Tech Tips

Examining C++ Divide-by-Zero Behavior on Windows
by Boris Eligulashvili

DBNull-Like Types and Equality
by Stephen Quattlebaum

The AutoResourceT Template
by Gigi Sayfan

December 01, 2003
URL:http://www.drdobbs.com/tech-tips/184416715

Treatment of arithmetic expressions that can result in divide-by-zero exceptions in C++ and other programming languages is laxly "governed" by standards that usually leave some or all of the behavior to software tool vendors. This makes the design of error handling problematic in cross-platform tools that involve heavy use of division. The ISO/IEC C++ Standard states, for example, that "If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined, unless such an expression is a constant expression (5.19), in which case the program is ill formed. [Note: most existing implementations of C++ ignore integer overflows. Treatment of division by zero, forming a remainder using a zero divisor, and all floating point exceptions vary among machines, and is usually adjustable by a library function.]"

Thus, the behavior is governed by a combination of hardware implementation, vendor software workarounds, and to a lesser extent, the standards that guide the software vendors.

So it is interesting to review the behavior of divide-by-zero situations for the Microsoft Visual C++ compiler. I've taken an experimental approach and compiled a summary of conditions that give rise to differing behavior when a divide by zero is attempted in a program built using Visual C++ 6.0 (Table 1). Visual C++ is sometimes able to catch problems at compile time. In the trivial case where the denominator is the integer constant 0, for example, Visual C++ issues Error C2124: divide or mod by zero. For floating-point arithmetic only, a warning is issued if a constant expression evaluates to zero — Warning C4723: potential divide by 0.

Table 2 shows the results that were observed at runtime on a Windows NT machine: The results are sometimes surprising if you were not aware of them. The system displays the different text emitted when exceptions are thrown in Debug or Release mode (they sometimes differ). It is possible to catch a hardware exception in Release mode if the variables are local integers. The result variable in the catch block is not changed, even if the result variable in the assignment is not used. For floating-point operations, you get infinity or indeterminate values. Don't expect your program to crash in those cases — you need to check the result and this can differ between architectures. Still even on Windows alone, the results aren't always as you would expect. Consider a floating-point variable named "Var":

1. If Var equals the indeterminate value (0.0/0.0), after evaluating b in bool b = (0.0 = = Var), b is True.

2. If Var equals NaN, after evaluating b in bool b = (0.0 = = Var), b is False.

3. If Var equals NaN, after evaluating i in int i = -100 + Var, i is 0.

In conclusion, catching divide-by-zero exceptions will work in some cases. Basic programming hygiene dictates that care should be taken to destroy the objects that were allocated on the heap by allocations in the **try** block, if applicable. One example of a possible cause of an integer denominator being zero is a loss of precision. The obvious fail-safe error-checking idiom is to always check denominators in the case of integers, but remember that these checks come with a price and might be performance inhibitive. Of course, for floating-point arithmetic, there are applications where infinity results are desired (such as when calculating a function that has poles).

The bottom line is that problems arise when programmers do not know what to expect when dividing where zero can somehow creep into the denominator. Hopefully, this analysis will provide you with some useful insight into indeterminate situations.

DBNull-Like Types and Equality

By Stephen Quattlebaum

stephen@covidimus.net

The .NET framework includes a type called **DBNull**, which exhibits singleton semantics: It has a private constructor and a single statically created instance that can be accessed through the **DBNull.Value** property. In theory, all references to **DBNull** instances in a particular AppDomain point to the same object.

This allows code like the following to work:

```
if(myVal == DBNull.Value)

    do something...
```

where otherwise one would have to write code like:

```
if(myVal is DBNull)

    do something...
```

There are more benefits to this than may immediately be apparent. Most importantly, it means that an object can be compared to **DBNull** by a generic comparator that only knows how to call **Equals()** on its parameters, instead of having to be treated specially. Unfortunately, this way of doing things sometimes malfunctions.

