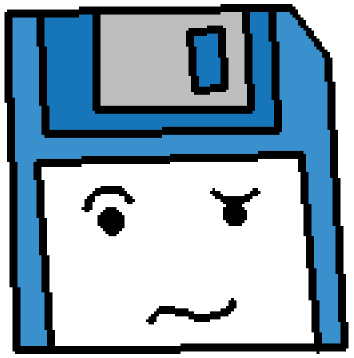
The Messiness Of Software Development

Written by Rachel J. Morris, last updated 2015-01-24

What could possibly go wrong?



Errors

Syntax errors are easy to protect against – your program won't build if there is a syntax error in the code.

Logic errors can be very difficult to spot, because they don't cause the program to crash or not build – the program runs, but it doesn't do that you intended for it to do.

Run-time errors can cause your program to crash, or generate errors, or otherwise make it so your program no longer functions.

This could be caused by a logic error, a user error, an error by somebody extending your original work, or by something else.

Errors

We can try to write our software so that no errors ever occur, but can you ever be certain that it is 100% bug free, or that the user will never cause an error?

There are techniques we can use to mitigate these problems.

Error Mitigation

Testing

We can use testing to search for run-time errors and logic errors, either by having a **tester** manually test the software, by writing **automated tests** that continue testing the software after every change, and writing **unit tests** that test each unit of the code (single functions, functionality, etc.)

Method

We can also follow certain design principles to try to minimize the likelihood of errors manifesting due to bad code, bad changes, bad process, and by ensuring code is reviewed by multiple developers before being added to the overall codebase.

Error Mitigation

Error Handling

We can also add code into our program to look out for errors that we know might happen, and handle it appropriately and cleanly if it does occur – so the program can still function, or at least clean up without causing problems to saved data or the system.

“Engineering”

Software can be extremely fragile, and the software engineering field is *not* a field like architecture, where there is a definite set of guidelines in order to ensure that the product **functions**, **is safe**, or even to ensure that development happens in a certain amount of time and with a certain budget.

Error Mitigation

Who's responsible for making sure software functions, is secure, or at least breaks *safely?*

How big is the risk associated with any given product?

Video games

Social Media

Vehicle Software

Media players

Legal Software

Medical Software

Organizer apps

Defense Software

Error Mitigation

If an airplane's software encounters an error, it can't just reboot its system or restart its software mid-flight.

If a legal case depends on documents from a business, loss/corruption of those documents due to software failure could be costly.

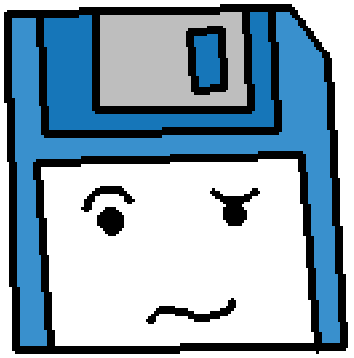
For personal software, it could just be losing some data or some time, but low-quality software can lead to loss of money, privacy, security, or even life.

Luckily...

Nobody's life is depending on our programming assignments.

But what can we do to try to build stable software?

Handling Errors



Sources of errors?

Trying to open a file that doesn't exist.

A memory access violations.

