Robust Programming

Matt Bishop[1]
Department of Computer Science
University of California at Davis
Davis, CA 95616-8562

Introduction

Robust programming, also called *bomb-proof programming*, is a style of programming that prevents abnormal termination or unexpected actions. Basically, it requires code to handle bad (invalid or absurd) inputs in a reasonable way. If an internal error occurs, the program or library terminates gracefully, and provides enough information so the programmer can debug the program or routine.

This handout discusses the principles of bomb-proof coding, and gives you a detailed example of how to do it right. Our example is library for managing queues (FIFO lists) of numbers. This allows the example to consider parameters and global variables. The principles apply to programs, also; specifically, input and parameters are equivalent, and the environment is like global variables.

Principles of Robust Programming

A robust program differs from a non-robust, or *fragile*, program by its adherence to the following four principles:

Paranoia. Don't trust anything you don't generate! Whenever someone uses your program or library routine, assume they will try to break it. When you call another function, check that it succeeds. Most importantly, assume that your own code will have problems, and program defensively, so those problems can be detected as quickly as possible.

Stupidity. Assume that the caller or user is an idiot, and cannot read any manual pages or documentation. Program so that the code you write will handle incorrect, bogus, and malformed inputs and parameters in a reasonable fashion, "reasonable" being defined by the environment. For example, if you print an error message, the message should be self-explanatory and not require the user to look up error codes in a manual. If you return an error code to the caller (for example, from a library routine) make the error codes unambiguous and detailed. Also, as soon as you detect the problem, take corrective action (or stop). This keeps the error from propagating.

Part of the problem is that in a week, you most likely will have forgotten the details of your program, and may call it incorrectly or give it bogus input. This programming style is also a form of defensive programming.

Dangerous Implements. A "dangerous implement" is anything that your routines expect to remain consistent across calls. For example, in the standard I/O library, the contents of the FILE structure for allocated files is expected to remain fixed across calls to the standard I/O library. That makes it a "dangerous implement." Don't let users gain access to these! They might accidentally (or deliberately) modify the data in that data structure, causing your library functions to fail—badly. Never return pointers or indices into arrays; always hide the true addresses (or indices) by using something called a "token."

Hiding data structures also makes your program or library more modular. For example, the queue manager uses arrays to represent queues. This gives them a maximum size. You might decide that linked lists would be more suitable and want to rewrite the routines. If you have properly designed the interface and hidden as much information and data as possible, the calling program need not be changed; however, if you neglected this style of information hiding and information abstraction, programs that correctly function with the current representation might break if you changed that representation (because the caller assumed that the queue elements are in sequential integer locations, for example).

Can't happen. As the old saw goes, "never say 'never.'" Impossible cases are rarely that; most often, they are merely highly unlikely. Think about how those cases should be handled, and implement that type of handling. In the worst case, check for what you think is impossible, and print an error message if it occurs. After all, if you modify the code

1.

repeatedly, it is very likely that one modification will affect other parts of the code and cause inconsistent effects, which may lead to "impossible" cases occurring.

Exercise 1. Relate the informal descriptions of the principles of robust programming to the more formal principles of good programming style, such as cohesion and coupling, information abstraction, information hiding, and good documentation. The point of this exercise is to show you that robust programs arise from writing code in a good style; what you learn in school ***is*** useful!

Robust programming is defensive, and the defensive nature protects the program not only from those who use your routine but also from yourself. Programming is not mathematics; errors occur. Good programming assumes this, and takes steps to detect and report those errors, internal as well as external. Beware of everything – even your own code!

Example of a Fragile Library

This library implements queues in a very straightforward way. It's typical of what many C programmers would write. It's also very fragile code. We'll analyze it in detail to support that statement.

The library can handle any number of queues, and when a queue is created, its pointer is returned to the caller. Three entry points are provided: *qmanage*, for creating or deleting a queue; *put_on_queue*, for adding an element to a queue; and *take_off_queue*, for deleting an element from a queue. All files calling this routine must include the header file *fqlib.h*, which defines the queue structure.

Reviewing the structure and library functions will illuminate the problems with fragile code, and show why the usual C coding style is so fragile.

The Queue Structure

The file *fqlib.h* defines the queue structure and the maximum number of elements in the queue. Because programs calling the queue functions must pass a pointer to the queue, the structure must be visible to the calling procedures (which need to know the structure to define the queue pointer type QUEUE *). Hence this file must be included in programs that call the queue library functions.

The header file contains:

```
/*
 * the queue structure
 */
typedef struct queue {
      int *que;         /* the actual array of queue elements */
      int head;         /* head index in que of the queue */
      int count;        /* number of elements in queue */
      int size;         /* max number of elements in queue */
} QUEUE;

/*
 * the library functions«
 */
void qmanage(QUEUE **, int, int);         /* create or delete a queue */
void put_on_queue(QUEUE *, int);          /* add to queue */
void take_off_queue(QUEUE *, int *);      /* pull off queue */
```

As indicated, the queue management functions are:

qmanage() . . . . . create and delete queues
put_on_queue() . . . . . .add an element to the end of the queue
take_off_queue() . . . . . take an element from the head of the queue

This organization provides the first evidence of fragility. The caller will use a pointer to a QUEUE structure; and as the layout of that structure, and its location, is known, the caller can bypass the queue library to obtain data from the queues directly – or alter information in the queues, or information that the library uses to manage the queues. For

example, if one wanted to change the number of elements in the queue without calling a queue management function, one can say:

```
qptr->count = newvalue;
```

where *qptr* is a pointer to the relevant queue and *newvalue* the expression for the new value to be assigned to the queue's counter.

