Exercise

- Write a program that reads two integers, n and m, and stores 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]);
    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

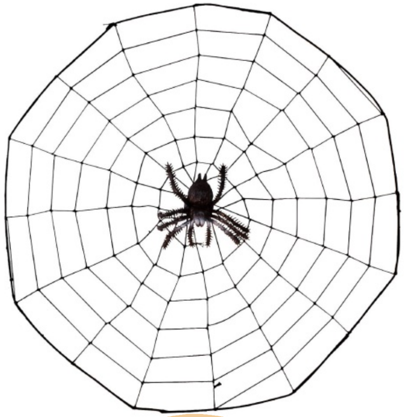
Pointers and Initialization

- Uninitialized pointer has GARBAGE value, NOT NULL
- Memory returned by malloc is **not** initialized.
- Brothers of malloc
  - calloc(n, size): allocates memory for n-element array of size bytes each. Memory is initialized to 0.
  - realloc(ptr, size): changes the size of the memory block pointed to by ptr to size bytes.
    - Complicated semantics, try to avoid.

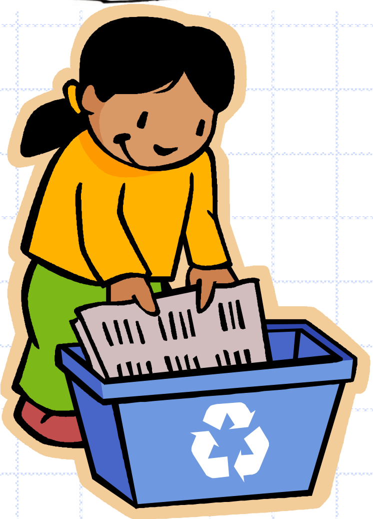
Both malloc, calloc return a **logically contiguous** block of memory.

Calloc also **clears-memory** with zeros.

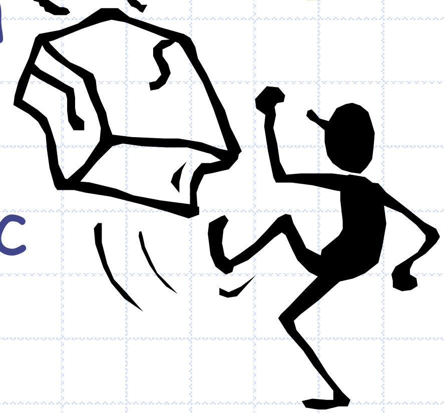
With great power comes great responsibility



