```
        DEC COUNT              ;判断多个汉字显示完否
        CMP COUNT,00H
        JNZ COUNT6
        POP DS                 ;恢复现象
        POP SI
        POP DI
        POP DX
        POP CX
        POP BX
        POP AX
        POP SP
        ;MOV AX,4C00H
        ;INT 21H
        IRET
DISP    ENDP                   ;远调返回
CODE    ENDS
        END START
```

浅谈 Borland C++4.0 的异常处理机制

贾晓东　(上海大学)

摘要:本文从抛出异常,捕获异常,对异常的处理以及异常指定等几个方面详细地讨论了 *Borland C++4* 的异常处理机制,分析了其不足之处并对解决办法进行了探讨。

异常指的是程序中出现了异常情况,通常指出错。异常处理是指让程序的一部分检测及报告异常情况而另一部分处理它们。Borland C++4.0 定义了异常处理的标准,这个标准确保程序始终受到面向对象的支持。在这个标准中值得强调的是引入了虚函数和利用对象定义异常,虚函数确保了程序具有最少的运行时间,若无异常抛出则程序额外的时间耗费将是零。Borland C++4.0 同时也支持 ANSI 标准,支持中止异常处理模式,程序运行时若有异常情况发生程序将中止,但是抛出一个异常后从抛出点收集信息,这些信息在研究程序失败的原因时大有用处。也可以在异常处理中指定程序终止前要采取的措施。

Borland C++4.0 允许异常为任意类型,但是最好把异常定义成对象,这样做得有好处,异常对象将和其他对象一样对待。

Borland C++4.0 的异常处理机制要求用到三个关键字:try,catch,throw,由 try 关键字来标识的 try 块后必须紧跟着由 catch 标识的异常处理程序块。如果 try 块抛出一个异常,程序控制权就转让给适当的异常处理程序块。程序应当试图捕获由任何函数抛出的异常,不能这样做时,将导致程序的异常终止。一个异常带着信息从抛出点来到捕获点,这些信息将告诉程序使用者在运行程序时遭遇到的异常。下面我们从几个方面来具体谈谈 Borland C++4.0 的异常处理机制。

一、抛出异常

C++中异常处理技术建立在非局部转移这一概念的基础上,当程序遭遇到一个异常情况时程序就抛出此异常情况,抛出异常会使程序在继续运行之前执行一个非局部转移把控制权转让给程序中另外一部分来处理这种问题。异常的抛出一般发生在 try 块中,一个有可能产生异常的代码块必须含有关键字 try 并具有下面的形式:

```
try{
  / / Take some action;
  }
```

这将指示程序准备测试异常,如果异常产生了程序流将中断,程序搜索匹配的处理程序,如果找到了堆栈回绕到这点,程序控制转让给处理程序块,如果没有找到程序将调用 terminati()函数。如果没有异常抛出程序将按正常顺序执行。异常的抛出方式有四种:

1.throw throw __ object,这种方式指定了 throw __ object 是传给异常处理程序块的异常。

2,throw,这种方式简单地指定最后抛出的异常被再抛出,至少要有一个异常抛出否则程序调用 terminate 函数中止程序。

3.void my_func() throw(A,B)

```
{
  / / Body of function;
  }
```

这种方式给出了函数 my func()能抛出的异常的表,除表中的异常没有其他的异常能传出函数体。如果一个既非 A 又非 B 的异常在 my func()函数中产生了,它将使得程序的控制转交给 unexpected()函数。

4.void my_func() throw()

```
{
   / / Body of function;
}
```

这种方式说明 my_func()不会抛出任何异常,若 my_func()中调用的函数抛出了异常,该异常在 my_func()函数体外将不可见。

二、处理异常

异常的处理部分由关键字 catch 来标识,处理部分必须紧接着 try 块。catch 关键字后可再接 catch 关键字,每个处理程序块只处理能和其关键字匹配或是能转换成参数表中类型的异常。每一个由程序抛出的异常都必须被捕获和交给处理程序处理。如果程序不能给一个抛出的异常提供一个处理块,程序将调用 terminate()函数。当和 catch 的参数表中参数匹配的异常被捕获程序控制权交给相应的处理程序块,堆栈回绕到处理块入口,处理程序决定采取处理程序异常的办法。Borland C++4.0 允许使用 goto 语句来转出处理程序块但不允许转入。处理程序执行完后程序将接着当前 try 块的最后一个处理程序块继续执行。

Borland C++4.0 的异常处理形式有两种,分别叙述如下:

```
1.try{
    / / include any code that might throw an exception;
  }
 catch(T X)
 {
   / / Take some action;
 }
```

这种形式指定了只能处理具有类型 T 的对象,如果参数是 T,T&,const T,const T&处理程序将处理满足下面条件之一的类型 X:

(1)T 和 X 具有同种类型;

(2)T 是 X 的基类;

(3)T 是指针,X 也是指针并且根据标准的异常指针类型转换能转换成 T;

```
2.try{
    / / Include any code that might throw an exception;
  }
catch(...){
    / / Take some actions;
}
```

catch(...)表明该处理程序能处理任何类型的异常,这种说明只能放在 try 块的最后一个处理程序块。

三、异常指定

Borland C++4.0 可以为一个函数指定它所能抛出的异常,这种异常的指定可以作为函数说明的后缀,其形式如下:

```
void f1(void)throw();/ / 该函数不抛任何异常;
void f2(void) throw(BETA);/ / 只能抛出 BETA 型异常;
void f3(void);/ / 该函数可以抛出任何异常;
```