So, the problem with including this header file in the callers' files is:

☞ The callers have access to the internal elements of the queue structure.

The Queue Management Function

The first function controls the creation and deletion of stacks:

```
/*
 * create or delete a queue
 *
 * PARAMETERS:    QUEUE **qptr      space for, or pointer to, queue
 *                int flag          1 for create, 0 for delete
 *                int size          max elements in queue
 */
void qmanage(QUEUE **qptr, int flag, int size)
{
      if (flag){
            /* allocate a new queue */
            *qptr = malloc(sizeof(QUEUE));
            (*qptr)->head = (*qptr)->count = 0;
            (*qptr)->que = malloc(size * sizeof(int));
            (*qptr)->size = size;
      }
      else{
            /* delete the current queue */
            (void) free((*qptr)->que);
            (void) free(*qptr);
      }
}
```

Glancing over it, we see it uses logical cohesion because the parameter *flag* indicates which of two distinct, logically separate operations are to be performed. The operations could be written as separate functions. That this routine has such poor cohesion does not speak well for its robustness.

Consider the parameter list, and the calling sequence. The *flag* parameter is an integer that indicates whether a queue is to be created (if *flag* is non-zero) or deleted (if *flag* is zero). The *size* parameter gives the maximum number of integers to be allowed in the queue. Suppose a caller interchanged the two:

```
qmanage(&qptr, 85, 1);
```

The *qmanage* routine would not detect this as an error, and will allocate a queue with room for 1 element rather than 85. This ease of confusion in the parameters is the first problem, and one that cannot be checked for easily.

☞ The order of elements in the parameter list is not checked.

Next, consider the *flag* argument. When does it mean to create a queue and when does it mean to delete a queue? For this function, the intention is that 1 means create and 0 means delete, but the coding style has changed this to allow any non-zero value to mean create. But there is little connection between 1 and create, and 0 and delete. So psychologically, the programmer may not remember which number to use and this can cause a queue to be destroyed when it should have been created. Using *enum*s would help here if the library is compiled with the program, but if the library is simply loaded, *enum*s do not help as the elements are translated into integers (so no type checking can be done).

☞ The value of the flag parameter is arbitrary.

The first parameter is a pointer to a pointer – necessary when a value is returned through the parameter list, as all C function arguments are passed by value. Passing a pointer provides the call by reference mechanism. However, a call

by reference usually uses a singly indirect pointer; if a doubly indirect pointer is used, programmers will make mistakes (specifically, pass a singly indirect pointer). In general, it is better to avoid call by reference; when it is necessary, do not use multiple levels of indirection unless absolutely necessary.

☞ Using pointers to pointers causes errors in function calls.

The third set of problems arises from a failure to check parameters. First look at queue creation. Suppose *qptr* is **NULL** or an invalid pointer. Then the line containing *malloc* will cause a segmentation fault. Also, suppose *size* is non-positive. What happens when the queue is allocated (the second *malloc*)? If *size* is 0, most *malloc*s will return a **NULL** pointer, but this is not universal. If *size* is negative, the result is implementation dependent and may cause a segmentation violation. In either case, the result is unpredictable.

Now look at queue deletion. The parameter *size* is irrelevant, but suppose *qptr* or **qptr* is **NULL**. Then the result of *free* is undefined and may result in a segmentation fault. Worse, imagine the parameter is not **NULL** but instead a meaningless address. This is almost impossible to catch before the call, and causes segmentation violations (if lucky) or very odd behavior afterwards (if not).

☞ The parameter values are not sanity checked.

The calling sequence is not checked either. Suppose one deletes a queue before creating it:

```
qmanage(&qptr, 0, 1);
     /* ... */
qmanage(&qptr, 1, 100);
```

This would either cause a segmentation violation when called, or the releasing of unallocated memory; in the latter case, the program will probably crash later on, with little indication of why. Again, the problem is that *qmanage* does not check that *qptr* refers to a valid queue. However, here's a more subtle variation of this problem:

```
qmanage(&qptr, 1, 100);
/* ... */
qmanage(&qptr, 0, 1);
/* ... */
qmanage(&qptr, 0, 1);
```

Now a queue is deleted twice. Attempting to *free* space that has already been deallocated causes an undefined action, usually a segmentation violation.

☞ The user can delete an unallocated queue, or a previously deleted queue.

Consider the body of the routine. What happens if either *malloc* fails, and returns **NULL**? The subsequent references to *qptr* to fault, as they are references through a **NULL** pointer. Hence:

☞ Check all return values.

One subtle problem arises from overflow. Consider the expression

```
size * sizeof(int)
```

in the first call to *malloc*. Suppose *size* is 2^{31}, and an integer is 4 bytes (a common value). Then this expression evaluates to 2^{33}. On a 32 bit machine, this overflows, and (most likely) produces a value of 0. Such a flaw will most likely not cause any problems during the call, but will cause the program to produce a segmentation fault in a seemingly unrelated place later on. Overflow (and underflow, in floating point calculations) are very pernicious and nasty problems; watch out for them.

☞ Look out for integer (or floating point) overflow (and underflow, when appropriate).

