What Is Robust Code?

- Robust code
  - A style of programming that prevents abnormal termination or unexpected actions
    - Handles bad input gracefully
    - Detects internal errors and handles them gracefully
    - On failure, provides information to aid in recovery or analysis
- Fragile code
  - Non-robust code

Example of Fragile Code

- It's always fun to pick apart someone else's code!
- Library: implement standard queues (LIFO structures)
  - Written in C, in typical way
- Files
  - queue.h
    - Header file containing QUEUE structure and prototypes
  - queue.c
    - Library functions; compiled and linked into programs

Queue Structure

- In queue.h:

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

Interfaces

- In queue.h:
  - Create, delete queues

```
void qmanage(QUEUE **, int, int);
```

  - Add element to tail of queue

```
void put_on_queue(QUEUE *, int);
```

  - Take element from head of queue

```
void take_off_queue(QUEUE *, int *);
```

How To Mess This Up

- Create queue
- Change counter value

```
QUEUE *xxx;
…
qmanage(&xxx, 1, 100);
xxx->count = 99;
```

- Now the queue structure says there are 99 elements in queue

qmanage

```
/* create or delete a queue
 * PARAMETERS:     QUEUE **qptr       pointer to, queue
 *                 int flag           1 for create, 0 for delete
 *                 int sizemax        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);
  }
}
```

Adding to a Queue

```
/* 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++;
}
```

Taking from a Queue

```
/* 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;
}
```

Robust Programming

- Basic Principles
  - Paranoia: don't trust what you don't generate
  - Stupidity: if it can be called (invoked) incorrectly, it will be
  - Dangerous implements: if something is to remain consistent across calls (invocations), make sure no-on else can access it
  - Can't happen: check for "impossible" errors
- Think "program defensively"

Queue Structure

- It's a dangerous implement
  - We never make it available to the user
    - Use *token* to index into array of queues
  - Use this trick to prevent "dangling reference"
    - Include in each created token a *nonce*
    - When referring to queue using token, check that index *and nonce* are both active
  - But won't token of 0 or 1 be valid always?
    - Construct token so they are not

Example Token

- Need to be able to extract index and nonce from it

```
token = ((index + 0x1221)<<16)|(nonce+0x0502)
```

  - Question: what assumptions does this token structure make?
- Define a type for convenience

```
typedef long int QTICKET;
```

- Lesson: don't return pointers to *internal* structures; use tokens

Error Handling

- Need to distinguish error codes from legitimate results
  - Convention: all error codes are *negative*
  - Convention: every error produces a *text* message saved in an externally visible buffer

```
    /* true if x is a qlib error code */
#define QE_ISERROR(x) ((x) < 0)
#define QE_NONE 0/* no errors */
    /* error buffer; contains message describing
     * last error; visible to callers */
extern char qe_errbuf[256];
```