ESC101: Introduction to Computing

Pointers

Array of Pointers

```
int *arr[10];
```

- arr is a 10-sized array of pointers to integers
- An equivalent dynamic allocation

```
int **arr;
arr = (int **)malloc ( 10 *  sizeof(int *)  );
```

- Individual elements in arr (arr[0], … arr[9]) are NOT allocated any space.

```
for (j = 0; j < 10; j++)
    arr[j] = (int*) malloc (5*sizeof(int));
```

Exercise: All Substrings

- Read a string and create an array containing all its substrings (i.e. contiguous).
- Display the substrings.

Input: ESC

Output:

E
ES
ESC
S
SC
C

All Substrings: Solution Strategy

- What are the possible substrings for a string having length len?
- For $0 \le i < len$ and for every $i \le j < len$, consider the substring between the i^{th} and j^{th} index.
- Allocate a 2D char array having $\frac{len \times (len+1)}{2}$ rows (Why ? How many columns?)
- Copy the substrings into different rows of this array.

Solution: Version 1

```
int len, i, j, k=0, nsubstr;
char st[100], **substrs;
scanf("%s",st);
len = strlen(st);
nsubstr = len*(len+1)/2;
substrs = (char**)malloc(sizeof(char*) * nsubstr);
for (i=0; i<nsubstr; i++)
    substrs[i] = (char*)malloc(sizeof(char) * (len+1));

for (i=0; i<len; i++){
    for (j=i; j<len; j++){
        strncpy(substrs[k], st+i, j-i+1);
        k++;
    }
}
for (i=0; i<k; i++)
    printf("%s\n",substrs[i]);
```

```
for (i=0; i<k; i++)
    free(substrs[i]);
free(substrs);
```

Too much wastage…

E	'\0'		
E	S	'\0'	
E	S	C	'\0'
S	'\0'		
S	C	'\0'	
C	'\0'		

Solution: Version 1

```
int len, i, j, k=0, nsubstr;
char st[100], **substrs;
scanf("%s",st);
len = strlen(st);
nsubstr = len*(len+1)/2;
substrs = (char**)malloc(sizeof(char*) * nsubstr);
for (i=0; i<nsubstr; i++)
    substrs[i] = (char*)malloc(sizeof(char) * (len+1));

for (i=0; i<len; i++){
    for (j=i; j<len; j++){
        strncpy(substrs[k], st+i, j-i+1);
        k++;
    }
}
for (i=0; i<k; i++)
    printf("%s\n",substrs[i]);
```

```
for (i=0; i<k; i++)
    free(substrs[i]);
free(substrs);
```

Solution: Version 2

```
int len, i, j, k=0,nsubstr; char st[100], **substrs;
scanf("%s",st);
len = strlen(st);
nsubstr = len*(len+1)/2;
substrs = (char**)malloc(sizeof(char*) * nsubstr);

for (i=0; i<len; i++)
   for (j=i; j<len; j++){
      substrs[k] = (char*)malloc(sizeof(char) * (j-i+2));
      strncpy(substrs[k], st+i, j-i+1);
      k++;
   }
for (i=0; i<k; i++)
   printf("%s\n",substrs[i]);
```

```
for (i=0; i<k; i++)
   free(substrs[i]);
free(substrs);
```

This version uses much less memory compared to version 1

Solution: Version 3

```
int len, i, j, k=0,nsubstr;
char st[100], **substrs;
scanf("%s",st);
len = strlen(st);
nsubstr = len*(len+1)/2;
substrs = (char**)malloc(sizeof(char*) * nsubstr);

for (i=0; i<len; i++){
   for (j=i; j<len; j++){
      substrs[k] = strndup(st+i, j-i+1);
      k++;
   }
}
for (i=0; i<k; i++)
   printf("%s\n",substrs[i]);
```

```
for (i=0; i<k; i++)
   free(substrs[i]);
free(substrs);
```

Less code => more readable, fewer bugs!
possibly faster!

