1. Dynamic Memory Allocation
   1. Normally, when variables are declared, memory is allocated to them at runtime.
      1. Programmer does not have to worry.
      2. Limiting when one does not know how large an array will be.
      3. C allows programmer to allocate chunks of memory created during runtime as needed
         1. **Dynamically-allocated memory**
   2. Memory chunks created dynamically.
      1. Accessible only by pointer
         1. Pointer is deleted/re-directed 🡪 access to that chunk of memory is lost
   3. Memory management left up to programmer
      1. Keep track of pointer
      2. Free up data when no longer necessary
         1. Otherwise, memory leaks can crash the program.
         2. Memory leaks can still occur if memory is allocated in a loop.
2. The Size to Set Aside
   1. sizeof(): standard C operator to determine how much memory a data type consumes.
      1. Argument = data type.
      2. Returns amount of memory required by data type specified in argument.
      3. Example
         1. sizeof(int)
            1. 4
      4. Useful for structs, as there is no practical way to tell how large a memory block needs to dynamically allocate a struct instance.
   2. malloc(): allocates uninitialized memory.
      1. Stands for *memory allocation*.
      2. Argument = size of chunk of memory.
         1. Insert sizeof() inside malloc()
      3. Returns pointer to the chunk of memory allocated dynamically
      4. Frequently used for structs
      5. Example
         1. struct planet \*planet\_ptr;
         2. planet\_ptr = malloc(sizeof(struct planet));
         3. planet\_ptr points to the chunk of memory type planet
      6. Preferable example
         1. typedef struct planet Planet;
         2. planet\_ptr = malloc(sizeof(Planet));
      7. If malloc() fails, pointer returned will be NULL.
         1. Must always check to make sure NULL was not returned.
         2. If so, redirect or cease execution.
   3. free(): frees up memory that has been dynamically allocated and gives it back to the system.
      1. Example
         1. free(planet\_ptr);
      2. Without freeing, the program will eventually run out memory.
   4. NULL: special pointer value to indicate that the pointer is not pointing to anything.
      1. Trying to dereference NULL 🡪 runtime error
3. Allocating Arrays
   1. After a pointer is declared, the variable can be used like an array.
   2. If bar is a pointer to an array, the following are equivalent.
      1. \*bar;
      2. bar[0]
   3. To allocate arrays, use…
      1. malloc()
      2. calloc()
         1. Contiguous allocation
         2. Used specifically for allocating arrays
         3. Arguments = size of each element, number of elements
         4. Automatically initializes memory to 0
4. Allocating Structures
   1. struct is for dynamic allocation of variables
   2. Must now use arrow operator
      1. Dot operator is irrelevant in dynamically-allocated memory
      2. Name of pointer that points to that structure instance replaces the name of the instance
   3. Example
      1. struct entry{
      2. char name[20];
      3. char due\_date[10];
      4. int priority;
      5. };
      6. typedef struct entry Entry;
      7. typedef Entry \* Entryptr; //defining a pointer data type that points to a variable type Entry – avoids use of asterisk
      8. entryptr ptr = malloc(sizeof(Entry));
      9. strcpy(ptr->name, “Job #3216”);
      10. strcpy(ptr->due\_date, “11-28-2015”);
      11. ptr->priority = 6;
5. Miscellaneous
   1. A pointer type can be defined just like a variable type.
   2. Dot and arrow operators are similar from a practical standpoint
   3. strcpy() was used to set values of members on instances of structures
   4. Code must check for NULL in ptr