ECE 220

Lecture x0010 - 10/22/24
Dynamic Memory Allocation

Slides based on material originally by: Yuting Chen & Thomas Moon

Announcements

- Midterm 2 will be held on 10/31
  - The conflict exam sign-up link is live
    - Deadline is Sunday on the week of the exam.
- Practice material is posted
- Check HKN website: https://hkn.illinois.edu/services for review session
- Thursday's lecture by Mike Montano.

Recap

- Last few weeks
  - Streams, buffers, queue (FIFO)
  - File I/O, formatted IO
  - Structs
    - Arrays of structs
    - Pointers to structs
    - Structs within structs
    - Passing structs in functions
    - Writing structs to files
  - Examples

Exercise

- Last time we wrote a function to write flight details to a binary file and then we read the data back from the file.

- Modify `airport_1.c` code from last time to now use *functions* to

  A. write struct to file

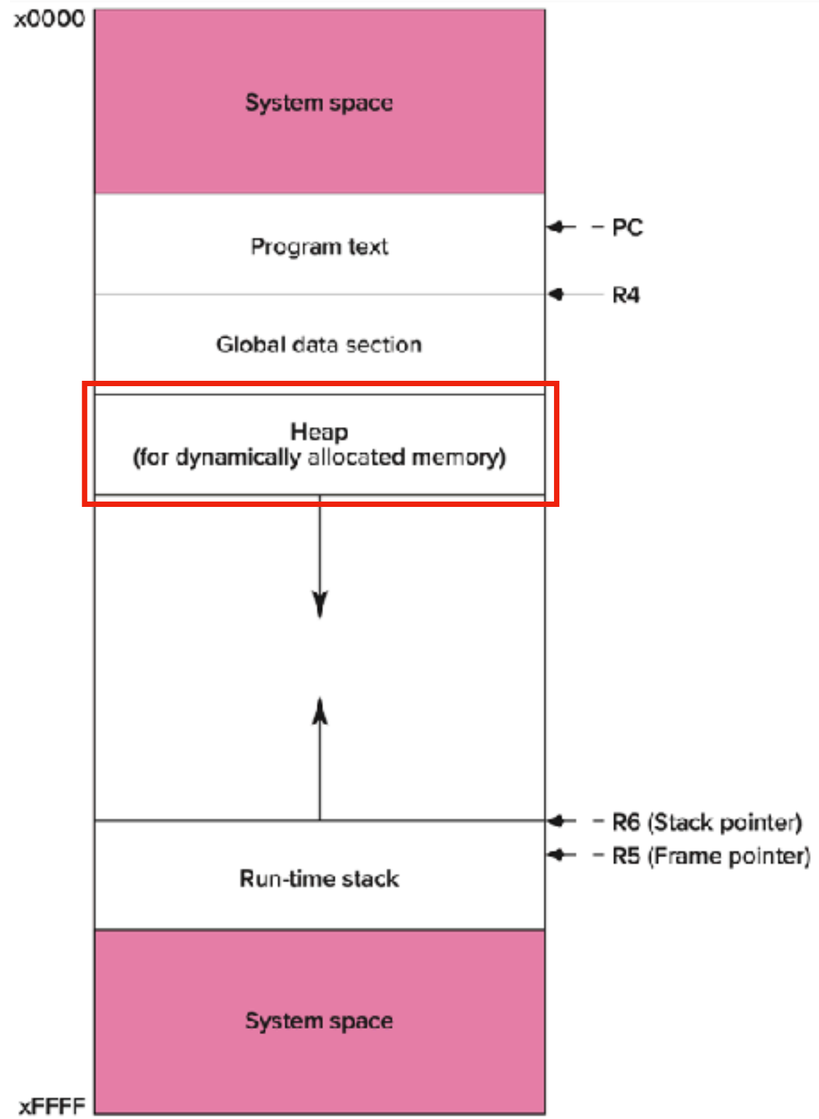
  B. load struct from file

Dynamic memory allocation

- We ask for *N*, the number of planes each time to set up the loops. Nevertheless the array size is fixed at 10.
  - If usually only ~3 flights, then memory is wasted.
  - If we read in a large file >50 then not enough memory is allocated.
- Ideally, we want to allocate as much memory as needed rather than a pre-set amount.
- In most cases, this memory comes from an area of the architecture called the *heap*.