While doing some database work, I came across the need for additional special values that needed to exhibit the same semantics as **DBNull**. **DBIncrement** is one such type, implemented to have the same singleton semantics as **DBNull**. It worked great until one day I found that "myVal == DBIncrement.Value" had started to evaluate to False, even though I could see in the debugger that the type of **myVal** was **DBIncrement**. After struggling with it for a while (and seriously contemplating the thought that Microsoft might have fundamentally changed the C# language between VS 2002 and VS 2003), the problem turned out to be that **myVal** referred to an instance of **DBIncrement** that had been deserialized using .NET serialization. The deserialization engine created a new **DBIncrement** value in memory to reflect the one that had been serialized during a previous run, and it was not the same object that **DBIncrement.Value** pointed to for the current run.

The upshot of this is that, in the presence of object activation techniques such as remoting and serialization, relying on reference-equality for **DBNull**-like types simply doesn't work as expected. It is never safe to say something like "myVal == DBIncrement.Value" if there's a chance that the object came into the current AppDomain by deserialization or some other exotic activation mechanism. This means that in library-level code, where you can't be certain of the origin of the objects your clients give you, you should always use "myVal is DBIncrement" instead, even if the type you're working with is a singleton type with **DBNull**-like semantics.

The AutoResourceT Template

By Gigi Sayfan

the_gigi@hotmail.com

Resource lifetime management is a difficult problem. Memory is what comes to mind immediately. However, every nontrivial program must handle lots of other resources as well, such as network connections, database connections, file handles, graphic objects, and the like. The famous RAII idiom "Resource acquisition is initialization" solves this problem by wrapping every resource in an object that cleans up the resource upon destruction. Here is a simple example:

```
 class AutoFile

{

public:

  AutoFile(const char * n, const char * a)

  { p = fopen(n,a); if (p==0) throw                     FileOpenException(errno); }

  AutoFile(FILE* pp)

  { p = pp; if (p==0) throw                             InvalidFilePtrException(errno); }

  ~AutoFile() { fclose(p); }

  operator FILE*() { return p; }

  // ...

private:

  FILE * p;

};

void f(const char* fn)

{

    File_handle f(fn,"rw");     // open fn for                  // reading and                  // writing

    // use file through f

}
```

There are some problems with this solution:

1. You have to write a special class for every resource.

2. The class does too much: It creates the resource and manages its lifetime. To acquire the resource, you must understand the implementation of the constructor and how it is using the arguments.

3. It assumes you are using exceptions.

4. There are issues with ownership when using a copy constructor (the resource will be released twice).

5. You can't detach the internal resource and use it outside the scope it was acquired in.

We can solve points 2 through 5 by using this class instead:

```
class AutoFile

{

public:

  explicit AutoFile(FILE* pp = 0)

  { p = pp; }

  ~AutoFile() { if (p != 0) fclose(p); }

  operator FILE*() { return p; }

  FILE * Detach() { FILE * temp = p; p = 0;                     return temp; }

  // ...

private:

  AutoFile(const AutoFile & other);

//

explicitely hide copy constructor

  AutoFile & operator=(const AutoFile & other);

// explicitely hide assignment operator

  FILE * p;

};
```

Now you must pass an initialized FILE pointer, which solves point (2). No exceptions are used (3). Copy construction is forbidden and the explicit constructor prevents the automatic conversion of the **AutoFile** object to a FILE *. Finally, the **Detach()** method allows (you guessed it) the detaching of internal resources, if necessary.

The reason I didn't introduce the second class immediately is to demonstrate how complicated and involved a deceptively simple wrapper class like that can be. For your information, I based the first class implementation on a similar class from Bjarn Stroustrup's own FAQ: http://www .research.att.com/~bs /bs_faq2.html#finally.

The moral of the story is that you don't want everybody and his sister running around creating their own versions of these wrapper classes with ever so slightly different assumptions. The solution lies in the realm of generic programming and templates. If you observe the **AutoFile** class closely, you will notice that we can generalize it to any resource if we have the following:

