CSE216 Programming Abstractions
Reference Counting

YoungMin Kwon

Reference Counting

- Example:
  - Increase cnt_ref (addref) when the object is referenced
  - Decrease cnt_ref (release) when the object is no longer referenced
  - cnt_ref is set to 1 when an object is created
  - If cnt_ref becomes 0, free the object

```
typedef struct refobj {
    tag_t tag;
    int cnt_ref;    //reference count
    void ( *addref  )(struct refobj *self);
    void ( *release )(struct refobj *self);
} refobj_t;
```

Managing Reference Counts

- Programming guidelines
  - Call addref
    - A1: writes address to a local variable or a field of an object
    - A2: callee writes to [out] or [in, out] parameter
    - A3: callee returns an address
  - Call release
    - R1: before overwriting a local variable or a field of an object
    - R2: before leaving the scope of local variables
    - R3: before callee writes to [in, out] parameter
      - [out] parameters are assumed to null (don't release them)
  - Skip addref, release
    - S1: caller passes an address to [in] parameter
      - Caller lives longer than callee

Reference Counting Exercise

- Download reci_refcounting.zip
- Implement TODOs.

```
//
// rat.h
//
#ifndef __RAT__     //to avoid multiple inclusion
#define __RAT__
#include "refobj.h"

typedef struct rat rat_t;
struct rat {
    refobj_t ref;   //ref is at the beginning of rat (they have the same address)
    int  ( *get_num )(/*in*/ rat_t* r);
    int  ( *get_den )(/*in*/ rat_t* r);
    void ( *print   )(/*in*/ rat_t* r);
    void ( *add     )(/*in*/ rat_t* a, /*in*/ rat_t* b, /*out*/ rat_t** pp_res);
    void ( *sub     )(/*in*/ rat_t* a, /*in*/ rat_t* b, /*out*/ rat_t** pp_res);
    void ( *mul     )(/*in*/ rat_t* a, /*in*/ rat_t* b, /*out*/ rat_t** pp_res);
    void ( *div     )(/*in*/ rat_t* a, /*in*/ rat_t* b, /*out*/ rat_t** pp_res);

    int num;
    int den;
};

//extern: make it visible to other files
extern void rat_make(int num, int den, /*out*/ rat_t** pp_res);

#endif
```

```
//
// expr.h
//
#ifndef __EXPR__
#define __EXPR__

#include "refobj.h"
#include "rat.h"

typedef struct expr expr_t;
struct expr {
    refobj_t ref;   //ref is at the beginning of rat
    void ( *eval  )(/*in*/ expr_t *self, /*out*/ rat_t **pp_res);
    void ( *print )(/*in*/ expr_t *self);
};

typedef struct expr_num {
    refobj_t ref;   //first part is the same as expr_t
    void ( *eval  )(/*in*/ expr_t *self, /*out*/ rat_t **pp_res);
    void ( *print )(/*in*/ expr_t *self);

    rat_t *n; //number
} expr_num_t;
```

```
typedef enum {
    OBJ_RAT,
    OBJ_EXPR_NUM,
    OBJ_EXPR_OPR,
    OBJ_COUNT,
} tag_t;
```

```
typedef struct expr_opr {
    refobj_t ref;   //first part is the same as expr_t
    void ( *eval  )(/*in*/ expr_t *self, /*out*/ rat_t **pp_res);
    void ( *print )(/*in*/ expr_t *self);

    void ( *opr ) (/*in*/ rat_t *a, /*in*/ rat_t *b, /*out*/ rat_t **pp_res);
    expr_t *a;     //operand 1
    expr_t *b;     //operand 2
} expr_opr_t;

extern void expr_make_num(/*in*/ rat_t *n,  /*out*/ expr_t **pp_res);
extern void expr_make_add(/*in*/ expr_t *a, /*in*/  expr_t *b,
                                            /*out*/ expr_t **pp_res);
extern void expr_make_sub(/*in*/ expr_t *a, /*in*/  expr_t *b,
                                            /*out*/ expr_t **pp_res);
extern void expr_make_mul(/*in*/ expr_t *a, /*in*/  expr_t *b,