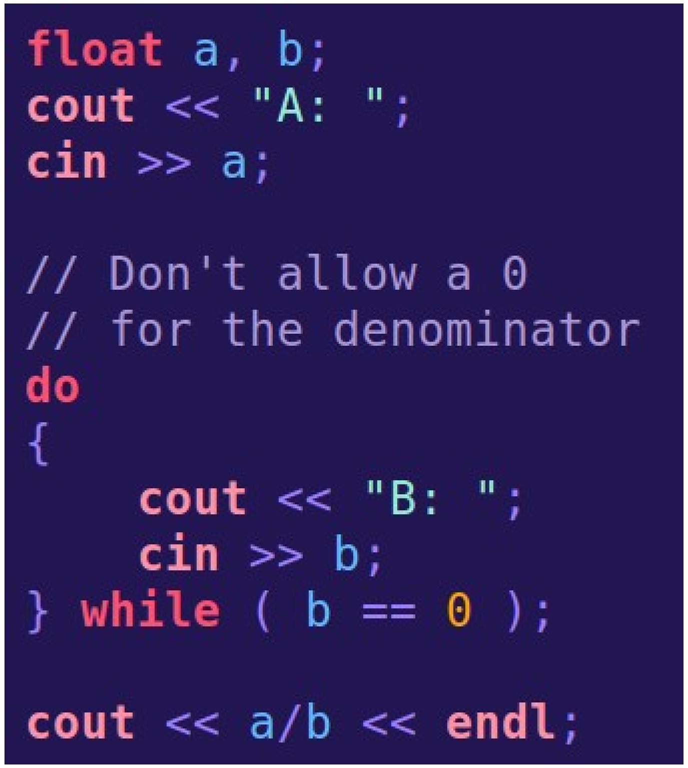
Invalid math (Division by 0...)

Not enough memory.

Receiving a data type as input that is incompatible with the data type we're expecting

What can we do?

We could simply check against invalid input...



```
float a, b;
cout << "A: ";
cin >> a;

// Don't allow a 0
// for the denominator
do
{
    cout << "B: ";
    cin >> b;
} while ( b == 0 );

cout << a/b << endl;
```

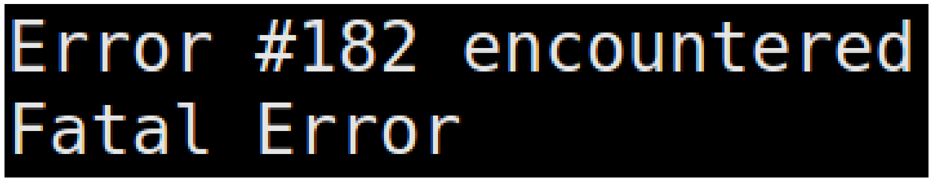
But are we always going to know what input is invalid ahead of time?

What if we're using someone else's code and there's an error?
How do we detect it?

What do we do when the program has an error?

Old school...

An old way of handling errors is to return an **error code** – 0 for “fine” and other numbers for other problems (which usually the programmer “would” know...)



```
Error #182 encountered
Fatal Error
```

The user can't tell what went wrong and can't hope to try to avoid the problematic behavior, or fix it!

Another way to report errors

We can write our software to handle errors so that:

An error happens. An **exception** is **thrown.**

The caller knows that exceptions could occur while using this code, so it **tries** the code, and is ready with a **catch** for exceptions.

The **catch** code looks at the **exception** and safely handles the exception.

Another way to report errors

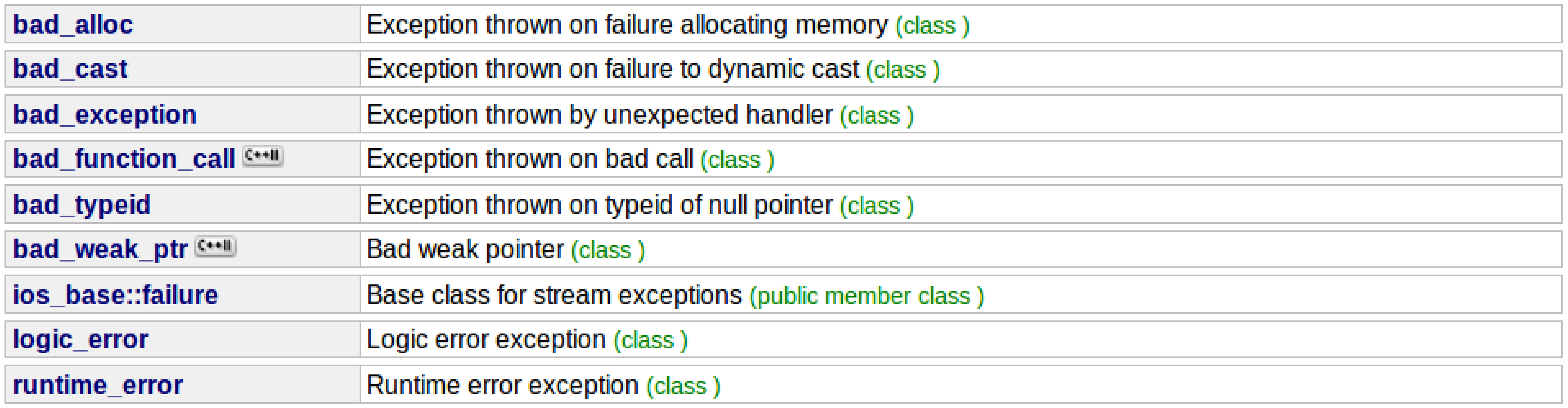
When an exception is **caught**, control of the program is transferred to a **handler**.

