    for (int i = 0; i < row; i ++)
        pArray[i] = new T[col];
    return pArray; }
```

若要定义 point 类对象的二维数组,其方法是:

```
point a, * * p;
p = Create2Array(a, row, col);
```

删除二维对象数组的操作为:

```
for(int i = 0; i < row; ++ i) delete[] p[i];
delete p;
```

同理,可以有动态生成多维数组的函数模板[6]。尽管用函数模板能动态生成数组,且模板也能重用,但其建立数组和删除数组的过程是分开进行的,这使得程序显得繁琐。更好的方法是将数组的建立和删除过程封装起来,形成一个动态数组类[7]。

## 2 用动态数组类生成数组

用动态数组类生成的数组,若希望象普通数组一样可以用下标进行运算,需要对下标运算符"[]"进行重载。当运算符用于类的实例时,用关键字 operator 声明一个指定运算符的函数,这给运算符赋予多重含义,即运算符重载[8]。若希望像一般的类对象一样可以复制,需要写可以进行深拷贝的拷贝构造函数;若重载了"="运算符,可对数组进行整体赋值;若重载了"<<"运算符,可对数组整体输出,等等。

### 2.1 能生成一维数组的动态数组类

不失一般性,以生成一维整型数组的动态数组类为例。类中只重载了下标运算符"[]",且省去了与下标运算符无关的成员函数,类的声明如下:

```
class ArrayOfInt
{public:
  ArrayOfInt(int n ) //构造函数
    { number= n; i_point=new int[ n ];}
  ~ArrayOfInt() //析构函数
    {   number=0;delete[ ] i_point;}
  int& operator[ ] (int n) //重载运算符"[]"
    { if( n < 0 || n > number-1) Error();
    return i_point[ n ];}
private:
    int * i_point,number;
  void Error() ; //错误处理函数
};
```

用类 ArrayOfInt 声明一个有 $n$ 个元素的一维整型数组的语句为:

```
ArrayOfInt A(n);
```

若要声明其它类型的对象数组,只需将 new int [ $n$ ]中的 int, $i$_point 的类型以及重载下标"[]"运算符函数的返回值类型,由 int 换成其他类型即可。

### 2.2 生成二维整型数组的动态数组类

仿照一维动态数组类的结构,声明一个二维动态数组类作为基类,并派生出 Matrix 类。在基类中重载下标运算符"[]",在派生类中重载"="运算符,同时还重载了"+"运算符,"-"运算符和"<<"运算符。

下面以生成二维整型数组的动态数组类为例,类的声明如下所示:

```
class Array2Int
{public:
  Array2Int(int n = 0, int m = 0);
   ~Array2Int(){}
   int * operator[ ] (int i)
   { return i_point[ i ]; }
protected:
    int * * i_point,row,col; //在派生类中可访问
};
class Matrix:public Array2Int
{public:
Matrix(int n = 0, int m =0):Array2Int (n,m)
{}
Matrix(Matrix&); //拷贝构造函数
~Matrix();
Matrix& operator+ (Matrix &mhs); //重载"+"运算符
Matrix& operator- (Matrix &mhs);
Matrix& operator= (const Matrix &mhs);
friend ostream& operator<<(ostream& os,Matrix &mhs);
```