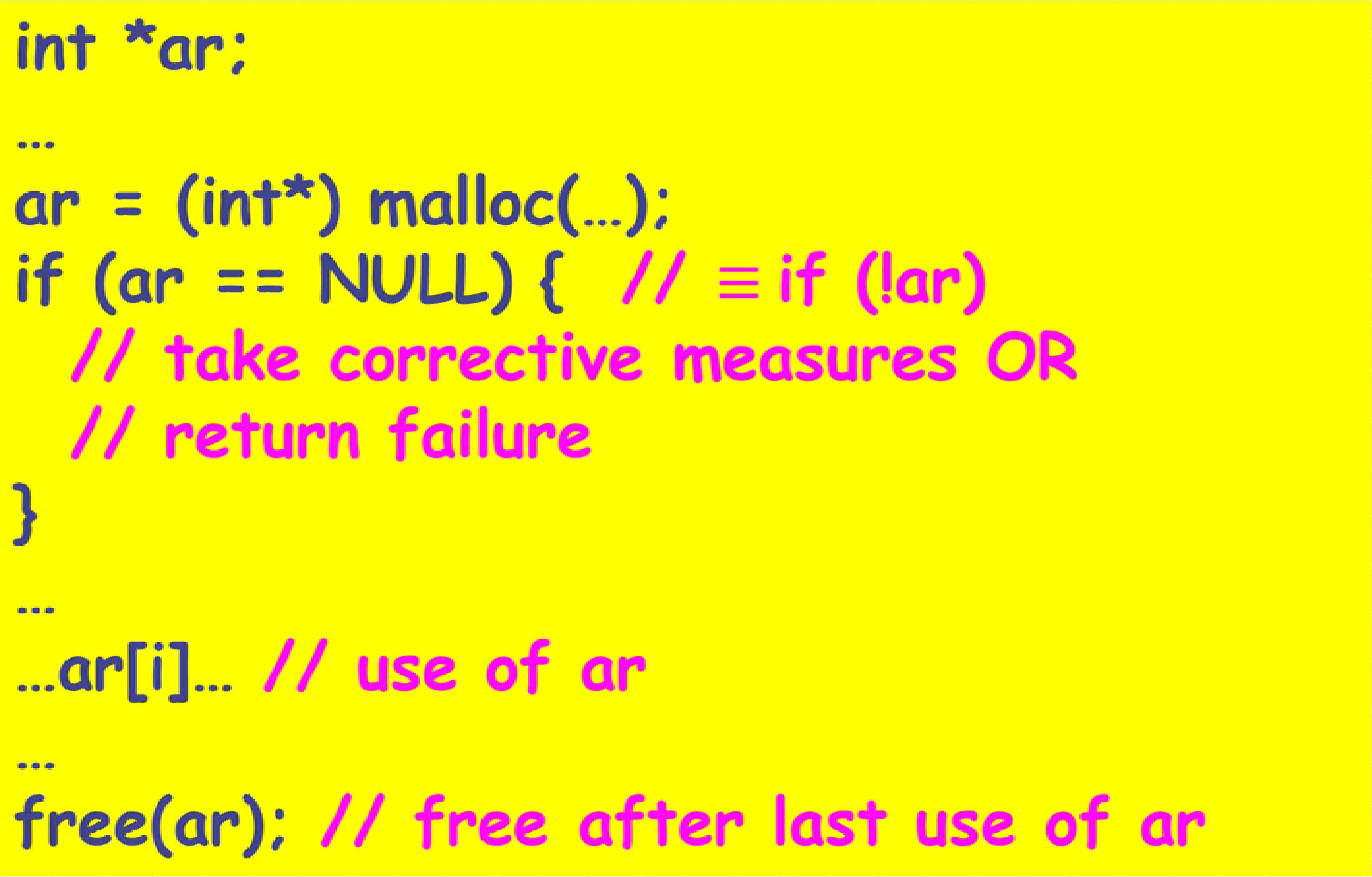
- Power to allocate memory when needed must be complimented by the responsibility to de-allocate memory when no longer needed!
  - free unused pointers



- Be prepared to face rejection of demand
  - Check the return value of malloc (and its variants)



Typical dynamic allocation



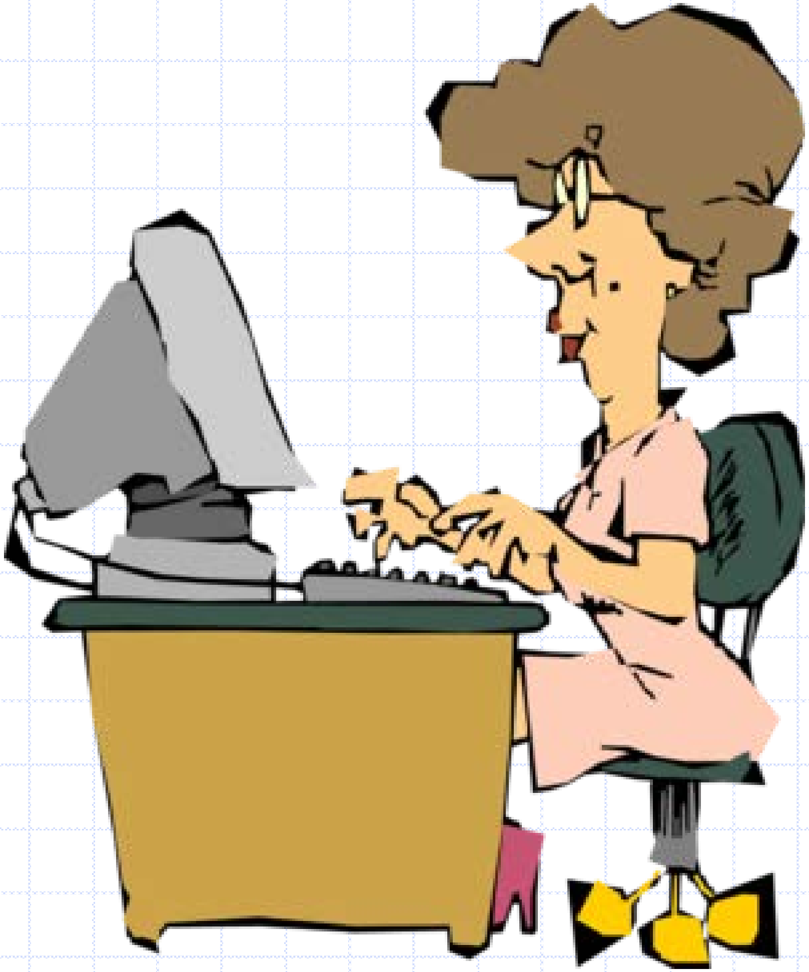
```
int *ar;
...
ar = (int*) malloc(...);
if (ar == NULL) {  // ≡ if (!ar)
  // take corrective measures OR
  // return failure
}
...
...ar[i]... // use of ar
...
free(ar); // free after last use of ar
```

