Lecture 22

Dynamic Data Structures (I)
Why dynamic?

• We have declared arrays for various data types in our programs
  
  int array[SIZE];
  double array[SIZE];
  char array[SIZE];
  student_t array[SIZE];  /*student_t is a user-defined type*/

• What is common in all these declarations, is that we need to indicate the type and size of the array in the declaration (size either as a constant number or a constant macro, but not a variable).

• Often we don’t know the size of the list we should create before the runtime; Or maybe not even the type.
Why dynamic?

- Often we don’t know the size of the list we should create before the runtime.
- Sometimes we don’t know the type of the list we need to create when writing the program.
- We want to be able to allocate memory for different data types and for different number of values during the runtime (whenever the user asks).
- This memory is allocated dynamically.
- The data structures using dynamic memory can grow or shrink in size while the program is running.
Dynamic Memory Allocation

- We can use four functions from stdlib.h library for dynamic memory allocation:
  - `malloc` – allocates “size” bytes of memory
  - `calloc` – allocates memory for “n elements” of size “size” each
  - `realloc` – changes the size of a previously allocated memory block to “size”
  - `free` – deallocates a memory previously allocated by the three functions above
On the sidenote – void*

- void* is a pointer with no data type
- void* is a generic pointer.
- void* can refer to data of any type, but to dereference it (to use the indirection operator) we need to cast it to the proper type.

```c
int a = 3;
void* vp = &a;  /* void* refers to an int – it can refer to any other type */
printf("%d\n", *vp);  /* compilation error – needs to be type-casted*/
printf("%d\n", *(int*)vp);  /* prints 3 – typecast void* to int* and then indirect it*/
```
malloc

void* malloc (size_t size);

- malloc allocates a memory block of size “size” and returns a pointer (void*) to the beginning of that block.
- size_t is a positive integer number
- We can pass in any positive integer as the argument to malloc, however, we want to store a variable of a known size in it, so we prefer to use the sizeof() operator.
  
  malloc(8) is valid but we prefer malloc(sizeof(int))
- It returns a generic pointer (void*) to the beginning of the allocated memory block. We need to type cast it before use.
- If memory allocation is unsuccessful, malloc will return NULL. Always check for NULL pointer after malloc.
Every dynamically allocated memory needs to be freed manually, otherwise it will stay on the memory, wasting space. (memory leak)

Free deallocates the dynamically allocated memory that the pointer is pointing to.

After freeing a memory, you can access that space, but the value in it will be garbage.

Deallocation a memory only if you are certain you won’t need it anymore.

After freeing a memory, set its pointer to NULL to avoid later reference to an already freed block. (good practice)
Example

• Write a program that asks the user whether the memory for an integer needs to be allocated. If the user answers yes, then the memory will be allocated and a value will be stored in it.

• If memory is allocated, remember to deallocate it before the program ends.
```c
int main()
{
    char c = '\0';
    int* nump;
    printf("do you want to allocate memory for an integer? (y/n)\n");
    scanf("%c", &c);
    switch (c)
    {
    case 'n':
        printf("no memory allocated\n");
        break;
    case 'y':
        nump = (int*) malloc( sizeof(int) );
        if (nump != NULL) {
            printf("memory allocated\n");
            *nump = 4;
            printf("freeing the memory...\n");
            free(nump);
            nump = NULL;
        }
        break;
    }
    break;
}
```

Example

This will allocate a memory of size an int and return a pointer to that memory cell.

Deallocates the memory pointed to by nump.
calloc

void* calloc(size_t count, size_t size)

- calloc will allocate memory for “count” number of elements, each of size “size” (an array of size “count” where each element can be of size “size”)
- It also initializes each element to zero
- size_t is a positive integer number
- Similar to malloc, we will use sizeof() operator to decide the size we need.
- It returns a generic pointer (void*) to the beginning of the allocated memory block. We need to type cast it before use.
- Freeing the memory allocated using calloc, will free the whole block (all of the count elements)
- If memory allocation is unsuccessful returns NULL. Always check for NULL pointer after call to calloc.
Example

• Ask the user to input the size of an array.
• Create an int array of the specific size
• In element i of the array store the value i+1
• Print the array
• Don’t forget to deallocate the memory
int main()
{
    int numElements, i;
    int* array;
    printf("What is the size of the array: ");
    scanf("%d", &numElements);
    array = (int*)calloc(numElements, sizeof(int));
    if (array != NULL) {
        for (i = 0; i < numElements; i++)
            array[i] = i + 1;
    } /* finished initializing */
    for (i = 0; i < numElements; i++)
        printf("array[%d] = %d
", i, array[i]);
    } /* finished printing */
    free(array);
    array = NULL;
}
Checks if the calloc was successful
malloc vs calloc

