Dynamic Allocation and Structures

Class 23

Compile Time

- all of the variables we have seen so far have been declared at compile time
- they are written into the program code
- you can see by looking at the program how many variables will exist in the entire program
- this is fine when you know exactly how many variables you will need in the entire program

- but sometimes you do not know in advance how many variables you will need

Run Time

- it is possible to create (and destroy) variables at runtime
- this is called dynamic allocation
- while a program is running, the logic may dictate that a new variable is needed, not specifically known about when the program was written

- there is an area of RAM that is available to be drawn from at runtime
- this area is called the heap

malloc

- a program can allocate a piece of this memory at runtime by using the malloc function
  `int* value = malloc(sizeof(int));`
- this line of code requests enough bytes of memory from the heap to store one int (4 bytes on sand)
- the operating system responds with the address of a 4-byte chunk in the heap
- since malloc returns an address, the return value must be assigned to a pointer

Dynamic Allocation

- a program fragment that uses a dynamically allocated variable

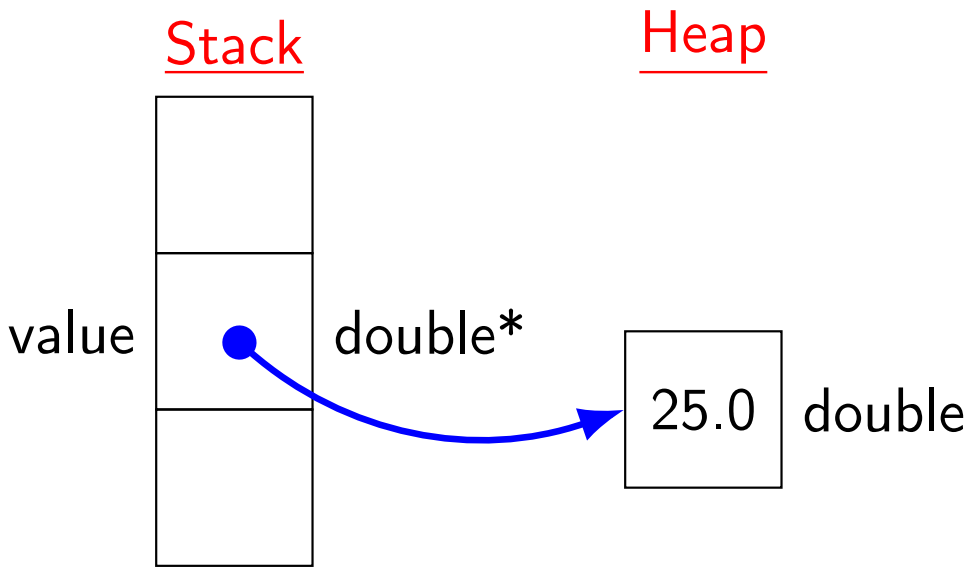
```
double* value = malloc(sizeof(double));
*value = 25.0;
*value *= 2.0;
printf("the value is: %f\n", *value);

50.0
```

Diagramming Dynamic Memory

- even though a computer's memory is just one huge linear list of locations
- we often diagram different regions separately

```
double* value = malloc(sizeof(double));
*value = 25.0;
```



Dynamic Allocation of Arrays

- in reality, allocating a single variable isn't very useful
- the true power of dynamic allocation is in allocating an array at runtime

- the biggest drawback of arrays is that their size is fixed at compile time
- there is a constant tension between not wasting RAM by making an array small and having enough room by making the array big

- with dynamic allocation, we can size an array just right

```
int* values;

printf("how many values do you need to store? ");
scanf("%d", &value_count);
values = malloc(sizeof(int) * value_count);
```

Stack and Heap

- the run-time stack area of memory is where all locally defined variables are located
- the operating system gives it to your program when your program starts running and reclaims it when your program finishes

- the heap is where all dynamically allocated variables are located
- the programmer is responsible for allocating variables on the heap, and the programmer is responsible for de-allocating them before your program finishes

- when you use memory from the heap
- you have to give it back before the program finishes
- failure to do so is called a memory leak

De-Allocating Memory

- the malloc function allocates a piece of memory
- the free function gives it back

```
double* value = malloc(sizeof(double));
```

- must always be followed eventually by:

```
free(value);
```

```
int* values = malloc(sizeof(int) * 50);
```

- must always be followed eventually by:

```
free(values);
```

- every program must have exactly as many calls to malloc as calls to free

Structures

- in C++, a struct is a class, with data and behavior
- in C, a struct only has data
- imagine you wish to build a system for maintaining information about movies
- you might define a Movie structure like this:

```
typedef struct
{
  char[MAXSTRING] title;
  char[MAXSTRING] director;
  unsigned year_released;
  double running_time;
} Movie;
```

- the struct tag name Movie typically starts with an Uppercase letter
- the data fields are variables of types that already exist