But, to detect any **thrown** exceptions, we have to be “listening” for it, with the **try** keyword.

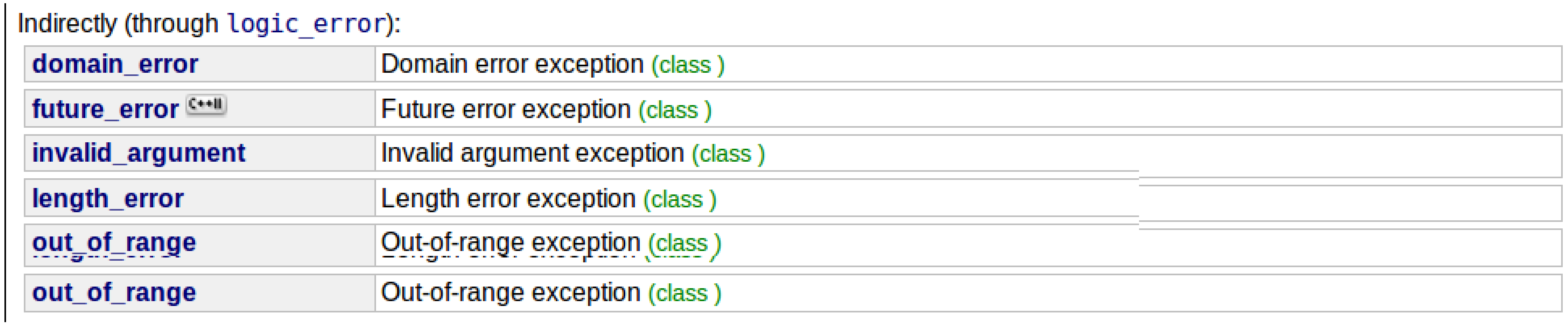
The C++ Standard Library provides an exception **class object**, which we can extend if we want.

http://www.cplusplus.com/reference/exception/exception/

C++ Standard Library Exception



bad_alloc	Exception thrown on failure allocating memory (class)
bad_cast	Exception thrown on failure to dynamic cast (class)
bad_exception	Exception thrown by unexpected handler (class)
bad_function_call C++11	Exception thrown on bad call (class)
bad_typeid	Exception thrown on typeid of null pointer (class)
bad_weak_ptr C++11	Bad weak pointer (class)
ios_base::failure	Base class for stream exceptions (public member class)
logic_error	Logic error exception (class)
runtime_error	Runtime error exception (class)

