CSE216 Programming Abstractions

Introduction to C

YoungMin Kwon

Hello.c

```c
//
// Hello.c
//
#include <stdio.h>        //header files have macros and declarations
                          //stdio.h has the declaration for printf

int main() {              //process begins from main
                          //main function returns the exit status
                          //main function can take parameters
                          //     (will be discussed later)

    printf("Hello World!\n"); //calling printf with the string argument
                          //in stdio.h: int printf(const char *format, ...);
                          //similar to System.out.format(...) in java

    return 0;             //returning 0 from main means normal termination
}
```

```c
// Scanf.c
//
#include <stdio.h>

int main() {
    int i;      //int type variable
    float f;    //float type variable

    printf("Enter an integer: ");
    scanf("%d", &i);    //"%d" in scanf means reading an int
                        //&i: passing the address of i so that
                        //    scanf can change the caller's variable
                        //scanf is declared in stdio.h
                        //int scanf(const char *format, ...);

    printf("Enter a float:    ");
    scanf("%f", &f);    //"%f" in scanf means reading a float
                        //&f: passing the address of f so that
                        //    scanf can change the caller's variable

    printf("int: %d, float: %f\n", i, f);
                        //%d, %f in printf mean printing an int
                        //and printing a float type variable
    return 0;
}
```

```
> a.exe
Enter an integer: 1
Enter a float:    3.14
int: 1, float: 3.140000
```

Procedures

- Procedures are a main abstraction mechanism
  - C is a procedural programming language
- Similar to java's static methods


```
int factorial(int n) {
    if(n <= 1)
        return 1;
    return n * factorial (n - 1);
}
```

- Call by value parameter passing mode
  - Pass addresses to change the caller's variables

Expressions and Statements

- Expressions
  - Evaluate to a value
  - E.g. arithmetic, boolean expressions
  - E.g. 1 + 2, x > y, arr[99]

- Statements
  - Represent actions
  - E.g. assignment, loop, conditional statements
  - E.g. x = 3;, while(true)…, if(false)…

```c
// factorial.c
//
#include <stdio.h>

//forward declaration: declarations must appear before their use
int factorial(int n);

int main() {
    int i;
    printf("Enter a number: ");
    scanf("%d", &i);
    printf("factorial %d = %d\n", i, factorial(i));
    return 0;
}

int factorial(int n) {
    if(n <= 1)      //n, 1   : arithmetic expression
                    //n <= 1 : boolean expression
                    //if ... : conditional statement
        return 1;   //1      : arithmetic expression
                    //return ... : return statement
    return n * factorial (n - 1);
                    //n, n - 1, factorial(n - 1): arithmetic expr
                    //n * factorial(n - 1)      : arithmetic expr
                    //return ...                : return statement
}
```

Call By Value

- Terms
  - Formal parameters: parameter names in the declaration of a subroutine
  - Actual parameters (arguments): expressions that are passed to a subroutine

```
void square(int x) {
    x = x * x;
}

void foo() {
    square(1 + 2);
}
```

Call by Value

- First, the arguments to a function are fully evaluated before invoking the function (eager evaluation)

- Call by value: copies of the arguments are passed

```
void square(int x) {
    x = x * x;
}

void foo() {
    int y = 1 + 2;
    square(y);
}
```

```
void remove(Object o) {
    o = null;
}

void foo() {
    Object o = new Object();
    remove(o);
}
```

Call by Value

- To change caller's variables
  - Explicitly pass the addresses of variables (pointers in C)

```
void square(int* x) {
    int y = *x;
    *x = y * y;
}

void foo() {
    int y = 1 + 2;
    square( &y );
}
```

```
int main() {
    int i;
    printf("Enter a number: ");
    scanf("%d", &i);
    return 0;
}
```

C Data Types

- Basic Data Types
  - char, short, int, long, long long, …
  - float, double, long double, …
- Composite Types
  - struct, union

```
struct pair {
    char c;
    int  i;
};
```

```
union pair {
    char c;
    int  i;
};
```

C Array Data Types

- Array Types
  - Array of other types.
  - E.g. `int arr[10];`
  - Array name is a constant pointing to the address of the first element of the array.

```
#include <stdio.h>
int main() {
    int arr[10];
    int *p = arr;
    printf("%p, %p, %p\n", arr, p, &arr[0]);
}

> a.exe
0061FEA4, 0061FEA4, 0061FEA4
```