Returning Pointers

Source: http://www.xkcd.com/138

Example Function that Returns Pointer

```
char *strdup(const char *s);
```

- strdup creates a copy of the string (char array) passed as arguments
  - copy is created in dynamically allocated memory block of sufficient size
- returns a pointer to the copy created
- C does not allow returning an array of any type from a function
  - We can use a pointer to simulate return of an array (or multiple values of same type)

Returning Pointer

```c
#include<stdio.h>
int *fun();
int main() {
  printf("%d",*fun());
}

int *fun() {
  int *p, i;
  p = &i;
  i = 10;
  return p;
}
```

OUTPUT

```c
#include<stdio.h>
int *fun();
int main() {
  printf("%d",*fun());
}

int *fun() {
  int *p;
  p = (int*)malloc(sizeof(int));
  *p = 10;
  return p;
}
```

OUTPUT: 10

Returning Pointer: Beware

- The function stack (except for the return value) is gone once the function completes its execution.
  - All addresses of local variables and formal arguments become invalid
  - available for "reuse"
- But the heap memory, once allocated, remains until it is explicitly "freed"
  - even beyond the function that allocated it.
- addresses of static and global variables remain valid throughout the program.

An Intuition

- Think of executing a function as writing on a classroom blackboard.
- Once the function finishes execution (the class is over), everything on the blackboard is erased.
- What if we want to retain a message, after class is over?
- Solution could be to post essential information on a "notice board", which is globally accessible to all classrooms.
- The blackboard of a class is like the stack (possibly erased/overwritten in the next class), and the notice board is like the heap.

Exercise: Generating strings

- Initial condition: A string such as "aba";
- Rule for generating next string
  - If there is a character 'a' in input string then output a character 'b' in output string
  - If there is a character 'b' in input string then output characters 'aa' in output string
- Input: number of times to generate a string such as 4

Output

0 aba
1 baab
2 aabbaa
3 bbaaaabb

Solution

```
char *genstring( char *inp)
{
   int ilen = strlen(inp), bcnt=0, k=0;

   for( int i=0; i<ilen; i++)
      if(inp[i] == 'b')
         bcnt++;

   int olen = ilen+bcnt+1;
   char *out = (char *) malloc(sizeof(char) * olen);

  for(int i=0; i<ilen; i++)
   {
      if(inp[i] == ‘a') {
         out[k] = 'b';  k++;
      }
      else  {
         out[k] = 'a'; k++;  out[k] = 'a'; k++;
      }
   }
   out[k] = ‘\0';

   free(inp);
   return out;
}
```

Solution

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{
   char *str;
   int n;

  //initialization
   str = (char*) malloc(sizeof(char)*4);
   strcpy(str,"aba");
   scanf("%d",&n);
   for(int i=0; i<n; i++)
   {
      printf("%d %s\n",i, str);
      str = genstring(str);
   }
   free(str);
   return 0;
}
```

Multi-dimensional Array vs. Multi-level pointer

Are these two equivalent

