# Robust Programming

- Style of programming that prevents abnormal termination and unexpected actions
  - Code handles bad inputs reasonably
  - Code assumes errors will occur and takes appropriate action
  - Also called *bomb-proof programming*

# Principles

- Paranoia
  - Don’t trust anything you don’t generate
- Stupidity
  - Assume caller or user will make mistakes calling or using the routine or program
- Dangerous implements
  - Hide anything your routines expect to remain consistent across calls
- Can’t happen
  - Handle impossible cases; they may be(come) possible

# Fragile Library

- fqlib.h

```
/* 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);     /* manage queue */
void put_on_queue(QUEUE *, int);      /* add to queue */
void take_off_queue(QUEUE *, int *);  /* pull off queue */
```

# Problem

- Callers have access to internal elements of queue structure
  - Note pointer given so user can reference queue contents directly
  - User sets:

```
qptr->count = 755
```

    This says queue now has 755 elements, regardless of how many it actually has!

# Managing Queues

```
/* create or delete a 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);
        }
}
```

# Problems

- Order of elements in parameter list is not checked
  - Is it flag, size or size, flag?
- Value of flag is arbitrary
  - Why is 1 create, 0 delete? What if it's 2?
- Pointers to pointers causes problems
  - Is it a singly indirect or doubly indirect reference?

# Problems

- Parameter values not sanity checked
  - Error, or may not be possible (pointers)
- Delete unallocated queue, or previously deleted queue

```
qmanage(&qptr, 1, 100);
qmanage(&qptr, 0, 1);
qmanage(&qptr, 0, 1);
```

- Return values not checked
  - What happens if *malloc* returns **NULL**?

# Problems

- What about integer overflow?
  - Say *size* is $2^{31}$ and integers are 4 bytes
  - Argument to *malloc* is $2^{31} \times 4$ or … $2^{33}$

# Adding to a Queue

```
/* add an element to an existing queue */
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++;
}
```

# Problems

- Parameters not checked
  - What if *qptr* is **NULL**, or refers to a deleted queue?
- Values in structures not checked
  - What if *qptr* is valid but *qptr->que* is not?
- Array overflow not checked
  - What if queue is full when this is called?

# Removing From a Queue

```
/* take element off the front of existing queue */
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;
}
```

# Problems

- Parameters not checked
  - What if *qptr* is **NULL**, or refers to a deleted queue?
  - What if *n* is invalid pointer?
- Values in structures not checked
  - What if *qptr* is valid but *qptr->que* is not?
- Array underflow not checked
  - What if queue is empty when this is called?

# Summary: Problems with *fqlib*

- 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.

# Summary: Problems with *fqlib*

- 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.

# Robust Library

- Hide queue structure
  - Internally: array of queues
- Interface: a *token*
  - Integer derived from index of queue
    - Make sure 0 is not a valid token!
- Problem: dangling references
  - Use a *nonce* (or *generation number*)
- Result: token is function of index, nonce

# Why a Nonce?

- Suppose queue A generated with index 7, used, deleted
- Now queue B generated with index 7, used
- Caller refers to queue A
  - Without nonce: token of A and B are *f*(7)
  - With nonce: token of A is *f*(7, 124) and token of B is *f*(7, 125), *which are different*

# Token Generation

$$f(i, n) = ((i + 0x1221)<<16)|(n+0x0502)$$

- Type of ticket:

```
typedef long int QTICKET;
```

- Use tokens, which are QTICKETs, rather than integers, to refer to queues
  - Prevents direct references to queues by callers
  - Enables detecting references to defunct queues

# Error Handling

- Print error messages
  - Can mess up interfaces of callers
- Return error results only
  - Allows callers to handle error appropriately *for its own purposes*
  - Important: handle errors consistently

# Error Handling Interface

- Return value indicates whether error:

```
#define QE_ISERROR(x)      ((x) < 0) /* true if x is error code */
#define QE_NONE            0         /* no errors */
/* … various other QE_ error codes defined throughout … */
```

- Error message in buffer:

```
extern char qe_errbuf[256];
```

- Error manipulation functions (in qlib.c):

```
#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)
```

# Function Interfaces

Goal: eliminate low cohesion of *qmanage*:

```
QTICKET create_queue(void);    /* create a queue */
int delete_queue(QTICKET);     /* delete a queue */
             /* put number on end of queue */
int put_on_queue(QTICKET, int);
             /* pull number off front of queue */
int take_off_queue(QTICKET);
```

