Data Structure and Algorithm

Class 3

Seongjin Lee

Updated: 2017-03-06
DSA_2017_03

insight@gnu.ac.kr
http://resourceful.github.io
Systems Research Lab.
GNU

Table of contents

Array

Array

an **array** is a set of pairs, **<index, value>**, such that each index that is defined has a value associated with it

- "a consecutive set of memory locations" in C
- logical order is the same as physical order

operations

- creating a new array
- retrieves a value
- stores a value
- insert a value into array - delete a value at the array

Array

A one-dimensional array in C is declared implicitly by appending brackets to the name of a variable

```
int list[5], *plist[5];
```

arrays start at index 0 in C

Array

consider the implementation of one-dimensional arrays

```
int list[5];
```

- allocates five consecutive memory locations
- each memory location is large enough to hold a single integer
- base address address of the first element

```
list = &list[0]
```

Array

Variable	Memory Address
`&list[0]`	base address = α
`&list[1]`	$\alpha+$ `sizeof(int)`
`&list[2]`	$\alpha + 2\cdot$ `sizeof(int)`
`&list[3]`	$\alpha + 3\cdot$ `sizeof(int)`
`&list[4]`	$\alpha + 4\cdot$ `sizeof(int)`

`&list[i]` in a C programs

- C interprets it as a pointer to an integer or its value

Array

```
int *list1; // pointer variable to an int
```

```
int *list2[5]; // list2 : pointer constant to an int,
                  // and five memory locations for holding
                  // integers are reserved
```

`(list2+i)` equals `&list2[i]`, and `*(list2+i)` equals `list2[i]`

- regardless of the type of the array list2

Array

consider the way C treats an array when it is a parameter to a function

- the parameter passing is done using call-by-value in C
- but array parameters have their values altered

Array: Example 2.1

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
int i;

void main(void) {
    for(i = 0; i < MAX_SIZE; i++) input[i] = i;
        answer = sum(input, MAX_SIZE);
    printf{"The sum is: %f\n", answer);
}

float sum(float list[], inst n) {
    int i;
    float tempsum = 0;
    for(i = 0; i < n; i++)
        tempsum += list[i];
    return tempsum;
}
```

Recap: On Pointer

Pointer Variable stores address

- & : Starting address of allocated variable
- * : Value stored on the address of the pointer variable

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

int main(){
    int nNum, *pNum;

    nNum = 10;
    pNum = &nNum;

    printf("nNum = %d, &nNum = %d \n", nNum, (int)&nNum);
    printf("pNum = %d, pNum = %d, &pNum = %d \n",
                          *pNum, (int)pNum, (int)&pNum);
}
```

Recap: On Pointer

Do Not!

- pointer variable is not referencing an address, so cannot store a value

```
int *ptr;
* ptr = 100;
```

- the data type must equal

```
double Pi = 3.14;
int *pPi = &Pi;
```

- cannot dereference a non-pointer variable

```
int num;
*num = 100;
```

Do!

- it is recommended to initialize a pointer value with NULL ('\0')
  See the example

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

int main(){
    int *pNum = NULL, Num=103;
    if (pNum == '\0')
        pNum = &Num;
    else
        *pNum = 100;
    printf("pNum %d", (int)*pNum);
    return 0;
}
```

Recap: On Pointer

```
#include <stdlib.h>

void *malloc(size_t size); // allocates size bytes of memory and returns
     a pointer to the allocated memory
void *free(void *ptr); // frees allocation that were created via the
    preceding allocation function
void *calloc(size_t count, size_t size); // contiguously allocates
    enough sapce for count objects that are size bytes of memory each
    and returns a pointer to the allocated memory. The allocated memory
     is filled with bytes of value zero.
void *realloc(void *ptr, size_t size); // change the size of the
    allocation pointed to by ptr to size, and returns ptr
```

Array: Ex 2.2, 1-dimensional array addressing

```
int one[] = {0, 1, 2, 3, 4};
```

write a function that prints out both the address of the i^{th} element of the array and the value found at this address

Array: Ex 2.2 One dimensional array accessed by address

```c
// codes/array_address.c
#include <stdio.h>

void print1(int *ptr,int rows) {
    int i;
    printf("Address\t\tContents\n");
    for(i=0;i<rows;i++)
        printf("%8u\t%5d\n", (unsigned int)ptr+i, *(ptr+i));
    printf("\n");
}

int main() {
  int one[] = {0, 1, 2, 3, 4};
  print1(one, 5);
  return 0;
}
```

One-dimensional array accessed by address

- address of i th element `ptr + i`
- obtain the value of the i th value `*(ptr + i)`

Array