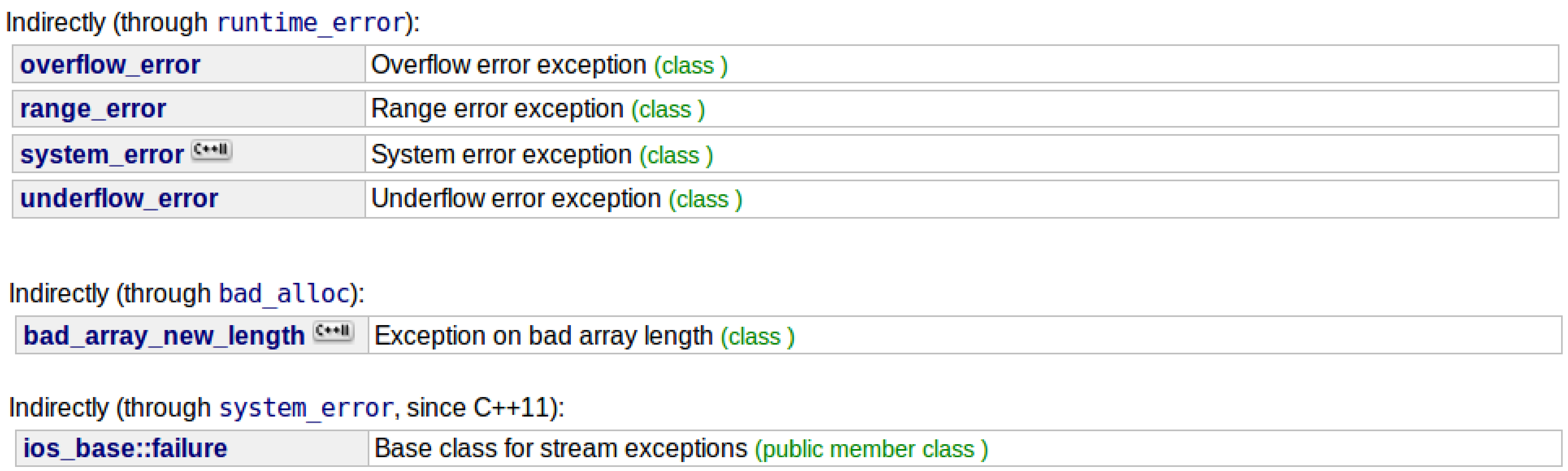


Indirectly (through `logic_error`):

domain_error	Domain error exception (class)
future_error C++11	Future error exception (class)
invalid_argument	Invalid argument exception (class)
length_error	Length error exception (class)
out_of_range	Out-of-range exception (class)
out_of_range	Out-of-range exception (class)

http://www.cplusplus.com/reference/exception/exception/

C++ Standard Library Exception



Indirectly (through `runtime_error`):

overflow_error	Overflow error exception (class)
range_error	Range error exception (class)
system_error C++11	System error exception (class)
underflow_error	Underflow error exception (class)

Indirectly (through `bad_alloc`):

bad_array_new_length C++11	Exception on bad array length (class)

Indirectly (through `system_error`, since C++11):

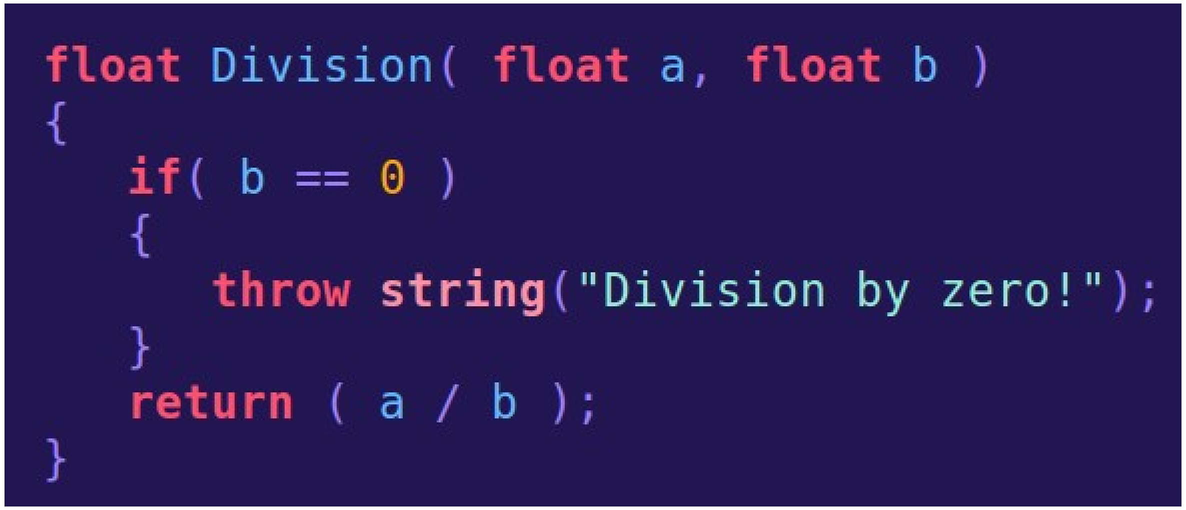
ios_base::failure	Base class for stream exceptions (public member class)

