C Programming
Lecture 11 : Structure

Data Type

- struct
- union
- enum
- typedef

structure

- A collection of one or more variables possibly of different types, grouped together under a single name.

ex) employee payroll record
name, address, salary, …

struct

- Create a new data type that has a structure
- Ex) point (x,y) coordinate

```
struct point {
  double x;
  double y;
};

struct point pt;

1) struct point pt = {320.0,200.0};
2) pt.x=320.0; pt.y=200.0;
```

struct example

- using in function : distance from origin (0,0)

```
double dist_from_origin(struct point pt1)
{
   return sqrt(p1.x*p1.x + p2.x*p2.x);
}
```

- point return

```
struct point makepoint(double x, double y)
{
   struct point temp;
   temp.x = x;
   temp.y = y;
   return temp;
}
```

struct array and pointer

```
struct point pointarray[100];

pointarray[0].x=10;
pointarray[0].y=20;



struct point* pointPtr;
struct point pt = makepoint(20.0,30.0);
pointPtr = &pt;

pointPtr->x = 30.0;  // same as (*pointPtr).x = 30.0;
pointPtr->y = 50.0;  // same as (*pointPtr).y = 50.0;
```

struct and class

- **class**
  - Extend struct to the concept of abstract data type(ADT)
- ADT = attributes + methods

↓

OOP (Object Oriented Programming)

union

- A user-defined data type that may hold objects of different types and sizes
- At any given time, contains only one object from its members
- Variable Size : the largest member in its member-list.

```
union u_tag {
   int ival;
   float fval;
   char cval;
} u;

if (utype == INT) u.ival=3;
else if (utype == FLOAT) u.fval=3.5;
else if (utype == CHAR) u.cval='c';
```

union example

```
#include <stdio.h>

union NumericType
{
    int         iValue;
    long        lValue;
    double      dValue;
};

int main()
{
    union NumericType Values = { 10 }; // first member(iValue)=10
    printf("%d\n", Values.iValue);
    Values.dValue = 3.1416;
    printf("%f\n", Values.dValue);
    printf("%d\n",sizeof(Values));
    return 0;
}
```

Output

```
10
3.141600
8
```

enum

- enumerate? , enumerator?
- A user-defined type
- defining a set of named constants (integer)
  - Ordered list of integers
  - Default integer starts from 0
- Makes code easier to read
- Can use in expressions like ints

enum

```
enum days {mon, tue, wed, thu, fri, sat, sun};
// Same as:
// #define mon 0
// #define tue 1
// ...
// #define sun 6
```

```
enum days {mon=3, tue, wed, thu=9, fri, sat, sun};
// Same as:
// #define mon 3
// #define tue 4
// #define wed 5
// #define thu 9
// #define fri 10
// #define sat 11
// #define sun 12
```

enum example code

```
#include <stdio.h>

enum days {mon, tue, wed, thu, fri, sat, sun};

int main()
{
  enum days day;
  // Same as: int day;
  for(day = mon; day <= sun; day++) {
      if (day == wed) {
      printf("WEDNESDAY\n");
    } else {
      printf("day = %d\n", day);
    }
  }
}
```

typedef

- Creates new data type names

```
typedef int Length;
Length len, maxlen;


typedef char *Str;
Str p, *lineptr;
const int MAXLINES=100;

lineptr = new Str[MAXLINES];
```

typedef and struct

```
typedef struct p {
  double x;
  double y;
} Point;

Point x, y;
Point makepoint(double x, double y);
```

A new data type, **Point**, is created.

```
Point *z;
z=(Point*)malloc(sizeof(Point)); // dynamic allocation
z->x=3.5;
z->y=2.7;
```

Ex) complex number

```
#include <stdio.h>

typedef struct {
    float re;       // real number
    float im;       // imaginary number
} COMPLEX ;

int main()
{
    COMPLEX x, y, z, *p;
    x.re = 1.0;
    x.im = 2.0;
    y.re = 5.0;
    y.im = 10.0;
    p = &z;
    p->re = x.re + y.re;
    p->im = x.im + y.im;

    printf("x     = %3.1f + %3.1f i \n", x.re, x.im);
    printf("y     = %3.1f + %3.1f i \n", y.re, y.im);
    printf("x + y = %3.1f + %3.1f i \n", p->re, p->im);

    return 0;
}
```

```
output:
x     = 1.0 + 2.0 i
y     = 5.0 + 10.0 i
x + y = 6.0 + 12.0 i
```