```
Address  Contents
1518325632 0
1518325633 1
1518325634 2
1518325635 3
1518325636 4
```

One-dimensional array addressing

- the addresses increase by two on an Intel 386 machine
- Example shown is the result of Mac OS X on Intel Core i5 Machine

Structures and Unions

Structures and Unions: struct

struct

- structure or record
- the mechanism of grouping data
- permits the data to vary in type

collection of data items where

- each item is identified as to its type and name

Structures and Unions: struct

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

creating a variable

- whose name is person and
- has three fields
  1. a name that is a character array
  2. an integer value representing the age of the person
  3. a float value representing the salary of the individual

use of the . as the structure member operator

```
strcpy(person.name, "james");
person.age = 30;
person.salaray = 35000;
```

- select a particular member of the structure

Structures and Unions: struct

`typedef statement`

- create our own structure data type

type 1

```
typedef struct human_being {
    char name[10];
    int age;
    float salary;
};
```

type 2

```
typedef struct {
    char name[10];
    int age;
    float salary;
} human_being;
```

Structures and Unions: struct

human_being

- the name of the type defined by the structure definition

```
human_being person1, person2;

if(strcmp(person1.name, person2.name))
    printf("The two people do not have the same name\n");
else
    printf("The two people have the same name\n");
```

Structures and Unions: Assignment

assignment

- permits structure assignment in ANSI C

```
    person1 = person2;
```

- but, in most earlier versions of C assignment of structures is not permitted

```
    strcpy(person1.name,person2.name);
    person1.age=person2.age;
    person1.salary=person2.salary;
```

Structures and Unions: Equality or Inequality

equality or inequality

- cannot be directly checked
- Example function to check equality of struct

```
int humans_equal(human_being person1, human_being person2) {
    if(strcmp(person1.name,person2.name))
        return FALSE;
    if(person1.age!=person2.age)
        return FALSE;
    if(person1.salary!=person2.salary)
        return FALSE;
    return TRUE;
}
```

Structures and Unions: Embedding Structure

Embedding of a structure within a structure

```
typedef struct {
   int month;
   int day;
   int year;
} date;

typedef struct human_being {
   char name[10];
   int age;
   float salary;
   date dob; // embedded structure
};
```

Structures and Unions: Embedding Structure

Ex. A person boar on Feb 14 1992

```
person1.dob.month = 2;
person1.dob.day = 14;
person1.dob.year = 1992;
```

Structures and Unions: Unions

Unions

- similar to a structure, but
- the fields of a union must share their memory space
- only one field of the union is “active” at any given time

Structures and Unions: Unions

```
typedef struct sex_type {
    enum tag_field {female,male} sex;
    union {
        int children;
        int beard; } u;
};

typedef struct human_being {
    char name[10];
    int age;
    float salary;
    date dob;
    sex_type sex_info;
};

human_being person1,person2;
```

Structures and Unions: Unions

Assign values to `person1` and `person2`

```
persone1.sex_info.sex = male;
person1.sex_info.u.beard = FALSE; /* FALSE: 0 */
```

and

```
person2.sex_info.sex = female;
person2.sex_info.u.children = 4;
```

```
struct {
    int i, j; float a, b;
}
```

or

```
struct {
    int i; int j; float a; float b;
};
```

stored in the same way

- increasing address locations in the order specified in the structure definition

size of an object of a struct or union type

- the amount of storage necessary to represent the largest component

- one or more of its components is a pointer to itself
- usually require dynamic storage management routines to explicitly obtain and release memory

```
typedef struct list {
    char data;
    list *link;
};
```

the value of `link`

- address in memory of an instance of `list` or `null` pointer

Structures and Unions: Self-referential structures

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;
```

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;
```

attach these structures together

```
item1.link = &item2;
item2.link = &item3;
```

Sparse Matrices

The Sparse Matrix: representing matrix by using array

- m by n (m rows, n columns)
- use two-dimensional array
- space complexity
  $S(m,n) = m * n$

$$
\begin{array}{c c}
 & \begin{array}{ccc} col0 & col1 & col2 \end{array} \\
\begin{array}{c} row0 \\ row1 \\ row2 \\ row3 \\ row4 \end{array} &
\begin{bmatrix}
-27 & 3 & 4 \\
6 & 82 & -2 \\
109 & -64 & 11 \\
12 & 8 & 9 \\
48 & 27 & 47
\end{bmatrix}
\end{array}
$$

The Sparse Matrix

A[6,6]

$$
\begin{array}{c c}
 & \begin{array}{cccccc} col0 & col1 & col2 & col3 & col4 & col5 \end{array} \\
\begin{array}{c} row0 \\ row1 \\ row2 \\ row3 \\ row4 \\ row5 \end{array} &
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & 15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

The Sparse Matrix

common characteristics

- most elements are zero's
- inefficient memory utilization

solutions

- store only nonzero elements
- using the triple `<row,col,value>`
- must know
  - the number of rows
  - the number of columns
  - the number of non-zero elements

The Sparse Matrix

