Pointers and Arrays

December, 2022

Dynamic Memory Allocation

Malloc

```
void *malloc(size_t size);
```

Calloc

```
void *calloc(size_t nitem, size_t size);
```

Realloc

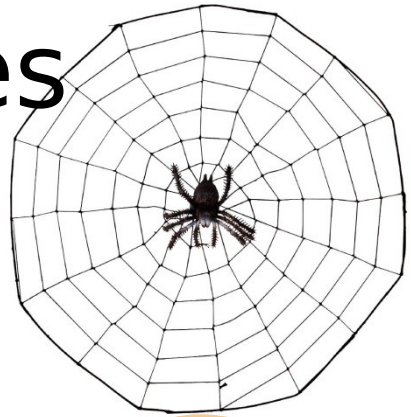
```
void *realloc(void *ptr, size_t size);
```

Free

```
void free(void *ptr);
```

Size_t is an alias for unsigned int. It is defined in the header file stdio.h and stdlib.h

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)

Dynamic Memory Management is Similar to Library Management

Pointer Declaration = Registration

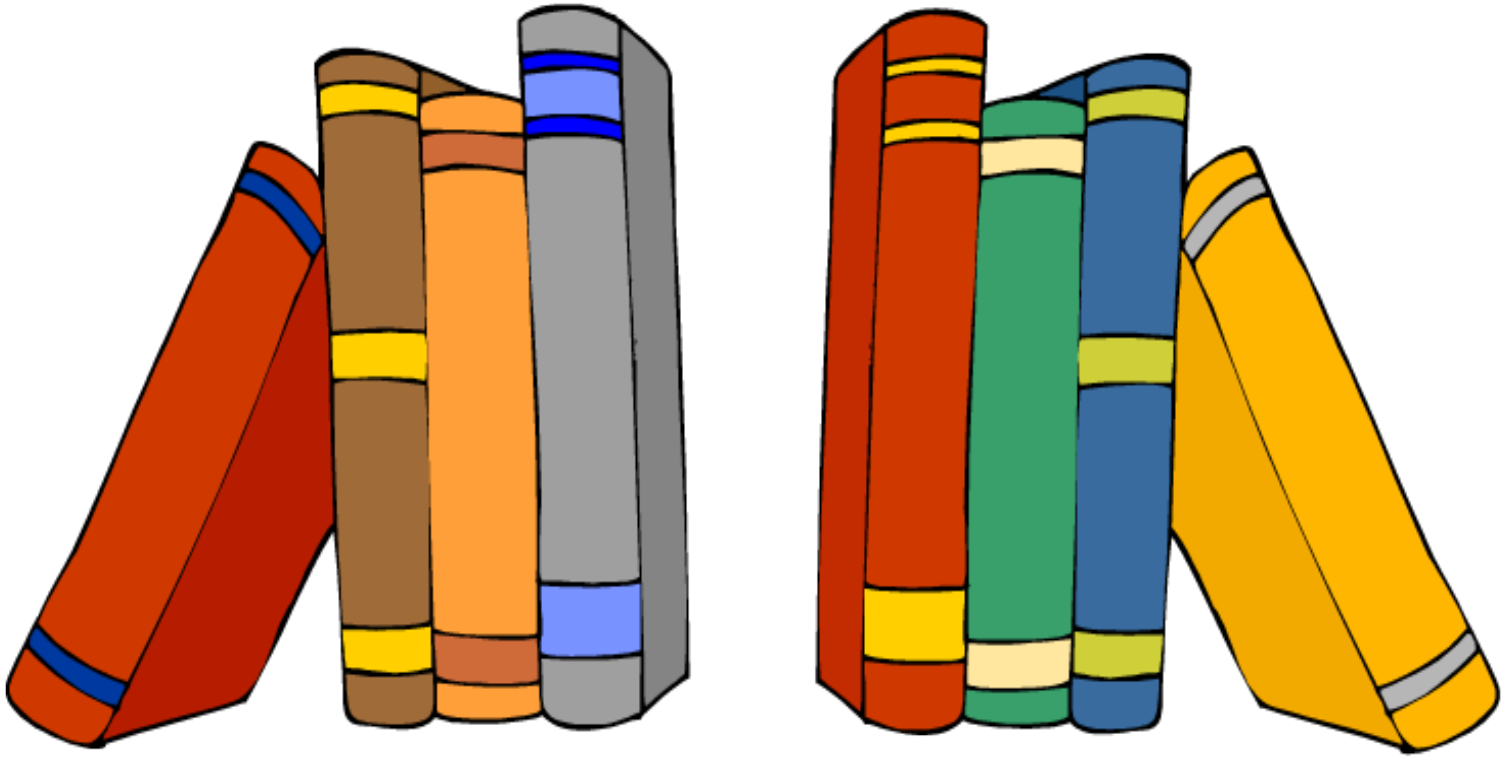
int *ar;