Exercise 2. The obvious way to test for overflow is to multiply the absolute value of *size* and *sizeof(int)* and see if the result is smaller than the absolute value of *size* (because if `|a * b| < |a|` when `|a| > 1` and `|b| > 1`, then overflow has occurred). Does this always work? What problems does it introduce? (*Hint*: think about architectures allowing arithmetic overflow to cause a trap.) Suggest an alternate method without these problems.

Adding to a Queue

This function adds an element to the queue. It adds the index of the head element to the current count (modulo the queue size), and places the new element at that location; it then increments the count:

```
/*
 * add an element to an existing queue
 *
 * PARAMETERS:    QUEUE *qptr pointer for queue involved
 *                int n       element to be appended
 */
void put_on_queue(QUEUE *qptr, int n)
{
      /* add new element to tail of queue */
      qptr->que[(qptr->head + qptr->count) % qptr->size] = n;
      qptr->count++;
}
```

Two basic problems arise. First, the *qptr* parameter is not checked to ensure it refers to a valid queue; it may refer to a queue that has been deleted, or to random memory, or may be **NULL**. Any of these will cause problems; if the caller is lucky, the problem will arise in this routine; if the caller is unlucky, the symptom will not appear until later on in the program, with no indication of what caused it.

☞ Check all parameters.

As an offshoot of this, suppose *qptr* is valid but *que* is not. Then the routine will not work correctly:

☞ Check for incorrect values in structures and variables.

Second, suppose the queue is full (that is, *qptr->count* equals *qptr->size*). If this function is called, it will overwrite the head of the queue. There is no check for an overflowing queue, doubtless because the author assumed it would never happen.

☞ Check for array overflow when inserting items.

A more sophisticated problem is the placing of trust in the values of the queue structure. The integrity of the queue structure depends on the consistency of the *count*, *size*, and *head* fields. If *size* increases between calls, the routine will think that the queue has been allocated more space than it actually has, leading to a segmentation fault. If *size* decreases between calls, some elements might be lost.

Exercise 3. Write a program that demonstrates when decreasing *size* between calls to *add_to_queue* causes elements previously added to the queue to become inaccessible. Describe the problems that can arise if the values of *head* and/or *count* are changed across calls to *put_on_queue*.

Given the accessibility of the queue structure elements to the callers, these elements may change (accidentally or deliberately).

Removing Elements from the Queue

Taking elements off the queue begins by getting the element at index *head*. Then count is decremented, and *head* is incremented (modulo *size*):

```
/*
 * take an element off the front of an existing queue
 *
 * PARAMETERS:    QUEUE *qptr pointer for queue involved
 *                int *n      storage for the return element
 */
void take_off_queue(QUEUE *qptr, int *n)
{
      /* return the element at the head of the queue */
      *n = qptr->que[qptr->head++];
      qptr->count--;
      qptr->head %= qptr->size;
}
```

The parameter problems described in the previous section exist here too; *qptr* may be invalid, **NULL**, or point to an invalid queue. Moreover, *n* may also be an invalid integer pointer. So:

☞ Check all parameters.

☞ Check for incorrect values in structures and variables.

Here. the danger is underflow, not overflow. Suppose there are no elements in the queue. The value returned through *n* will be garbage, and *count* will be set to a bogus value. Hence:

☞ Check for array underflow when extracting items.

The problem of variable consistency across calls occurs in this routine, also.

Exercise 4. What problems might an invalid pointer for *n* cause? Specifically, suppose in the call

```
take_off_queue(qptr, c)
```

the variable `c` is declared as a *char* * or a *float* *? How would you solve these problems in a portable manner?

Summary

The library *fqlib.c*, the contents of which are presented in this section, is very fragile code. Among its flaws are:

- The callers have access to the internal elements of the queue structure.
- The order of elements in parameter lists is not checked.
- The value of command parameters (which tell the function what operation to perform) is arbitrary.
- Using pointers to pointers causes errors in function calls.
- The parameter values are not sanity checked.
- The user can delete an unallocated queue, or a previously deleted queue.
- Return values from library functions are not checked.
- Integer (or floating point) overflow (and underflow, when appropriate) is ignored.
- The values in structures and variables are not sanity checked.
- Neither array underflow nor overflow is checked for.

All these flaws make the library susceptible to failure.

Example of a Robust Library

In this section, we study an alternate implementation of the same library. This version, however, is very robust; it performs sanity checking, and attempts to anticipate problems and handle them gracefully. If the library cannot recover, it returns an error code to the caller indicating the problem. This code is more complex to write, but—as the callers can rely on it—makes debugging the calling applications much simpler.

The Queue Structure

The queue structure is to be unavailable to the caller, so we need to define two items: the structure itself, and an interface to it. We deal with the interface first. The object that the caller uses to represent a queue will be called a *token*.

If we make the token a pointer to a structure, the user will be able to manipulate the data in the structure directly. Hence we need some other mechanism, and indexing into an array is the obvious solution. However, if we use simple indices, then the user can refer to queue 0, and have a high degree of certainty of getting a valid queue. So instead we use a function of the index such that 0 is not in the range of the function. Thus, we will represent the queues as entries in an array of queues, and the token will be an invertible mathematical function of their index.