1. A cleanup function that accepts a single argument of the resource type (FILE * in the case of **AutoFile**).

2. An invalid/uninitialized value (0 in the case of FILE* or any pointer) .

3. In order for detach to work, the resource should have value assignment (support operator= in an intuitive way).

Listing 1 shows the templates needed to realize this. Before delving into the details, let's see how easy it is to use it. It comes with a default configuration class that works out of the box on any pointer-like resource. Pointer-like resources are objects (or real pointers) whose cleanup function is the **delete** operator and their invalid/uninitialized value is 0. For example, any object allocated on the heap:

```
DoSomethingWithObjectPtr(Object * pObj) {...};

DoSomethingWithObjectRef(Object & obj) {...};

{

    AutoResourceT<Object *> pObj(new Object("Argument 1",

"Argument 2", 45));

    DoSomethingWithObjectPtr(pObj);

    DoSomethingWithObjectRef(*Pobj);

    // No need to delete. Automatically deleted

    // by the template

}
```

I know, I know. why not use **auto_ptr**? Two reasons:

1. **auto_ptr** has funny transfer semantics when you assign one **auto_ptr** to another. The original **auto_ptr** loses its resource. This is what I (and others) call unfair surprise. **AutoResourceT** prohibits this. Note that assignment is dangerous in general when it comes to managing resources because you can

perform assignments multiple times; then the whole issue of what to do with the previous resource arises.

2. **auto_ptr** deals with pointers only, while **AutoResource** can manage any resource.

Now, suppose your database API has **open_db()** and **close_db()** functions and an invalid value of DB_INVALID. We need to specialize the **AutoResourceConfigT** template. It may sound a little frightening, but there is nothing to it really:

```
template <db_handle>

struct AutoResourceConfigT

{

    static db_handle  GetInvalidValue()

     {

     return DB_INVALID;

     }

    static void Release(db_handle handle)

     {

      close_db(handle);

     }

};
```

You just have to replace **T** with your resource type, return the proper invalid value from **GetInvalidValue()**, and call the proper cleanup function in **Release()**. This is what they call "full specialization" since you replace all the template arguments with your own type.

Then you have to instantiate the **AutoResourceT** template itself:

```
//Declare a typedef for ease of use

typedef AutoDB AutoResourceT<db_handle>

// takes out specialization of

// AutoResourceConfigT<db_handle> by default

{

    AutoDb db(open_db("DB name));

    // do some useful db stuff

    ...

    // No need to cleanup

}
```

AutoResourceT uses Generative Programming techniques to configure a template with a necessary type. The problem was how to supply a function pointer (the **Release** function) and an invalid value of type **T** to **AutoResourceT**. I couldn't find a way to pass a function pointer directly, so I decided to use another template with static functions that provide the functionality I needed.

The constructor has a default parameter of of type **T** and a default value of **Config::GetInvalidValue()**. This ensures that the **Config::Release()** function will not be called on invalid/uninitialized resource. The destructor checks if the resource is valid and, if it is, it releases it by calling the **Config::Release** function. The operator **T** is an implicit conversion operator that simply returns the managed resource. This means that an instance of **AutoResourceT<T>**, such as **AutoDB**, can be passed to functions tht expect **dn_handle**. Finally, the **Detach()** method returns the resource after copying it to a temporary, and assigns the **Config::GetInvalidValue()** to the **m_resource** data member. This ensures that the detached resource will not be released when exiting the scope.
w::d