                                            /*out*/ expr_t **pp_res);
extern void expr_make_div(/*in*/ expr_t *a, /*in*/  expr_t *b,
                                            /*out*/ expr_t **pp_res);

#endif
```

```
//
// expr.c
//
…

static void eval_num(/*in*/ expr_t *self, /*out*/ rat_t **pp_res) {
    expr_num_t *expr = (expr_num_t*) self;
    ON_FALSE_EXIT(self->ref.tag == OBJ_EXPR_NUM,
                  strmsg("tag (%d) is not OBJ_EXPR_NUM", self->ref.tag));

    //TODO: return expr->n

}

static void print_num(/*in*/ expr_t *self) {
    expr_num_t *expr = (expr_num_t*) self;
    ON_FALSE_EXIT(self->ref.tag == OBJ_EXPR_NUM,
                  strmsg("tag (%d) is not OBJ_EXPR_NUM", self->ref.tag));
    expr->n->print(expr->n);
}
```

```
static void eval_num(/*in*/ expr_t *self, /*out*/ rat_t **pp_res) {
    expr_num_t *expr = (expr_num_t*) self;
    ON_FALSE_EXIT(self->ref.tag == OBJ_EXPR_NUM,
                  strmsg("tag (%d) is not OBJ_EXPR_NUM", self->ref.tag));

    //TODO: return expr->n
    *pp_res = expr->n;
    (*pp_res)->ref.addref(&(*pp_res)->ref);
}
```

```
static void release_num(refobj_t *ref) {
    expr_num_t *expr = (expr_num_t*) ref;

    //TODO: implement release
    //  - call refobj_decref
    //  - if cnt_ref is 0, release expr->n and
    //    free expr->ref (refobj_free)
}
```

```
static void release_num(refobj_t *ref) {
    expr_num_t *expr = (expr_num_t*) ref;

    //TODO: implement release
    //  - call refobj_decref
    //  - if cnt_ref is 0, release expr->n and
    //    free expr->ref (refobj_free)
    refobj_decref(&expr->ref);
    if(expr->ref.cnt_ref == 0) {
        expr->n->ref.release(&expr->n->ref);
        refobj_free(&expr->ref);
    }
}
```

```
void expr_make_num(/*in*/ rat_t *n, /*out*/ expr_t **pp_res) {
    expr_num_t* expr  = refobj_alloc(OBJ_EXPR_NUM, sizeof(expr_num_t));
    expr->ref.release = release_num;
    expr->eval        = eval_num;
    expr->print       = print_num;

    //TODO: copy n to expr->n


    //TODO: return expr

}
```

```
void expr_make_num(/*in*/ rat_t *n, /*out*/ expr_t **pp_res) {
    expr_num_t* expr  = refobj_alloc(OBJ_EXPR_NUM, sizeof(expr_num_t));
    expr->ref.release = release_num;
    expr->eval        = eval_num;
    expr->print       = print_num;

    //TODO: copy n to expr->n
    expr->n = n;
    expr->n->ref.addref(&expr->n->ref);

    //TODO: return expr
    *pp_res = (expr_t*)expr;
    (*pp_res)->ref.addref(&(*pp_res)->ref);
    expr->ref.release(&expr->ref);
}
```

```
static void eval_opr(/*in*/  expr_t *self,
                     /*out*/ rat_t **pp_res) {
    expr_opr_t *expr = (expr_opr_t*) self;
    ON_FALSE_EXIT(self->ref.tag == OBJ_EXPR_OPR,
                  strmsg("tag (%d) is not OBJ_EXPR_OPR", self->ref.tag));

    //TODO: evaluate expr->a and expr->b
    rat_t *a, *b;

    //TODO: call opr with a and b and get the result in pp_res

    //TODO: return
}
```

```
static void eval_opr(/*in*/  expr_t *self,
                     /*out*/ rat_t **pp_res) {
    expr_opr_t *expr = (expr_opr_t*) self;
    ON_FALSE_EXIT(self->ref.tag == OBJ_EXPR_OPR,
                  strmsg("tag (%d) is not OBJ_EXPR_OPR", self->ref.tag));

    //TODO: evaluate expr->a and expr->b
    rat_t *a, *b;
    expr->a->eval(expr->a, &a);
    expr->b->eval(expr->b, &b);

    //TODO: call opr with a and b and get the result in pp_res
    expr->opr(a, b, pp_res);

    //TODO: return
    a->ref.release(&a->ref);