In addition, we must prevent a "dangling reference" problem (in which a program references queue A after that queue is deleted). Suppose a programmer uses the library to create queue A. Queue A is subsequently deleted and queue B created; queue B happens to occupy the same slot in the array of queues as queue A did. A subsequent reference to queue A by token will get queue B. To avoid this problem, we associate with each queue a unique integer (called a *nonce*) and merge this into the token. In the above example, queue might have nonce 124 and queue B might have nonce 3086. The token for queue A is $f(7, 124)$ and the token for queue B is $f(7, 3085)$. As these values differ, the reference to queue A will be detected and rejected.

We choose as our function the following:

$$f(index, nonce) = ((index + 0x1221)<<16) \mid (nonce + 0x0502)$$

where << and | are the usual C operators. We emphasize, however, that ***any*** function invertible in either argument is acceptable. In the above,

```
index = (f(index, nonce) >> 16) & 0xffff
```

and

```
nonce = f(index, nonce) & 0xffff
```

where & and >> are the usual C operators.

This simplifies the interface considerably, but at the cost of assuming a 32-bit quantity (or greater). Fortunately, most systems have a datatype supporting integers of this length. So, in the header file, we put:

```
/* queue identifier; contains internal index and nonce mashed together */
typedef long int QTICKET;
```

With this token defined, calling routines need know nothing about the internal structures.

☞ Don't hand out pointers to internal data structures; use tokens instead.

The second issue is errors; how to handle them? We can print error messages (and take action, possibly even terminating), we can return error results and allow the caller to handle the error, or we can set up special error handlers (if the language supports these; they are typically written as trap or exception handlers). Unfortunately, C does not provide exception handlers for errors. The method of returning error codes to the caller allows much greater flexibility than handling the error in the routine, and is equally simple to perform. The complication is that a set of error codes must indicate the errors that could occur. So, as we proceed through our library, we shall track errors and define error codes.

☞ Handle errors in a consistent manner: either print error messages from a centralized printing routine, or return error codes to the caller and let the caller report the error.

We make some decisions about the library functions for this purpose. The return value will indicate whether an error has occurred; if so, the function returns an error code and an expository message in a buffer. If not, it returns a flag indicating no error. So, we define all error codes to be negative:

```
/*
 * error return values
 * all the queue manipulation functions return these;
 * you can interpret them yourself, or print the error
 * message in qe_errbuf, which describes these codes
 */
#define QE_ISERROR(x)   ((x) < 0)   /* true if x is a qlib error code */
#define QE_NONE         0           /* no errors */
/*
 * the error buffer; contains a message describing the last queue
 * error (but is NUL if no error encountered); not cleared on success
 */
extern char qe_errbuf[256];
```

Like the UNIX system variable *errno*(3), *qe_errbuf* is set on an error but not cleared on success. The buffer will contain additional information (such as in which routine the error occurred and relevant numbers). The following macros aid this process:

```
                /* macros to fill qe_errbuf */
#define ERRBUF(str)     (void) strncpy(qe_errbuf, str, sizeof(qe_errbuf))
#define ERRBUF2(str,n)  (void) sprintf(qe_errbuf, str, n)
#define ERRBUF3(str,n,m)(void) sprintf(qe_errbuf, str, n, m)
```

These are defined in *qlib.c* because they format messages placed in *qe_errbuf*; the functions that call the library have no use for them.

We also redefine the function interface to eliminate the low cohesion of the *qmanage* routine:

```
QTICKET create_queue(void);         /* create a queue */
int delete_queue(QTICKET);          /* delete a queue */
```

```
int put_on_queue(QTICKET, int);     /* put number on end of queue */
int take_off_queue(QTICKET);        /* pull number off front of queue */
```

This eliminates the use of a flag variable to manage creation or deletion.

In the *qlib.c* file we place the definition of the queue structure and the related variables:

```
/* macros for the queue structure (limits) */
#define MAXQ      1024              /* max number of queues */
#define MAXELT    1024              /* max number of elements per queue */

/* the queue structure */
typedef int QELT;                   /* type of element being queued */
typedef struct queue {
      QTICKET ticket;               /* contains unique queue ID */
      QELT que[MAXELT];             /* the actual queue */
      int head;                     /* head index in que of the queue */
      int count;                    /* number of elements in queue */
} QUEUE;
/* variables shared by library routines */
static QUEUE *queues[MAXQ];         /* the array of queues */
                                        /* nonce generator -- this */
static unsigned int noncectr = NOFFSET;   /* MUST be non-zero always */
```

We made one change to the queue definition: all queues are to be of fixed size. This was for simplicity (see the exercise below). Also, all globals are declared *static* so they are not accessible to any functions outside the library file.

We distinguish between an *empty* queue and a *nonexistent* queue. The former has its *count* field set to 0 (so the queue exists but contains no elements); the latter has the relevant element in *queues* set to **NULL** (so the queue does not exist).

Exercise 5. The macros *ERRBUF2* and *ERRBUF3* use *sprintf* to format the error message. What problem does this function not guard against? Why can we ignore this problem in our library?

Exercise 6. What problems does static allocation of space for each queue's contents and for all queues introduce? What advantages?

Token Creation and Analysis

One function internal to the library creates a token from an index, and another takes a token, validates it, and returns the appropriate index.

These are separate routines because we need to be able to change the token's representation if the library is ported to a system without a 32-bit quantity to store the token in. Or, we may prefer to modify the mathematical function involved. In either case, this increases cohesion and decreases coupling, laudable goals from the software engineering (and maintenance!) perspectives.