```
#define MAX_TERMS 101

typedef struct {
    int col;
    int row;
    int value;
} term;

term a[MAX_TERMS];
```

- a[0].row: the number of rows
- a[0].col: the number of columns
- a[0].value: the total number of non-zoros
- choose row-major order

The Sparse Matrix: As Triples

	row	col	value
a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

- space complexity (*variable space requirement)
  `S(m,n) = 3 * t` where
  `t` : the number of non-zero's
- independent of the size of rows and columns

The Sparse Matrix: Transpose

Transpose a matrix

- interchange rows and columns
- move `a[i][j]` to `a[j][i]`

	row	col	value
b[0]	6	6	8
b[1]	0	0	15
b[2]	0	4	91
b[3]	1	1	11
b[4]	2	1	3
b[5]	2	5	28
b[6]	3	0	22
b[7]	3	2	-6
b[8]	5	0	-15

```
algorithm BAD_TRANS
for each row i
    take element <i,j,value>;
    store it as element <j,i,value> of the transpose;
end;
```

The problem

The Sparse Matrix

```
algorithm BAD_TRANS
for each row i
    take element <i,j,value>;
    store it as element <j,i,value> of the transpose;
end;
```

The problem

- data movement

The Sparse Matrix

```
algorithm BAD_TRANS
for each row i
    take element <i,j,value>;
    store it as element <j,i,value> of the transpose;
end;
```

The problem

- data movement

```
algorithm TRANS
for all elements in column j
    place element <i,j,value> in element <j,i,value> end;
```

The problem

The Sparse Matrix

```
algorithm BAD_TRANS
for each row i
    take element <i,j,value>;
    store it as element <j,i,value> of the transpose;
end;
```

The problem

- data movement

```
algorithm TRANS
for all elements in column j
    place element <i,j,value> in element <j,i,value> end;
```

The problem

- unnecessary loop for each column

The Sparse Matrix: Transpose

```
void transpose(term a[], term b[]) {
    int n, i, j, currentb;
    n = a[0].value;
    b[0].row = a[0].col;
    b[0].col = a[0].row;
    b[0].value = n;
    if(n > 0){
        currentb = 1;
        for (i = 0; i < a[0].col; i++)
            for(j = 1; j <= n; j++)
                if(a[j].col == i) {
                  b[currentb].row = a[j].col;
                  b[currentb].col = a[j].row;
                  b[currentb].value = a[j].value;
                  cuurentb++;
            }
    }
}
```

Space and Time complexity?

The Sparse Matrix: Transpose

Complexity

- space: $3\times$ t
- time: O(cols$\times$ t)

create better algorithm by using a little more storage

- row_terms the number of element in each row
- starting_pos the starting point of each row

The Sparse Matrix: Better Transpose I

```
void fast_transpose(term a[], term b[]){
    int row_terms[MAX_COL];
    int starting_pos[MAX_COL];

    int i, j;

    int num_cols = b[0].row = a[0].col;
    int num_terms = b[0].value = a[0].value;

    b[0].col = a[0].row;

    if(num_terms > 0) {
        for(i = 0; i < num_cols; i++)
            row_terms[i] = 0;

        for(i = 1; i <= num_terms; i++)
            row_terms[a[i].col]++;

        starting_pos[0] = 1;
```

The Sparse Matrix: Better Transpose II

```
        for(i = 1; i < num_cols; i++)
            starting_pos[i] = starting_pos[i-1] + row_terms[i-1];

        for(i = 1; i <= num_terms; i++){
            j = starting_pos[a[i].col]++;
            b[j].row = a[i].col;
            b[j].col = a[i].row;
            b[j].value = a[i].value;
        }
    }
}
```

The Sparse Matrix: Better Transpose

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms=	2	1	2	2	0	1
starting_pos=	1	3	4	6	8	8

complexity

- space: $3\times$ t + extra
- time: O(cols + t)