• malloc and calloc can be used interchangeably
• However malloc does not initialize the allocated memory while calloc initializes all to zero
• malloc receives one parameter (size)
• calloc receives two parameters (count and size)

malloc( sizeof(int) * n ) ≈ calloc(n, sizeof(int) )
Example

• Re-write the previous example using malloc
  ▪ Ask the user to input the size of an array.
  ▪ Create an int array of the specified size.
  ▪ In element i of the array store the value i+1.
  ▪ Print the array.
  ▪ Don’t forget to deallocate the memory.
int main()
{
    int numElements, i;
    int* array;
    printf("What is the size of the array: ");
    scanf("%d", &numElements);
    array = (int*)malloc( sizeof(int) * numElements );
    if (array != NULL) {
        for (i = 0; i < numElements; i++)
        {
            array[i] = i + 1;
        }
        for (i = 0; i < numElements; i++)
        {
            printf("array[%d] = %d\n", i, array[i]);
        }
        free(array);
        array = NULL;
    }
}
void* realloc (pointer, size);

- Realloc allows for changing the size of a previously allocated dynamic memory.
- If we request a smaller size than it currently has, it will free the portion of the memory we don’t need
- If we request a larger size than what we asked for previously:
  - If there is enough empty space right after the current block, will append it to the current block
  - If there the memory next to the current block is used, then finds a new empty block of the given “size” and copies the current values into that new block, returning a pointer to it.
Example

• Write a program that asks for the size of an array, reads this size, creates an array of that size and initializes its index \( i \) to contain the number \( i+1 \).

• After printing the array, it asks for a new size and resizes and reinitializes the array.

• Repeat this resizing until the user asks for size 0 array.
```c
int main()
{
    int size, i;
    int* array = NULL;
    do
    {
        printf("Enter a size for the array: ");
        scanf("%d", &size);
        array = (int*) realloc(array, size * sizeof(int));
        for (i = 0; i < size; i++)
        {
            array[i] = i + 1;
        }
        for (i = 0; i < size; i++)
        {
            printf("array[%d]=%d\n", i, array[i]);
        }
    } while (size > 0);
    free(array);
    array = NULL;
    return 0;
}
```
Note on Free

- Be careful about freeing an allocated memory.
- If there are multiple pointers referring to the same block, freeing one of them will cause all of them to lose their values.

```c
int main()
{
    int* ip, *ipc;
    ip = (int*)malloc(sizeof(int)); /*check if ip is NULL after this*/
    *ip = 5;
    ipc = ip;
    printf("*ip=%d t *ipc=%d\n", *ip, *ipc); /*both print 5*/
    free(ip);
    ip = NULL;
    printf("ip freed, *ipc=%d\n", *ipc); /* *ipc is garbage */
    return 0;
}
```

- `ip` and `ipc` refer to the same memory
- We free only `ip`
- We have lost `ipc` as well
Example

• We can give any positive integer as size to malloc/calloc/realloc.
• sizeof operator works on all data types
• Result: we can dynamically allocate memory for any data type.
• Example: declare a struct student_t with the components: name, ID, and gpa.
• Ask the user for the number of students in the class.
• Create an array of students of the given size.
typedef struct {
    char name[20];
    int id;
    double gpa;
} student_t;

void scanStudent(student_t* sp);

int main() {
    student_t* sp = NULL;
    int studentCount = 0, i = 0;
    printf("how many students are in the class: ");
    scanf("%d", &studentCount);
    sp = (student_t*)calloc(studentCount, sizeof(student_t));
    if (sp != NULL)
    {
        for (i = 0; i < studentCount; i++)
        {
            scanStudent(&sp[i]);
        }
    }
    free(sp);
    sp = NULL;
    return 0;
}
Stack vs Heap

- When you start running a program, there is a memory associated with it. This memory is organized as:

  - **Stack**: Local variables for functions are stored here and the stack grows upward as we make more function calls.
  - **Heap**: Dynamic memory requests are assigned from the heap and grows downward as we request more memory.
  - **Data segment**: Global variables are stored here.
  - **Code segment**: Program instructions are stored here.
Stack vs heap

• The stack is neat
  ▪ At the beginning of each function we already know the type and count of local variables for that function
  ▪ Just enough memory space is reserved on top of the stack as we call a function.
  ▪ When we return from a function the memory for that function is popped out from the stack.
  ▪ Memory allocation/deallocation is done automatically

• Heap can be unorderly
  ▪ You can request memory dynamically
  ▪ You can request a new memory block
  ▪ If you free the first block of memory, there will be a gap in place of it. (which might be re-used later)
  ▪ It should be managed by the programmer (don’t forget to free the memory. – after freeing it, all references to it will be lost).
References