In what follows, *IOFFSET* is the offset added to the index of the element and *NOFFSET* is the initial nonce. Both are defined in *qlib.c*:

```
#define IOFFSET   0x1221      /* used to hide index number in ticket */
#define NOFFSET   0x0502      /* initial nonce */
```

Here is the function to generate a token:

```
/*
 * generate a token; this is an integer: index number + OFFSET,,nonce
 * WARNING: each quantity must fit into 16 bits
 *
 * PARAMETERS:    int index   index number
 * RETURNED:      QTICKET     ticket of corresponding queue
 * ERRORS:        QE_INTINCON * index + OFFSET is too big
 *                            * nonce is too big
 *                            * index is out of range
```

```
 *                            (qe_errbuf has disambiguating string)
 * EXCEPTIONS:    none
 */
static QTICKET qtktref(unsigned int index)
{
      unsigned int high;      /* high 16 bits of ticket (index) */
      unsigned int low;       /* low 16 bits of ticket (nonce) */

      /* sanity check argument; called internally ... */
      if (index > MAXQ){
            ERRBUF3("qtktref: index %u exceeds %d", index, MAXQ);
            return(QE_INTINCON);
      }

      /*
       * get the high part of the ticket (index into queues array,
       *   offset by some arbitrary amount)
       * SANITY CHECK: be sure index + OFFSET fits into 16 bits as positive
       */
      high = (index + IOFFSET)&0x7fff;
      if (high != index + IOFFSET){
            ERRBUF3("qtktref: index %u larger than %u", index,
                                                    0x7fff - IOFFSET);
            return(QE_INTINCON);
      }

      /*
       * get the low part of the ticket (nonce)
       * SANITY CHECK: be sure nonce fits into 16 bits
       */
      low = noncectr & 0xffff;
      if ((low != noncectr++) || low == 0){
            ERRBUF3("qtktref: generation number %u exceeds %u\n",
                                       noncectr - 1, 0xffff - NOFFSET);
            return(QE_INTINCON);
      }

      /* construct and return the ticket */
      return((QTICKET) ((high << 16) | low));
}
```

The function is declared *static* so that only functions in the library may access it.

Rather than return a value through the parameter list, we compute the token and return it as the function value. This allows us to return error codes as well (since tokens are always positive, and error codes always negative). The single parameter is an index for which the token is to be computed.

☞ Make interfaces simple, even if they are for routines internal to the library.

The next *if* statement checks the value of the parameter; in this case, it must be a valid array index (between 0 and *MAXQ* inclusive). As the parameter is unsigned, only the upper bound need be checked. This may seem excessive; after all, this function is only called within our library, so can't we ensure the parameter is always in the expected range? The principle of "can't happen" applies here. We can indeed assure the index always lies within the range, but suppose someone else one day modifies our code and makes an error. That error could cause the library to fail. So it's best to program defensively.

☞ Always check parameters to make sure they are reasonable.

If an error occurs, it should be identified precisely. Two techniques are combined here. The first is an error message, giving the name of the routine and an exact description of the problem. It is in *qe_errbuf*, and available to the caller. The second is a return value indicating an error (specifically, an internal inconsistency):

```
#define QE_INTINCON     -8    /* internal inconsistency */
```

The calling routine must detect this error and act accordingly.

☞ Give useful and informative error messages. Include the name of the routine in which the error occurs. Allow numbers in the error message. Use error codes that precisely describe the error.

The error code here indicates an internal inconsistency (because an error indicates another library routine is calling *qtktref* incorrectly). An error message or code simply indicating an error occurred would be less helpful, because we would not know why the error occurred, or (depending on the error message) where.

The next statements add the offset to the index. As this is to be the upper half of a 32-bit number, it must fit into 16 bits as a signed number. The code checks that this requirement is met. Again, if it is not met, a detailed error message is given.

Exercise 7. Explain how the check works, in detail.

Exercise 8. The code uses `0x7fff` to mask *index* for the comparison instead of using `0xffff`. Why is the mask 15 bits instead of 16?

Exercise 9. The check for *noncectr* is similar, but uses `0xffff` as a mask. Explain why it does not need to use `0x7fff`.

The routine to break down a token into its component parts, and check the queue, is similar:

```
/*
 * check a ticket number and turn it into an index
 *
 * PARAMETERS:    QTICKET qno       queue ticket from the user
 * RETURNED:      int               index from the ticket
 * ERRORS:        QE_BADTICKET      queue ticket is invalid because:
 *                                  * index out of range [0 .. MAXQ)
 *                                  * index is for unused slot
 *                                  * nonce is of old queue
 *                                  (qe_errbuf has disambiguating string)
 *                QE_INTINCON       queue is internally inconsistent because:
 *                                  * one of head, count is uninitialized
 *                                  * nonce is 0
 *                                  (qe_errbuf has disambiguating string)
 * EXCEPTIONS:    none
 */
static int readref(QTICKET qno)
{
      register unsigned index;      /* index of current queue */
      register QUEUE *q;            /* pointer to queue structure */

      /* get the index number and check it for validity */
      index = ((qno >> 16) & 0xffff) - IOFFSET;
      if (index >= MAXQ){
            ERRBUF3("readref: index %u exceeds %d", index, MAXQ);
            return(QE_BADTICKET);
      }
      if (queues[index] == NULL){
            ERRBUF2("readref: ticket refers to unused queue index %u",
                                                      index);
            return(QE_BADTICKET);
      }
```

```
	/*
	 * you have a valid index; now validate the nonce; note we
	 * store the ticket in the queue, so just check that (same
	 * thing)
	 */
	if (queues[index]->ticket != qno){
		ERRBUF3("readref: ticket refers to old queue (new=%u, old=%u)",
				((queues[index]->ticket)&0xffff) - IOFFSET,
				(qno&0xffff) - NOFFSET);
		return(QE_BADTICKET);
	}

	/*
	 * check for internal consistencies
	 */
	if ((q = queues[index])->head < 0 || q->head >= MAXELT ||
					q->count < 0 || q->count > MAXELT){
		ERRBUF3("readref: internal inconsistency: head=%u,count=%u",
					q->head, q->count);
		return(QE_INTINCON);
	}
	if (((q->ticket)&0xffff) == 0){
		ERRBUF("readref: internal inconsistency: nonce=0");
		return(QE_INTINCON);
	}

	/* all's well -- return index */
	return(index);
}
```