    b->ref.release(&b->ref);
}
```

```
static void release_opr(refobj_t *ref) {
    expr_opr_t *expr = (expr_opr_t*)ref;

    //TODO: implement release
    //  - call refobj_decref
    //  - if cnt_ref is 0, release expr->a, expr->b and
    //    free expr->ref (refobj_free)
}
```

```
static void release_opr(refobj_t *ref) {
    expr_opr_t *expr = (expr_opr_t*)ref;

    //TODO: implement release
    //  - call refobj_decref
    //  - if cnt_ref is 0, release expr->a, expr->b and
    //    free expr->ref (refobj_free)
    refobj_decref(&expr->ref);
    if(expr->ref.cnt_ref == 0) {
        expr->a->ref.release(&expr->a->ref);
        expr->b->ref.release(&expr->b->ref);
        refobj_free(&expr->ref);
    }
}
```

```
//make expr_opr without opr
static void make_expr_opr(/*in*/ expr_t *a, /*in*/ expr_t *b, char *str_opr,
                          /*out*/ expr_t **pp_res) {
    expr_opr_t* expr  = refobj_alloc(OBJ_EXPR_OPR, sizeof(expr_opr_t));
    expr->ref.release = release_opr;
    expr->eval        = eval_opr;
    expr->print       = print_opr;

    //TODO: copy a to expr->a and b to expr->b

    //update opr using rat_op

    //return the result
}
```

```
//make expr_opr without opr
static void make_expr_opr(/*in*/ expr_t *a, /*in*/ expr_t *b, char *str_opr,
                          /*out*/ expr_t **pp_res) {
    expr_opr_t* expr  = refobj_alloc(OBJ_EXPR_OPR, sizeof(expr_opr_t));
    expr->ref.release = release_opr;
    expr->eval        = eval_opr;
    expr->print       = print_opr;

    //TODO: copy a to expr->a and b to expr->b
    expr->a = a;
    expr->b = b;
    expr->a->ref.addref(&expr->a->ref);
    expr->b->ref.addref(&expr->b->ref);

    //update opr using rat_op
    expr->opr = rat_op(str_opr);

    //return the result
    *pp_res = (expr_t*)expr;
    (*pp_res)->ref.addref(&(*pp_res)->ref);
    expr->ref.release(&expr->ref);
}
```

```
//
// app.c
//
#include "rat.h"
#include "expr.h"
#include <stdio.h>

int main() {
    rat_t* a, *b, *c;
    rat_make(1, 2, &a);
    rat_make(1, 3, &b);
    rat_make(1, 5, &c);

    expr_t *na, *nb, *nc;
    expr_make_num(a, &na);
    expr_make_num(b, &nb);
    expr_make_num(c, &nc);

    //a * b + b * c
    expr_t *x, *y, *z;
    expr_make_mul(na, nb, &x);
    expr_make_mul(nb, nc, &y);
    expr_make_add(x, y, &z);

    z->print(z);
    printf("\n");

    rat_t *d;
    z->eval(z, &d);
    d->print(d);
    printf("\n");


    a->ref.release(&a->ref);
    b->ref.release(&b->ref);
    c->ref.release(&c->ref);
    d->ref.release(&d->ref);

    na->ref.release(&na->ref);
    nb->ref.release(&nb->ref);
    nc->ref.release(&nc->ref);

    x->ref.release(&x->ref);
    y->ref.release(&y->ref);
    z->ref.release(&z->ref);

    refobj_check_dealloc();
}
```

```
$ a.out
[1 / 2] * [1 / 3] + [1 / 3] * [1 / 5]
[7 / 30]
```

X Window System

- Install GTK (GIMP Toolkit)
  - We will use GTK+ 3.0 in the class
    - GNOME environment uses GTK+ as a base
    - GTK+ uses the C programming language
  - Install GTK and other development environments

```
;; preferred method
$ sudo apt-get install gnome-devel

;; if above method does not work, try
$ sudo apt-get install libgtk-3-dev
$ sudo apt-get install libgtk-3-doc
```

GIMP 2.2

X Window System

- Check the installation
  - Download hello.c and Makefile
  - Compile hello.c
    - make hello
  - Run Xming/Xquartz from your local machine
  - Run hello from AWS