File I/O and ADTs in C

010.133
Digital Computer Concept and Practice
Spring 2013

Lecture 12

File I/O

File I/O

- In C, a stream is a source of input or a destination of output
  - A text stream is a sequence of lines and each line has zero or more characters and is terminated by '\n'

File I/O (contd.)

- <stdio.h> provides three standard streams
  - stdin (standard input): keyboard
  - stdout (standard output): screen
  - stderr (standard error): screen
- Printf gets input from stdin, and scanf sends output to stdout
- C also provides two simple I/O functions:
  - int getchar(void)
    - Reads the next character from stdin and returns it
  - int putchar(int c)
    - Prints character c into stdout

File I/O (contd.)

- Instead of standard input and output, a program can access a file for its input and output
  - A file pointer (stream) that points to a file (whose type is FILE)
    - FILE *fp;

File I/O Operations

- `FILE *fopen(const char *filename, const char *mode)`
  - Opens the filename file and returns a file pointer, or it returns NULL if the filename file does not exist
- The modes for text files are,
  - "r" open for reading (the file must exist)
  - "w" open for writing (discard previous contents if any)
  - "a" open for appending (the file is created if it does not exist)
  - "r+" open for reading and writing (the file must exist)
  - "w+" open for reading and writing (discard previous contents if any)
  - "a+" open for reading and appending (the file is created if it does not exist)

File I/O Operations (contd.)

- `int fclose(FILE *stream)`
  - Closes stream
- `int fgetc(FILE *stream)`
  - Returns the next character of stream, or EOF if end of file or error occurs
- `int fputc(int c, FILE *stream)`
  - Writes character c on stream
- `char *fgets(char *s, int n, FILE *stream)`
  - Reads at most the next n-1 characters into array s, stops if a newline is encountered (the newline is included in s)
  - Returns s, or NULL if end of file or error occurs

File I/O operations (contd.)

- **`int fputs(const char *s, FILE *stream)`**
  - Writes string s (which need not to contain a newline) on stream
- **`long ftell(FILE *stream)`**
  - Returns the current file position of stream
  - Data type long means a long integer
- **`int fseek(FILE *stream, long offset, int origin)`**
  - Sets the file position for stream
  - A subsequent read or write will access data at the new position.
  - origin may be SEEK_SET (beginning), SEEK_CUR (current position), or SEEK_END (end of file)
  - For a text stream, offset must be zero, or a value returned by ftell in which case origin must be SEEK_SET (fseek moves the position to the beginning or end of a text stream, or to a place that was visited previously)

File I/O Example

- Copies the contents of input.txt to output.txt
- EOF is an integer constant defined in <stdio.h> and it indicates 'end of file'

File I/O Example (contd.)

```
#include <stdio.h>

int main(void)
{
    FILE *fp1, *fp2;
    int c;

    fp1 = fopen("input.txt", "r");
    fp2 = fopen("output.txt", "w");

    while ((c = fgetc(fp1)) != EOF)
        fputc(c, fp2);

    fclose(fp1);
    fclose(fp2);
    return 0;
}
```

File I/O Example (contd.)

- Writes two strings into test.txt, sets the file position to the beginning, and reads the first line
- It will print “alphabet” when test.txt contains the following two lines:
  - alphabet
  - abcdefghijklmnopqrstuvwxyz

File I/O Example (contd.)

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char *pstr = "abcdefghijklmnopqrstuvwxyz";
    char buf[30];

    fp = fopen("test.txt", "w+");
    if(fp == NULL)
    {
        printf("file open error\n");
        return -1;
    }
         fputs("alphabet\n", fp);
    fputs(pstr, fp);

    fseek(fp, 0, SEEK_SET);
    fgets(buf, 30, fp);
    printf("%s", buf);

    fclose(fp);
    return 0;
}
```

Structures and Unions

Structures

- A structure is a collection of one or more variables, possibly of different types
- Struct person introduces a structure which contains three members, i.e., name, age, and sex
  - Person is called a structure tag
  - Once the structure tag is defined, we can declare structure variables
    - `struct person per1, per2;`

```
struct person {
    char name[10];
    int age;
    char sex;
};
```

Structure (contd.)

- A structure variable can be initialized by initializing its members
  - `struct person per1 = {"Lee", 20, 'f'};`

Operations on Structures

- A member of a structure variable is referred to by the following form:
  - structure-variable.member
- For example, we can print the members of per1 as follows:
  - `printf("%s, %d, %c", per1.name, per1.age, per1.sex);`
- A structure can be copied as a unit
  - For example,
    - `per2 = per1;`
  - copies per1.name to per2.name, per1.age to per2.age, and per1.sex to per2.sex

Operations on Structures (contd.)

- We can take the address of a structure with **&**
- **pp** is declared as a pointer to struct person, and the address of **per1** is assigned to **pp** (i.e., **pp** points to **per1**)
- ***pp** means **per1**, and **(*pp).name**, **(*pp).age**, **(*pp).sex** refer to **per1**'s members

