CSE216 Programming Abstractions

Memory Management

YoungMin Kwon

Memory Management

- Dynamic memory management
  - It is difficult to track whether a memory block is being used or not
- Garbage collection
  - Find all reachable memory blocks from the process
  - Free all unreachable memory blocks
- Reference counting
  - Track the number of pointers referencing the memory block
  - If the reference counter reaches 0, free the block

Garbage Collection

- Garbage
  - Any variable not reachable from your program

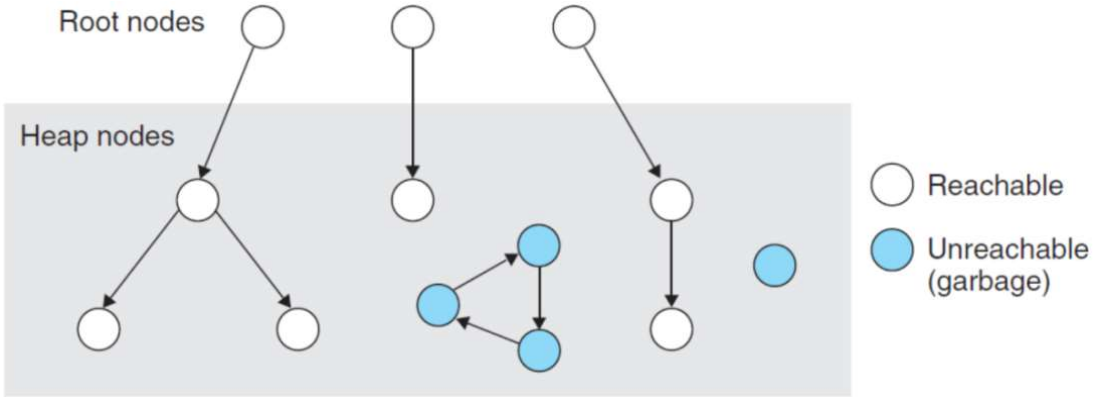
```
void MakeGarbage()
{
    // After returning from MakeGarbage,
    // p becomes unreachable

    int *p = (int*) malloc(100);

    return;
}
```

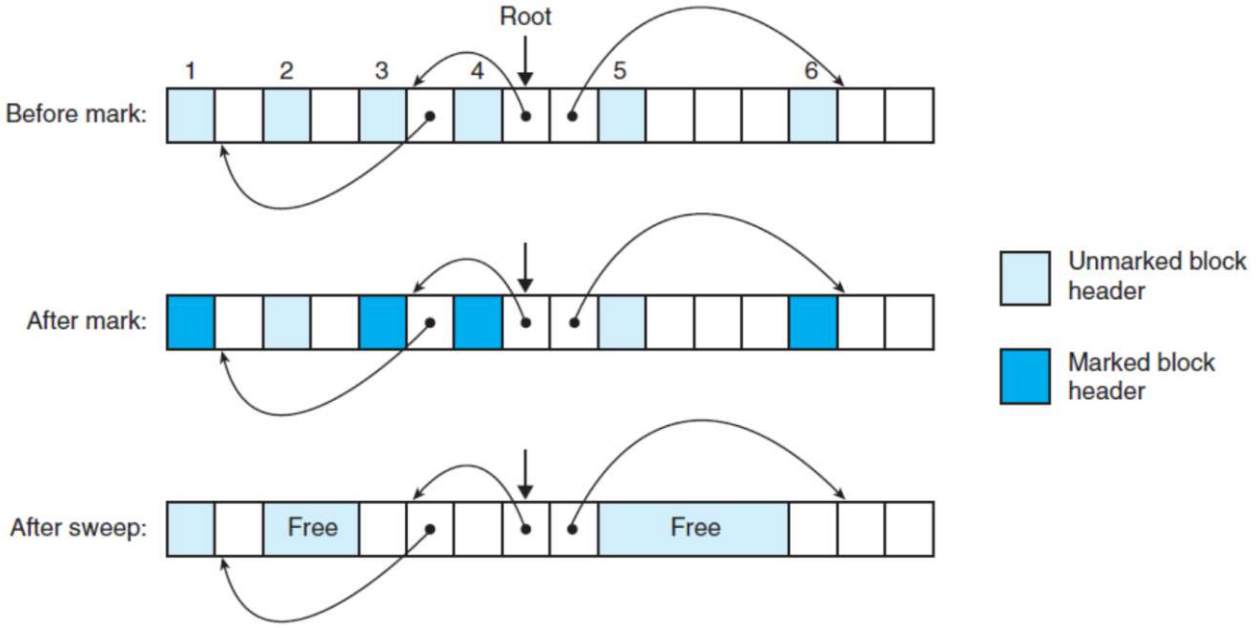
Garbage Collection

- Reachability Graph
  - Nodes are variables
  - There is an edge $v_i \rightarrow v_j$ if a pointer variable v_i is pointing to another variable v_j
  - A variable v_i is reachable if there is a path to v_i from any root variables (live variables not in the heap)



Garbage Collection

- Mark and Sweep garbage collector
  - Mark phase: mark all variables reachable from any root variables
  - Sweep phase: free the variables not marked during the mark phase.