There are a lot of types of exceptions!

http://www.cplusplus.com/reference/exception/exception/

Writing a throw

Say that we're writing a function where there could potentially be a division by zero. We need to **throw** an exception if this happens.



```
float Division( float a, float b )
{
   if( b == 0 )
   {
      throw string("Division by zero!");
   }
   return ( a / b );
}
```

We could throw different data types – an integer for an error code, a string for a simple message, or an **exception** object from the Standard Library.

Writing a throw

Then, when our function is put into practice, it would need to have **try** surrounding it.

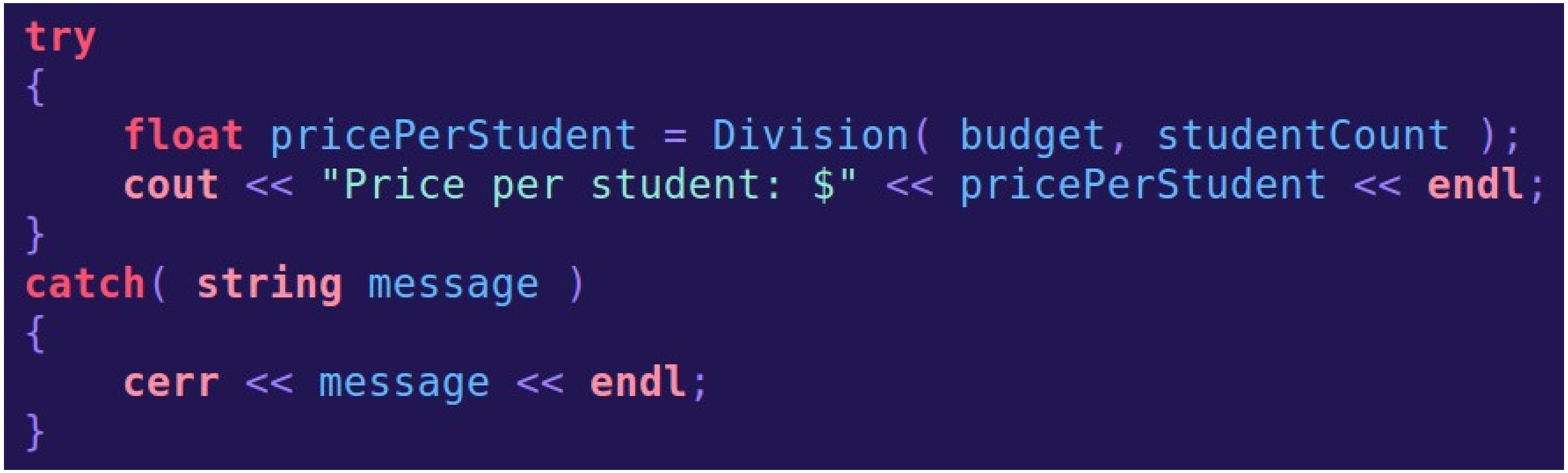
```
try
{
    float pricePerStudent = Division( budget, studentCount );
    cout << "Price per student: $" << pricePerStudent << endl;
}
catch( string message )
{
    cerr << message << endl;
}
```

The **catch** parameter needs to match the data-type we are **throwing**.

