ESC101: Introduction to Computing

Pointers

malloc: Example

A pointer to float

```
float *f;
f= (float*) malloc(10 * sizeof(float));
```

Size big enough to hold 10 floats.

Explicit type casting to convey users intent

Note the use of **sizeof** to keep it machine independent

malloc evaluates its arguments at runtime to allocate (reserve) space. Returns a **void***, pointer to first address of allocated space.

free: Example

malloc: allows us to allocate memory. It is our job to release the memory once we are done using the memory

```
float *f;
f= (float*) malloc(10 * sizeof(float));
//use f
free(f);
```

memory in f is released, future references to f will result in error if memory in f not initialised

Exercise

- Write a program to read two integers, n, m and store powers of n from 0 up to m (n^0, n^1, …, n^m)

Exercise

- Write a program to read two integers, n, m and store powers of n from 0 up to m (n^0, n^1, …, n^m)

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int *pow, i, n, m;
    scanf("%d %d", &n, &m); // m>= 0
    pow = (int *) malloc ((m+1) * sizeof(int));
    pow[0] = 1;
    for (i=1; i<=m; i++)
        pow[i] = pow[i-1]*n;
    for (i=0; i<=m; i++)
        printf("%d\n",pow[i]);
    free(pow);
    return 0;
}
```

Note that instead of writing **pow[i]**, we can also write
***(pow + i)**

NULL

- A special pointer value to denote “points-to-nothing”
- C uses the value 0 or name NULL
- In Boolean context, NULL is equivalent to false, any other pointer value is equivalent to true
- A malloc call can return NULL if it is not possible to satisfy memory request
  - negative or ZERO size argument
  - TOO BIG size argument

Typical dynamic allocation

```
int *ar;

…
ar = (int*) malloc(…);
if (ar == NULL) {  // ≡ if (!ar)
  // take corrective measures OR
  // return failure
}

…
…ar[i]… // use of ar

…
free(ar); // free after last use of ar
```

Arrays and Pointers

- In C, array names are nothing but pointers.
  - Can be used interchangeably in most cases
- However, array names can not be assigned, but pointer variables can be.
  - Array name is not a variable. It gets evaluated in C.

```
int ar[10], *b;

ar = ar + 2; ✗

ar = b; ✗

b = ar; ✓

b = b + 1; ✓

b = ar + 2; ✓

b++; ✓
```

What is the output of following code?

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

int main()
{
        char a[]="A long line of text with many words";
        char ch, *p;
        int i=strlen(a);
        while(i>=0)
        {
                if(a[i]==' ')
                {
                        a[i] = '\0';
                        p = &a[i+1];
                        printf("%s\n",p);
                }
                i--;
        }
        p = &a[i+1];
        printf("%s\n",p);
        return 0;
}
```

What is the output of following code?

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

int main()
{
        char a[]="A long line of text with many words";
        char ch, *p;
        int i=strlen(a);
        while(i>=0)
        {
                if(a[i]==' ')
                {
                        a[i] = '\0';
                        p = &a[i+1];
                        printf("%s\n",p);
                }
                i--;
        }
        p = &a[i+1];
        printf("%s\n",p);
        return 0;
}
```

Output is:

```
words
many
with
text
of
line
long
A
```

Array of Pointers

- Consider the following declaration

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

- arr is a 10-sized array of pointers to integers
- How can we have equivalent dynamic array?

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

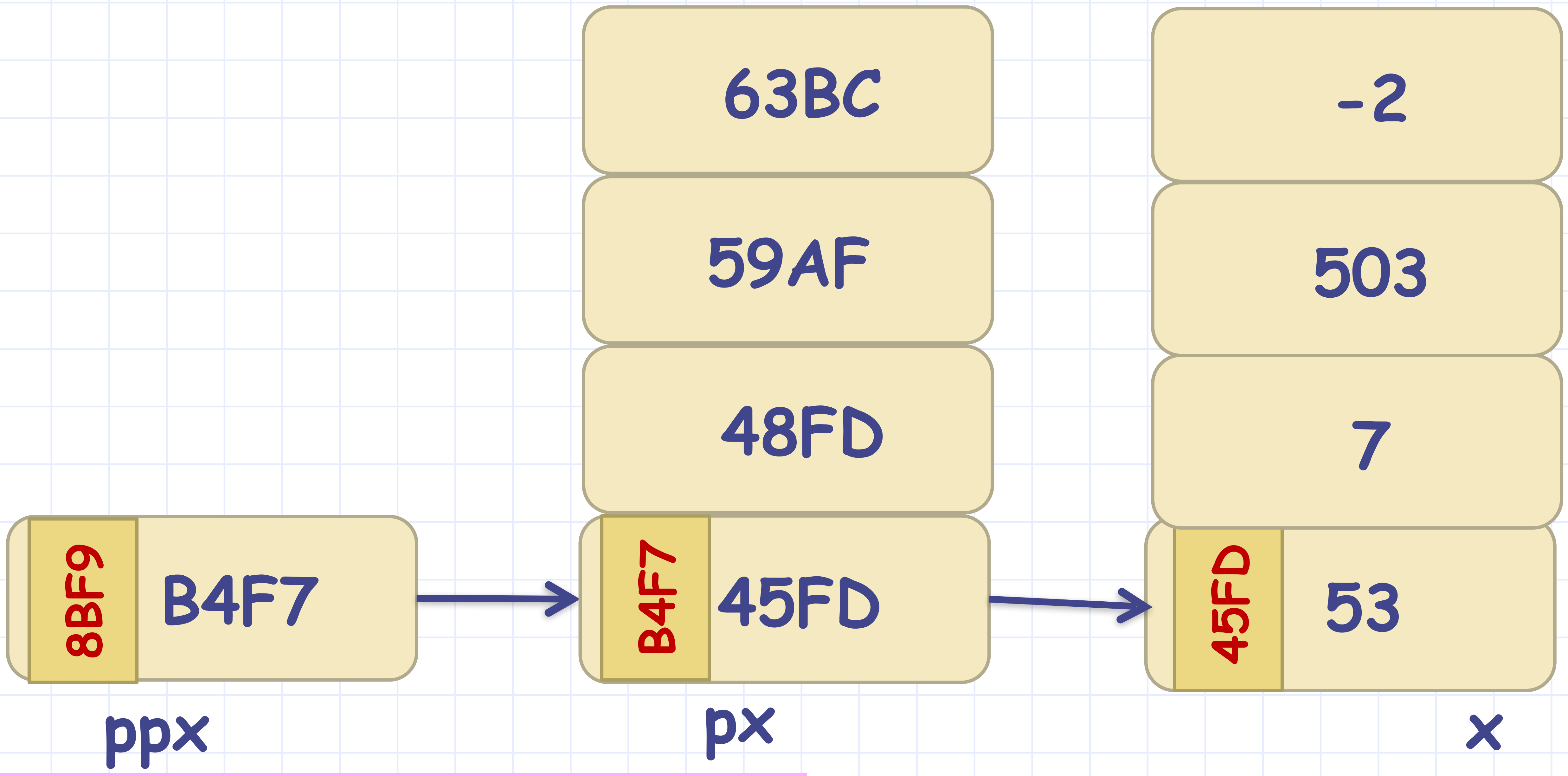
Array of Pointers

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

- Note that individual elements in the array arr (arr[0], … arr[9]) are NOT allocated any space. Uninitialized.
- We need to do it (directly or indirectly) before using them.

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

Pointer to a pointer



We are showing addresses for explanation only.
Ideally, the program should not depend on actual addresses.

```
int x[4]; x[0]=53;
int *px = x;
int **ppx = &px;
```

Exercise

- Write a program to read in names and output the length of the longest name

Solution for exercise

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAXSTRING 100
int main()
{
   char **names;
   int i, n;
   int imax, lenmax=0, len;
   scanf("%d",&n);
   names = (char **) malloc(sizeof(char*) * n);
   for(i=0; i<n; i++) {
      names[i] = (char *) malloc(sizeof(char)*MAXSTRING);
      scanf("%s",names[i]);
   }
   for(int i=0; i<n; i++)
   {
      len = strlen(names[i]);
      if( len > lenmax) {
         imax = i;   lenmax = len;
      }
   }
   printf("longest name is %s\n",names[imax]);
   for( int i=0; i<n; i++)
      free(names[i]);
   free(names);
   return 0;
}
```

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 $0 \leq i < len$ and for every $i \leq j < len$, 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'		