Reference Counting

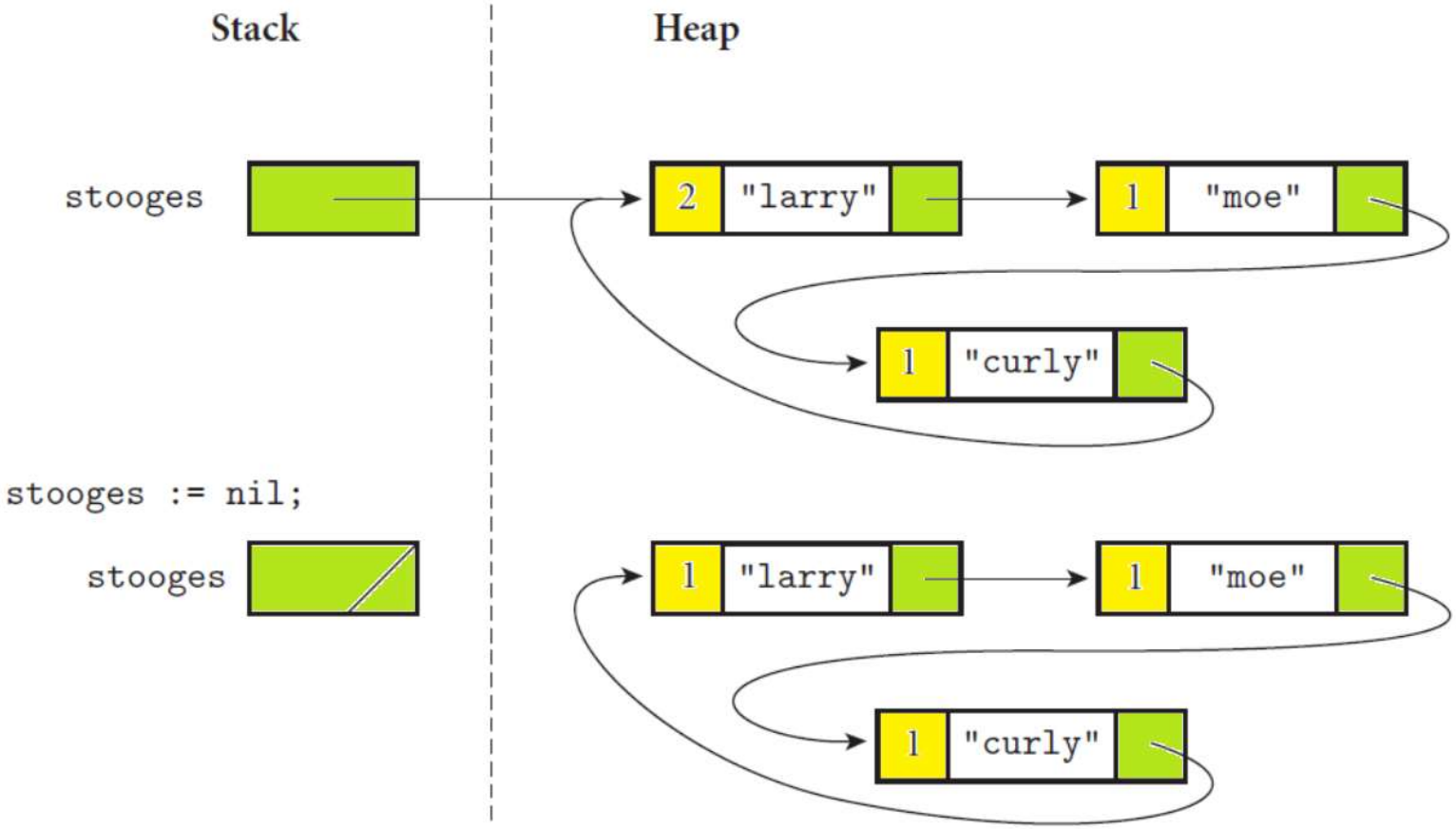
- Reference counting
  - A simple garbage collection algorithm
  - Place a counter in each object
  - The counter keeps track of the number of pointers referencing the object