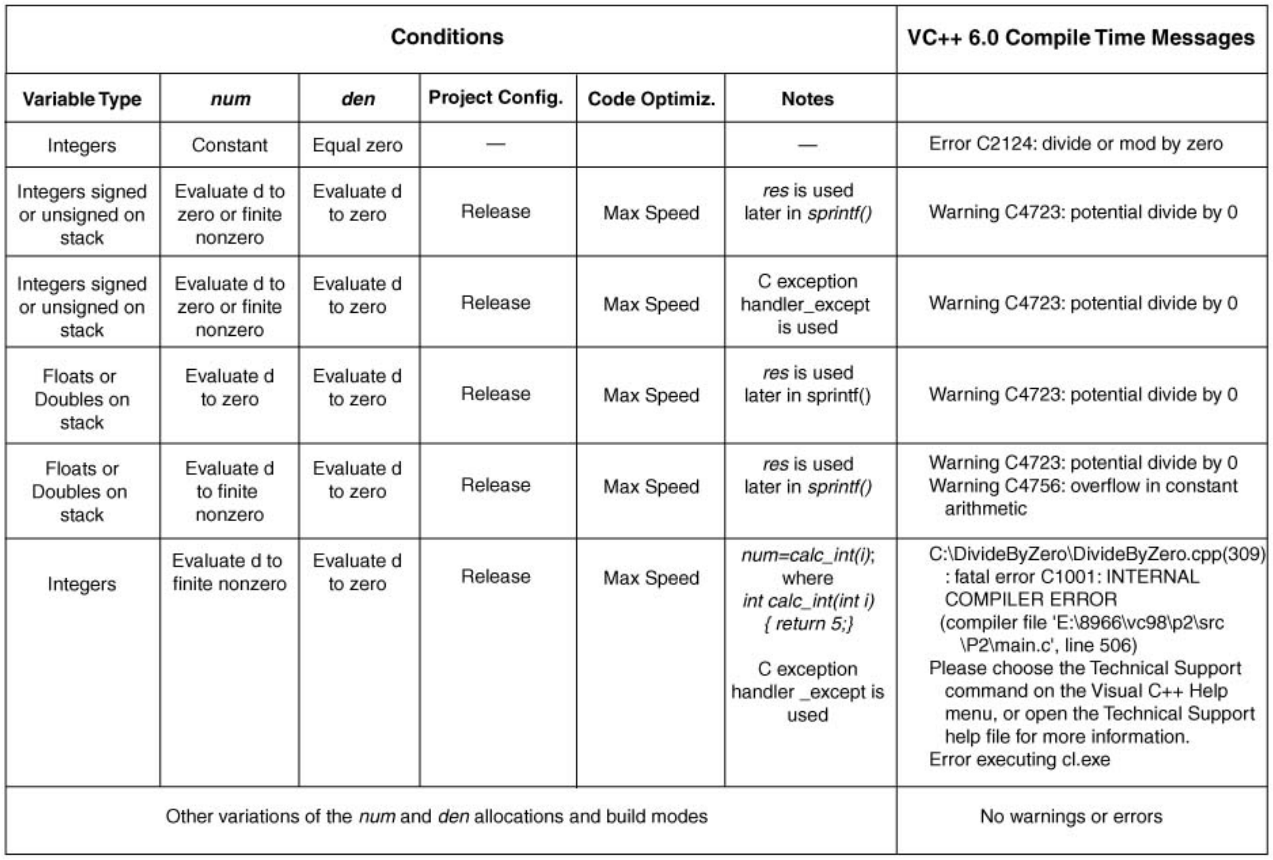
George Frazier is a software engineer in the System Design and Verification group at Cadence Design Systems Inc. and has been programming for Windows since 1991. He can be reached at georgefrazier@ yahoo.com.

Listing 1 The AutoResourceT template

```
// Config struct template for pointer-like resources. Need to
// specialize for non-pointers
template <typename T>
struct AutoResourceConfigT
{
             static T          GetInvalidValue()             { return (T)0; }
             static void Release(T t)                        { delete t; }
};

template <typename T, typename Config = AutoResourceConfigT<T> >
class AutoResourceT
{
             public:
```