C Array Data Types

- Array name is a constant pointer

```c
#include <stdio.h>

void update(int *a) {
    *a = 0;
    *(a + 1) = 1;
    *(a + 2) = 2;
    a[3] = 3;
}

void array() {
    int arr[10] = {0,};
    int *p = arr;

    printf("before: %d, %d, %d, %d\n", arr[0], arr[1], p[2], p[3]);
    update(arr);
    printf("after : %d, %d, %d, %d\n", arr[0], arr[1], p[2], p[3]);
}
```

```
> a.exe
before: 0, 0, 0, 0
after : 0, 1, 2, 3
```

Pointer Types

- Pointer Types
  - Variables holding the address of other variables:
  - & operator returns the address of a variable
  - * operator returns the value of the address

```
void pointer() {
    int i = 0, j;
    int *p;      //p is a pointer type variable
    p = &i;      //p has the address of i
    j = *p;      //j has the value of p
    *p = j + 1; //p's value is updated to j + 1
}
```

String

- String is a null ('\0') terminated char array

```
#include <stdio.h>

void print_str(char *str) { //str points to the beginning
                            //  of the char array
    for(; *str; str++)      //*str:  until the null char
                            //str++: increate the pointer by 1
        printf("%c", *str); //*str:  contents of the str
}

int main() {
    //string is a null (0 or '\0') terminated char array
    char str[] = { 'H', 'e', 'l', 'l', 'o', '\n', 0 };

    char *pstr = str;   //str is a constant pointer
                        //pstr is a pointer
    print_str(str);
    print_str(pstr);
}
```

```
> a.exe
Hello
Hello
```

Some String Functions

```
void test_str() {
    char str[100] = "Hello";
    printf("str: %s\n", str);

    printf("len: %d\n", strlen(str));//length of string

    strcat(str, " world");          //concatenate strings
    printf("str: %s\n", str);

    printf("cmp: %d, %d, %d\n",     //compare strings
        strcmp(str, "Hello world"),
        strcmp(str, "AAA"),
        strcmp(str, "ZZZ"));

    printf("strchr: %s\n", strchr(str, ' ')); //find char in a string
    printf("strstr: %s\n", strstr(str, "lo"));//find substring in a string
}
```

```
> a.exe
str: Hello
len: 5
str: Hello world
cmp: 0, 1, -1
strchr:  world
strstr: lo world
```

C Pointer Types for struct

- Pointer to struct and union types
  - The address of a composite type is the address of the first field
  - -> operator returns the field's value from the address of a composite type

```
struct rat {
    int num, den;
};
int main() {
    struct rat r = {.num = 1, .den = 2};
    struct rat *p = &r;

    printf("r = %d/%d\n", (*p).num //equiv to p->num
                        , p->den); //equiv to (*p).den
    return 0;
}
```

```c
struct pair {
    char c;
    int  i;
};

void update(struct pair *ps) {
    printf("before- %c, %d, %p\n",
                ps->c,  //c field of the address pointed by ps
                ps->i,  //i field of the address pointed by ps
                ps);    //ps has an address of a struct pair type variable
    ps->c = 'z';
    ps->i = 2;
    printf("after - %c, %d, %p\n", ps->c, ps->i, ps);
}

int main() {
    struct pair s = { .c = 'a', .i = 1};

    printf("before: %c, %d, %p\n", s.c, s.i, &s);
    update(&s);
    printf("after:  %c, %d, %p\n", s.c, s.i, &s);

    return 0;
}
```

```
>a.exe
before: a, 1, 0061FEC8
before- a, 1, 0061FEC8
after - z, 2, 0061FEC8
after:  z, 2, 0061FEC8
```

