Exceptions in Libraries

Ulrich W. Eisenecker
Daimler Benz AG
Research Center
Ulm, Germany
eisenecker@dbag.ulm.DaimlerBenz.COM

Doc No: WG21/N0606, X3J16/95-0006
January 17, 1995

Abstract

This article describes a way to introduce a checked and unchecked version of a library functionality based on a technique which separates a single class into one class, called BODY, and a familiy of classes, called BEHAVIOR. A library should at least provide two BEHAVIOR classes for a BODY class, namely checked and unchecked behavior. Additionaly special versions of BEHAVIOR classes can be provided by the programmer. Of course they must cover all necessary functions for the BEHAVIOR family. The proposed technique could help to substitute some of the *undefined behavior* places in the *C++ Standard Library*, where exceptions could do much better work.

1 Introduction

In chapter 26 "Numerics library" of the current draft I read, that valarray and associated classes lack any discussion of possible exceptions (box 101). Furthermore the phrase "if (some condition) ..., the behavior is undefined" can be found very often. This is unsatisfactory to me because of the following reasons:

- C++ supports exceptions (because of many good reasons). The libraries included in the standard should therefore use exceptions as much as sensible.
- At least for developing and testing a safe version of a library is needed. By a safe version I understand a library which avoids undefined behavior by throwing exceptions, if not possible otherwise.
- Even if a library serves as a building block for other libraries, I do not want to write either safe wrapper classes or adopt a sloppy programming style.

2 Discussion

Exceptions were introduced in C++ - among other reasons - because of enabling a better design of libraries. With exceptions functions do not need to return a error/success value which has to be checked in the calling program. Instead an error is detected in the function being called. Then the error is thrown as an exception to the calling program which knows more about possible reasons for this error or how to deal with possible reasons of the error. The classic example for this is range checking of an array access: the calling program causes an error which is detected in the access function. But the reason, if known at all, can be found in the calling program. Sometimes the detection of errors can be time or space consuming. If there are programs which are believed to be free of wrong assumptions, error checking and throwing of exceptions may be unwanted. Therefore I prefer libraries which provide checked functions with exceptions in all those cases where is no penalty in time or space. Functions, for which error checking and exceptions cost time and space, should be available in a safe variant and a variant without error checking, so that a minimum of time and space is required. It is now left to the programmer, which functions he/she uses. C and C++ always pursued this philosophy. Admittedly it may be annoying to replace the checked version of a function by its unchecked alternative and vice a versa by editor commands. It would be much more comfortable to change the behavior of a class only at one single point, so that all of their relevant functions change their behavior to checked or unchecked. This could be achieved via a compiler option. But, if the programmer should be enabled to provide his own desired mixture of checked and unchecked behavior, the compiler should not be burdened with something that could be expressed better in the language directly.

3 Solution

One possibility could be using defines and conditional compilation. Another technique is to separate a single class into one class, called BODY, and a familiy of classes, called BEHAVIOR. A library should at least provide two BEHAVIOR classes for a BODY class, namely checked and unchecked behavior. Additionaly special versions of BEHAVIOR classes can be provided by the programmer. Of course they must cover all necessary functions for the BEHAVIOR family. To realize this idiom with inheritance and polymorphism is not free of time and cost when the unchecked behavior is used and it contradicts the philosophy of the template classes, which are the main part of the coming *C++ Standard Library.*

4 The idea

So the idiom should be realized using templates and inline functions. The following program [1] illustrates this idea. Instead of constructors other inline member functions (maybe static) could be used of course. Every compiler should be able to optimize away the call of an empty function.

```
#include <iostream.h>
template <class T,class Behavior>
class Array
{
public:
    Array(int size);
    ~Array();
    T& operator[](int i);
```

[1] compiled and tested with Borland C++ 4.5

```
    T sum();
    const int& size() const;
private:
    T* data;
    int mySize;
};

template <class T,class Behavior>
Array<T,Behavior>::Array(int size) : mySize(size)
{
    (Behavior(mySize));
    data = new T[mySize];
}

template <class T,class Behavior>
Array<T,Behavior>::~Array()
{
    delete[] data;
}

template <class T,class Behavior>
T& Array<T,Behavior>::operator[](int i)
{
    (Behavior(0,size(),i));
    return data[i];
}

template <class T,class Behavior>
T Array<T,Behavior>::sum()
{
    (Behavior(size()));
    T s = (*this)[0];
    for (int i = 1;i < size();i++)
    s += data[i];
    return s;
}

template <class T,class Behavior>
const int& Array<T,Behavior>::size() const
```

```
{
    return mySize;
}

class Checking
{
public:
    Checking(int lowerLimit,int upperLimit,int index);
    Checking(int size);
};

class NoChecking
{
public:
    NoChecking(int lowerLimit,int upperLimit,int index);
    NoChecking(int size);
};

inline
Checking::Checking(int lowerLimit,int upperLimit,int index)
{
    if (index < lowerLimit || index > upperLimit)
        throw "range error in T& Array<T,Behavior>::operator[](int i)";
}

inline
Checking::Checking(int size)
{
    if (size <= 0)
        throw "size of Array<T,Behavior> must be positive";
}

inline
NoChecking::NoChecking(int lowerLimit,int upperLimit,int index)
{
};

inline
NoChecking::NoChecking(int size)
```

```
{
}

void main()
{
    try {
        Array<int,Checking> myArray(1);
        myArray[0] = 5;
        myArray[1] = 6;
        myArray[2] = 7;
        myArray[3] = 8;
        myArray[4] = 9;
        cout << "Sum of " << myArray.size() << " elements is "
             << myArray.sum() << endl;
    }
    catch (char* error) {
        cout << endl << error << endl;
    };
}
```

5 Problems

There is one problem using this realization: BEHAVIOR classes can not access any member of the BODY class. Interestingly this problem does not occur until one tries to instantiate the template for the BODY class. I do not know if I oversaw something or if perhaps nested templates could help.

6 Conclusion

The BODY BEHAVIOR idiom allows to parameterize template classes with error detection behavior. Additionaly a programmer can easily provide his or her own version of error detection on a fine grained level. In the absence of error checking no additional code is generated by the compiler.