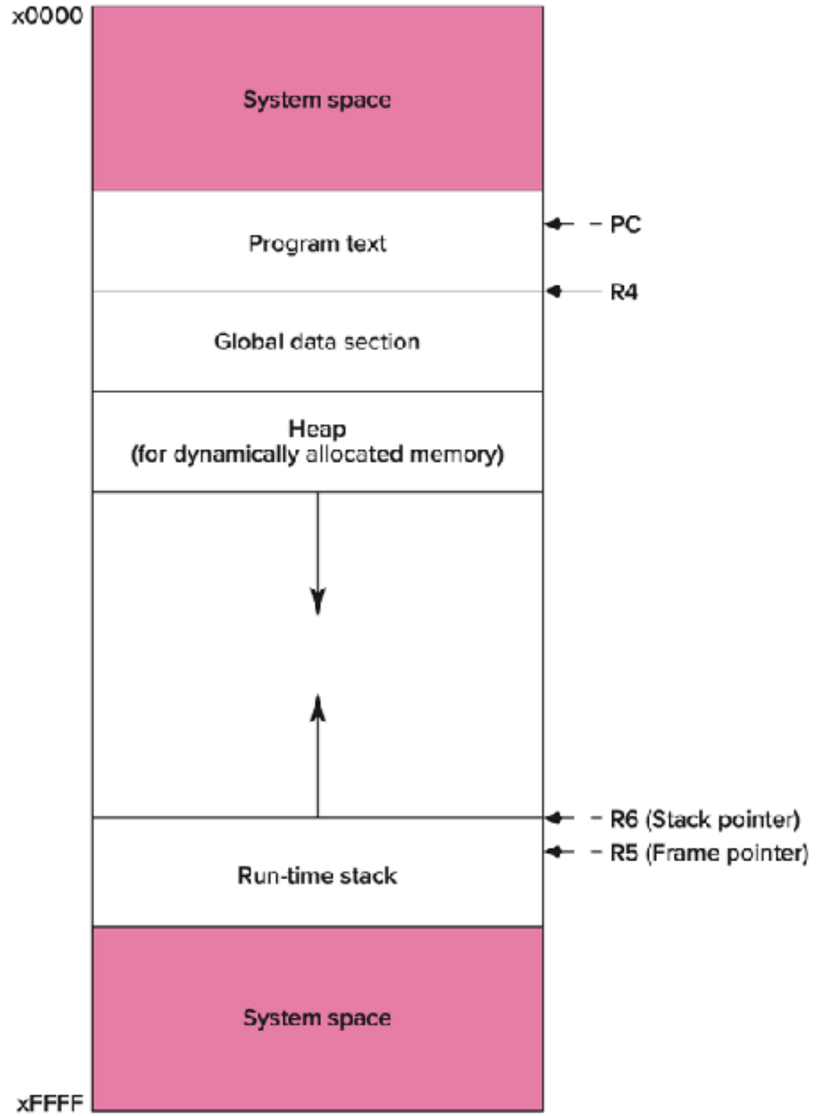
Dynamic memory allocation

- During the execution, a program makes a request to the memory allocator for a contiguous piece of memory of a particular size
- The allocator reserves the memory and returns a pointer to it. We interact with the memory allocation manager by using *malloc* family & *free* functions.



Automatic vs dynamic memory

	Automatic	Dynamic
Mechanism	Automatic	Use `malloc` family
Lifetime	Compiler makes decisions; variables “die” when functions & blocks end	Programmer makes decision, must use `free()` to deallocate
Location	Stack or global data area	Heap
Size	Fixed	Adjustable



The malloc function

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

- Parameters
  - `size`: Number of bytes to allocate
  - `size_t`: A *type* defined in the user library ~ unsigned integer
- Return value: NULL (failure) or pointer to beginning of allocated block (success).

https://en.cppreference.com/w/c/memory/malloc

Using malloc

- Memory allocated by `malloc` is not initialized (there could be garbage values or leftover values).