Stack

- LIFO(Last In First Out)
  - Last data pushed is popped out first.
- Stack Operations (stack : s , d : data)
  - `Push(s,d)`
    - Insert data at top of stack
  - `Pop(s)`
    - Delete and return the data at top of stack
  - `IS_EMPTY(s)`
  - `IS_FULL(s)`

Implementation of Stack (using array and structure)

- Interface design
  - `Stack *mkStack();`
  - `void push(Stack *s, int item);`
  - `int pop(Stack *s);`
  - `int isEmpty(const Stack *s);`
  - `int isFull(const Stack *s);`

Stack.h

```
#define MAX_SIZE 10

struct _stack {
    int data[MAX_SIZE];
    int top;
};
typedef struct _stack Stack;

Stack *mkStack();
void push(Stack *s, int item);
int pop(Stack *s);
int isEmpty(const Stack *s);
int isFull(const Stack *s);

void error(const char *msg);
```

Stack.c

```
#include "stack.h"
#include <stdlib.h>

void error(const char *msg)
{
    printf("error : %s ",msg);
    printf("Program Finished.\n");
    exit(-1);
}

Stack * mkStack()
{
    Stack *s = (Stack *)
    malloc(sizeof(Stack));
    s->top = 0;
    return s;
}

void push(Stack *s, int item)
{
    if (isFull(s)) error("Stack Full");
    s->data[s->top++] = item;
}
```

```
int pop(Stack *s)
{
    if (isEmpty(s)) error("Stack Empty");
    return s->data[--s->top];
}

int isEmpty(const Stack *s)
{
    if (s->top == 0) return 1;
    else return 0;
}

int isFull(const Stack *s)
{
    if (s->top == MAX_SIZE) return 1;
    else return 0;
}
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "stack.h"

int main()
{
    int data;
    int n, i;
    Stack *s;

    printf("Get # of input :\n");
    printf("(between 1 and 10)");
    scanf("%d", &n);
    if (n <= 0 || n > 10)
        error("wrong number of stack size");

    s = mkStack();
```

```
    printf("%d integer input : ", n);
    for (i = 0; i < n; ++i) {
        scanf("%d", &data);
        push(s, data);
    }
    printf("printed in reverse order: ");
    while (! isEmpty(s)) {
        data = pop(s);
        printf("%d ", data);
    }
    printf("\n");

    return 0;
}
```

Implementation of Stack (using dynamic allocation)

- Interface design
  - `Stack *mkStack();`
  - `void push(Stack *s, int item);`
  - `int pop(Stack *s);`
  - `int isEmpty(const Stack *s);`
  - `int isFull(const Stack *s);`

Stack2.h

```
struct _stack {
    int *data;
    int top;
    int size;
};
typedef struct _stack Stack;

Stack *mkStack(int size);
void push(Stack *s, int item);
int pop(Stack *s);
int isEmpty(const Stack *s);
int isFull(const Stack *s);

void error(const char *msg);
```

Stack2.c

```
#include "stack2.h"
#include <stdlib.h>
#include <assert.h>

void error(const char *msg)
{
    printf("error: %s ",msg);
    printf("program exit\n");
    exit(-1);
}

Stack * mkStack(int size)
{
    Stack *s = (Stack *) malloc(sizeof(Stack));
    assert(size > 0);
    assert(s);
    s->size = size;
    s->data = (int *) malloc(sizeof(int)*size);
    assert(s->data);
    s->top = 0;
    return s;
}
```

```
void push(Stack *s, int item)
{
    if (isFull(s)) error("stack full.");
    s->data[s->top++] = item;
}

int pop(Stack *s)
{
    if (isEmpty(s)) error("stack empty.");
    return s->data[--s->top];
}

int isEmpty(const Stack *s)
{
    if (s->top == 0) return 1;
    else return 0;
}

int isFull(const Stack *s)
{
    if (s->top == MAX_SIZE) return 1;
    else return 0;
}
```

main2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "stack2.h"

int main()
{
    int data;
    int n, i;
    Stack *s;    // pointer that points to stack

    printf("how many inputs? ");
    scanf("%d", &n);
    if (n <= 0)
        error("wrong # of inputs.");

    s = mkStack(n);
```

```
    printf("%d integer input : ", n);
    for (i = 0; i < n; ++i) {
        scanf("%d", &data);
        push(s, data);
    }
    printf("printed in reverse order: ");
    while (! isEmpty(s)) {
        data = pop(s);
        printf("%d ", data);
    }
    printf("\n");

    return 0;
}
```