Reference Counting

- Rules
  - Object creation: ref count is set to 1
  - Assignment to a pointer: decrease ref count of the LHS of =, increase ref count of the RHS of =
  - Procedure return: decrease ref counts of all local pointers
  - When ref count reaches 0: free the object, decrease the ref counts of all objects the object was referencing

Reference Counting

- May fail to collect circular structures



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;
```

```c
// refobj.h
//
#ifndef __REFOBJ__
#define __REFOBJ__
#include "common.h"

typedef enum {
    OBJ_RAT,
    OBJ_COUNT
} tag_t;

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

typedef struct refstat {    //to track which object is not released
    //total ref count
    int cnt_ref[OBJ_COUNT];

    //live object count
    int cnt_obj[OBJ_COUNT];
} refstat_t;
```

```
//allocate container of type tag and initialize ref obj
extern void *refobj_alloc(tag_t tag, size_t size);

//
//use the next 3 functions only from release
//

//free the container object of ref
extern void refobj_free(refobj_t *ref);

//increase cnt_ref of ref
extern void refobj_incref(refobj_t *ref);

//decrease cnt_ref of ref
extern void refobj_decref(refobj_t *ref);


//check whether all refobjs are deallocated correctly
extern void refobj_check_dealloc();

#endif
```

```c
// refobj.c
//
#include "refobj.h"
#include "common.h"
#include <stdlib.h>

//statistics
static refstat_t stat;

//forward definitions
static void default_addref(refobj_t *ref);
static void default_release(refobj_t *ref);

//allocate container and init ref obj
void *refobj_alloc(tag_t tag, size_t size) {
    char *obj = calloc(1, size);     //allocate memory
    refobj_t *ref = (refobj_t*)obj;
    ref->tag      = tag;             //tag of the object
    ref->cnt_ref  = 1;               //reference count is set to 1
    ref->addref   = default_addref;  //default addref
    ref->release  = default_release; //default release
    stat.cnt_obj[ref->tag]++;        //update stat
    stat.cnt_ref[ref->tag]++;
    return obj;
}
```

```c
//free the container of ref
void refobj_free(refobj_t *ref) {
    ON_FALSE_EXIT(ref->cnt_ref == 0,
        strmsg("destroying live refobj, cnt_ref: %d", ref->cnt_ref));
    stat.cnt_obj[ref->tag]--;          //update stat
    free(ref);                         //deallocate memory
}

//increase cnt_ref of ref
void refobj_incref(refobj_t *ref) {
    ON_FALSE_EXIT(ref->cnt_ref > 0,
        strmsg("nonpositive cnt_ref: %d", ref->cnt_ref));
    ref->cnt_ref++;                    //increase reference count
    stat.cnt_ref[ref->tag]++;          //update stat
}

//decrease cnt_ref of ref
void refobj_decref(refobj_t *ref) {
    ON_FALSE_EXIT(ref->cnt_ref > 0,
        strmsg("nonpositive cnt_ref: %d", ref->cnt_ref));
    ref->cnt_ref--;                    //decrease reference count
    stat.cnt_ref[ref->tag]--;          //update stat
}
```

```
//check whether all refobjs are deallocated correctly
void refobj_check_dealloc() {
    int i, allzero = 1;

    for(i = 0; i < OBJ_COUNT; i++) {
        allzero &= stat.cnt_ref[i] == 0;
        allzero &= stat.cnt_obj[i] == 0;
    }

    if(!allzero) {
        fprintf(stderr, "cnt_ref = [ ");
        for(i = 0; i < OBJ_COUNT; i++)
            fprintf(stderr, "%d, ", stat.cnt_ref[i]);
        fprintf(stderr, "]\n");

        fprintf(stderr, "cnt_obj = [ ");
        for(i = 0; i < OBJ_COUNT; i++)
            fprintf(stderr, "%d, ", stat.cnt_obj[i]);
        fprintf(stderr, "]\n");

        ON_FALSE_EXIT(0, "error: deallocating refobjs");
    }
}
```

```c
//default addref for reference counting
static void default_addref(refobj_t *ref) {
    refobj_incref(ref);          //increase reference count
}

//default release for reference counting
static void default_release(refobj_t *ref) {
    refobj_decref(ref);          //decrease the reference count
    if(ref->cnt_ref == 0)        //free the object if the count is 0
        refobj_free(ref);
}
```

Managing Reference Counts

- General rules
  - Call addref for every copy of the pointer
  - Call release for every destruction of the pointer
  - Redundant addref/release can be canceled

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