- To use `malloc`, we need to know how many bytes to allocate. The `sizeof` operator asks the compiler to calculate the size of a particular type.

- We also need to change the type of the return value to the proper kind of pointer- this is called "*casting*".

Standard pointer declaration

malloc returns *void* pointer

```
int *ptr = (int *) malloc(sizeof(int));
```

Juxtaposition with *(int *) casts* the *void* pointer as an *int* pointer

The free function

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

- Parameters
  - *`*ptr`*: Pointer to beginning of block to be *deallocated*. Should have been generated by the `malloc` family.

- Memory allocated via `malloc` ***must*** be deallocated via `free` or reallocated via `realloc` to prevent memory leaks!

- Use `valgrind` to check for memory leaks

https://valgrind.org/

The calloc function

```
void *calloc(size_t n_items, size_t item_size)
```

- Parameters
  - *`size`*: Number of items to be allocated
  - *`item_size`*: Size of each item
- Return value: NULL (failure) or pointer to beginning of allocated block (success).
- Identical to `malloc`, except `calloc` initializes memory to zero.

The realloc function

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

- Parameters
  - *ptr*: Pointer to memory block to be reallocated
  - *size*: New size of block
- Return value: NULL (failure) or pointer to beginning of allocated block (success).

The realloc function

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

- The content of the memory block is preserved, even if the block is moved to a new location (if the new size is larger than the old size, the added memory will not be initialized).
  - If `ptr` is NULL, it is same as malloc
  - If `size` is 0 and `ptr` is not NULL, *implementation dependent!*
  - `ptr` must have been returned by the `malloc` family

Example of malloc & free

- Casting:

```
int *ptr = (int *) malloc(sizeof(int));
Flight *ptr = (Flight *) malloc(numFlight*sizeof(Flight));
```

- Why: recall C is *statically* typed; so compiler needs to know what type to assign to allocated memory locations.
  - Sorta-kinda a fib (C can tell by looking at LHS, but C++ won't)
  - Types can be built-in or user-defined.

Example of malloc & free

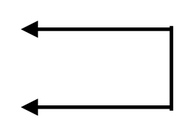
```
int main(){
  int *ptr1 = (int *) malloc(sizeof(int));
  if(ptr1==NULL){
    printf("Error - malloc failure\n");
    return -1;
  }
  *ptr1 = 10;
  int *ptr2 = (int *) malloc(sizeof(int));
  *ptr2 = 5;
}
```

What is wrong with this code?

Didn’t free memory allocated!

Example of malloc & free

```
int main(){
  int *ptr1 = (int *) malloc(sizeof(int));
  if(ptr1==NULL){
    printf("Error - malloc failure\n");
    return -1;
  }
  *ptr1 = 10;
  int *ptr2 = (int *) malloc(sizeof(int));
  *ptr2 = 5;

  ptr1 = ptr2;
  free(ptr1);
  free(ptr2);
}
```



This one frees the memory, but has a bug. What should we do?

Example of realloc

```c
int *ptr;
int *ptr_new;

// What does this code do?
ptr = (int *) calloc(2, sizeof(int));
*ptr = 10;

// What is the contents of memory now?
ptr_new = (int *) realloc(ptr, 4*sizeof(int));
*(ptr_new+2) = 30;
*(ptr_new+3) = 40;

// How much memory are we deallocating here?
free(ptr_new)
```

Do we need `free(ptr)`?