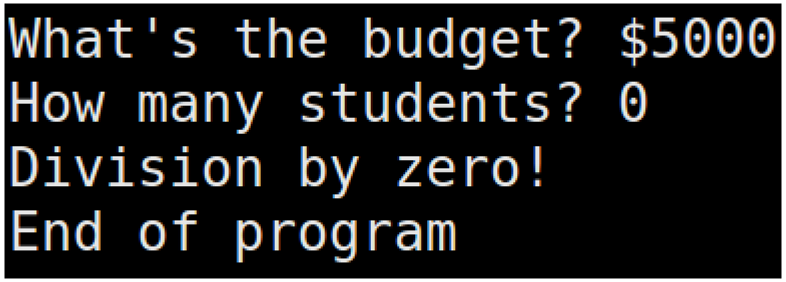
```
throw string("Division by zero!");
```

Writing a throw

Then, when our function is put into practice, it would need to have **try** surrounding it.



```
try
{
    float pricePerStudent = Division( budget, studentCount );
    cout << "Price per student: $" << pricePerStudent << endl;
}
catch( string message )
{
    cerr << message << endl;
}
```



```
What's the budget? $5000
How many students? 0
Division by zero!
End of program
```

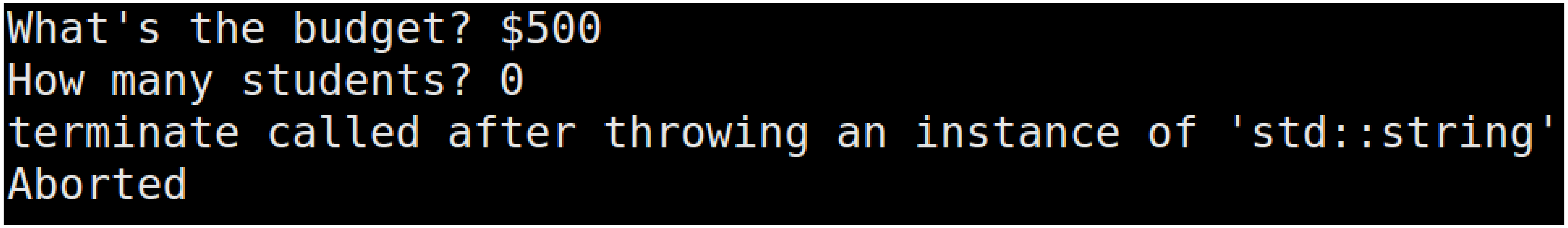
When our program runs, it will display the error message (if we want it to) then continue.

Try / Catch / Throw

try	Try a function or portion of code that we know has a **throw** command within it.
catch	Handle the error we receive. There can be multiple **catches** for different types of exceptions.
throw	From within code that may cause a problem, we **throw** if that problem is detected.

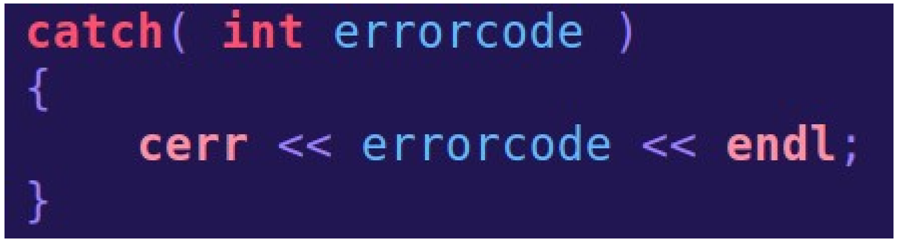
Try / Catch / Throw

If you do not **catch** an exception, the default behavior for C++ is to **terminate** the program.



```
What's the budget? $500
How many students? 0
terminate called after throwing an instance of 'std::string'
Aborted
```

Even if you have a **catch**, it might not be looking for the right type of exception. For the example above, it is looking for a **catch** to handle a string, but the program was modified to only handle an **int**...