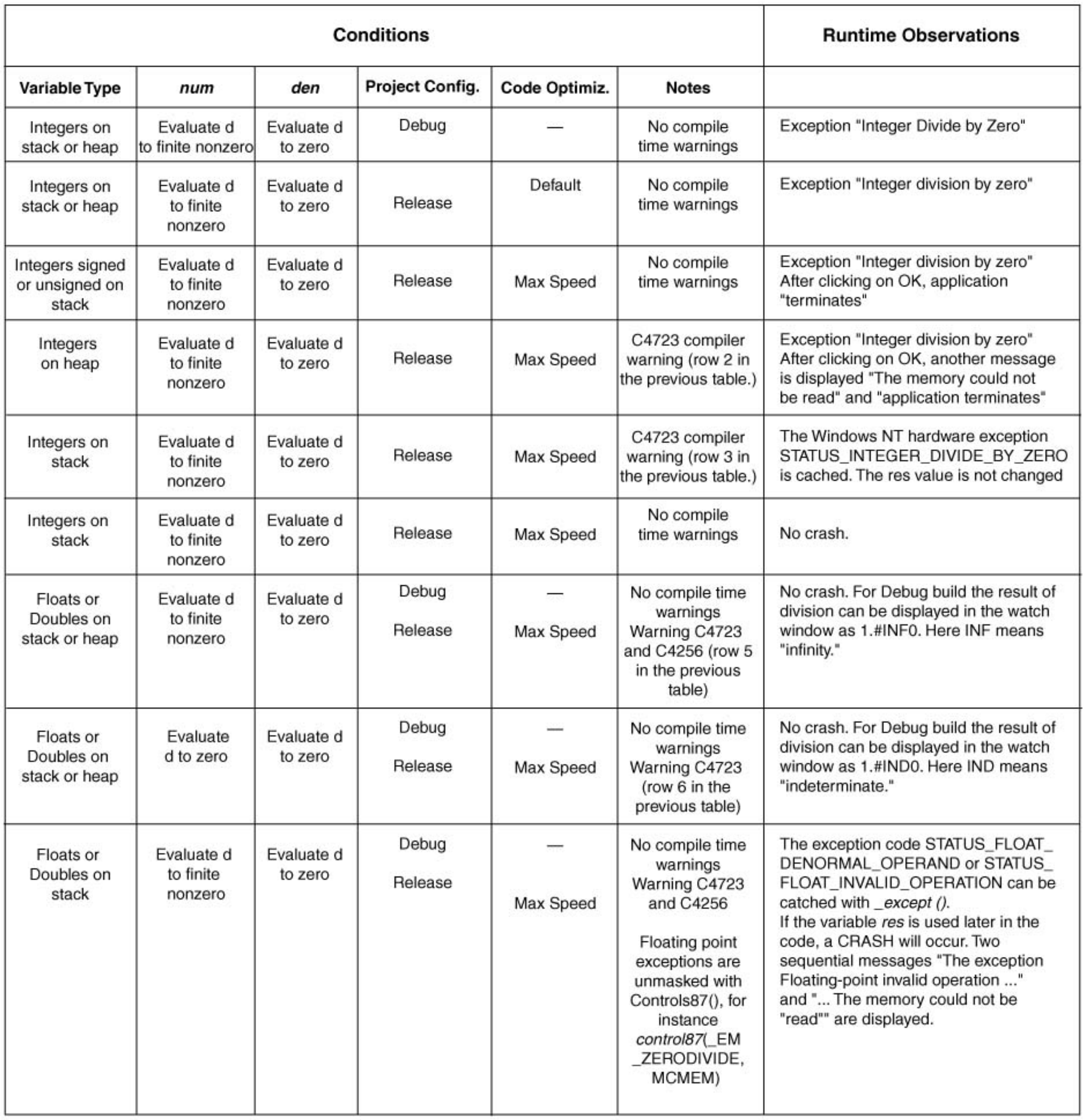
```
                        // Takes ownership of passed resource, or initializes internal resource
                        // member to invalid value
                        explicit AutoResourceT(T resource = Config::GetInvalidValue()) :
                                   m_resource(resource)
                        {
                        }

                        // If owns a valid resource, release it using the Config::Release
                        ~AutoResourceT(void)
                        {
                                    if (m_resource != Config::GetInvalidValue())
                                    {
                                      Config::Release(m_resource);
                                    }
                        }

                        // Returns the owned resource for normal use. Retains ownership.
                        operator T()
                        {
                                    return m_resource;
                        }

                        // Detaches a resource and returns it, so it is not release automatically
                        // when leaving current scope. Note that the resource type must support
                        // the assingment operator and must have value semantics.
                        T Detach()
                        {
                                    T temp = m_resource;
                                    m_resource = Config::GetInvalidValue();
                                    return temp;
                        }

            private:
                        AutoResourceT(const AutoResourceT &);
                        // hide copy constructor
                        AutoResourceT & operator=(const AutoResourceT &);
                       // hide assignment operator
            private:
                        T           m_resource;
};
```