Allocating 2D arrays

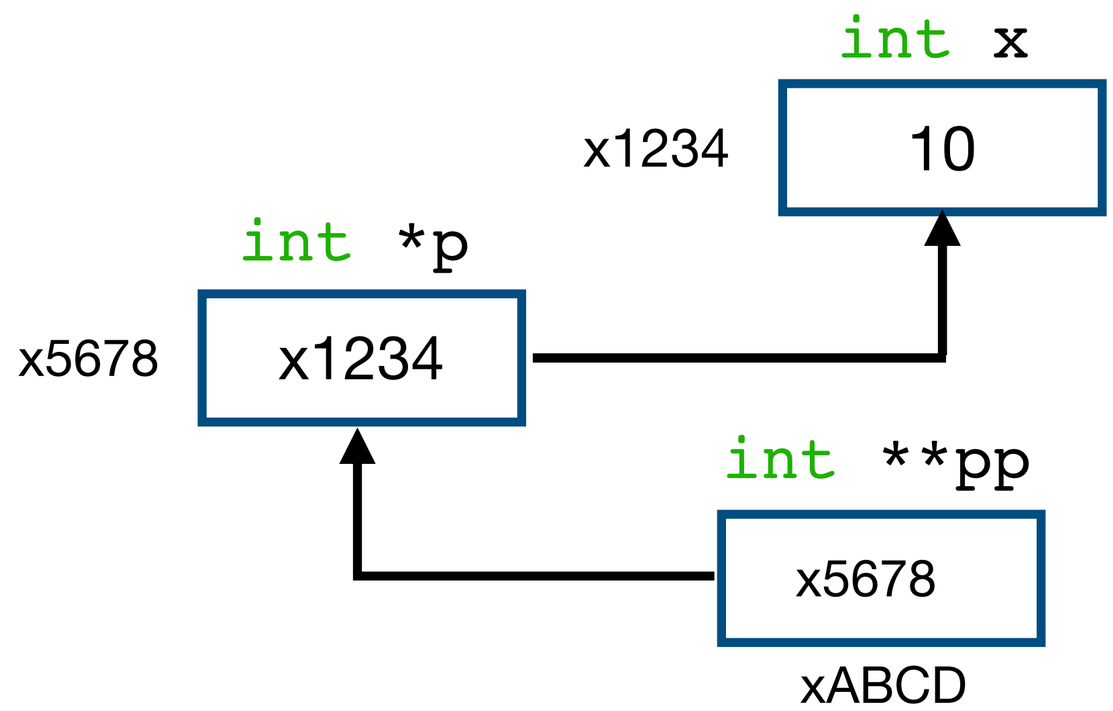
- Here is **one** method of allocating 2D arrays:

```
FILE *infile = fopen("mat.csv", "r");
int nr, nc;

fscanf(infile, "%d, %d", &nr, &nc);
int *mat = (int *) malloc(sizeof(int)*nr*nc);

for (int i=0; i < nr; i++)
  for (int j=0; j< nc; j++)
    fscanf(infile, "%d, ", &mat[i*nc+j]);

fclose(infile);
```