Dynamic memory management is similar to library management

Pointer Declaration = Registration

```
int *ar;
```

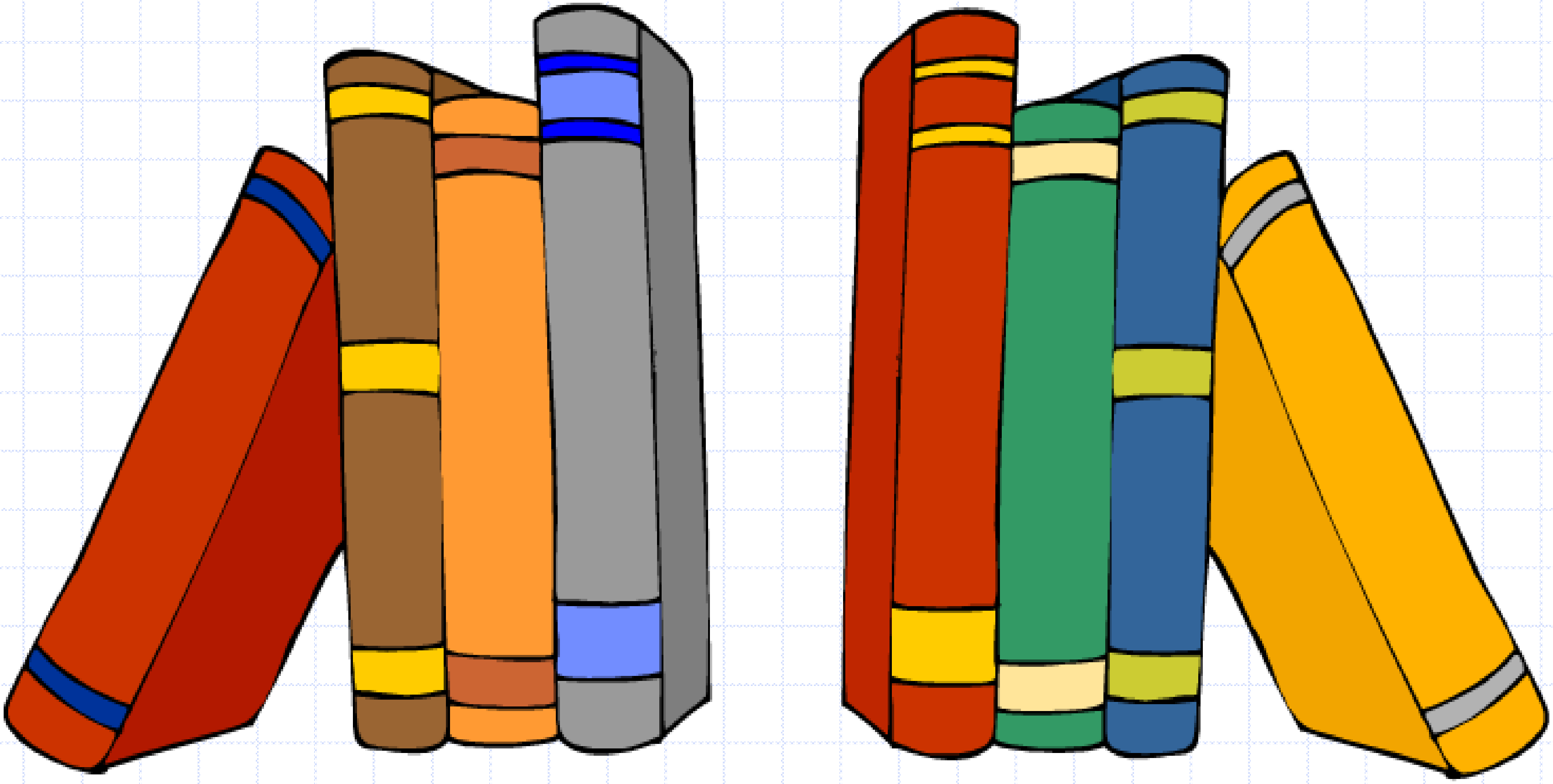
Declare your intent that you will use books from the library



