As mentioned in class, some large C programs make use of the goto statement to unify code for handling exceptional conditions within individual functions. In this note, we explore the implications in both space (code size) and time (performance) for using functions or exceptions in place of goto. We find that while hand-coded jump instructions (i.e., use of goto) do reduce code size, the performance of optimized code is actually slightly better when functions are used. We also note that the g++ implementation of the C++ exception handling mechanism is quite slow when used, and thus must be used with caution.

The basis for our comparison is a synthetic example of local exception handling within a function. A version of this function using goto appears in Figure 1. The handle_func function takes two arguments; the first specifies the type of operation to be performed, and the second is an operand. The function provides four types of computation. For each type, the function first checks the validity of the operand. Error handling for invalid operands is then unified using goto. For any type of error, the type of computation is recorded, and the number of errors seen is incremented and returned. The return value from handle_func is not meaningful, since success cases can also return non-zero values. The number of errors seen is a global variable (miscount) to enable a simple sanity check across the approaches that we compare, while the type of the last error seen is a file scope variable (errortype).

We test five variants of the code and report both code size and performance results with and without compiler optimizations. The first variant, GOTO, is the goto version shown in Figure 1. The second variant, FUNCTION, replaces the block of code labeled mistake in the figure with a static function mark_bad that implements the same functionality using the fact that both variables are accessible directly and passing only the arg parameter from handle_func. All goto statements are changed to calls to mark_bad, and the return value of mark_bad is returned directly from handle_func. The third variant, INDIRECT, avoids making use of the direct accessibility of the variables by adding two pointer arguments to the mark_bad function. Note that these cases are highly favorable to compiler optimization because the variables are in the file, and the mark_bad function can readily be inlined. However, in the context of exception handling within a specific function, any additional exception handling functions should be near the original function, and any benefits from variables in the file are equivalent for all three of these variants.

The last two variants are written in C++ and make use of exceptions to transfer control within the handle_func function. Code for the fourth variant, EXCEPTION, appears in Figure 2. The BadCall exception class carries a single integer representing the type of operation that has failed, and the catch block takes the place of the block labeled mistake in Figure 1. Only one of the case statement switches is shown; the others are changed in an analogous manner to throw BadCall exceptions. The fifth variant, EXCINDIRECT, extends the BadCall exception class data with two integer pointers, and the corresponding catch clause makes use of these pointers rather than using the variables errortype and miscount directly.

We tested all five variants by compiling the driver code using gcc/g++ 4.3.2 with maximum optimization (~09) and the variant code at several levels of optimization, then executing all codes on a somewhat old x86 machine. Table 1 reports performance and code size results. Notably, for all variants, the code generated for low (~02), medium (~06), and high (~09) levels of optimization was identical. With optimization turned on, gcc is able to eliminate the indirection from the INDIRECT variant, producing code that is identical to that generated for the FUNCTION variant.

We estimate the relative performance of the different approaches by measuring the execution time of a large number of calls that cycle repeatedly through 100 combinations of the two parameters and dividing measured wall clock time for the program by the number of calls. For the three C variants, we make 500 million calls, while for the two C++ variants we make only 5 million calls to allow the runs to finish in under a minute. The arg parameter ranges from 0 to 4, and the value parameter ranges from 0 to 19. For each data point in the table, the program was run three times. The minimum wall clock time across runs was then used to calculate the result, with the reported precision based on the measured differences across the three runs.
<table>
<thead>
<tr>
<th></th>
<th>unoptimized (-O0)</th>
<th>optimized (-O2, -O6, or -O9)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Execution Time</td>
<td>Total Bytes</td>
</tr>
<tr>
<td>GOTO</td>
<td>25.6 nsec</td>
<td>3476</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>28.30 nsec</td>
<td>3540</td>
</tr>
<tr>
<td>INDIRECT</td>
<td>31.4 nsec</td>
<td>3604</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>4.32 µsec</td>
<td>5252</td>
</tr>
<tr>
<td>EXCINDIRECT</td>
<td>4.30 µsec</td>
<td>5380</td>
</tr>
</tbody>
</table>

Table 1: Execution time, total code size, and function and support size for five intra-function exception-handling variants. Total code size includes the benchmark driver and all standard C/C++ wrapper code.

```c
static int errortype = 0;
int miscount = 0;

int handle_func (int arg, int value)
{
    static int table[8] = {0, 1, 1, 0, 1, 0, 1, 1};
    int answer;

    switch (arg) {
        case 0:
            if (3 == (value & 3)) {goto mistake;}
            answer = value - 8;
            break;
        case 1:
            if (10 > value) {goto mistake;}
            answer = value / 2;
            break;
        case 2:
            if (table[value % 8]) {goto mistake;}
            answer = table[(value + 1) % 8];
            break;
        default:
            if (0 != value) {goto mistake;}
            answer = 42;
            break;
    }
    return answer;

    mistake:
        errortype = arg;
        return ++miscount;
}
```

Figure 1: The synthetic benchmark routine `handle_func` as written using `goto`.

We report the minimum based on the notion that random interference from other system activities can only increase execution time. No attempt was made to extract the loop overhead cost of the driver code; although this time makes up a non-negligible portion of the reported execution time for the C variants, the amount should be constant across all variants.
static int errortype = 0;
int miscount = 0;

class BadCall : public std::exception {
private:
    int value;
public:
    BadCall (int v) {value = v;}
    int getValue () {return value;}
};

extern "C" { // support C linkage from driver code
    int handle_func (int arg, int value);
}

int handle_func (int arg, int value)
{
    ... some code omitted ...
    try {
        switch (arg) {
            case 0:
                if (3 == (value & 3)) {throw BadCall (arg);}
                answer = value - 8;
                break;
            ... some code omitted ...
        }
    } catch (BadCall& bad) {
        errortype = bad.getValue ();
        return ++miscount;
    }
    return answer;
}

Figure 2: The synthetic benchmark routine handle_func as written using C++ exceptions.

Code size is reported in two ways. First, the binaries are stripped of all symbols, and the total number of bytes necessary for each program is reported. This value includes both the driver code and all C/C++ wrapper functions necessary to a full program. Second, the number of bytes required for the handle_func function itself is reported along with the number of bytes in all supporting functions. For this purpose, we used the tool objdump to view disassembled output along with byte counts. For the C variants, supporting functions include only the mark_bad routine, which disappears in the optimized version. For C++, the unoptimized code includes constructors, destructors, and class support functions. With optimization turned on, most of these functions are inlined, and standalone versions are not generated, but destructors are still needed to support catching exceptions.

The results for unoptimized code match intuitive expectations: telling the compiler to use jump instructions (that is, goto) is slightly better than making use of a function call, and requires less code. Similarly, adding extra indirection produces even more code and leads to further slowdown. Making use of C++ exceptions is more than a hundred times more costly for performance and requires three or four times as much code as well.

The results for optimized code are less intuitive. The C variants are shorter and faster, but the function-based approach now beats the goto-based approach, even though the code size of the former remains larger. The explicit use of goto may place too many constraints on the compiler’s register allocation and code generation optimization strategies. The C++ variants generate less code when optimized, but also get slightly slower.