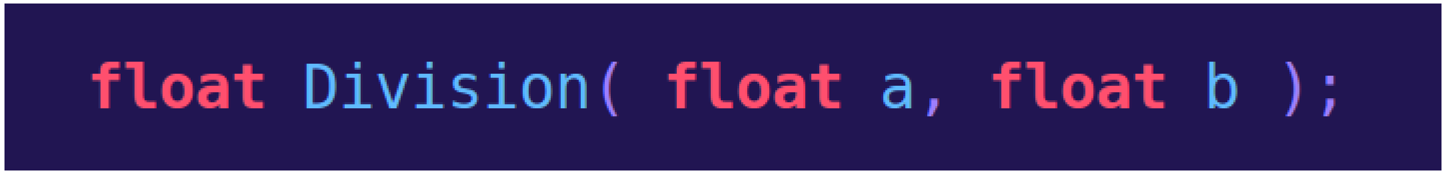


```
catch( int errorcode )
{
    cerr << errorcode << endl;
}
```

Specifying that your function might “throw up”

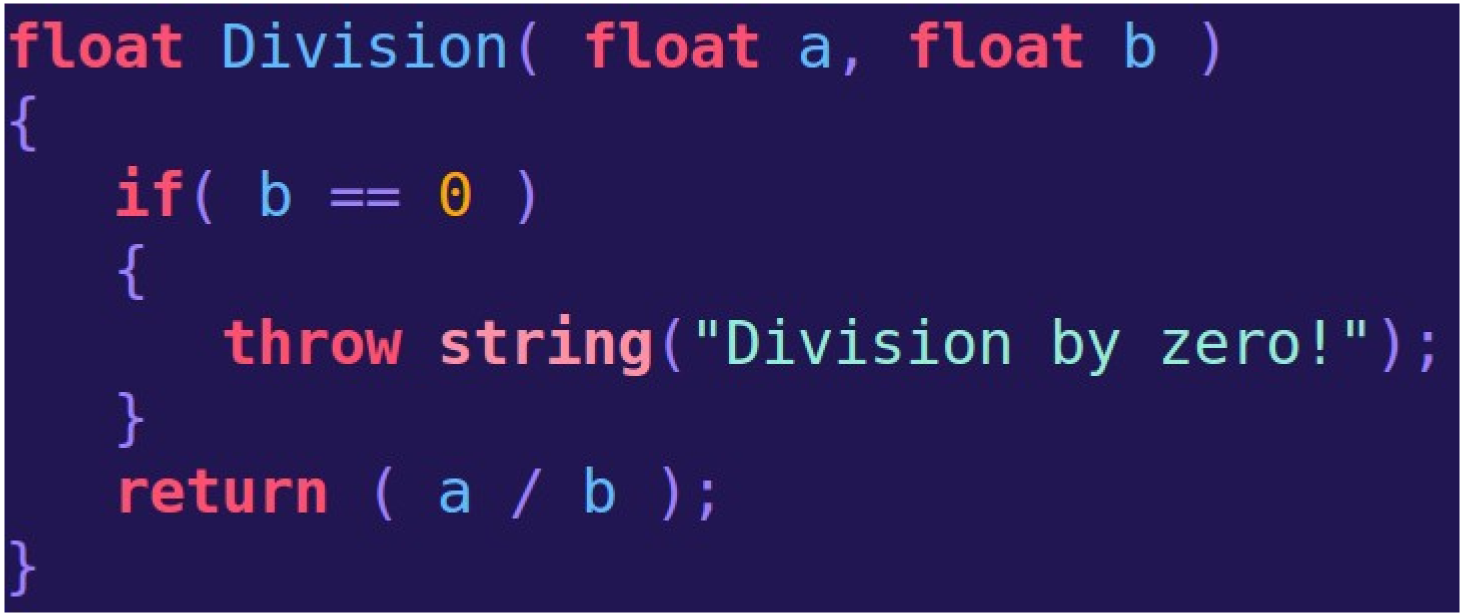
You can specify that your program might **throw**, and what kind of exceptions it will throw, during the **function declaration** / **function definition**

Nothing specified - This function might throw anything.



```
float Division( float a, float b );
```

