CS1010E: Programming Methodology

Tutorial 09: Arrays & Pointers

03 Apr 2017 - 07 Apr 2017

1. Discussion Questions

(a) [Pointer] What is/are the output of the code fragments below?

i.
```c
int foo(int *a) {
  *a++;
  return *a-1;
}
int main() {
  int a = 1, b = foo(&a);
  printf("%d %d", a, b);
  return 0;
}
```

i. _____

ii.
```c
void foo(int *a, int *b) {
  int t = *a; *a = *b; *b = t;
}

int main() {
  int a = 1, b = 2; printf("%d %d ", a, b);
  foo(&a, &b);      printf("%d %d ", a, b);
  return 0;
}
```

ii. _____

iii.
```c
void foo(int *a, int *b) {
  int *t = a; a = b; b = t;
}
int main() {
  int a = 1, b = 2; printf("%d %d ", a, b);
  foo(&a, &b);      printf("%d %d ", a, b);
  return 0;
}
```

iii. _____

(b) [Equivalence] What is/are the output of the code fragments below?

i.
```c
int foo(int *a, int n) {
  int s=0, i;
  for(i=0; i<n; i++) s += a[i];
  return s;
}
```

```c
int main() {
  int a[] = {1, 2, 3, 4, 5};
  printf("%d %d", foo(a, 3), foo(a+2, 3));
  return 0;
}
```

i. _____

ii.
```c
void foo(int *a) {
  while(*a != 0) {
    *a = *(a+1);
    a++;
  }
}
int main() {
  int a[] = {1, 2, 3, 4, 5, 0}, i;
  foo(a);
  for(i=0; i<4; i++) printf("%d ", a[i]);
  return 0;
}
```

ii. _____

2. Program Analysis

(a) [Pointer Reasoning] What is/are the output of code fragments below?

i.
```c
int* bar(int *p) {
  p++;
  return p - 1;
}
int foo(int *p) {
  int *q;
  *p = *p + 5;
   p =  p + 1;
   q = bar(p);
  return *p;
}
int main() {
  int arr[] = {1,2,3,4,5}, ptr[] = {0,0,0,0,0}, i;
  for(i=0; i<4; i++)
    if(i%2) ptr[i] = foo(arr + i);
    else    ptr[i] = foo(&(arr[i]));

  for(i=0; i<5; i++)
    printf("%d %d ", arr[i], ptr[i]);
  return 0;
}
```

i. 6 2 7 3 8 4 9 5 5 0 (array as pointer)

ii.
```c
int **qtr, arr[] = {1,2,3,4,5}, i;
int *ptr[] = {&arr[4], &arr[3], &arr[2], &arr[1], &arr[0]};
qtr = ptr;
for(i=0; i<4; i++) { (*qtr)--; **qtr+=2; qtr++; }
for(i=0; i<5; i++)
  printf("%d ", arr[i]);
```

ii. 3 4 5 6 5 (array of pointers)

iii.
```
int M[5][5] = {
  { 1, 2, 3, 4, 5},
  { 6, 7, 8, 9,10},
  {11,12,13,14,15},
  {16,17,18,19,20},
  {21,22,23,24,25}
}, *ptr[] = {&M[0], &M[1], &M[2], &M[3], &M[4]};
int **qtr = ptr;
qtr  += 2;
*qtr += 3;
printf("%d", **qtr);
```

iii. 14 (array of pointers to matrix)

3. Designing a Solution

(a) [Array; Pointer] In this question, we will consider operations on **two (2)** arrays of different size. Furthermore, the size of each array is *unknown* to the function. However, it is an array of positive `integer` terminated by `-1` at the end of the array. An example will be an array `[2, 3, 5, 1, 2, -1]`.

i. **Zip Operation**

Zip algorithm takes as input **two (2)** random array. The arrays are given as pointers satisfying the condition above. The result of the execution is an array such that the first element of the result array is the sum of the first element of both the input arrays, the second is the sum of second element, and so on. However, since the two arrays can be of different size, you should only take up to the minimum size.

For instance, given the input `[1, 3, 5, 2, 3, -1]` and `[9, 2, 5, -1]`, the result should be `[10, 5, 10, -1]`. You can assume the result array has enough space to store all the values from both input arrays. Note that the function is a `void` function as the return value is given as a parameter.

```
void zip(int* arr1, int* arr2, int* res) {
  /* Zip arr1 and arr2 into res */
  while(*arr1 != -1 && *arr2 != -1) {
    *res = *arr1 + *arr2;
    *res++;
    *arr1++;
    *arr2++;
  }
  *res = -1;

}
```

(b) [Array; Pointer] In this question, we will consider operations on **two (2)** arrays of different size. Furthermore, the size of each array is *unknown* to the function. However, it is an array of positive `integer` terminated by `-1` at the end of the array. An example will be an array `[2, 3, 5, 1, 2, -1]`.

i. **Merge Operation**

Merge algorithm takes as input **two (2)** sorted array. The arrays are sorted from smallest to largest and are given as pointers satisfying the condition above. The result of the execution is an array that contains all the positive `integer` from both input arrays and sorted from smallest to largest.