# Internal Structures

- Offsets:

```
#define IOFFSET 0x1221  /* hides index number */
#define NOFFSET 0x0502  /* hides nonce in ticket */
```

- 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 iundex in que of the queue */
   int count;          /* number of elements in queue */
} QUEUE;
```

- Shared variables:

```
static QUEUE *queues[MAXQ];           /* the queues */
static unsigned int noncectr = 1;     /* the nonce */
```

# Token Creation

```
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 too large (assumed less than %d)",
                                                    index, MAXQ);
        return(QE_INTINCON);
   }
   /* get the high part of the ticket */
   high = (index + IOFFSET)&0x7fff;
   if (high != index + IOFFSET){
        ERRBUF3("qtktref: index %u too large (assumed less than %u)",
                                           index, 0x7fff – IOFFSET);
        return(QE_INTINCON);
   }
```

# More Token Creation

```
    /*
     * get the low part of the ticket (nonce)
     * SANITY CHECK: be sure nonce fits into 16 bits
     */
    low = (noncectr + NOFFSET) & 0xffff;
    if (low != (noncectr++ + NOFFSET) || low == 0){
         ERRBUF2("qtktref: generation number too large (max %u)\n",
                                                  0xffff – NOFFSET);
         return(QE_INTINCON);
    }

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

# Points

- Only internal functions can reference it
- Return token as value, not via parameter list
  - Keep interfaces simple, even when internal
- Check parameters
  - Checks index for validity
- Error values, messages must be informative
  - Error code identifies precise problem
  - Error message gives details of problem

# Token Analysis

```
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);
   }
```

# Tokn Analysis

```
/* you have a valid index; now validate the nonce */
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);
}
```

# Token Analysis

```
   /* more internal consistencies checking */
   if (((q->ticket)&0xffff) == 0){
       ERRBUF(
          "readref: internal inconsistency: nonce=0");
       return(QE_INTINCON);
   }

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

# Points

- Make parameters quantities that *can* be checked for validity, and do so
  - Here, it's a QTICKET, and is checked
- Check for references to outdated data structures
  - The check on the nonce does this
- Check for internal consistency
  - The last check does this

# Alternate Approach

- Make consistency checks conditionally compilable and omit them in production code
  - This makes code run (slightly) faster, but much harder to debug when a problem occurs in production
  - Use #ifdef … #endif or *assert*() macro:

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

  - On error, prints and then core dumps:

```
assertion "(q->ticket)&0xffff) != 0" failed file "qlib.c",
   line 179
```

# Point

- Assume debugged code isn't; if you move it to different environments, previously unknown bugs may appear
  - Don't make checks in code conditionally compilable
  - If you use *assert*(), do **not** define **NDEBUG**
    - If defined, all *assert*s become empty code

# Queue Creation

```
QTICKET create_queue(void)
{
   register int cur;   /* index of current queue */
   register int 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);
   }
```

# Queue Creation

```
/* 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 — abend procedure */
     (void) free(queues[cur]);
     queues[cur] = NULL;
     return(tkt);
}
```

# Queue Creation

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

    return(tkt);
}
```

# Points

- Keep parameter lists consistent
  - Passing pointer to token inconsistent with other routines' parameters
- Avoid passing pointers
  - Means avoid changing variables through parameter lists
  - Pointers hard to check for validity
- Check for array overflow
  - Checks here to be sure there is room for new array
  - On error, error code and message indicate cause

# Points

- Check C library function, system calls for errors
  - These can fail!
- Check your own library functions for failure
  - These can fail too, especially if there is an internal error
- Initialize on creation
  - This ensures the variables are initialized

# Queue Deletion

```
int delete_queue(QTICKET qno)
{
   register int cur;   /* index of current queue */

   /* check that qno refers to an existing queue */
   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);
}
```

# Points

- Check that parameter refers to valid data structure
  - Note if queue is already deleted, this check fails
- Always clean up deleted information
  - Here, ensures queue creation routine can re-use slot, and memory released for re-use

# Adding To a Queue

```
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; */
   if (QE_ISERROR(cur = readref(qno)))
       return(cur);
```

# Adding To a Queue

```
    /* 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);
}
```

# Points

- Allow error messages to contain numbers and variable data
  - Tells others limits and enables messages to reflect what happened
- No pointers are passed
  - All parameters passed by value
  - Validity of tokens can be checked

# Removing From a Queue

```
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; */
   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);
   }
```

# Removing From a Queue

```
    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);
}
```

# Detecting Errors

- Problem: negative number may be error code ***or*** a valid integer
- Solution: note *qe_errbuf* set on error, so call as follows:

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