```
struct person *pp;
      …
pp = &per1;
```

Operations on Structures (contd.)

- If **`pp`** is a pointer to a structure, an alternative notation to refer to a member is
  - pp->structure-member
- **`pp->name`**, **`pp->age`**, and **`pp->sex`** refer to the members of **`per1`**

Array of Structures

- When we need to maintain a list of persons, we can declare an array of structures:
  - `struct person per[10];`

Array of Structures (contd.)

```
int main(void)
{
    int i;
    struct person per[3]={
        {"Lee", 20, 'f'},
        {"Kim", 25, 'm'},
        {"Park", 22, 'f'}
    };

    for (i=0; i<3; i++)
    PrintPerson(&per[i]);
    return 0;
}
```

```
#include <stdio.h>

struct person {
    char name[10];
    int age;
    char sex;
};

void PrintPerson(struct person *pp)
{
    printf("name: %s, age: %d, sex: %c\n",
           pp->name, pp->age, pp->sex);
}
```

Structure Example

- The following program declares a structure for complex numbers and a function for complex number multiplication

```
#include <stdio.h>

struct complex {
     double x;
     double y;
};

struct complex cmult(struct complex a, struct complex b)
{
    struct complex c;
    c.x = a.x * b.x - a.y * b.y;
    c.y = a.x * b.y + a.y * b.x;
    return c;
}
```

Structure Example (contd.)

```
int main(void)
{
    struct complex a, b, c;

    printf("Enter a: ");
    scanf("%lf %lf", &a.x, &a.y);
    printf("Enter b: ");
    scanf("%lf %lf", &b.x, &b.y);
    c = cmult(a, b);
    printf("Mult: %f + %f i\n", c.x, c.y);
    return 0;
}
```

Typedef (revisited)

```
typedef complex {
    double x;
    double y;
} Complex;
...
Complex a, b, c;
```

Unions

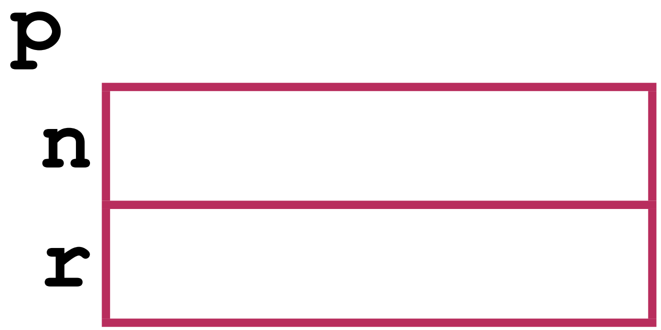
- The same syntax as structures
  - But, members share storage

```
typedef union foo {
    int n;
    float r;
} number;

int main(void)
{
    number m;
    m.n = 2345;
    printf("n: %10d  r: %16.10e\n", m.n, m.r);
    m.r = 2.345;
    printf("n: %10d  r: %16.10e\n", m.n, m.r);
    return 0;
}
```

Unions (contd.)

```
struct foo {
     int n;
     float r;
} p;
```




```
union foo {
     int n;
     float r;
} q;
```

calloc() and malloc()

- In the standard library (stdlib.h)
- To dynamically create storage for arrays, structures, and unions
- **`void* calloc( size_t n, size_t s )`**
  - Contiguous allocation
    - Allocates contiguous space in memory with a size of n × s bytes
  - The space is initialized with 0's
  - If successful, returns a pointer to the base of the space
  - Otherwise, returns NULL
- Typically "**`typedef unsigned int size_t;`**" in stdlib.h
- **`x = calloc( n, sizeof( int ));`**

calloc() and malloc() (contd.)

- `void* malloc( size_t s )`
  - Similar to calloc()
  - Allocates contiguous space in memory with a size of s bytes without initialization
  - `x = malloc( n * sizeof( int ));`

calloc() and malloc() (contd.)

- The programmer should explicitly return the space
  - **`free( ptr );`**
  - Makes the space in memory pointer by **`ptr`** to be deallocated
  - **`ptr`** must be the base address of space previously allocated

Abstract Data Types

Abstract Data Types

- An abstract data type is a collection of objects together with a collection of operations
- Lists, sets, and stacks are examples of abstract data types

Abstract Data Types (contd.)

- A list is simply a list of elements a_1, a_2, …, and a_k
  - Two operations on the list
    - Linsert(list, x) inserts element x into list
    - Lsearch(list, x) searches list for element x
  - We may define more operations such as Ldelete
- A set $\{a_1, a_2, \ldots, a_k\}$ along with some set operations such as union and intersection can be an abstract data type
- A stack is an important abstract data type which has many applications in programming

Linked Lists

- A linked list is a list of nodes linked by pointers
  - A node of a linked list requires a structure
  - It has two members element and next which is a pointer to struct node