Declaration



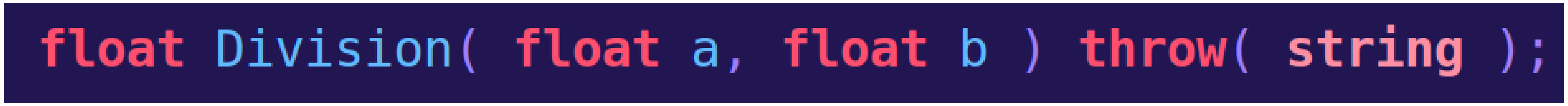
```
float Division( float a, float b )
{
   if( b == 0 )
   {
      throw string("Division by zero!");
   }
   return ( a / b );
}
```

Definition

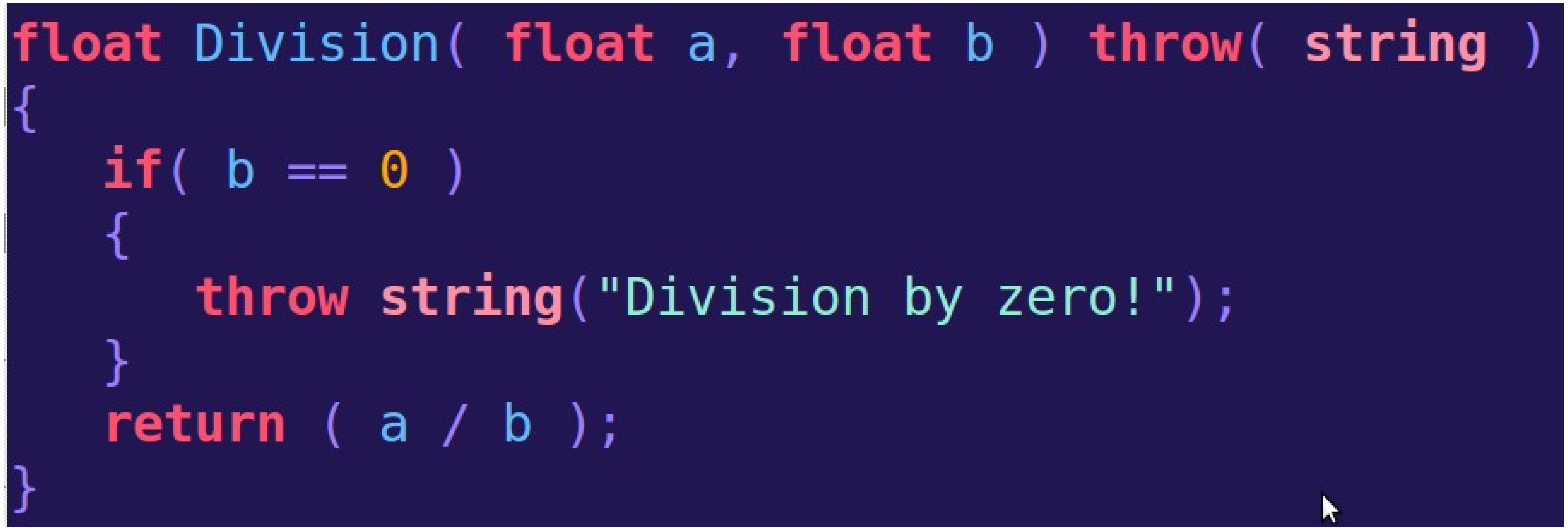
Specifying that your function might “throw up”

You can specify that your program might **throw**, and what kind of exceptions it will throw, during the **function declaration** / **function definition**

Only a string will be thrown



```
float Division( float a, float b ) throw( string );
```



```
float Division( float a, float b ) throw( string )
{
    if( b == 0 )
    {
        throw string("Division by zero!");
    }
    return ( a / b );
}
```

Bad Practice

Don't wrap your **entire program** in a **try / catch** block!

Try to wrap your try / catch around the smallest amount of code possible!