Allocating Memory at Runtime

Dynamic Memory Allocation

Up until now, all memory allocation has been *static* or *automatic*

- The programmer (you) didn't have to worry about finding available memory; the compiler did it for you.
- You also didn't have to worry about releasing the memory when you were finished with it; it happened automatically.
- Static memory allocation is easy and effortless, but it has limitations.
- Dynamic memory allocation is under complete control of the programmer.
- This means that you will be responsible for allocating and de-allocating memory.
- Failing to understand how to manage the memory yourself will lead to programs that behave badly (crash).

The two primary functions required for dynamic memory management are *malloc* and *free*.

```
void *malloc( size_t size ); /* Allocate a block of memory   */
void free( void *pointer );  /* Deallocate a block of memory */
```

To use `malloc` and `free`:

```
#include <stdlib.h> /* malloc, free */
```

The argument to *malloc* is the number of bytes to allocate:

```
char *pc = malloc(10); /* allocate memory for 10 chars */
int *pi = malloc(40);  /* allocate memory for 10 ints  */
```

10 chars (10 bytes) **10 ints (40 bytes)**

Notice that there is no type information associated with `malloc`, so the return from `malloc` may need to be cast to the correct type:

```
  /* Casting the return from malloc to the proper type */
char *pc = (char *) malloc(10); /* allocate memory for 10 chars */
int *pi = (int *) malloc(40);   /* allocate memory for 10 ints  */
```

You should never hard-code the size of the data types, since they may change. Do this instead:

```
  /* Proper memory allocation for 10 chars */
char *pc = (char *) malloc(10 * sizeof(char));

  /* Proper memory allocation for 10 ints  */
int *pi = (int *) malloc(10 * sizeof(int));
```

If the allocation fails, NULL is returned so you should check the pointer after calling *malloc*.

```
  /* Allocate some memory for a string */
char *pc = (char *) malloc(10 * sizeof(char));

  /* If the memory allocation was successful */
if (pc != NULL)
{
  strcpy(pc, "Digipen"); /* Copy some text into the memory */
  printf("%s\n", pc);    /* Print out the text             */
  free(pc);              /* Release the memory             */
}
else
  printf("Memory allocation failed!\n");
```

After allocation **After `strcpy`**

Notes:

- The memory allocated by `malloc` is uninitialized (random values).
- You need to initialize the memory yourself.
- If you want all of the memory to be set to zeros, you can use the `calloc` function instead:

  ```
  void *calloc( size_t num, size_t size ); /* Allocates memory and sets all bytes to 0 */
  ```

- Notice that `calloc` has two parameters: 1) the number of elements and 2) the size of each element.

  ```
    /* Allocate and initialize 10 chars to 0 */
  char *pc = (char *) calloc(10, sizeof(char));
  ```

  After calling `calloc`:

- `malloc` and `calloc` are essentially the same, but, for obvious reasons, `malloc` is faster.
- If you are going to set the values of the memory yourself **DO NOT** use `calloc`. (It's an unecessary waste of time.)

Examples:

Common usage

```
int main(void)
{
  int SIZE = 10;
  int *pi;

    /* allocate memory */
  pi = (int *)malloc(SIZE * sizeof(int));

    /* check for valid pointer */
  if (!pi)
  {
    printf("Failed to allocate memory.\n");
    return -1;
  }

  /* do stuff */

    /* free memory */
  free(pi);

  return 0;
}
```

Set and check in one statement

```
int main(void)
{
  int SIZE = 1000 * 1000 * 1000; /* 1 billion */
  int *pi;

    /* allocate and check memory */
  if ((pi = (int *)malloc(SIZE * sizeof(int))) == NULL)
  {
    printf("Failed to allocate memory.\n");
    return -1;
  }

  /* do stuff */

    /* free memory */
  free(pi);

  return 0;
}
```

Accessing the allocated block:

```
void test_malloc(void)
{
  int SIZE = 10;
  int i, *pi;

    /* allocate memory */
  if ((pi = (int *)malloc(SIZE * sizeof(int))) == NULL)
  {
    printf("Failed to allocate memory.\n");
    return;
  }

    /* using pointer notation */
  for (i = 0; i < SIZE; i++)
    *(pi + i) = i;

    /* using subscripting */
  for (i = 0; i < SIZE; i++)
    pi[i] = i;

  for (i = 0; i < SIZE; i++)
    printf("%i \n", *(pi + i));

    /* free memory */
  free(pi);
}
```

By now it should be clear why we learned that pointers can be used to access array elements. With dynamic memory allocation, there are no *named* arrays, just pointers to contiguous (array-like) memory and pointers *must* be used.