Allocating 2D arrays - another way

- Recall pointers to pointers?

```
int x = 10;
int *p = &x;
int **pp = &p;
```



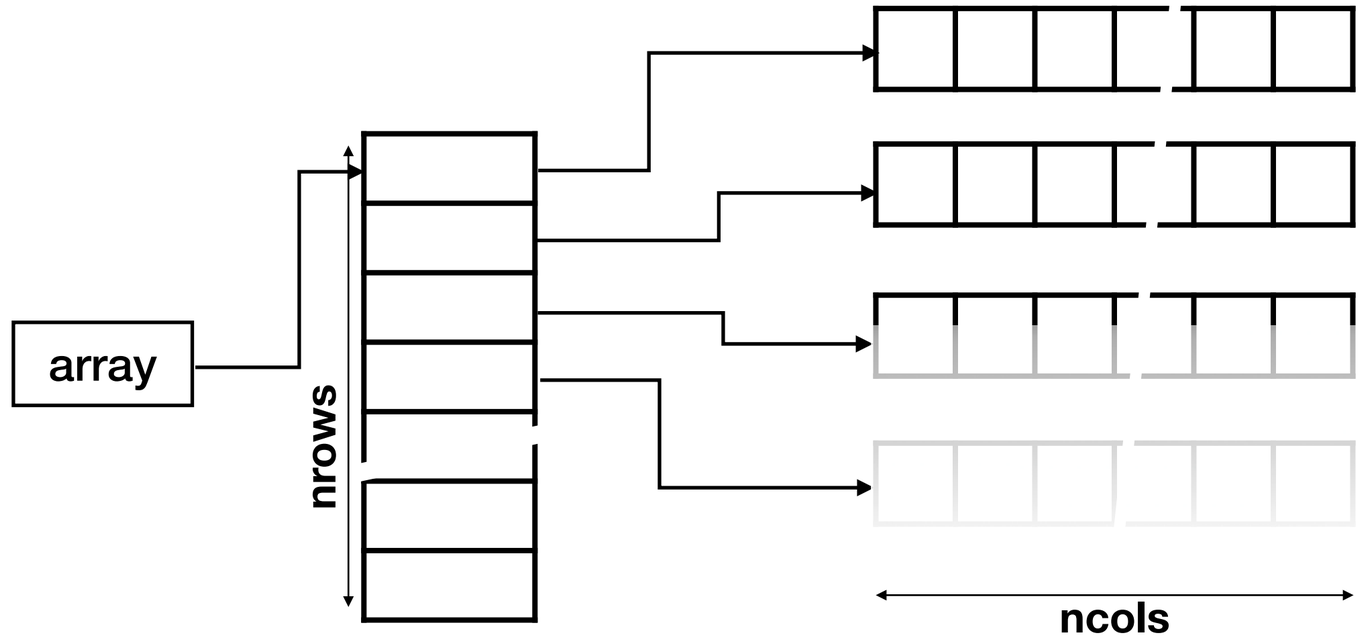
Allocating 2D arrays - another way

- Recall pointers to pointers?
- We can use that:

```
int **array;

array = (int**) malloc(nrows*sizeof(int*));
for(i=0;i<nrows;i++)
  array[i] = (int*) malloc(ncols*sizeof(int));
array[0][0] = 3;
…
```

Allocating 2D arrays - another way

```
int **array;

array = (int**) malloc(nrows*sizeof(int*));
for(i=0;i<nrows;i++)
  array[i] = (int*) malloc(ncols*sizeof(int));
array[0][0] = 3;
```



Pointer to pointer - caveat

- How do you **deallocate** a 2D array?
  - Method 1: Free the single pointer: `int * mat`
  - Method 2: Need to free ***each*** pointer separately!!
    - **Not** enough to free the top level pointer `(int **array)`
      - Unless made free, lower level pointers `(int *)` will leak memory!

Exercise

- Use this second method of memory allocation for 2D arrays to read in a given file (`matrix.csv`) and print out its transpose.

- The first row of the file lists the number of rows and columns of the matrix.

Aside: Variable Length Arrays

- You could still define an array size using user input.
  - Array still allocated on the stack
  - Mechanism is far more complicated
- Still cannot modify size after definition
- We pay that performance overhead for convenience

```
void fun(int n)
{
 int arr[n];
 /* More code follows
 ...
 ...
 */
}
int main()
{
   fun(6);
}
```

Example with *valgrind*

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

int main(void){
  char *p;

  /* Allocation #1 of 19 bytes */
  p = (char *) malloc(19);

  /* Allocation #2 of 12 bytes */
  p = (char *) malloc(12);
  free(p);

  /* Allocation #3 of 16 bytes */
  p = (char *) malloc(16);

  return 0;
}
```

- Get on to EWS. Compile the standard way. Then run:

```
> valgrind ./a.out
```

- Can you figure out where the leaks are?

Exercise

- Recall how to use `malloc` for our struct

```
Flight *ptr = (Flight *) malloc(numFlight*sizeof(Flight));
```

- Write a function to read the provided binary file and return a struct containing the n-th flight record. Discard the first n-1.

```
Flight * nth_flight(char *filename, int num_total, int N)
```

- Make sure to free memory!

Next time - important

- So far our use of `malloc` has been to load records or data from a file
- Thus we no longer have to know the sizes at compile time
- Nevertheless `realloc/malloc/free` is cumbersome to keep using
- Need a data structure that takes care of this automatically - enter *linked-lists*.

Time permitting - *key idea*

- Basic idea of a linked list:

```
typedef struct node{
  char *name;
  struct node * next;
}node;
```

- Definition is *recursive*; a node is either
  - *NULL* or
  - Contains a reference to another node

Thursday - learn it from the GOAT!