Function Pointers

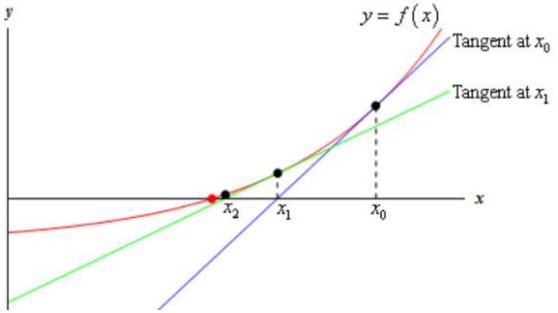
- Function names are addresses of the beginning of the function's code

```
int inc(int a) {
    return a + 1;
}
int add(int a, int b) {
    return a + b;
}
int main() {
    int (*pinc)(int a) = inc;          //pinc is a pointer to function inc
    int (*padd)(int a, int b) = add; //padd is a pointer to function add

    printf("inc: %d, %p, %p\n", pinc(1),    inc, pinc);
    printf("add: %d, %p, %p\n", padd(1, 2), add, padd);

    return 0;
}
```

```
> a.exe
inc: 2, 004015C0, 004015C0
add: 3, 004015CB, 004015CB
```

```c
/* newton.c
   to compile: gcc newton.c -lm
*/
#include <math.h>
#include <stdio.h>
#define EPS (1e-8)

double dfdx(double (*f)(double x), double x) {
    return (f(x+EPS) - f(x)) / EPS;
}
double next(double (*f)(double x), double x) {
    return x - f(x) / dfdx(f, x);
}
double solve(double (*f)(double x), double x0) {
    for(int i = 0 ; i < 1000; i++) {
        double x = next(f, x0);
        if(fabs(x - x0) < EPS)
            return x;
        x0 = x;
    }
    return x0;
}
int main() {
    printf("pi = %lf\n", solve(sin, 3));
}
```

to link with math library



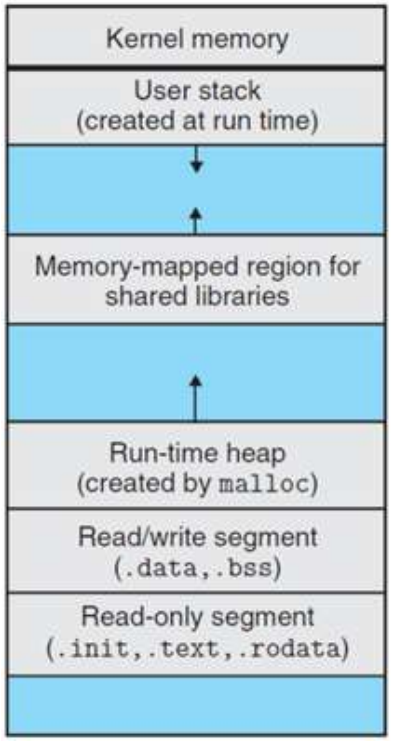
Typedef

- One can define a custom type using typedef
  - After typedef keyword, define a type like defining a variable

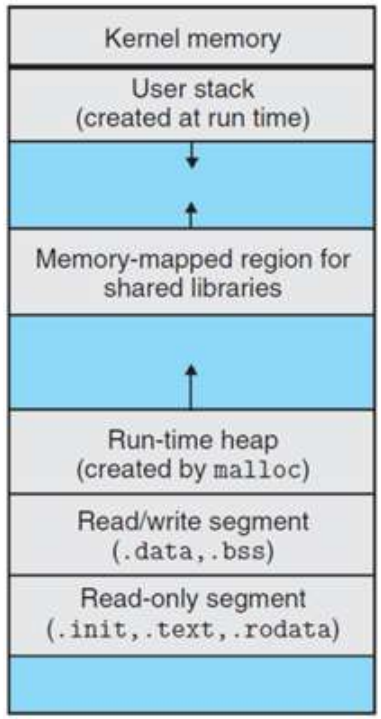
```
typedef int my_int;                     //int type
typedef int *int_ptr;                   //int pointer type

typedef double vect[3];                 //double array type

struct dual {
    int a, b;
};
typedef struct dual dual_t;             //struct type
typedef struct dual *dual_ptr_t;        //struct pointer type
typedef struct triple {                 //another struct type
    int a, b, c;
} triple_t;

typedef int (*binop)(int a, int b); //function pointer type
```

Variables

- Local variables and parameters
  - Allocated in the user stack
- Static and extern variables
  - Allocated in .data, .bss
- Initialized read only data (e.g. "hello")
  - Allocated in .rodata
- Instruction code
  - Allocated in .text
- Dynamic memory allocation
  - Heap