For instance, given the input `[1, 4, 7, 9, -1]` and `[1, 3, 5, -1]`, the result should be `[1, 1, 3, 4, 5, 7, 9, -1]`. You can assume the result array has enough space to store all the values from both input arrays. Note that the function is a `void` function as the return value is given as a parameter.

```
void merge(int* arr1, int* arr2, int* res) {
  /* Merge arr1 and arr2 into res */
  while(*arr1 != -1 && *arr2 != -1) {
    if(*arr1 <= *arr2) {
      *res = *arr1;
      arr1++; res++;
    } else {
      *res = *arr2;
      arr2++; res++;
    }
  }
  while(*arr1 != -1) {
    *res = *arr1;
    arr1++; res++;
  }
  while(*arr2 != -1) {
    *res = *arr2;
    arr2++; res++;
  }
  *res = -1;
}
```

4. Challenge

(a) [Problem Solving; Past Question (*from 14/15 Sem2 Take Home Lab 4*)] A floor plan is a drawing that consists of room dividers. We consider a room to be an *enclosed space* if it is surrounded completely by dividers. The floor plan will consist of a series of *diagonal walls* [either / or \].

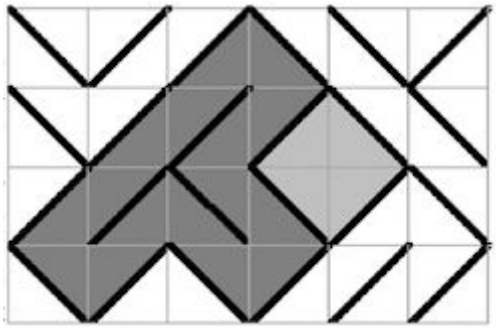

Diagram 1: An example floor plan with diagonal walls and two enclosed spaces.

You are asked to write a program to count how many enclosed space are there in a given floor plan. For instance, the floor plan in Diagram 1 shows **two (2)** enclosed spaces given as darker shades of grey. Write your program below (*you may add additional helper functions*):

```
#define TILE  0     #define WALL      1     #define DUMMY   2     #define MARKED  3
#define NONE  0     #define ENCLOSED  1     #define BOUNDED 2
#define ABS(x)          (x<0?-x:x)
#define NEIGHBOR(i,j)   (((ABS(i)) + (ABS(j))) == 1)
int countEnclosedSpace(char plan[][SIZE+1], int n, int m) {
  int i = 0, j = 0, ctx = 0, state;
  int f_plan[3*SIZE+2][3*SIZE+2] = {0}; convert(plan, f_plan, n, m);
  for(i=1; i<=3*n; i++)
    for(j=1; j<=3*m; j++)
      if(mark(f_plan, n, m, i, j) == ENCLOSED) ctx++;
  return ctx;
}
/* Mark all connected floor plans and check if it touches the "dummy"
-- NONE (already marked); ENCLOSED (enclosed space); BOUNDED (touched "dummy"); */
int mark(int f_plan[][3*SIZE+2], int n, int m, int x, int y) {
  int i, j, bound, state = NONE;
  if(f_plan[x][y] == MARKED || f_plan[x][y] == WALL) return NONE; // already marked
  state = ENCLOSED;
  if(f_plan[x][y] == TILE) f_plan[x][y] = MARKED;
  for(i=-1; i<=1; i++)  // mark all the partition
    for(j=-1; j<=1; j++)
      if(NEIGHBOR(i,j)) // 4 direction neighbors
        if(f_plan[x+i][y+j] == TILE) {
          bound = mark(f_plan, n, m, x+i, y+j); // recursively mark neighbors
          if(state != BOUNDED) // not bounded by "dummy"
            if(bound == BOUNDED) state = BOUNDED;
            else if(state != BOUNDED && bound == FOUND) state = FOUND;
        } else if(f_plan[x+i][y+j] == DUMMY) state = BOUNDED;
  return state;
}
```

```c
/* Convert each tile in floor plan to 3x3 tiles of WALL and TILE */
void convert(char plan[][SIZE+1], int f_plan[][3*SIZE+2], int n, int m) {
  int i, j, x, y; // f_plan contains "dummy" boundary at index 0, n+1, and m+1
  for(i=0,x=1; i<n; i++,x+=3)
    for(j=0,y=1; j<m; j++,y+=3)
      if(plan[i][j] == '/') {
        f_plan[x+0][y+0] = TILE; f_plan[x+0][y+1] = TILE; f_plan[x+0][y+2] = WALL;
        f_plan[x+1][y+0] = TILE; f_plan[x+1][y+1] = WALL; f_plan[x+1][y+2] = TILE;
        f_plan[x+2][y+0] = WALL; f_plan[x+2][y+1] = TILE; f_plan[x+2][y+2] = TILE;
      } else {
        f_plan[x+0][y+0] = WALL; f_plan[x+0][y+1] = TILE; f_plan[x+0][y+2] = TILE;
        f_plan[x+1][y+0] = TILE; f_plan[x+1][y+1] = WALL; f_plan[x+1][y+2] = TILE;
        f_plan[x+2][y+0] = TILE; f_plan[x+2][y+1] = TILE; f_plan[x+2][y+2] = WALL;
      }
  for(i=0; i<n+2; i++)
    floor[i][0] = floor[i][n+1] = DUMMY; // mark "dummy" boundaries
  for(j=0; j<n+2; j++)
    floor[0][j] = floor[m+1][j] = DUMMY; // mark "dummy" boundaries
}
```

Example*: the following is the input from Diagram 1.*

```
4 6     | size: 4 x 6
\//\\/  | Wall #1
\///\\  | Wall #2
//\\/\  | Wall #3
\/\///  | Wall #4
```