CSC215 Lecture

Memory Management

Outline

- Static vs Dynamic Allocation
- Dynamic allocation functions
  `malloc, realloc, calloc, free`
- Implementation
- Common errors

Static Allocation

- Allocation of memory at compile-time
  - before the associated program is executed
- Let’s say we need a list of 1000 names:
  - We can create an array statically
    char names[1000][20]
  - allocates 20000 bytes at compile time
  - wastes space
  - restricts the size of the names

Dynamic allocation of memory

- Heap is a chunk of memory that users can use to dynamically allocated memory
  - Lasts until freed, or program exits.
- Allocate memory during runtime as needed

  `#include <stdlib.h>`
- Use sizeof number to return the number of bytes of a data type.
- To reserve a specified amount of free memory and returns a void pointer to it, use:
  - `malloc`
  - `calloc`
  - `Realloc`
- To release a previously allocated memory block, use:
  - `free`

Dynamic Allocation: `malloc`

- C library function allocates the requested memory and returns a pointer to it

  ```
  void *malloc(size_t size)
  ```
  - `size_t: unsigned integer type`
  - size: the size of the requested memory block, in bytes
  - return value: a pointer to the allocated memory, or NULL if the request fails
  - memory block is not cleared (undefined)

- Example:

  ```
  char *str = (char *) malloc(3*sizeof(char));
  *str = 'O';
  *(str+1) = 'K';
  *(str+2) = '\0';
  ```

Dynamic Allocation: `realloc`

- C library function attempts to resize the memory block pointed to by a pointer

  `void *realloc(void *ptr, size_t size)`
  - ptr: a previously allocated pointer (using malloc, calloc or realloc)
    - if `NULL`, a new block is allocated ⇔ `malloc`
  - size: the total size of the requested memory block, in bytes
    - if `0`, the memory pointed to by ptr is freed ⇔ `free`
  - return value: a pointer to the allocated memory, or NULL if the request fails
  - may move the memory block to a new location

- Example:

```
char *str = (char *) malloc( 3 * sizeof(char) );
*str = 'H';        *(str+1) = 'i';        *(str+2) = '\0';

str = (char *) realloc( str , 6 * sizeof(char) );
*(str+1) = 'e';   *(str+2) = 'l';        *(str+3) = 'l';
*(str+4) = 'o';   *(str+5) = '\0';
```

What is considered
a bad practice here?

Dynamic Allocation: `calloc`

- Dynamically allocating arrays:
  - allows the user to avoid fixing array size at declaration
  - use malloc to allocate memory for array when needed:

```
int *a = (int *)malloc(sizeof(int)*10);
a[0]=1;
```

- Alternatively, use:

```
void *calloc(size_t nitems, size_t size)
```

  - nittems: the number of elements to be allocated
  - size: the size of the requested memory block, in bytes
  - return value: a pointer to the allocated memory, or NULL if the request fails
  - sets allocated memory to 0s

- Example:

```
int size;    char *s;
printf("How many characters?\n"); scanf("%d", &size);
s = (char *)calloc(size+1, 1);
printf("type string\n");  gets(s);
```

Dynamic Deallocation: `free`

- C library function deallocates the memory previously allocated
    - by a call to calloc, malloc, or realloc

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

    - ptr : the pointer to a memory block previously allocated with malloc, calloc or realloc to be deallocated
    - If a null pointer is passed as argument, no action occurs.
- Can only be used on pointers that are dynamically allocated
- It is an error to free:
    - A pointer that has already been freed
    - Any memory address that has not been directly returned by a dynamic memory allocation routine
- Example:

```
char *str = (char *)malloc(3*sizeof(char));
/* use str */
free(str);
```

How It Is Done

- Best-fit method:
  an area with m bytes is selected, where m is the smallest available chunk of contiguous memory equal to or larger than n.
- First-fit method:
  returns the first chunk encountered containing n or more bytes.
- Prevention of fragmentation
  a memory manager may allocate chunks that are larger than the requested size if the space remaining is too small to be useful.
- When free is called:
  returns chunks to the available space list as soon as they become free and consolidate adjacent areas