The argument for this function is a token representing a queue; the purpose of this function is to validate the token and return the corresponding index. The first section of *readref* does this; it derives the index number from the token, and checks first that the index is a legal index, then that there is a queue with that index. If either fails, an appropriate error message is given. Notice that the error code simply indicates a problem with the parameter, although the message in *qe_errbuf* distinguishes between the two.

☞ Make parameters quantities that can be checked for validity, and check them.

As the caller of the library has supplied the token, a bogus token is not an internal error. So we use another error code to indicate the problem:

```
#define QE_BADTICKET    -3    /* bad ticket for the queue */
```

Next, we check that the queue with the same index as the token is the queue the token refers to. This is the "dangling reference" problem mentioned earlier. The current queue's token is stored in each queue structure, so we simply ensure the current token is the queue's token. If not, we handle the error appropriately.

☞ Check for references to outdated data structures.

The last section of code checks for internal consistency. The goal is to detect problems internal to the queue library. The consistency checks are:

1. The position of the queue *head* must lie between 0 and **MAXELT**.
2. The *count* of elements in the queue must be nonnegative and no greater than **MAXELT**.
3. The nonce can never be 0. This prevents a random integer 0 from being taken as a valid token.

When any of these are detected, the routine reports an error.

An alternate approach, favored by some experts, is to make this code conditionally compilable, and omit it on production versions. They either use `#ifdefs` to surround the code:

```
#ifdef DEBUG
	/* the code goes here */
```

```
#endif
```

or use the *assert*() macro. This saves space when the library is provided for production, but can make tracking down any problems more difficult when they occur in production software, because less information is provided than in a development environment.

☞ Assume "debugged code" isn't. When it's moved to other environments, previously unknown bugs may appear.

The *assert*() macro is described in the manual as *assert*(3) Its argument is an expression, and that expression is evaluated. If the expression is false, the macro writes a message to the standard error, aborts the program and forces a core dump for debugging purposes. For example, the internal consistency checking code could be replaced with:

```
assert((q = queues[index])->head < 0 || q->head >= MAXELT);
assert(q->count < 0 || q->count > MAXELT);
assert((q->ticket)&0xffff) != 0);
```

If the middle *assert* expression were false, the error message would be:

```
assertion "q->count < 0 || q->count > MAXELT" failed file "qlib.c", line 178
```

If the compile-time constant **NDEBUG** is defined, all *assert*() macros are empty, so they are in effect deleted from the program.

Exercise 10. The *assert* macro aborts the program if the condition fails. It applies the theory that "if the library is internally inconsistent, the entire set of queues cannot be trusted." The other methods allow the caller to attempt to recover. Which is better? When?

Queue Creation

The routine *create_queue* creates queues:

```
/*
 * create a new queue
 *
 * PARAMETERS:    none
 * RETURNED:      QTICKET          token (if > 0); error number (if < 0)
 * ERRORS:        QE_BADPARAM      parameter is NULL
 *                QE_TOOMANYQS     too many queues allocated already
 *                QE_NOROOM        no memory to allocate new queue
 *                                 (qe_errbuf has descriptive string)
 * EXCEPTIONS:    none
 */
QTICKET create_queue(void)
{
      register int cur;       /* index of current queue */
      register QTICKET tkt;   /* new ticket for current queue */

      /* check for array full */
      for(cur = 0; cur < MAXQ; cur++)
            if (queues[cur] == NULL)
                  break;
      if (cur == MAXQ){
            ERRBUF2("create_queue: too many queues (max %d)", MAXQ);
            return(QE_TOOMANYQS);
      }

      /* allocate a new queue */
      if ((queues[cur] = malloc(sizeof(QUEUE))) == NULL){
            ERRBUF("create_queue: malloc: no more memory");
            return(QE_NOROOM);
```

```
	}

	/* generate ticket */
	if (QE_ISERROR(tkt = qtktref(cur))){
		/* error in ticket generation -- clean up and return */
		(void) free(queues[cur]);
		queues[cur] = NULL;
		return(tkt);
	}

	/* now initialize queue entry */
	queues[cur]->head = queues[cur]->count = 0;
	queues[cur]->ticket = tkt;

	return(tkt);
}
```

The parameter list for this routine is empty; all information is returned as a function value. An alternate approach would be to pass the QTICKET back through the parameter list with the declaration

```
int create_queue(QTICKET *tkt)
```

and have the return value be the error code. But this can lead to confusion. Some library routines require pointers as arguments; others do not. Programmers may become confused, or suffer memory lapses, about which routines require pointers and which do not. The routine should guard against these potential problems.