malloc = check out

`ar = (int*) malloc(…);`

Reserve book(s) for your use



What if the book is not available?

```
if (ar == NULL) {
  // take corrective measures
  // OR return failure
}
```

Book not available:
Purchase the book?
Share with a friend?
Not study ☹

If the check out is successful

```
…ar[i]… // use of ar
```

Read it.

If the check out is successful

```
br = ar; // copy the address
…
ar[i] = …; // change the content
…
br[i] = …; // change the content indirectly
…
```

Share it.
Use it!

free = return the book

```
free(ar); // free after last use of
          // alloc'ed memory
```

Your job is done, return the book so that others can use it.

Return the book

```
br = ar;
…
free(br); // free after last use
~~free(ar);~~ // multiple free of same loc not allowed
```

Your friend can also return the book for you.

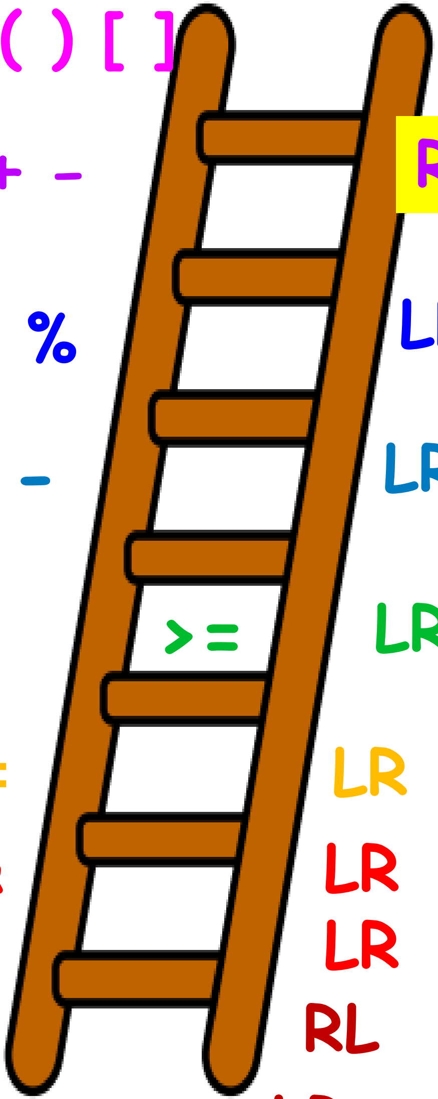
But a book can be returned only once per check out!

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++; ✓
```

Precedence (Unary Refined)



( ) [ ]	LR
* (deref) ++ -- ! & + -	RL
* / %	LR
+ -	LR
< <= > >=	LR
== !=	LR
&&	LR
||	LR
=	RL
,	LR

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));
```

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’		

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!