Variables



- Where are these variables?

```
extern int e;           //e: extern variable
static int s = 10;      //s: static variable
void square(int x, int *p) {
                        //x, p: parameters,
                        //square: points to the
                        //  beginning of this code
    int y = x * x;      //y: local variable
    *p = y;             //the value of p in heap is updated
}

int main() {            //main points to the beginning of this code
    int *p;             //p: local variable
    p = malloc(sizeof(int)); //p is an address in heap
    square(3, p);       //P: passing the address in heap
    printf("%d", *p);   //"%d": read only data
    free(p);
}
```

Variable Scope

- Variable scope
  - Extern variables
    - Visible from other files
  - Static variables
    - Visible within the file or
    - Visible within the defining { … } block
  - Auto variables
    - Visible within the function or
    - Visible with the defining { … } block

Variable Lifetime

- Variable lifetime
  - Global variables (static, extern vars)
    - Valid through out the lifetime of the process
  - Auto variables (local, parameter vars)
    - Valid until the function returns
    - Valid while the control remains in the defining { … } block

Macros

- #define
  - #define macro_name (macro parameters…)

```
#define PI (3.141592)
#define ADD(a, b)   ((a) + (b))
#define MUL(a, b)   ((a) * (b))
#define MAX(a, b)   ((a) > (b) ? (a) : (b))

#define ON_FALSE_GOTO(exp, label, msg) {\
    if(!(exp)) {\
        char *str = (char*)msg;\
        if(str && str[0] != '\0')\
            fprintf(stderr, "%s in file: %s, function: %s, line: %d\n",\
                str, __FILE__, __FUNCTION__, __LINE__);\
        goto label;\
    }\
}
```

Macros

- Using #define
  - Substituting macro parameters
  - Similar to beta-reduction in lambda calculus

```
void macro_test() {
    printf("PI: %f\n", PI);

    int a = ADD(1, 2);
    printf("ADD(1, 2): %d\n", a);

    int b = MUL(2, 3);
    printf("MUL(2, 3): %d\n", b);

    int c = MAX(1, 2);
    printf("MAX(1, 2): %d\n", c);
}
```

```
> a.exe
PI: 3.141592
ADD(1, 2): 3
MUL(2, 3): 6
MAX(1, 2): 2
```

Macros

- Fallacies in using #define

```
#define ADD_(a, b)    a + b
#define MUL_(a, b)    a * b

void macro_errors() {
    int i = 0;
    int j = MAX(++i, 0);
    printf("MAX(++i, 0): %d\n", j);

    int a = MUL (2, ADD (3, 4));
    printf("MUL (2, ADD (3, 4)): %d\n", a);
    ON_FALSE_GOTO(a == 14, err, "unexpected");

    int b = MUL_(2, ADD_(3, 4));
    printf("MUL_(2, ADD_(3, 4)): %d\n", b);
    ON_FALSE_GOTO(b == 14, err, "unexpected");

err:
    return;
}
```

```
> a.exe
MAX(++i, 0): 2
MUL (2, ADD (3, 4)): 14
MUL_(2, ADD_(3, 4)): 10
unexpected in file: macro.c,
     function: macro_errors,
     line: 45
```

Multiple Source Code

```
/* rat.h
*/
#ifndef __RAT__ //to avoid multiple inclusion
#define __RAT__ //equivalent to #pragma once

typedef struct rat rat_t;
struct rat {
    int num, den;
    int   (*get_num)(rat_t a);
    int   (*get_den)(rat_t a);
    void  (*print)  (rat_t a);
    rat_t (*add)    (rat_t a, rat_t b);
    rat_t (*sub)    (rat_t a, rat_t b);
    rat_t (*mul)    (rat_t a, rat_t b);
    rat_t (*div)    (rat_t a, rat_t b);
};

//extern: make the symbol visible to other files
extern rat_t make_rat(int num, int den);

#endif
```

```c
/* rat.c
*/
#include <stdio.h>
#include "rat.h"                //to include declarations in rat.h

static int abs(int x) {         //static functions are invisible
    return x < 0 ? -x : x;      //from outside of this file
}
static int sign(int x) {
    return x >= 0 ? 1 : -1;
}
static int gcd(int a, int b) {
    return a > b ? gcd(a - b, b)
         :  a < b ? gcd(b - a, a)
         :  a
         ;
}
static int get_num(rat_t r) {
    return r.num;
}
static int get_den(rat_t r) {
    return r.den;
}
```