☞ Keep parameter lists consistent.

Making all parameters be pointers is not suitable. First, pointers are difficult to check for validity; one can (and should) check for **NULL** pointers, but how can one portably determine if a non-**NULL** pointer contains an address in the process' address space or that the address is not properly aligned for the quantity to be stored there (on some systems, notably RISC systems, this can cause an alignment fault and terminate the program)? Using a style of programming akin to functional programming languages avoids these problems.

☞ Avoid changing variables through a parameter list; whenever possible, avoid passing pointers.

This routine checks if there is room for another queue; if so, it determines the index of the queue; if not, it reports an error and returns an error code. The error code is returned to the calling routine, which is not a part of the library:

```
#define QE_TOOMANYQS    -7    /* too many queues in use (max 100) */
```

In the error message we supply helpful information, namely the maximum number of queues allowed. This enables the programmers to know why the routine failed and tailor their code accordingly.

☞ Check for array overflow and report an error when it would occur (or take other corrective action).

The next part allocates space for the queue. Again, the routine checks for a failure in *malloc*, and reports it should it happen. Again, a special error code is used. This is of special importance since a *malloc* failure is usually due to a system call failure, and *perror*(3) will print a more informative message that the caller may desire. The queue library's error indicator is:

```
#define QE_NOROOM       -6    /* can't allocate space (sys err) */
```

☞ Check for failure in C library functions and system calls.

The routine then obtains a token for the new queue, checking again for failure. If the token cannot be obtained, the new queue is deleted and an error is returned. No error message is provided; the token generator *qtktref* provides that.

☞ Check for failure in other library functions in your library.

Finally, all quantities in the queue structure are set to default values (here, indicating an empty queue), and the token is returned. This way, we need not worry about initialization later in the library, when it might be harder to determine if initialization is needed.

☞ Initialize on creation.

Exercise 11. If a token can't be generated, then the error message in *qe_errbuf* comes from *qtktref*, but there is no indication of what routine called *qtktref*. Write a macro that will append this to the error message in *qe_errbuf*. Remember to check for bounds problems when you append to the contents of *qe_errbuf*.

Queue Deletion

This routine deletes queues.

```
/*
 * delete an existing queue
 *
 * PARAMETERS:    QTICKET qno       ticket for the queue to be deleted
 * RETURNED:      int               error code
 * ERRORS:        QE_BADPARAM       parameter refers to deleted, unallocated,
 *                                  or invalid queue (from readref()).
 *                QE_INTINCON       queue is internally inconsistent (from
 *                                  readref()).
 * EXCEPTIONS:    none
 */
int delete_queue(QTICKET qno)
{
      register int cur;       /* index of current queue */

      /*
       * check that qno refers to an existing queue;
       * readref sets error code
       */
      if (QE_ISERROR(cur = readref(qno)))
            return(cur);

      /*
       * free the queue and reset the array element
       */
      (void) free(queues[cur]);
      queues[cur] = NULL;

      return(QE_NONE);
}
```

This routine takes as a parameter the token of the queue to be deleted. It determines if the token is valid, and if not returns the error code. Thus, a queue which is not created, or one that has already been deleted, will report an error.

☞ Check that the parameter refers to a valid data structure.

The queue is then freed. The entry in the *queues* array is reset to indicate that an element of the array is available for reassignment.

☞ Always clean up deleted information – it prevents errors later on.

Exercise 12. The *free* statement is not protected by an *if* that checks to see whether *queues*[*cur*] is **NULL**. Is this a bug? If not, why don't we need to make the check?

Exercise 13. What prevents a caller from deleting the same queue twice?

Adding an Element to a Queue

Adding an element to a queue requires that it be placed at the tail:

```
/*
 * add an element to an existing queue
```

```
 *
 * PARAMETERS:    QTICKET qno       ticket for the queue involved
 *                int n             element to be appended
 * RETURNED:      int               error code
 * ERRORS:        QE_BADPARAM       parameter refers to deleted, unallocated,
 *                                  or invalid queue (from readref()).
 *                QE_INTINCON       queue is internally inconsistent (from
 *                                  readref()).
 *                QE_TOOFULL        queue has MAXELT elements and a new one
 *                                  can't be added
 * EXCEPTIONS:    none
 */
int put_on_queue(QTICKET qno, int n)
{
      register int cur;       /* index of current queue */
      register QUEUE *q;      /* pointer to queue structure */

      /*
       * check that qno refers to an existing queue;
       * readref sets error code
       */
      if (QE_ISERROR(cur = readref(qno)))
            return(cur);

      /*
       * add new element to tail of queue
       */
      if ((q = queues[cur])->count == MAXELT){
            /* queue is full; give error */
            ERRBUF2("put_on_queue: queue full (max %d elts)", MAXELT);
            return(QE_TOOFULL);
      }
      else{
            /* append element to end */
            q->que[(q->head+q->count)%MAXELT] = n;
            /* one more in the queue */
            q->count++;
      }

      return(QE_NONE);
}
```

The variables in the parameter list are not pointers; all are passed by value. As before, the validity of the token is first checked, and as *readref* builds the error message and code, if the token is not valid, the error is simply returned.