```
};
Array2Int::Array2Int(int n, int m) //基类构造函数的实现
{  row= n; col= m;
   i_point=new int*[row];
   for (int i = 0; i < row; i++)
   i_point[ i ] = new int[col];
}
ostream& operator << (ostream& os, Matrix &mhs) //重载"<<"运算符
{ for(int i = 0; i <mhs.row; i++)
{ for(int j = 0;j < mhs.col; j++) os<<mhs[i][j]<<"\t";
os<<endl; }
return os;
}
```

派生类 Matrix 中析构函数的实现与 1.2 中所述删除二维对象数组的操作类似,在删除指针变量后,将 row 和 col 清零。其余成员函数的实现算法如图 1～图 3 所示。

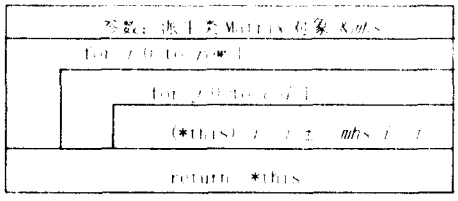

图 1　重载"+"或"-"运算符函数的算法

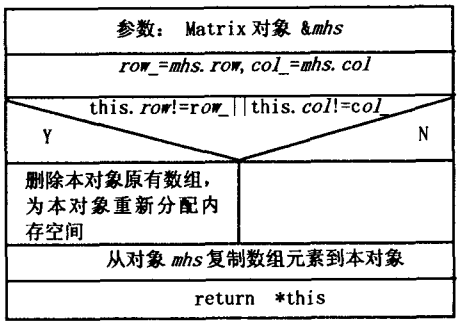


图 2　重载"="运算符函数的算法

用 Matrix 类声明一个二维整型数组的语句是:Matrix $A$(row,col);若 $A$ 数组已赋值,要将 $A$ 数组的元素复制给 $B$ 数组,用语句 Matrix $B$($A$);即可。可以计算 $A+B$,$A-B$,$C=A$;要将数组 $A$ 整体输出,用语句 cout<<$A$;即可。

采用动态数组类的方法生成数组,优点是:只要将类的声明放在一个头文件里,在程序中包含该头文件,就可以生成大小自定的对象数组。缺点是不同数据类型的对象数组,就要声明不同数据类型的动态数组类。若将动态数组类的功能抽象出来,将数据类型作为参数,这样得到了数组类模板。

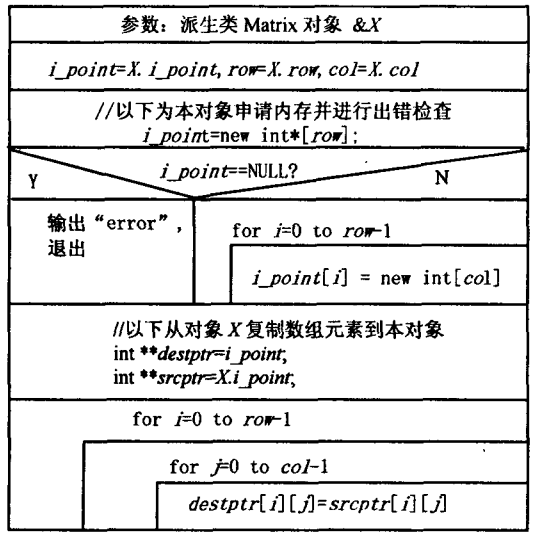


图 3　派生类 Matrix 拷贝构造函数的算法

## 3　用数组类模板动态生成二维数组

一维数组类模板的声明及应用文献[7]中有详细介绍,此处不赘述。这里详细介绍二维数组的数组类模板。二维数组类模板的成员数据与成员函数是二维动态数组类 Array2Int 和其派生类 Matrix 的综合。只要将类 Array2Int 和 Matrix 中的相关数据类型由 int 改为模板参数,将对象类型改为 Array$T2<T>$ 即可。为简化叙述,不妨将数组类模板的定义用 UML 的类图表示之。模板是一个参数化的模型元素,在 UML 中模板类像普通类一样表示,只是模板参数用虚矩形显示在类图标的右上角[9]。二维数组类模板的类图如图 4 所示。

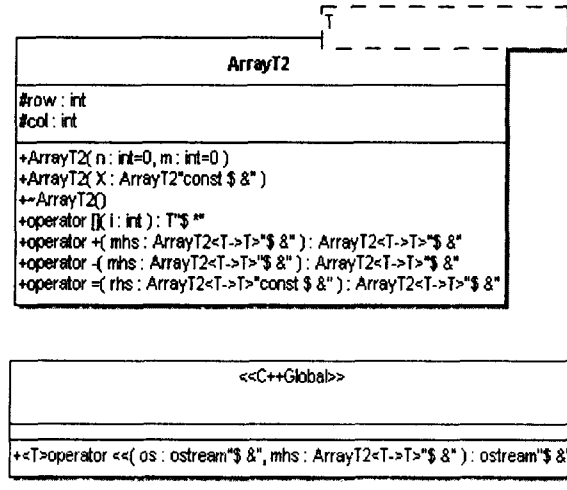


图 4　二维数组类模板的 UML 图形表示

数组类模板中构造函数、析构函数和运算符重载函数的实现,与 Array2Int 类中的构造函数,Matrix 类中析构函数和运算符重载函数的实现类似,只是在每

一个函数的实现代码中，在函数头前需加上 template <class $T$>，下面仅以拷贝构造函数的实现为例。