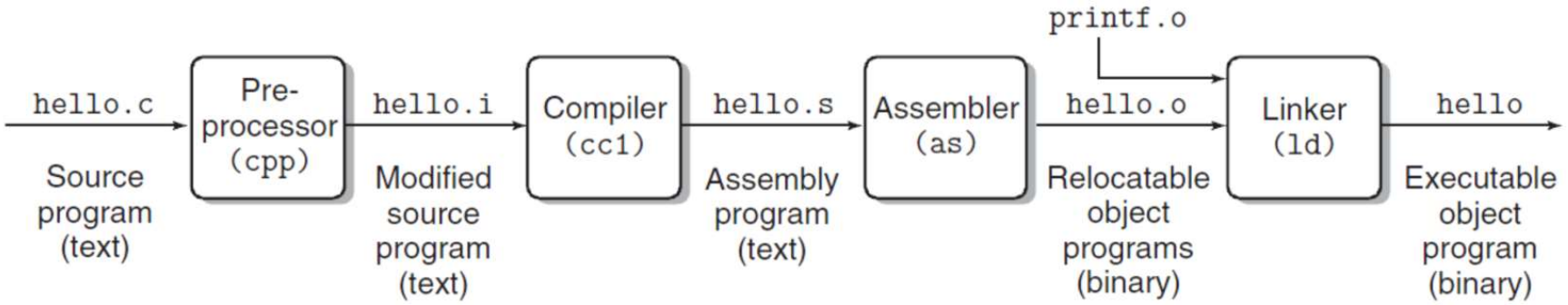
```c
static void print(rat_t a) {
    printf("%d / %d\n", a.num, a.den);
}
static rat_t add(rat_t a, rat_t b) {
    int num = a.num * b.den + b.num * a.den;
    int den = a.den * b.den;
    return make_rat(num, den);
}
static rat_t sub(rat_t a, rat_t b) {
    int num = a.num * b.den - b.num * a.den;
    int den = a.den * b.den;
    return make_rat(num, den);
}
static rat_t mul(rat_t a, rat_t b) {
    int num = a.num * b.num;
    int den = a.den * b.den;
    return make_rat(num, den);
}
static rat_t div(rat_t a, rat_t b) {
    int num = a.num * b.den;
    int den = a.den * b.num;
    return make_rat(num, den);
}
```

```
rat_t make_rat(int num, int den) {
    int s = sign(num * den);
    int n = abs(num);
    int d = abs(den);
    int g = gcd(n, d);
    rat_t rat = {
        .num = s*n/g, .den = d/g,
        //copy function pointers
        .get_num = get_num, .get_den = get_den, .print = print,
        .add = add, .sub = sub, .mul = mul, .div = div
    };
    return rat;
}
```

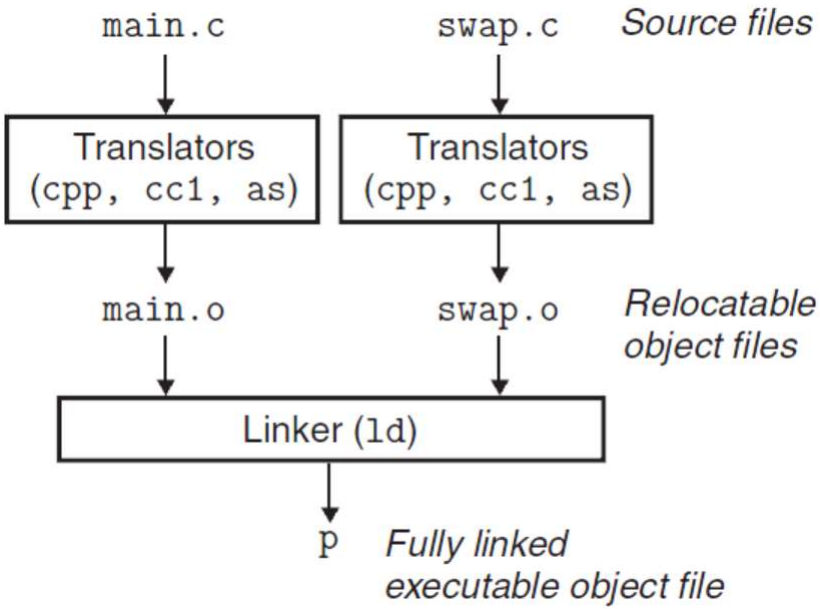
```c
/* rat_test.c
   to compile: gcc rat.c rat_test.c
*/
#include <stdio.h>
#include "rat.h"

int main() {
    rat_t rat = make_rat(1, 1);
    rat_t a = make_rat(2, 3);
    rat_t b = make_rat(1, 2);

    printf("%d, %d\n", a.get_num(a), a.get_den(a));
    rat.print(rat.add(a, b));
    rat.print(rat.sub(a, b));
    rat.print(rat.mul(a, b));
    rat.print(rat.div(a, b));
}
```

```
** output **
$ ./a.out
2, 3
7 / 6
1 / 6
1 / 3
4 / 3
```