Common Dynamic Allocation Errors

- **Initialization errors**
  do not assume memory returned by malloc and realloc to be filled with zeros
- **Failing to check return values**
  since memory is a limited resource, allocation is not always guaranteed to succeed
- **Memory leak**
  Forgetting to call free when the allocated memory is no more needed
- **Writing to already freed memory**
  if pointer is not set to NULL it is still possible to read/write from where it points to
- **Freeing the same memory multiple times**
  may corrupt data structure
- **Improper use of allocation functions**
  malloc(0): insure non-zero length

Example

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int input, n, count = 0;
  int *numbers = NULL, *more_numbers = NULL;
  do {
    printf ("Enter an integer (0 to end): ");     scanf("%d", &input);
    count++;
    more_numbers = (int*)realloc(numbers, count * sizeof(int));
    if (more_numbers!=NULL) {
      numbers = more_numbers;
      numbers[count-1]=input;
    else {
      free(numbers);
      puts("Error (re)allocating memory");
      return 1;
    }
  } while (input!=0);
  printf ("Numbers entered: ");
  for (n=0;n<count;n++) printf ("%d ",numbers[n]);
  free (numbers);
  return 0;
```

Example: mat.c

```
#include <stdio.h>
#include <stdlib.h>
#include "mat.h"

int** get_matrix(int rows, int cols){
  int  i, **matrix;
  if (matrix = (int**)malloc(rows*sizeof(int*)))
    if (matrix[0] = (int*)calloc(rows*cols,sizeof(int))){
      for (i=1; i<rows; i++)
        matrix[i] = matrix[0] + cols * i;
      return matrix;
    }
  return NULL;
}
```

Compare with:

```
if (matrix =
      (int**) malloc(rows*sizeof(int*)))
  for (i=0; i<rows; i++)
    if(!(matrix[i] =
      (int*) calloc(cols,sizeof(int))))
      return NULL;
   return matrix;
```

```
void free_matrix(int** m){
  free(m[0]);
  free(m);
}
```

Compare with:

```
void free_matrix(int*** m){
  free(*m[0]);
  free(*m);
  *m = NULL;
}
```

Example: mat.c

```
void fill_matrix(int** m, int rows, int cols){
  int i, j;
  for (i=0; i < rows; i++)
    for (j=0; j < cols; j++){
      printf("Enter element [%d, %d]:", i, j); scanf("%d", &m[i][j]);
    }
}
void print_matrix(int** m, int rows, int cols){
  int i, j;
  for (i=0; i < rows; i++){
    for (j=0; j < cols; j++) printf("%d\t", m[i][j]);
    printf("\n");
  }
}
int** transpose(int** m, int rows, int cols){
  int i, j, **t = get_matrix(cols, rows);
  for (i=0; i < rows; i++)
    for (j=0; j < cols; j++) t[j][i] = m[i][j];
  return t;
}
```

Example: mat.h

```
#if !defined MAT
#define MAT

int** get_matrix(int, int);

void free_matrix(int**);       /* OR */  void free_matrix(int***);

void fill_matrix(int**, int, int);

void print_matrix(int**, int, int);

int** transpose(int**, int, int);

#endif
```

Example: test.c

```
#include <stdio.h>
#include "mat.h"

int main(){
  int r, c;
  printf("How many rows? "); scanf("%d", &r);
  printf("How many columns? "); scanf("%d", &c);

  int** mat = get_matrix(r, c);

  fill_matrix(mat, r, c);
  print_matrix(mat, r, c);

  int** tra = transpose(mat, r, c);
  print_matrix(tra, c, r);

  free_matrix(mat);       /* OR */              free_matrix(&mat);
  free_matrix(tra);                             free_matrix(&tra);
  return 0;
}
```