```
template <class T>
ArrayT2 < T >: :ArrayT2 (const ArrayT2 < T >
&X)
{ i_ point= X. i_ point;
    row= X. row;
    col= X. col;
    i_ point=new T *[row]; //为对象申请内存
    if( i_ point= =NULL)
    {cout<<"Error";exit(1);}
    for (int i = 0; i < row; i + +)
    i_ point[ i ] = new T [col];
    //从对象 X 复制数组元素到本对象
    T * *destptr= i_ point;
    T * *srcptr= X. i_ point;
    for( i = 0;i < row;i + +)
    for(int j = 0;j < col; j + +)
    destptr [i][j] =srcptr [i][j];
}
```

需要特别提及的是：类模板中成员函数的实现应与类模板的定义在同一文件中，否则，链接时会出错。类模板同样可以被继承，在派生类中添加新的成员，就可以满足应用程序的需要。

## 4 动态数组应用举例

若二维数组类模板保存在头文件“ArrayT2. h”中，则用它动态生成二维数组及完成矩阵加法运算和整体输出二维数组元素的代码如下：

```
#include <iostream. h>
#include "ArrayT2. h"
void main()
{ int n,m; cout<<"Enter n and m :";
cin>> n >> m;
ArrayT2 <int> A(n,m),C;
for(int i = 0;i < n;i ++) //为矩阵 A 中的元素
赋值
for(int j = 0;j < m;j ++)
A[i][j] = (i + 1) * j;
cout<<"Matrix A is"<<endl;
cout<<A;          //输出矩阵 A 中的元素
ArrayT2 <int> B(A); /将矩阵 A 的元素复制
给矩阵 B
cout<<"Matrix B is:"<<endl;
cout<<B;
C=A+B; //两矩阵相加
cout<<" C=A+B is:"<<endl;
cout<<C;
}
```

若输入 $n$ 值为 3，$m$ 值为 4，则程序运行后的输出为：

```
Matrix A is:
0  1  2  3
0  2  4  6
0  3  6  9
Matrix B is:
0  1  2  3
0  2  4  6
0  3  6  9...
C=A+B is:
0  2  4  6
0  4  8  12
0  6  12  18
```

## 5 结束语

从上面的分析可以看出，用函数模板实现动态数组比较简单，但因其函数的功能单一，且函数模板不能被继承，因而其重用受到一些限制；用动态数组类的方法，通过派生，能比较好地实现重用。但动态数组类因受数据类型固定的限制，对不同的数据类型，要写不同的数组类，增加了维护的成本。用数组类模板，能根据需要，动态生成不同数据类型的对象数组，利用 C++ 提供的继承机制，可以较好地实现代码重用。对数组类模板进行充分测试后，用于应用程序中，可以减少维护的开销，不但可以为程序开发节省人力和物力，还可以提高软件的可靠性。

**参考文献：**


[1] 杨进才，王奇玺. C++数组与指针深入剖析[J]. 微型机与应用，2000(10):7-9.

[2] 张雪彬，刘培国，曹 兵. 基于 C++语言的多维动态数组的实现[J]. 现代电子技术，2006(24):68-69.

[3] 蓝雯飞. 为 C++ 语言开发长度可变的数组类模板[J]. 微型机与应用，2004,25(3):381-384.

[4] 齐治昌，谭庆平，宁 洪. 软件工程[M]. 第 2 版. 北京：高等教育出版社，2004:365-366.

[5] msdn, VisualC++ DevelopeCentre, Library. Templates[OL]. http://msdn. microsoft. com/zh-cn/library/y097fkab. aspx, 2007.11

[6] 周美莲，李青. 动态分配多维数组及其应用[J]. 信息技术，


(下转第 86 页)

## 3 Dalvik 虚拟机进程间通信关于信号部分

Dalvik 虚拟机暴露的 API 中直接提供了对信号的支持,这体现在 android.os.Process 类中。其中包括了用于终结指定进程的 killProcess(int),以及用于向指定进程发送特定信号的 sendSignal(int,int)。目前可以发送的信号是 SIGNAL_KILL、SIGNAL_QUIT 和 SIGNAL_USR1,Dalvik 的源代码对这些信号都做了特殊处理。

上文对三个创建进程的 API 分析中省略了关于父子进程间通信的信号发送和接收的部分。信号处理也是进程间通信的主要手段,尤其是在 Linux 这样一个以进程为主体的操作系统之上。

以上这三个 API(对应的本地方法的源代码)中父进程都调用了函数 setSignalHandler(),它的实质是将进程中用于接收子进程的 SIGCHLD 信号的处理函数设为 sigchldHandler()。

在这个信号处理函数中,绝大多数部分代码用来根据不同的情况写日志,唯一一处真正用于进程间通信的程序逻辑是下面这段代码:

