# “Traditional” Error Handling

## Returning Error Values

One classic approach involves returning a predefined error constant from a function on an error condition. For example, the C memory management function malloc() returns NULL for out of memory, and the C file I/O function fopen() returns NULL for unsuccessful open.

However, users may not check for an error condition, either because they forget or because they don’t even know that they need to do so. For example, the C function `printf()` returns the number of arguments successfully printed. However, you will probably never encounter code that does anything with the return value of printf.

Additionally, choosing a predefined error value is not always possible. For example, if operator[] on an array or vector is out of bounds, there’s no value that is guaranteed not to appear in the contents of the array.

## Global Error Condition Flag

Standard C provides errono() and perror() functions to support a global C standard library error code. POSIX, the de-facto standard for how UNIX systems, defined a global integer value *errno*. However, just as with returning an error value, users may choose to ignore the error code.

Error checking bulks up normal code, making it less efficient and harder to read.

# Exception Handling

Exceptions are a mechanism for handling errors “out of band,” in separate code that obeys scopes and code flow.

```
class Vector {  //detects out of range errors as exceptions
   sz: number;
   getValue(i: number) {
      if (0 <= i && i < sz) return p[i];  //within range, OK
      throw new Error(“Out of range”);    //exception – throw!
   }
   // ...
};
```

Thus the code above “throws” an exception, when it detects an out of range subscript `i`.

## Handling an Exception

When an exception is thrown, the error must be “caught” or else the program will crash. If an exception is uncaught in a particular function, the call stack (the layers of functions calling other functions) is “unwound” until something that handles the exception is found. In this manner, you can almost imagine an exception to be an alternate means to return from a function.

```
foo(v: Array)
{
   let i: number
   // ...
   try
   {
         bar(v); // try block may or may not cause an exception
   }
   catch (e)
   {     // if try causes an exception, then catch a Range
object
         // do something about the exception
   }

   // ...
}

bar(v: Array)
{
   v[v.length + 1]; // trigger range error!
   …
}
```

The above code throws Range up to `bar()` (no handler here), then `foo()`, which catches the error.

## Advantages

Instead of terminating the program, we can write more robust, fault-tolerant code. Instead of returning a value representing error (e.g., 0 or NULL) that may not always be feasible if there is no acceptable "error value,” we explicitly separate error handling code form "ordinary" code. Exception handling is also more readable and more amenable to automated code tools.

Instead of returning a legal value and leaving the program in an illegal state, which lets the program runs, but may cause mysterious crashes later on, we force developers to get the program to run acceptably.

## Exceptions as Objects

You can throw any valid value: a number, a boolean, an object, etc. However, there are significant advantages to throwing objects. For example, we can pass data members as part of the exception information, such as error codes, a failure message, etc

## Exceptions in Constructors

Exceptions especially a good idea for constructors because constructors may fail, but cannot produce an error value. Exception handling lets failure be transmitted out of the constructor, e.g.:

```
constructor(size: number)
{
   if (size < 0 || max < size) throw Error(“Invalid size”); //
max is member of Vector in this example
     // ...
}
```

Code creating Vectors can catch Size errors:

```
let p: Vector;
try { let p = new Vector(i); }
catch (error) { p = new Vector(Vector.max)); }
return p;
```