Dynamically Allocated Structures

Revisiting our `FILEINFO` example:

```
#define MAX_PATH 12

struct DATE
{
  int month;
  int day;
  int year;
};

struct TIME
{
  int hours;
  int minutes;
  int seconds;
};

struct DATETIME
{
  struct DATE date;
  struct TIME time;
};

struct FILEINFO
{
  int length;
  char name[MAX_PATH];
  struct DATETIME dt;
};
```

Function to print a single FILEINFO structure:

```
void PrintFileInfo(const struct FILEINFO *fi)
{
  printf("Name: %s\n", fi->name);
  printf("Size: %i\n", fi->length);
  printf("Time: %2i:%02i:%02i\n", fi->dt.time.hours, fi->dt.time.minutes, fi->dt.time.seconds);
  printf("Date: %i/%i/%i\n", fi->dt.date.month, fi->dt.date.day, fi->dt.date.year);
}
```

Dynamically allocate a FILEINFO structure and print it:

```
void f14(void)
{
    /* Pointer to a FILEINFO struct (The 1 is redundant but instructive) */
  struct FILEINFO *pfi = (struct FILEINFO *)malloc(1 * sizeof(struct FILEINFO));

    /* Check that the allocation succeeded */
    /* Set the fields of the struct ....    */

  PrintFileInfo(pfi); /* Print the fields */
  free(pfi);          /* Free the memory  */
}
```

View of memory after allocation:

Function to print an array of FILEINFO structures:

```
void PrintFileInfos(const struct FILEINFO *records, int count)
{
  int i;
  for (i = 0; i < count; i++)
    PrintFileInfo(records++);
}
```

Remember, for function parameters, we could have written the function like this:

```
  /* Use array notation instead of pointer notation */
void PrintFileInfos(const struct FILEINFO records[], int count)
{
  . . .
}
```

```
#include <assert.h> /* assert */

#define SIZE 10
#define MAX_PATH 12

void TestHeapStruct(void)
{
  int i;
  struct FILEINFO *pfi;
  struct FILEINFO *saved;

    /* Allocate and initialize all fields of all structs to 0 */
  pfi = (struct FILEINFO *)calloc(SIZE, sizeof(struct FILEINFO));

  assert(pfi != NULL); /* Check that it was successful */
  saved = pfi;         /* Save pointer for later...    */

    /* Set the date and name of each structure */
  for (i = 0; i < SIZE; i++)
  {
    char name[MAX_PATH];

      /* Format dates from 12/1/2019 through 12/10/2019 */
    pfi->dt.date.month = 12;
    pfi->dt.date.day = i + 1;
    pfi->dt.date.year = 2019;

      /* Format and store the filenames (foo-1.txt through foo-10.txt) */
    sprintf(name, "foo-%i.txt", i + 1);
    strcpy(pfi->name, name);

      /* Point to next FILEINFO struct in the array */
    pfi++;
  }

    /* Reset pointer to beginning */
  pfi = saved;

    /* Print info */
  PrintFileInfos(pfi, SIZE);

    /* Release the memory */
  free(pfi);
}
```

Output:

```
Name: foo-1.txt
Size: 0
Time:  0:00:00
Date: 12/1/2019
Name: foo-2.txt
Size: 0
Time:  0:00:00
Date: 12/2/2019
Name: foo-3.txt
Size: 0
Time:  0:00:00
Date: 12/3/2019
Name: foo-4.txt
Size: 0
Time:  0:00:00
Date: 12/4/2019
Name: foo-5.txt
Size: 0
Time:  0:00:00
Date: 12/5/2019
Name: foo-6.txt
Size: 0
Time:  0:00:00
Date: 12/6/2019
Name: foo-7.txt
Size: 0
Time:  0:00:00
Date: 12/7/2019
Name: foo-8.txt
Size: 0
Time:  0:00:00
Date: 12/8/2019
Name: foo-9.txt
Size: 0
Time:  0:00:00
Date: 12/9/2019
Name: foo-10.txt
Size: 0
Time:  0:00:00
Date: 12/10/2019
```

Note: do review pointer arithmetic.

Note:

- In the code above, we didn't have to reset the pointer. We could have just used the *saved* pointer:

```
  /* Print info */
PrintFileInfos(saved, SIZE);

  /* Release the memory */
free(saved);
```

This is simply because *saved* has the same value that *pfi* had originally.

- Also, failure to save the original value of *pfi* could pose a serious problem. You may have no way to get back to the original address so that you can free the memory.

Summary

Summary for `malloc`

- The most common uses for dynamic memory allocation are arrays and structures. You will rarely allocate a single **`int`**, **`char`**, **`double`** dynamically, but you can, of course.
- Using memory allocated by `malloc` is no different than using memory allocated (statically) by the compiler.
- Like static allocation of arrays, there are no range checks on the subscripts, so the programmer has to be careful.
- If you try to access memory outside the block that you allocated, the behavior of your program is undefined.

Summary for `free`

- Forgetting to call `free` will result in a *memory leak*, which can lead to your program running out of memory. Memory leaks are **very, very bad**.
- Calling `free` twice on a pointer renders your entire program undefined. It may cause the program to crash or it may not. But the results of the running program will now be undefined (random).
- Calling `free` on a pointer not pointing to the beginning of memory allocated by `malloc` will also render your entire program undefined. In other words, the address you pass to `free` **MUST** be an address that you was returned by `malloc` (or `calloc`).
- If you allocate memory dynamically, you must call `free` on it at some point.
- Don't access memory after it has been freed as strange problems may occur.
- There are many other problems associated with dynamically allocated memory that you will discover in the coming semesters.