```
if (pid == gDvm.systemServerPid) { /* 子进程是一个系统服务进程 */
    LOG(.....);
    kill(getpid(), SIGKILL); /* 终止本父进程 */
}
```

即如果子进程是个服务进程,则将本进程(相对于这个子进程来说是父进程)终止。

之前在 fork system server 进程中也有类似的终结父进程的代码,笔者认为并不多余,这是个双保险以保证父进程的正确终结。

当创建非 zygote 进程和 system server 进程时,子进程似乎也有调用 setSignalHandler()。但是后面又调用了 unsetSignalHandler 把这个处理函数删除了。因此,完全符合 API 设计的语义。

如果虚拟机启动时,没有加上"-Xrs"和"-Xnoquithandler"参数,那么在创建非 zygote 进程时,将会触发一个线程运行 signalCatcherThreadStart 函数。这个线程的功能就是在一个无限循环中不断地监听两个信号:SIGQUIT 和 SIGUSR1。当 SIGQUIT 被捕获的时候,就打印 jni 全局参考表的信息;当接收到 SIGUSR1 信号时,会强制进行不回收软引用的垃圾收集,此时,在程序中向进程发送 SIGUSR1 信号,就可使进程进行不回收软引用的垃圾收集。

在 Dalvik 中没有暴露使用管道的 API。因此,在虚拟机初始化时就将管道信号屏蔽,以阻止管道端因读关闭,写入失败而直接退出。

## 4 结束语

Android 是一个开源的嵌入式操作系统,Dalvik 则是在它之上的核心组件,基于 Dalvik 的 Java 应用打破了 Sun 对 Java 世界的垄断,并使之成为当今最为流行的嵌入式应用开发标准之一。虽然随着 Android 的发展,新版本的不断推出,尚无法预测它的发展方向,因而文中内容可能存在时间和空间上的局限性,但是,对 Dalvik 的深入探索是极具意义的一种尝试,能够给 Dalvik 的使用、移植、研究以及虚拟机设计提供建议和参考,这是撰写本文的主要目的。

**参考文献:**


[1] 姚昱旻,刘卫国. Android 的架构与应用开发研究[J]. 计算机系统应用,2008,17(11):110-112.

[2] Sun Microsystems Inc. Java ME Technology API Documentation[EB/OL]. 2007. ttp://java.sun.com/javame/reference/apis.jsp.

[3] Google Inc. Android Documentation[EB/OL]. 2007. http://code.google.com/intl/en/android/Documentation.html.

[4] Venners B. 深入 java 虚拟机[M]. 第 2 版. 曹晓钢,蒋 靖 译. 北京:机械工业出版社,2003.

[5] 探矽工作室. 深入嵌入式 Java 虚拟机[M]. 北京:中国铁道出版社,2003.

[6] 陈闵中. Linux 在嵌入式操作系统中的应用[J]. 同济大学学报,2001,5(14):564-566.

[7] 河 秦,王洪涛. Linux 2.6 内核标准教程[M]. 北京:人民邮电出版社,2008.

[8] 李正平,徐 超,陈军宁,等. Linux 2.6 内核进程调度分析[J]. 计算机技术与发展,2006,16(9):82-84.

[9] 毛德操,胡希明. Linux 内核源代码情景分析[M]. 杭州:浙江大学出版社,2001.

[10] Sun Microsystems Inc. Porting Guide Sun Java(TM) Wireless Client Software2.0 Java Platform, Micro Edition[EB/OL]. 2007-05. http://java.sun.com/javame/reference/apis.jsp.


(上接第 82 页)


2007(2):21-23.

[7] 郑 莉,董 渊,张瑞丰. C++语言程序设计[M]. 第 3 版. 北京:清华大学出版社,2004:188-190,293-300.

[8] msdn, VisualC++ DevelopeCentre, Library. Operator Overloading[EB/OL]. 2007-11. http://msdn.microsoft.com/zh-cn/library/5tk49fh2.aspx,2007.11.

[9] Priestley M. Practical Object-orientec Design with UML [M]. 2nd ed(影印版). 北京:清华大学出版社,2004:176-177.