函数后缀并不认为是函数类型的一部分,指向函数的指针不受函数异常指定的影响,这是因为这种指针只检查函数的返值和参数的类型,故如下的语句是合法的:

```
void ( * fptr)();
fptr = f1;
fptr = f2;
```

在重载函数时必须特别小心,因为异常指定没有被当作是函数类型的一部分,很有可能重载后的函数并不是所期望的。下面我们看一个如何进行异常指定并处理所有异常的例子:

```
#include< iostream.h >
class EXCEPTION{};/ / 异常说明;
EXCEPTION e;
void f2(void) throw(EXCEPTION){
    cout< < "f3() was called"< < endl;
    throw(_e);
  }
void f2(void) throw(){
    try{
      cout< < "f2() was called"< < endl;
      f3();
    }
    catch(...){
      cout< < "f2() has elements with exceptions!"< < endl;
    }
}
int main(void){
  try{
    f2();
    return 0;
  }
  catch(...){
      cout< < "Need more handlers!"< < endl;
      return 1;
  }
  return 1;
}
```

程序输出结果为:f2() was called
f3() has called
f2() has element with exceptions!

四、异常处理中的构造函数与析构函数及资源问题

当一个异常对象被抛出时,调用这个异常对象构造函数的一个副本用来在抛出点初始化一个暂时对象。当程序流被一个异常打断,程序将为所有自动对象调用析构函

数,这些自动对象是在进入 try 块时构造的。很遗憾的是 C++ 中长期存在的构造函数中的异常处理在 Borland C++4.0 中也没有得到很好的解决,这主要的是因为构造函数不返回值,而且也没有直接的方法向调用者报告失败。如果异常在构造某个对象时被抛出了,只有已完全构造好的对象才能调用析构函数。例如,当一个对象数组正在构造时异常抛出了,只有那些已完全构造好了的数组元素才能调用析构函数。这样如果在构造函数中申请了资源而又抛出了异常,所申请的这部分资源就悬挂起来了。解决这个问题目前还没有最好的方案,值得提倡的是使用函数闭包来处理。所谓闭包是借用数学中的一个概念,这里其含义是指一个代码域,它的边界是由一个前文和一个后文组成的,前文和后文必须同时存在,否则是一个运行时错误,例如:

```
hDC=GetDC(hWnd);
TextOut(hDC,x,y,string,size);
ReleaseDC(hWnd,hDC);
```

我们可以把它看成一个整体,成为一个闭包。

要较好解决前面提到的资源悬挂问题,我们建立一个单独的类,该类围绕资源建立一个函数闭包,类的构造函数相当于前文,析构函数相当于后文。当然无论异常情况是否被抛出,都得编一个小类,虽然增加了代码量,但是却不浪费资源是值得的。我们用一个小例子来说明这一方法。

```
#include<windows.h>
class DC{
    HDC hDC;
  public:
    DC(){hDC=GetDC(NULL);}
    operator HDC(){return hDC;}
    ~DC(){ReleaseDC(NULL,hDC);}
};
```

使用这个类就不必担心异常抛出时设备上下文挂起,需要设备上下文的函数可以如下使用 DC 类:

```
void MyFunc();
    {
      DC MyDc;
      //call the GDI function on the DC;
      //...
    }
```

当在设备上下文上处理 GDI 操作时,如有异常抛出,那么 MyFunc()将被退出,但是在退出前程序将调用 DC 的析构函数释放设备上下文。

函数闭包可以说是处理异常情况中资源问题的一种很妙的方法,我们提倡把所有的资源都用如此的方法封闭起来,以免出现内存漏洞,死包围,和死锁定的代码。在处理动态内存分配时这显得更为重要。

五、结束语

由于程序利用异常处理能以有秩序,有组织及一致的方法截取及处理异常情况,所以异常处理将成为 C++4.0 语言的下一个热点,而 Borland C++4.0 处理异常的机制总的来说是近乎完美的,它必将成为广大程序员的好帮手。

参考文献:

[1]C++异常处理<<微型计算机>>1994,3,

[2]Borland C++4.0 程序设计 成都科技大学出版社

[3]C++高级编程技术 电子工业出版社

将 FoxBASE+的数据转换为 WATCOM SQL 的数据

庞建民　宁　钰　(解放军信息工程学院)

摘要:本文给出了一种将 *FoxBASE+*中的数据转化为 *WATCOM SQL* 中数据的实现途径。本文中的程序采用 *Powerscript* 语言编写。

一、引 言

Powerbuilder 是目前最流行的数据库前台开发工具,它支持 Sybase、Oracle、Informix、Ingress、DB2 等多种数据库管理系统。另外,它提供的内置数据库管理系统 WATCOM SQL,在没有后台数据库管理系统的情况下,也可用来开发大型应用。由于我国用 dBASE、FoxBASE+等数据库管理系统开发的软件较多,因此无论从系统的互连还是系统的更新换代来讲,都可能需要将 FoxBASE+的数据转换成 WATCOM SQL 中的数据,为此我们用 Powerscrip 语言编写了一个程序,它能完成将 FoxBASE+中的数据转换成 WATCOM SQL 中的数据。下面我们以分析 FoxBASE+中数据存放格式开始,介绍我们的实现途径。

二、FoxBASE+的数据库文件 (.DBF 文件)的结构

FoxBASE+中的数据以数据库文件为单位存放,有时