```
void GetObject([out] IUnknown **ppUnk);
void UseObject([in] IUnknown *pUnk);
void GetAndUse(/* [out] */ IUnknown ** ppUnkOut) {
  IUnknown *pUnk1 = 0, *pUnk2 = 0;
  *ppUnkOut = 0;                                          // R3

// get pointers to one (or two) objects
  GetObject(&pUnk1);                                      // A2
  GetObject(&pUnk2);                                      // A2

// set pUnk2 to point to first object
  if (pUnk2)  pUnk2->Release();                           // R1
  if (pUnk2 = pUnk1) pUnk2->AddRef();                     // A1

// pass pUnk2 to some other function
  UseObject(pUnk2);                                       // S1

// return pUnk2 to caller using ppUnkOut parameter
  if (*ppUnkOut = pUnk2)  (*ppUnkOut)->AddRef();          // A2

// falling out of scope so clean up
  if (pUnk1) pUnk1->Release();                            // R2
  if (pUnk2) pUnk2->Release();                            // R2
}
```

Reference Counting Example 1

- Rat for rational numbers
  - struct rat
    - has refobj_t type ref field for reference counting
    - has function pointers for its operations
    - has num and den fields
- Extern functions for arithmetic operations on rational numbers

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

typedef struct rat {
    refobj_t ref;   //ref is at the beginning of rat
                    //they have the same address
    int  ( *get_num )(struct rat* a/*in*/);
    int  ( *get_den )(struct rat* a/*in*/);
    void ( *print   )(struct rat* a/*in*/);
    int num;
    int den;
} rat_t;

//extern: make it visible to other files
extern rat_t *rat_make(int num, int den);
extern rat_t *rat_add(rat_t* a/*in*/, rat_t* b/*in*/);
extern rat_t *rat_sub(rat_t* a/*in*/, rat_t* b/*in*/);
extern rat_t *rat_mul(rat_t* a/*in*/, rat_t* b/*in*/);
extern rat_t *rat_div(rat_t* a/*in*/, rat_t* b/*in*/);

#endif
```

```c
//
// rat.c
//
#include "rat.h"        //include the declarations of rat.h
#include <stdio.h>

static int sign(int a) { return a < 0 ? -1 : 1; }

static int iabs(int a) { return a < 0 ? -a : a; }

static int gcd(int a, int b) {
    if(a == b)
        return a;
    else if (a > b)
        return gcd(a - b, b);
    else
        return gcd(b - a, a);
}
```

```c
static int get_num(rat_t* a) { return a->num; }

static int get_den(rat_t* a) { return a->den; }

static void print(rat_t* a)  {
    printf("[%d / %d]", a->num, a->den);
}

rat_t* rat_make(int num, int den) {
    rat_t* ret = refobj_alloc(OBJ_RAT, sizeof(rat_t));

    ret->get_num = get_num;  //copy the function pointers
    ret->get_den = get_den;
    ret->print   = print;

    int s = sign(num) * sign(den);
    int g = gcd(iabs(num), iabs(den)); //reduce the fraction
    ret->num = num / g * s;
    ret->den = den / g;

    return ret;
}
```

```
rat_t* rat_add(rat_t* a/*in*/, rat_t* b/*in*/) {
    int num = a->num * b->den + b->num * a->den;
    int den = a->den * b->den;
    return rat_make(num, den);   //return: rat_make will addref
}

rat_t* rat_sub(rat_t* a/*in*/, rat_t* b/*in*/) {
    int num = a->num * b->den - b->num * a->den;
    int den = a->den * b->den;
    return rat_make(num, den);
}

rat_t* rat_mul(rat_t* a/*in*/, rat_t* b/*in*/) {
    int num = a->num * b->num;
    int den = a->den * b->den;
    return rat_make(num, den);
}

rat_t* rat_div(rat_t* a/*in*/, rat_t* b/*in*/) {
    int num = a->num * b->den;
    int den = a->den * b->num;
    return rat_make(num, den);
}
```