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 on failure
}
```

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

If the check out is successful

…ar[i]… // use of ar

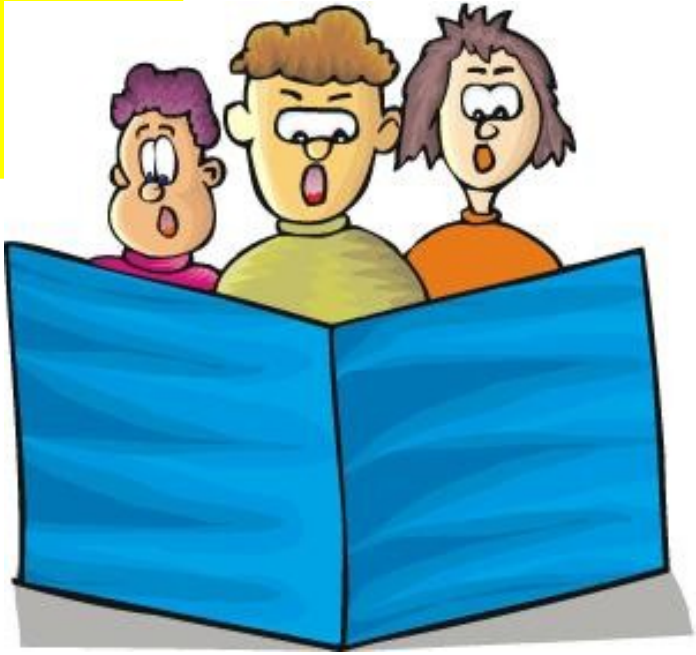
Read it.

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.

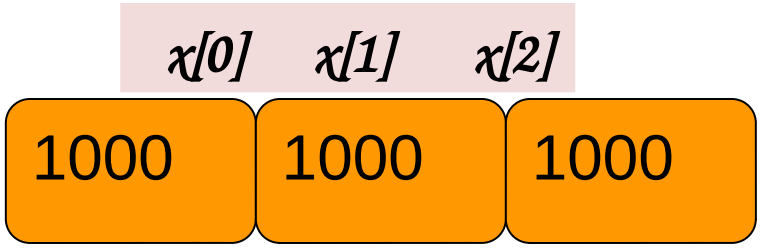
But a book can be returned only once per check out!

Typical Dynamic Allocation

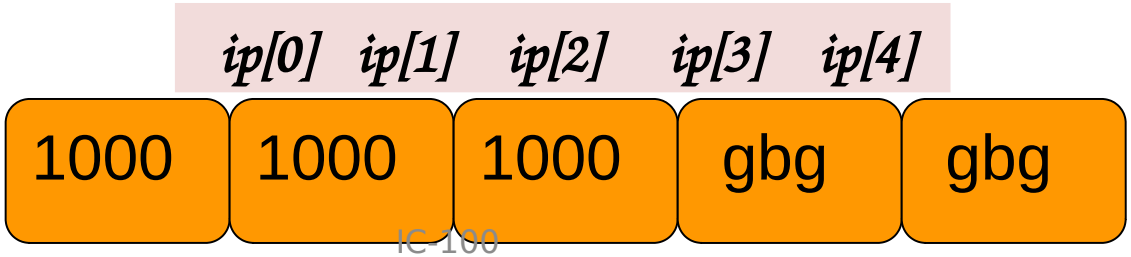
```
int * ar;
....
ar = (int *) malloc(...);  // allocate memory
if (ar == NULL) {  // or if (!ar)
   // take corrective measures and return
   failures
}
...ar[i] ...
free(ar);   // free after last use of ar
```

realloc

```
int *x;
x = (int*)malloc(3 * sizeof(int));
X[0] = 1000; x[1] = 1000; x[2] = 1000;
```

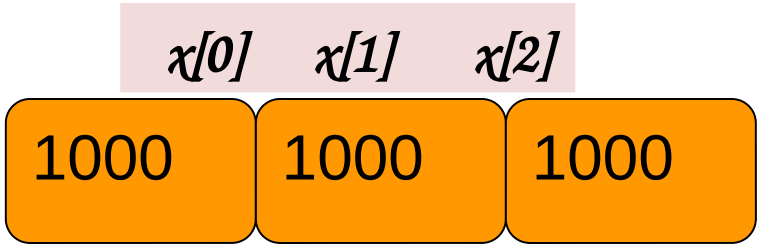



```
/* need more space */
ip = (int*)realloc(x, 5 * sizeof(int));
```

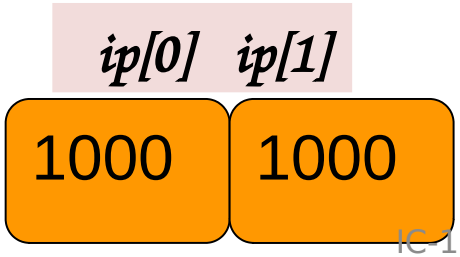


realloc

```
int *x;
x = (int*)malloc(3 * sizeof(int));
X[0] = 1000; x[1] = 1000; x[2] = 1000;
```




```
/* need less space */
ip = (int*)realloc(x, 2 * sizeof(int));
```



realloc and free

- How do we free a pointer we passed into realloc and returned out into the same variable name?

```
int *ip;
ip = (int*)malloc(100 * sizeof(int));
...
/* need twice as much space */
ip = (int*)realloc(ip, 200 * sizeof(int));
```

- If realloc succeeds in memory allocation, old memory is automatically freed
- If realloc fails, the old memory block is not freed and null pointer is returned

Question

- What is the difference between the *arr* we get from
  - int arr[10]
  - int *arr followed by arr = malloc(sizeof(int)*10)
- Both are pointers to the first block of memory assigned to the array
- Static assigned pointer cannot be re-assigned
- Dynamically assigned pointer can be re-assigned

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.

```
int ar[10], *b;
ar = ar + 2;
ar = b;
b = ar;
b = b + 1;
b = ar + 2;
b++;
```






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 n?
- Allocate a 2D char array having K= n(n+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

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

char *strndup(const char *s, size_t num);

- Returns a pointer to a null-terminated byte string, which contains copies of at most num bytes from the string pointed to by s.

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!