CS 210 **November 8, 2016**
Day 19

Chapter 13 Dynamic Allocation

- dynamic data structure
  - a structure that can expand and contract as a program executes
- nodes
  - dynamically allocated structures that are linked together to form a composite structure

Review of pointers.

Pointers to variables

Reference	Explanation	Value
num	Direct value of num	3
nump	Direct value of nump	Pointer to location containing 3
*nump	Indirect value of nump	3

```
int num;            //num is an integer
int *nump;          //nump is a pointer to an integer
nump = &num;        //Set nump to the address of num
*nump = 3;          // *nump is used to indirectly access num
```

Pointers to Functions
We can also have pointers to functions. We can pass a function as an argument to another function. In this case the function name is represented as a pointer to a function.

The program below passes two different functions to another function for evaluation. The functions Square and Cube are passed as functions to a third function called Evaluate. In this case F is just a pointer to the function being passed.

The C-compiler will also accept

```
void Evaluate(double (*F)(double d),double d1,double d2)
```

if you want to make it more explicit that you are passing a pointer to a function.

```c
#include<stdio.h>
/* This program passes a function as a parameter
     to another function for evaluation.
*/
double Square(double d);
double Cube(double d);
void Evaluate(double F(double d), double d1, double d2);
int main()
{
   double d1 = 9;
   double d2 = 12;
   Evaluate(Square, d1, d2);
   Evaluate(Cube, d1, d2);
}
void Evaluate(double F(double d),double d1,double d2)
{
    printf("F(%5.1f) = %5.1f\n", d1, F(d1));
    printf("F(%5.1f) = %5.1f\n", d2, F(d2));
}

double Square(double d)
{
    return d*d;
}
double Cube(double d)
{
    return d*d*d;
}
```

Pointers to Arrays and Strings
When we declare an array:
`double num[12];`

The C-compilers interprets the array name as the address of the first element in the array. In this case `num` is a pointer to the array. By default arrays are passed by reference.

Both of these are syntactically correct and produce the same result

```c
#include<stdio.h>
void MyFunction(int a[], char c[]);
int main()
{
    int a[] = {1, 2, 3, 4, 5};
    char c[] = "Hello Mom!";
    MyFunction(a, c);
}

void MyFunction(int a[],char c[])
{
    int i;
    for(i=0;i<sizeof(a);i++)
       printf("%d, ", a[i]);
    printf("\n");
    for(i=0;c[i] != 0;i++)
       printf("%c", c[i]);
    printf("\n");
}
```

```c
#include<stdio.h>
void MyFunction(int *a, char *c);
int main()
{
    int a[] = {1, 2, 3, 4, 5};
    char c[] = "Hello Mom!";
    MyFunction(&a[0], &c[0]);
}
void MyFunction(int *a,char *c)
{
    int i;
    for(i=0;i<sizeof(a);i++)
       printf("%d, ", a[i]);
    printf("\n");
    for(i=0;c[i] != 0;i++)
       printf("%c", c[i]);
    printf("\n");
}
```

Dynamic Memory Allocation

When you compile a program the compiler calculates the amount of memory space your program will need – including the stack space and the space for all of the variables. When your program is loaded the needed memory is allocated by the operating system. This is a static allocation for the duration of your program.

If your program tries to redimension an array or it goes outside of an array boundary, chances are good that it will get outside of its allocated memory. This will trigger an operating system fault and your program will be halted.

To allow a user to redimension an array, for example we need a way to make a call to the operating system as the program is running and request additional memory. This is called a dynamic allocation since it occurs after the program has started running.

In C there is a special function called *malloc* which does dynamic memory allocation. The syntax is:

```
malloc(sizeof(int));
```

This function returns a pointer to the memory block allocated – in this case just enough for one integer. In practice we would write:

```
int *nump;
nump = malloc(sizeof(int));
```

The memory allocated is located in the systems read/write memory in a location referred to as the *heap*.

malloc can be used on any built-in or user defined type (i.e. structures)

```c
#include<stdio.h>
#include<stdlib.h>   //needed for malloc
int main()
{
    int i = 5;
    int *iPtr;
    iPtr = (int *)malloc(sizeof(int));
    *iPtr = i + 9;
    printf("i = %d, *iPtr = %d\n ", i, *iPtr);
}
```

```
Prints:
i = 5, *iPtr = 14
Press any key to continue . . .
```

Or we can use malloc with a typedef

```c
typedef struct
{
   double x;
   double y;
}point_t;
#include<stdio.h>
#include<stdlib.h>   //needed for malloc
int main()
{
    int i = 5;
    int *iPtr;
    iPtr = (int *)malloc(sizeof(int));
    *iPtr = i + 9;
    printf("i = %d, *iPtr = %d\n ", i, *iPtr);
    point_t *p1;
    p1 = (point_t *)malloc(sizeof(point_t));
    (*p1).x = 5.4;
    (*p1).y = 9.1;
    //Alternatively we can write
    p1 -> x = 5.4;
    p1 -> y = 0.1;
}
```

Note that `(*p1).x` is the same notation as `p1 -> x`

Dynamic Array Allocation

It is a little unusual to dynamically allocate a single variable or even a single typedef. The common occurrence is the dynamic allocation of an array.

C uses the function `calloc` (continuous memory allocation) to allocate arrays.
The syntax for creating an array of 20 ints is:

```
int *aPtr;
aPtr = (int *)calloc(20, sizeof(int));
```

Here is an example which asks for an array size, creates an array of that size using calloc, and prints the array forward and backward.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i, aSize;
    int *a;
    printf("Enter the array size... ");
    scanf_s("%d", &aSize);
    a = (int *)calloc(aSize, sizeof(int));
    for(i=0;i<aSize;i++)
       a[i] = i;
    for(i=0;i<aSize;i++)
       printf("%d, %d\n", a[i], a[aSize-i-1]);
}
/*Prints the following
Enter the array size... 10
0, 9
1, 8
2, 7
3, 6
4, 5
5, 4
6, 3
7, 2
8, 1
9, 0
Press any key to continue . . .
```

Returning memory to the heap

A function called free will release allocated memory back to the heap so it can be used elsewhere.

```
free(a);  //deallocates the array in the program example above
```

CS 210 **November 8, 2016**
Dynamic Memory Allocation

The following program reads a txt file named MyData.txt and prints it to the console.
Part 1:
Copy the file "MyData.txt" from the web site and place it in the same directory as your c-code source file. Run the program below and verify that it can read and print the data in the file.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int i, status, dataIn, err;
    FILE *inp;  //Declare pointers to in file
    err = fopen_s(&inp, "MyData.txt", "r");
    status = fscanf_s(inp, "%d", &dataIn);
       //read the file and print to console
    while(status == 1)
       {
              printf("%d\n", dataIn);
        status = fscanf_s(inp, "%d", &dataIn);
       }
    fclose(inp);
}
```

Part 2:
Modify the program above to read the first line of the file which contains the number of entries that follow. Use `calloc` to allocate an integer array to hold the data (not including the first line). Print the data from the array in reverse order on a single line with each data item separated by a comma.

Allow your instructor to verify that your program works correctly and turn in a hard copy of your source file.