A Note on Style

- C has several different ways of defining structs
- K&R awkwardly use one style from the beginning of chapter 6 up through 6.6, and then suddenly switch to the more common style in 6.7
- we will exclusively use the later, more common style, which uses typedef and no structure tag

A Structure Variable

- a struct is a template or a blueprint for a composite variable
- this struct has four fields
- a struct is not a variable, it is a new type
- since it is a type, we can declare or define a variable of this type, using the structure tag as the type name:

```
Movie movie = {"Harry Potter", "Chris Columbus", 2001, 2.53};
```

A Structure Variable

- a struct is a template or a blueprint for a composite variable
- this struct has four fields
- a struct is not a variable, it is a new type
- since it is a type, we can declare or define a variable of this type, using the structure tag as the type name:

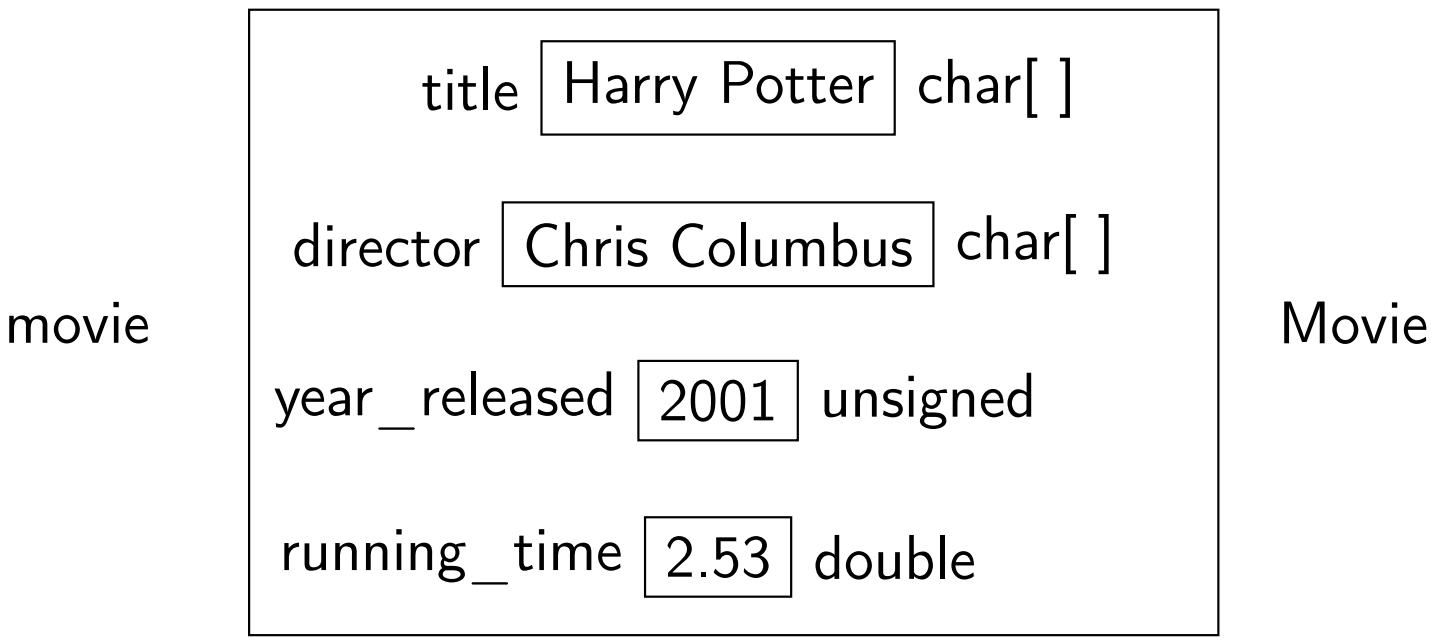
```
Movie movie = {"Harry Potter", "Chris Columbus", 2001, 2.53};
```

- movie is a variable that has four data members

Structure

```
Movie movie {"Harry Potter", "Chris Columbus", 2001, 2.53};
```

- this declares the variable movie of data type Movie
- movie is a composite variable
- we diagram this variable schematically like this:



Accessing Structure Members

- to access individual elements of an array, we use square brackets
- to access a individual structure member, we use the dot operator and dot notation

```
printf("%s\n", movie.title);
```

- note that the dot operator is one of two binary operators that is not surrounded by spaces

Initializing a Struct

- a struct variable can be initialized when it is declared by filling all the fields in order

```
Movie movie = {"Harry Potter", "Chris Columbus", 2001, 2.53};
```

- or by assigning them one-by-one after declaration:

```
Movie movie;
strcpy(movie.director, "Chris Columbus");
movie.year_released = 2001;
strcpy(movie.title, "Harry Potter");
movie.running_time = 2.53;
```

Note: serious security issues with strcpy; don't actually do this

Arrays of Structs

- it is perfectly legal to have a single struct variable
- but the real power of structs comes with collections of structs
- i.e., arrays of structs