Table 1 Error conditions and VC++ 6.0 messages



Conditions						VC++ 6.0 Compile Time Messages
Variable Type	*num*	*den*	Project Config.	Code Optimiz.	Notes	
Integers	Constant	Equal zero	—		—	Error C2124: divide or mod by zero
Integers signed or unsigned on stack	Evaluate d to zero or finite nonzero	Evaluate d to zero	Release	Max Speed	*res* is used later in *sprintf()*	Warning C4723: potential divide by 0
Integers signed or unsigned on stack	Evaluate d to zero or finite nonzero	Evaluate d to zero	Release	Max Speed	C exception handler_except is used	Warning C4723: potential divide by 0
Floats or Doubles on stack	Evaluate d to zero	Evaluate d to zero	Release	Max Speed	*res* is used later in sprintf()	Warning C4723: potential divide by 0
Floats or Doubles on stack	Evaluate d to finite nonzero	Evaluate d to zero	Release	Max Speed	*res* is used later in *sprintf()*	Warning C4723: potential divide by 0 Warning C4756: overflow in constant arithmetic
Integers	Evaluate d to finite nonzero	Evaluate d to zero	Release	Max Speed	*num=calc_int(i)*; where *int calc_int(int i) { return 5;}* C exception handler _except is used	C:\DivideByZero\DivideByZero.cpp(309) : fatal error C1001: INTERNAL COMPILER ERROR (compiler file 'E:\8966\vc98\p2\src \P2\main.c', line 506) Please choose the Technical Support command on the Visual C++ Help menu, or open the Technical Support help file for more information. Error executing cl.exe
Other variations of the *num* and *den* allocations and build modes						No warnings or errors

Table 2 Runtime results on Windows NT



Conditions						Runtime Observations
Variable Type	*num*	*den*	Project Config.	Code Optimiz.	Notes	
Integers on stack or heap	Evaluate d to finite nonzero	Evaluate d to zero	Debug	—	No compile time warnings	Exception "Integer Divide by Zero"
Integers on stack or heap	Evaluate d to finite nonzero	Evaluate d to zero	Release	Default	No compile time warnings	Exception "Integer division by zero"
Integers signed or unsigned on stack	Evaluate d to finite nonzero	Evaluate d to zero	Release	Max Speed	No compile time warnings	Exception "Integer division by zero" After clicking on OK, application "terminates"
Integers on heap	Evaluate d to finite nonzero	Evaluate d to zero	Release	Max Speed	C4723 compiler warning (row 2 in the previous table.)	Exception "Integer division by zero" After clicking on OK, another message is displayed "The memory could not be read" and "application terminates"
Integers on stack	Evaluate d to finite nonzero	Evaluate d to zero	Release	Max Speed	C4723 compiler warning (row 3 in the previous table.)	The Windows NT hardware exception STATUS_INTEGER_DIVIDE_BY_ZERO is cached. The res value is not changed
Integers on stack	Evaluate d to finite nonzero	Evaluate d to zero	Release	Max Speed	No compile time warnings	No crash.
Floats or Doubles on stack or heap	Evaluate d to finite nonzero	Evaluate d to zero	Debug Release	— Max Speed	No compile time warnings Warning C4723 and C4256 (row 5 in the previous table)	No crash. For Debug build the result of division can be displayed in the watch window as 1.#INF0. Here INF means "infinity."
Floats or Doubles on stack or heap	Evaluate d to zero	Evaluate d to zero	Debug Release	— Max Speed	No compile time warnings Warning C4723 (row 6 in the previous table)	No crash. For Debug build the result of division can be displayed in the watch window as 1.#IND0. Here IND means "indeterminate."
Floats or Doubles on stack	Evaluate d to finite nonzero	Evaluate d to zero	Debug Release	— Max Speed	No compile time warnings Warning C4723 and C4256 Floating point exceptions are unmasked with Controls87(), for instance *control87*(_EM _ZERODIVIDE, MCMEM)	The exception code STATUS_FLOAT_ DENORMAL_OPERAND or STATUS_ FLOAT_INVALID_OPERATION can be catched with *_except ()*. If the variable *res* is used later in the code, a CRASH will occur. Two sequential messages "The exception Floating-point invalid operation ..." and "... The memory could not be "read"" are displayed.