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

int main() {
    rat_t* a = rat_make(2, 3);
    rat_t* b = rat_make(1, 2);

    printf("%d / %d\n", a->get_num(a), a->get_den(a));

    rat_t* c = rat_add(a, b);
    c->print(c);
    c->ref.release(&c->ref);

    c = rat_sub(a, b);
    c->print(c);
    c->ref.release(&c->ref);

    c = rat_mul(a, b);
    c->print(c);
    c->ref.release(&c->ref);
```

```
    c = rat_div(a, b);
    c->print(c);
    c->ref.release(&c->ref);
    printf("\n");

    a->ref.release(&a->ref);
    //b->ref.release(&b->ref); //intentionally commented

    refobj_check_dealloc();
}
```

```
> a.exe
2 / 3
[7 / 6][1 / 6][1 / 3][4 / 3]
cnt_ref = [ 1, ]
cnt_obj = [ 1, ]
error: deallocating refobjs in file: refobj.c,
       function: refobj_check_dealloc, line: 72
```

Reference Counting Exercise

- Expr for a parse tree
  - struct expr
    - has `refobj_t` type `ref` field for reference counting
    - has function pointers for its operations
  - struct expr_num
    - First part is the same as struct expr
    - has `rat_t *n`
  - struct expr_opr
    - First part is the same as struct expr
    - has `struct expr *a, *b`
- Extern functions for arithmetic operations

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

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

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

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

    struct expr *a;     //operand 1
    struct expr *b;     //operand 2
} expr_opr_t;

extern expr_t *expr_make_num(rat_t *n);
extern expr_t *expr_make_add(expr_t *a, expr_t *b);
extern expr_t *expr_make_sub(expr_t *a, expr_t *b);
extern expr_t *expr_make_mul(expr_t *a, expr_t *b);
extern expr_t *expr_make_div(expr_t *a, expr_t *b);

#endif
```

```
//
// expr.c
//
…

static rat_t *eval_num(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));

    //TODO: return expr->n
}
static void print_num(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 rat_t *eval_num(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));

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

```
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);
    }
}
```

```
expr_t *expr_make_num(rat_t *n) {
    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

    return (expr_t*)expr;
}
```

```
expr_t *expr_make_num(rat_t *n) {
    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);

    return (expr_t*)expr;
}
```

```
static rat_t *eval_opr(expr_t *self,
                       rat_t *(*opr)(rat_t *a/*in*/, rat_t *b/*in*/)) {
    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));
    rat_t *a = expr->a->eval(expr->a);
    rat_t *b = expr->b->eval(expr->b);
    rat_t *c = opr(a, b);

    //TODO: return c
}
```

```
static rat_t *eval_opr(expr_t *self,
                       rat_t *(*opr)(rat_t *a/*in*/, rat_t *b/*in*/)) {
    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));
    rat_t *a = expr->a->eval(expr->a);
    rat_t *b = expr->b->eval(expr->b);
    rat_t *c = opr(a, b);

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

```
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);
    }
}
```

```
static expr_opr_t *make_opr(expr_t *a, expr_t *b) {
    expr_opr_t* expr = refobj_alloc(OBJ_EXPR_OPR, sizeof(expr_opr_t));
    expr->ref.release = release_opr;

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

    return expr;
}
```

```
static expr_opr_t *make_opr(expr_t *a, expr_t *b) {
    expr_opr_t* expr = refobj_alloc(OBJ_EXPR_OPR, sizeof(expr_opr_t));
    expr->ref.release = release_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);

    return expr;
}
```

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

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

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

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

    rat_t *d = z->eval(z);
    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.exe
[1 / 2] * [1 / 3] + [1 / 3] * [1 / 5]
[7 / 30]
```

Assignment 10

- Download hwa.zip and implement the TODOs
  - Implement complex add, sub, mul, and div
  - Implement complex in rectangular form
  - Implement complex in polar form
  - Implement vector 3
  - Upload the files in a single zip file
- Due date: 6/11/2024

Assignment 10

- Expected result

```
$ ./app
--test complex-------------------
1 + 1 i
1 e^i 0.785398
1 e^i 1.5708
1 + 2 i
--test vector--------------------
[ 1 + 0 i, 1 + 0 i, 1 + 0 i ]
[ 0 + 1 i, 0 + 1 i, 0 + 1 i ]
[ 1 + 1 i, 1 + 1 i, 1 + 1 i ]
[ 2 e^i 1.5708, 2 e^i 1.5708, 2 e^i 1.5708 ]
-6 + 6 i
```

Final Exam

- Date
  - 6/13/2024 (Thursday)
    9:00am ~ 11:30am
- Scope
  - Comprehensive, more weights are on C
- Format
  - Similar format as Midterm exams

Thank you for your attention during the semester!

Any questions or comments?

- Please submit your Course Evaluation at https://p22.courseval.net/etw/ets/et.asp?nxappid=SU2&nxmid=start

SUNY Korea