- an array of structs is created just like any array
- first you need the struct definition

```
typedef struct
{
  char title[MAXSTRING];
  char director[MAXSTRING];
  unsigned year_released;
  double running_time;
} Movie;
```

Array of Structs

- then you use it to create an array

```
size_t index;
Movie movies[3] = {{"Psycho", "Hitchcock", 1960, 1.82},
                   {"Vertigo", "Hitchcock", 1958, 2.13},
                   {"Repulsion", "Polanski", 1965, 1.75}};

for (index = 0; index < 3; index++)
{
  printf("%s %u\n", movies[index].title, movies[index].year_released);
}
```

```
Psycho 1960
Vertigo 1958
Repulsion 1965
```

Padding

- a note about compiling structs
- every program that uses this movie struct will have a compiler warning

```
foo.c:9:12: warning: padding struct 'Movie' with
    4 bytes to align 'year_released' [-Wpadded]
```

- this is strictly an informational message that indicates the struct has extra "wasted" space between the two members `director` and `year_released`
- the compiler does this for speed

Padding

- adding an extra int member eliminates the warning:

```
struct Movie
{
  char title[MAXSTRING];
  char director[MAXSTRING];
  int dummy;
  unsigned year_released;
  double running_time;
};
```

- but that would do no good and be confusing
- you can suppress the warning message by adding a switch to the compiler: `-Wno-padded`
- or you can just ignore this warning

Passing Structs to Functions

- a struct variable may be passed to a function in three different ways
  1. by value
  2. by pointer
  3. by constant pointer

Passing Structs to Functions

- a struct variable may be passed to a function in three different ways
  1. by value
  2. by pointer
  3. by constant pointer

- each has a purpose

Pass by Value

- passing a struct variable by value makes a copy of the variable in the function
- typically not used, because a struct may be large
- sometimes, used when both of these conditions hold:
  1. the struct is small, no more that a few simple members
  2. changes to the variable are not needed in the calling scope

Pass by Pointer and Constant Pointer

- typically this is done
- pass by constant pointer if no changes are allowed
- pass by pointer if changes are needed in the calling scope

Location of Struct Definition

- if a struct is going to be exclusively used in one function (rare, but theoretically possible) it may be defined within that function, right after any constants
- but almost always, the struct definition will be used in multiple functions and should be defined in global scope, right after global constants, and before function prototypes
- once we start using .h files, struct definitions will typically be there

Copying Struct Variables

- unlike arrays, it is perfectly legal to copy one struct to another in a single statement

```
Movie movie1 = {"Psycho", "Hitchcock", 1960, 1.82};
Movie movie2;

movie2 = movie1;
```

- now movie2 is an exact duplicate of movie1
- copied member by member, byte by byte

Comparing Struct Variables

- like arrays, you cannot compare struct variables with relops
- what would this even mean? `movie1 < movie2`

Pointers to Structure Variables

- a pointer variable can point to a structure location in memory

```
Movie movie = {"Psycho", "Hitchcock", 1960, 1.82};

Movie* mptr = &movie;
```

Pointers to Structure Variables

- a pointer variable can point to a structure location in memory

```
Movie movie = {"Psycho", "Hitchcock", 1960, 1.82};

Movie* mptr = &movie;
```

- immediately there is a problem, however
- to access a member, we want to use the dot operator after dereferencing the pointer:
  `printf("%s\n", *mptr.title);`

Pointers to Structure Variables

- a pointer variable can point to a structure location in memory

```
Movie movie = {"Psycho", "Hitchcock", 1960, 1.82};

Movie* mptr = &movie;
```

- immediately there is a problem, however
- to access a member, we want to use the dot operator after dereferencing the pointer:
  `printf("%s\n", *mptr.title);`
- but this doesn't work, because the precedence of dot is higher than the precedence of dereference
- the statement means: go to the variable mptr, select its title field (which doesn't exist), and then dereference that (which makes no sense) to find the thing to print (which doesn't work at all)

Pointers to Structure Variables

- instead we have to do this:

```
Movie movie = {"Psycho", "Hitchcock", 1960, 1.82};
Movie* mptr = &movie;

printf("%s\n", (*mptr).title);
```

- this syntax works, and you sometimes see it in old code
- considered awkward and old-fashioned
- instead use the dereference-then-select operator `->`

```
printf("%s\n", mptr->title);
```

- this operator means: first dereference mptr, then go to the title field of the thing mptr is pointing to, and print that
- this operator also has no spaces around it

Dynamically Allocating Structures

- with the ability to have pointers to structure variables, we can dynamically allocate them

```
mp = malloc(sizeof movie); /* or malloc(sizeof(Movie)); */
strcpy(mp->title, "Billy Jack");
strcpy(mp->director, "Tom Laughlin");
mp->year_released = 1971;

printf("%s %u\n", mp->title, mp->year_released);
free(mp);
```