```
struct node {
    int element;
    struct node *next;
};
```

Linked Lists (contd.)

- **`Linsert(list, x)`** gets list, which is a pointer to a linked list, and element x as its input
  - It inserts x at the front of the list
- **`Lsearch(list, x)`** gets list, which is a pointer to a linked list, and x as its input
  - If x is in the list, it returns the pointer of the node that contains x; NULL otherwise

Linked Lists (contd.)

- The next program creates a linked list by inserting elements, and searches it for an element x

```
#include <stdio.h>

struct node {
   int element;
   struct node *next;
};
```

Linked Lists (contd.)

```
struct node *Linsert(struct node *list, int x)
{
   struct node *pnew;

   pnew = malloc(sizeof(struct node));
   if (pnew == NULL) {
      printf("malloc error\n");
      return NULL;
   }
   pnew->element = x;
   pnew->next = list;

   return pnew;
}
```

Linked Lists (contd.)

```
struct node *Lsearch(struct node *list, int x)
{
   struct node *pn;

   for (pn = list; pn != NULL; pn = pn->next)
      if (pn->element == x) return pn;

   return NULL;
}
```

Linked Lists (contd.)

```
void PrintList(struct node *list)
{
   if (list == NULL) {
      printf("\n");
      return;
   }
   printf("%d ", list->element);
   PrintList(list->next);
}
```

Linked lists (contd.)

```
int main(void)
{
   struct node *list = NULL;
   int x;

   while (1) {
      printf("enter x (0 to terminate): ");
      scanf("%d", &x);
      if (x == 0) break;
      list = Linsert(list, x);
   }
   PrintList(list);
   printf("enter x: ");
   scanf("%d", &x);
   if (Lsearch(list, x) != NULL) printf("%d is in the list\n", x);
   else printf("%d is not in the list\n", x);

   return 0;
}
```

Stacks

- A stack is a list of elements with the restriction that elements are inserted and deleted only at one place called the top

Stacks (contd.)

- Fundamental stack operations are push and pop
  - `push(st, x)` gets st, which is a pointer to a stack, and element x as its input
    - It pushes x into the stack.
  - `pop(st)` gets st, which is a pointer to a stack, and it pops and returns the element at the top

Stacks (contd.)

```
#include <stdio.h>
#define MAX 100

struct stack {
   int starray[MAX];
   int top;
};
```

Stacks (contd.)

```
struct stack *create(void)
{
  struct stack *st;

  st = malloc(sizeof(struct stack));
  if (st == NULL) {
    printf("malloc error\n");
    return NULL;
  }
  st->top = 0;
  return st;
}
```

Stacks (contd.)

```
int is_empty(struct stack *st)
{
  return st->top == 0;
}

void push(struct stack *st, int x)
{
  if (st->top == MAX) {
    printf("stack is full\n");
    return;
  }
  st->starray[st->top++] = x;
}
```

Stacks (contd.)

```
int pop(struct stack *st)
{
   if (is_empty(st)) {
      printf("stack is empty\n");
      return;
   }

   return st->starray[--st->top];
}
```

Stacks (contd.)

```
int main(void)
{
   struct stack *st;
   int x;

   st = create();
   while (1) {
      printf("enter x (0 to terminate): ");
      scanf("%d", &x);
      if (x == 0) break;
      push(st, x);
   }
   while (!is_empty(st))
      printf("%d ", pop(st));
   printf("\n");

   return 0;
}
```

Programming style

- Programming style is a set of rules or guidelines for writing computer programs
- A good programming style is a subject matter, and thus it is difficult to define
- However, there are several elements common to many programming styles

Programming style (contd.)

- Indentation helps identify control flows
- Blank lines can divide a program into logical units
- Also spaces should be used properly to enhance readability of programs
- Variable names and function names: names should carry appropriate meanings

Programming style (contd.)

- Write clearly what you are doing in comments
- Modular design: use functions for independent tasks
- Write first in an easy-to-understand language, and then translate into a programming language