Adding an element to the queue requires checking that the queue is not full. The first part of the *if* … *else* … statement does this. The error message again gives the maximum number of elements that the queue can hold. If the queue is full, an appropriate error code is generated:

```
#define QE_TOOFULL      -5    /* append it to a full queue */
```

☞ Allow error messages to contain numbers or other variable data.

Removing an Element from the Queue

We remove elements from the head of the queue:

```
/*
```

```
 * take an element off the front of an existing queue
 *
 * PARAMETERS:    QTICKET qno       ticket for the queue involved
 * RETURNED:      int               error code or value
 * ERRORS:        QE_BADPARAM       bogus parameter because:
 *                                  * parameter refers to deleted, invalid,
 *                                    or unallocated queue (from readref())
 *                                  * pointer points to NULL address for
 *                                    returned element
 *                                  (qe_errbuf has descriptive string)
 *                QE_INTINCON       queue is internally inconsistent (from
 *                                  readref()).
 *                QE_EMPTY          no elements so none can be retrieved
 * EXCEPTIONS:    none
 */
int take_off_queue(QTICKET qno)
{
      register int cur;             /* index of current queue */
      register QUEUE *q;            /* pointer to queue structure */
      register int n;               /* index of element to be returned */

      /*
       * check that qno refers to an existing queue;
       * readref sets error code
       */
      if (QE_ISERROR(cur = readref(qno)))
            return(cur);

      /*
       * now pop the element at the head of the queue
       */
      if ((q = queues[cur])->count == 0){
            /* it's empty */
            ERRBUF("take_off_queue: queue empty");
            return(QE_EMPTY);
      }
      else{
            /* get the last element */
            q->count--;
            n = q->head;
            q->head = (q->head + 1) % MAXELT;
            return(q->que[n]);
      }

      /* should never reach here (sure ...) */
      ERRBUF("take_off_queue: reached end of routine despite no path there");
      return(QE_INTINCON);
}
```

Here we must distinguish between a return value that is an error code and a return value that comes from the queue. The solution is to check the error buffer *qe_errbuf*. Before this function is called, the first byte of that array is set to the NUL byte '\0'. If, on return, the error buffer contains a string of length 1 or greater, an error occurred and the returned value is an error code; otherwise, no error occurred. So the calling sequence is:

```
qe_errbuf[0] = '\0';
```

```
rv = take_off_queue(qno);
if (QE_ISERROR(rv) && qe_errbuf[0] != '\0')
```
… rv contains error code, qe_errbuf the error message …
```
else
```
… no error; rv is the value removed from the queue …

This way, we need not pass a pointer through the parameter list. The disadvantage of this method is the complexity of calling the function; however, that seems a small price to pay to avoid a possible segmentation fault within the library. The standard I/O library function *fseek* uses a similar technique to distinguish failure from success in some cases.

☞ Avoid changing variables through a parameter list; whenever possible, avoid passing pointers.

Exercise 14. Rewrite *take_off_queue* to use a second parameter, *int *n*, and return the value removed from the queue through that parameter. (Use the error code **BADPARAM**, defined as –1, to report an invalid pointer.) The function value is the error code. Compare and contrast this approach with the one used in the above version. When would you use each?

Exercise 15. How does *fseek*(3S) use *errno* to distinguish failure from success?

The rest of this routine is similar to *add_to_queue*. We check the parameter, and then validate the queue. We next check for an empty queue, and report an error if the queue is empty:

```
#define QE_EMPTY        -4    /* take it off an empty queue */
```

If it is not empty, it returns the element at the head of the queue. The *head* index is incremented to move to the next element in the queue, which becomes the element at the head of the queue.

Summary

The above routines give several examples of the differences between robust and fragile coding. The above routines are robust, because they cannot be crashed by poor or incorrect calls, or inconsistency in the caller. They form a module, and have informational cohesion and stamp coupling (the latter because they share data in common variables; specifically, *queues*, *noncectr*, and *qe_errbuf*. While the coding of these versions of the routines takes more time than the fragile routines, they take much less to debug, and will require less debugging time for applications using these routines.

Exercise 16. Rewrite this library to allocate space for each queue dynamically. Specifically, change *create_queue* to take the parameter *int size*, where *size* is to be the maximum number of elements allowed in the queue. Allocate space for the queue array dynamically.

Exercise 17. Rewrite this library to use a linked list structure for each queue. What are the advantages and disadvantages to this?

Conclusion

Many security problems arise from fragile coding. The UNIX operating system, and the C standard libraries, encourage this. They provide library calls that can cause severe problems; for example, *gets*(3S) does not check for buffer overflow. Avoid these routines!

Exercise 18. Why should the following functions be avoided when writing robust code: *gets*, *strcpy*, *strcat*, *sprintf*? How would you modify them to make them acceptable?

Exercise 19. When should members of the *scanf* family of functions be avoided while writing robust code? What should you use instead?

Acknowledgements: Chip Elliott (then of Dartmouth College, later of BBN Communications Corp.) provided the inspiration for this. His handout “Writing Bomb-Proof Code” for Dartmouth’s COSC 23, Introduction to Software Engineering class, succinctly described many of the principles and techniques which I have extended.

Thanks to Kim Knowles for her constructive suggestions on this document. Also, the students of ECS 40 and ECS 153 here at UC Davis helped me refine this exposition by asking questions and indicating what was difficult to understand. I hope this document is clear; it is certainly clearer than earlier versions!