Compilation Steps



Compile Multiple Source Files



How to Compile

- Compile all source files together

```
$ gcc rat.c rat_test.c

$ ./a.out
2, 3
7 / 6
1 / 6
2 / 6
4 / 3
```

How to Compile

- Compile each source file separately

```
$ gcc -c rat.c

$ gcc -c rat_test.c

$ ls rat*
rat.c        rat.h        rat.o        rat_test.c   rat_test.o

$ gcc rat.o rat_test.o

$ ./a.out
2, 3
7 / 6
1 / 6
1 / 3
4 / 3
```

How to Compile (Makefile)

```
TGT  = rat.exe
HSRC = rat.h
CSRC = rat.c rat_test.c
OBJS = rat.o rat_test.o

RM   = del  # rm in Linux,   del in Windows
TRUE = cd . # true in Linux, cd . in Windows

.SUFFIXES:          # reset all suffixes
.SUFFIXES: .c .o    # suffixes to consider

# convert .c to .o
.c.o:; gcc -c $< -o $@

$(TGT): $(HSRC) $(OBJS)
    gcc -o $@ $(OBJS)

clean:
    $(RM) *.o | $(TRUE)
```

Assignment 9

- Download hw9.zip and implement the TODOs
  - Make an array of words from a string
  - Sort the words
  - Find the frequency (the number of occurrences) of each word
  - Upload the files in a single zip file
- Due date: 6/4/2024

Assignment 9

- make_words:
  - Consecutive alphabets are a word
- sort_words:
  - Sort words by their dictionary order
  - Use strcmp
- freq_words:
  - After sorting words, the same words will be grouped
  - Count the number of occurrences of each unique word

```c
int main() {
    char *str =
        "Eeny, meeny, miny, moe,"
        "Catch a tiger by the toe."
        "If he hollers, let him go,"
        "Eeny, meeny, miny, moe.";

    char **words;                   //words
    word_freq_t *word_freqs;        //word frequencies
    int wc, uwc;                    //word count, unique word count

    //make words from str
    make_words(str, &words, &wc);
    print_words(words, wc);
    printf("word count: %d\n", word_count(str));

    //sort words
    sort_words(words, wc);
    print_words(words, wc);
    printf("word count: %d\n", word_count(str));

    //count the frequencies of words
    make_word_freqs(words, wc, &word_freqs, &uwc);
    print_word_freqs(word_freqs, uwc);
    printf("unique word count: %d\n", unique_word_count(words, wc));
```

…

Assignment 9: expected result

```
--print words----------
Eeny
meeny
miny
moe
Catch
a
tiger
by
the
toe
If
he
hollers
let
him
go
Eeny
meeny
miny
moe
word count: 20
```

```
--print words----------
Catch
Eeny
Eeny
If
a
by
go
he
him
hollers
let
meeny
meeny
miny
miny
moe
moe
the
tiger
toe
word count: 20
```

```
--print freqs----------
Catch: 1
Eeny: 2
If: 1
a: 1
by: 1
go: 1
he: 1
him: 1
hollers: 1
let: 1
meeny: 2
miny: 2
moe: 2
the: 1
tiger: 1
toe: 1
unique word count: 16
```