

C programming: Dynamic Memory Allocation, Structs and Other Features of C

Tying up loose ends

Remember that:

- You can declare multiple pointers on a line, but you need a * for each one
- There is a difference between passing by value and passing by reference. How is this difference manifest in C? How is it manifest in Java?
- Pointers in C are very flexible. We can even have pointers to functions. E.g.:

```
int f(char); // a function that takes a char and returns an int
int (*pf) (int) = &f; // a pointer to f
```

Types of Memory: Stack vs Heap

All local variables and parameters of a function are declared in a part of memory called the stack. Can you guess why it is called the stack?

On the other hand, global variables and dynamically allocated memory comes from a different part of memory called the heap.

Dynamic Memory Allocation

So far, we have only looked at static memory allocation. In static memory allocation, variables are allocated on the stack and they go away when the function returns. When we allocate memory dynamically, memory is allocated from the heap. In C++ and Java, memory is allocated dynamically using the *new* operator. In C, it is allocated using the *malloc()* function.

In Java, the garbage collector automatically gets rid of dynamically allocated memory that is no longer in use. In C and C++, **you are required to manually free dynamically allocated memory that is no longer in use.** In C, this is done using the *free()* function. In C++, it is done using the *delete* operator. If you do not free up dynamically allocated memory, you have what is called a *memory leak*, and your computer may run out of memory while running your program.

For example, suppose we want to create an array to hold the grades of the students in a class, but we don't know how many students are in the class until we are told.

```
int specifyGrades(int n)
{
    int i;
```

```

// allocate the grades array
int* grades = (int*) malloc(n*sizeof(int));

// populate the grades array
for (i=0; i<n; i++)
    grades[i] = (int) (rand()*(100.0 / RAND_MAX));

// print out the grades array
for (i=0; i<n; i++)
    printf("Student %d's got: %d% \n", i, grades[i]);

// free the grades array
free(grades);
}

```

The sizeof() operator returns the size of a data type or a variable in bytes. For example sizeof(int) will return 4. In the example above, the malloc() function is used to allocate enough memory to hold n ints. The total amount of memory needed is $n*4$ bytes. Note that the return type of malloc is void*. This can be interpreted as “a pointer to anything” or “a pointer to raw memory”. To use the allocated memory, we need to cast it to our desired type. In our case, the desired type is a pointer to an int.

Aside: Two related functions are **calloc**, and **realloc**. Learn about these using the man command.

Aside:

When we use the scanf function we introduced in the last lecture, we are required to provide the address of the variable / buffer where the read input must be stored. Thus, the following code reads a decimal integer from the console and stores it in the variable myNum.

```

int myNum;
scanf("%d", &myNum);

```

Note: The following code is incorrect. Why?

```

int myNum;
scanf("%d", myNum);

```



INCORRECT

Very important: When you use scanf, the memory in which the input will be stored must already have been allocated, either statically or dynamically. Otherwise, you will get a *segmentation fault* (an error that occurs when your code accesses an invalid region of memory). Examine the following code fragments. In the correct segments, what is the result? Can you find the problem with the incorrect fragments?

```

int myNum;
int* pMyNum = &myNum;
scanf("%d", pMyNum);

```



CORRECT

```
int myNum;
int* pMyNum;
scanf("%d", pMyNum);
```

 **INCORRECT**

```
char myChar;
scanf("%c", &myChar);
```

 **CORRECT**

```
char* myChar;
scanf("%c", myChar);
```

 **INCORRECT**

```
char val[10];
scanf("%s", val);
```

 **CORRECT**

```
char* val = (char*) malloc(10*sizeof(char));
scanf("%s", val);
free(val);
```

 **CORRECT**

```
char* val = (char*) malloc(10*sizeof(char));
scanf("%s", &val);
free(val);
```

 **INCORRECT**

```
char* val;
scanf("%s", val);
```

 **INCORRECT**

Initialization

Initialization of variables is very important in C. Otherwise, your program will give very unexpected results!

What is the output of the following program?

```
double x;
double y = x+3;

printf("x=%d, y=%d \n", x, y);
```

How about the following?

```
double *w;
double z = 5;
printf("w=%p, z=%f \n", w, z);

*w = 18;
printf("*w = %f \n", *w);
```

If you're lucky, the code above will give a *segmentation fault* – an error that occurs when you try to access memory that is not available to you. If you're not lucky, you will not get a segmentation fault but your program will give unexpected answers: this sort of error is much harder to track down.

Structures

As I mentioned in class, the C language does not have classes. However, it does have *structures*. A structure is a collection of items of different types. It is similar to a class in Java, but has no methods.

This is an example of how to declare a structure in C:

```
struct Position {
    int x;
    int y;
};
```

You can then declare a variable of a structure type in the same way that you would declare a variable of any other type, except that you need to include the keyword 'struct'. You can then access the individual fields of the struct using the dot operator.

```
int main() {
    struct Position mypos;

    mypos.x = 4;
    mypos.y = 5;

    printf("Position: (%d, %d) \n", mypos.x, mypos.y);
}
```

To avoid repeating the keyword struct, you can use a *typedef* outside of any function:

```
typedef struct Position Pos;
```

Now, you can declare a declare a variable of the Position type as follows:

```
Pos mypos;
```

The *typedef* command can be used for other types as well:

```
typedef unsigned int uint;
```

You can have pointers to structs, and arrays of structs.

-> is a shortcut for the dereferencing operator.

Resources

Programming in C, by A. D. Marshall, <http://www.cs.cf.ac.uk/Dave/C/>

Pointers and Memory, by Nick Parlante,
<http://cslibrary.stanford.edu/102/PointersAndMemory.pdf>

C Programming Tutorial – Structures from Georgia Tech
<http://vergil.chemistry.gatech.edu/resources/programming/c-tutorial/structs.html>