Robust Programming

ECS 153 Spring Quarter 2021

Module 3

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

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”

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

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];
```

Error Handling

```
                    /* 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];
                    /* useful macros */
#define ERRBUF(str)\
            (void) strncpy(qe_errbuf, str, sizeof(qe_errbuf)),\
            qe_errbuf[255] = '\0'
#define ERRBUF2(str,n)\
            (void) sprintf(qe_errbuf, str, n)
#define ERRBUF3(str,n,m)\
            (void) sprintf(qe_errbuf, str, n, m)
```

Cohesion

- How well parts of a function hang together
- *qmanage* had low cohesion
  - Two really independent parts, create and delete
  - Much simpler to do two separate functions

New Interfaces

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

Queue Structure

- Invisible to caller; can change easily

```
        /* the queue structure */
typedef int QELT;                   /* type being queued */
typedef struct queue {
            QTICKET ticket;             /* unique queue ID */
            QELT que[MAXELT];           /* actual queue */
            int head;                   /* index of head */
            int count;                  /* number of elts */
} QUEUE;
        /* array of queues */
static QUEUE *queues[MAXQ];
        /* current nonce */
static unsigned int noncectr = NOFFSET;
```

Token Generation

```
static QTICKET qtktref(unsigned int index)
{
        unsigned int high;  /* high part of token (index) */
        unsigned int low;   /* low part of ticket (nonce) */

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

Token Generation

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

Token Generation

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

Checklist

- Make interfaces simple, even when for internal use only
- Check everything, even internally generated parameters
- Give useful error messages, and describe the error precisely
  - For those caused by internal inconsistencies, name the routine to help whoever debugs it

Token Interpretation

```
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);
      }
```

Token Interpretation

```
/* you have a valid index
 * now validate the nonce; note we store the
 * ticket in the queue structure
 */
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);
}
```

Token Interpretation

```
  /* 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);
}
```

Checklist

- Make parameters quantities that can be checked for validity—and check them!
- Check for references to outdated (old, especially discarded) data
- Assumed “debugged” code isn’t. Leave the checks in!

Creating a Queue

```
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);
  }
```

Creating a Queue

```
/* 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);
}
```

Creating a Queue

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

Checklist

- Keep parameter lists consistent
  - Don’t have some require pointers and others not
- Check for (array) overflow and report it (or correct for it)
- Check for failure in library functions, system calls, and your own functions
  - Only time not to do this is when you don’t care if the called function fails

Deleting a Queue

```
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);
}
```

Checklist

- Check the parameter refers to a valid data structure
- Always clean up deleted information
  - It prevents errors later on

Adding an Element 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; readref
   * sets error code
   */
  if (QE_ISERROR(cur = readref(qno)))
              return(cur);
```

Adding an Element 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);
}
```

Removing an Element 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 elt to be returned */

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

Removing an Element from a Queue

```
  /* 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);
}
```

Calling Removing Function

```
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 …

Summary of Problems

- Order of parameters (arguments) not checked
- Values of parameters (arguments) arbitrary
- Calls with pointers to pointers
- Values of parameters not sanity checked
- Return values (especially from library functions) not checked
- Overflow, underflow ignored
  - Both integer and array

Summary of Problems

- Callers have access to internal structures
- Internal values in variables, structures not sanity checked
- Users can delete non-existent or already delete things
- Users can allocate already allocated things

Non-Robust Programming

- Introduces security problems
  - Fragile code makes assumptions about user, environment that are often wrong
  - Fragile code harder to fix when a security problem is found
- Introduces non-security problems
  - Maintenance more complex, takes more time
  - Easier for users, callers to make *accidental* errors in invocation