```
int a[2][3];
```

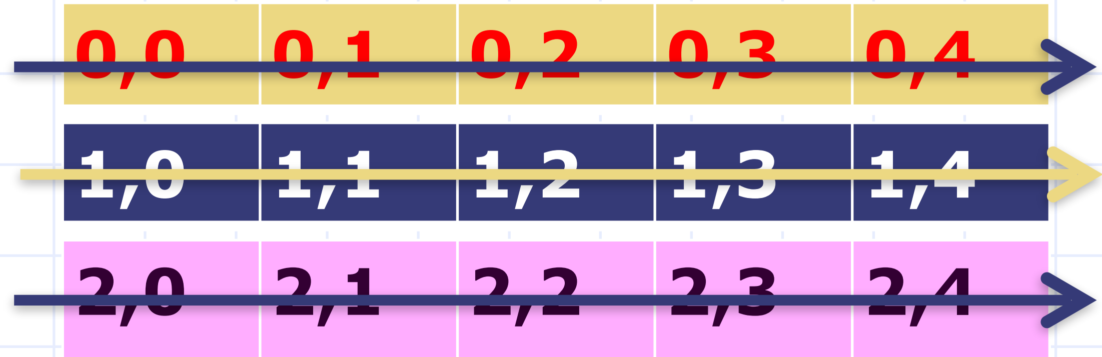
```
int **b;
b = (int**)malloc(2*sizeof(int*));
b[0] = (int*)malloc(3*sizeof(int));
b[1] = (int*)malloc(3*sizeof(int));
```

Row Major Layout

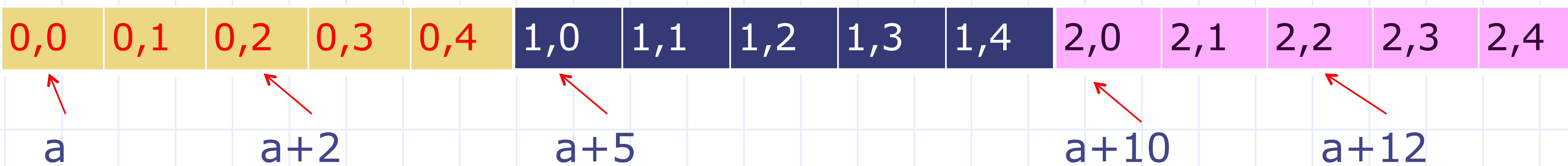
- 2D (or >2D) arrays are "flattened" into 1D to be stored in memory
- In C (and most other languages), arrays are flattened using Row-Major order
  - In case of 2D arrays, knowledge of number of columns is required to figure out where the next row starts.
  - Last n-1 dimensions required for nD arrays

Row Major Layout

mat[3][5]



Layout of mat[3][5] in memory



- **for a 2D array declared as mat[M][N], cell [i][j] is stored in memory at location i*N + j from start of mat.**

Row Major Layout

mat[3][5]

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

Layout of mat[3][5] in memory

0,0	0,1	0,2	0,3	0,4	1,0	1,1	1,2	1,3	1,4	2,0	2,1	2,2	2,3	2,4

mat[0] mat[1] mat[2]

int a[3][3]

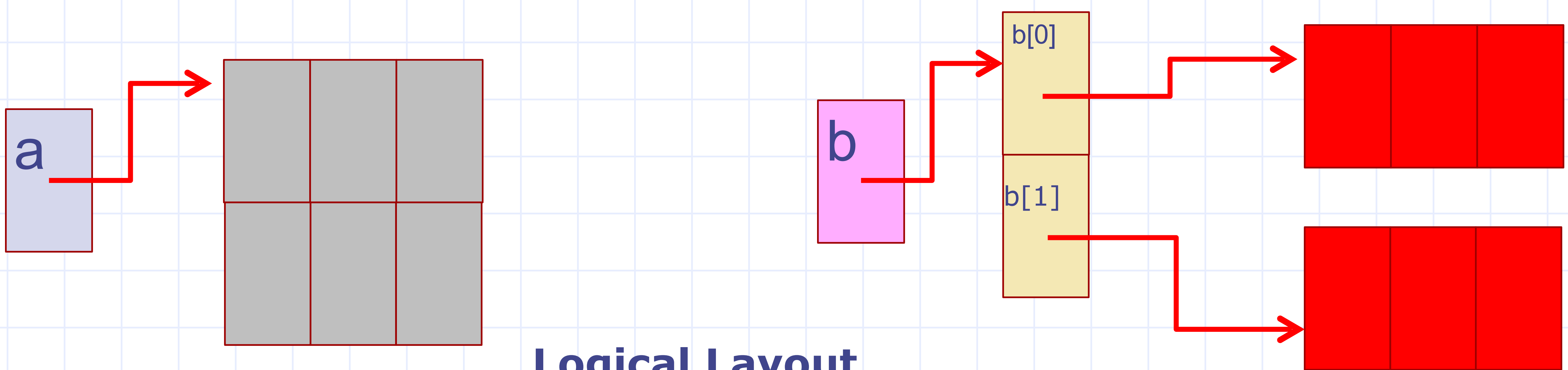
3	9	27
4	16	64
5	25	125

Expression	Value	Expression
*(*a+0)	a[0][0]=3	
*(*a+2)	a[0][2]=27	
*(*a+3)	a[1][0]=4	*(*(a+1)+0)
*(*a+7)	a[2][1]=25	*(*(a+2)+1)

Memory layout

```
int a[2][3];
```

```
int **b;
b = (int**)malloc(2*sizeof(int*));
b[0] = (int*)malloc(3*sizeof(int));
b[1] = (int*)malloc(3*sizeof(int));
```



Logical Layout

Warning:

- **(*b+3) may not point to b[1][0].**
- **(*a+3) points to a[1][0].**

int a[3][3]

3	9	27
4	16	64
5	25	125

Expression	Value	Expression
*(*a+0)	a[0][0]=3	
*(*a+2)	a[0][2]=27	
*(*a+3)	a[1][0]=4	*(*(a+1)+0)
*(*a+7)	a[2][1]=25	*(*(a+2)+1)