# Systems Programming Lecture 6: Dynamic Memory Management in C

## Memory Model

In C, the notion of type and the amount of memory required to store that type are strongly linked. Below are some basic types in C with their memory allocation requirements on 64-bit Linux:

- `char`: integer number or character: 1-byte
- `short`: integer number: 2-bytes (must be at least 2-bytes)
- `int`: integer number: 4-bytes (must be at least 2-bytes)
- `long`: integer number: 8-bytes (must be at least 4-bytes)
- `float`: floating point number: 4-bytes
- `double`: floating point number: 8-bytes
- `void *`, `int *` etc.: pointers: 8-bytes (on 64 bit machines)

Understanding the memory model in C will help to manage situations where we need to manage complex memory management and produce good programs. Making simple mistakes could potentially produce undesirable bugs that may fail the goal of our programs.

## Memory layout

Figure 1 is a basic representation of a memory layout on a machine. The memory consists of a number of spaces. The stack space is used for storing temporary data, such as local variables; the heap space stores more long-term data whose size may not be known at compile time and it normally takes more space than the stack; and the static data is allocated for data that lasts for the duration of the program, such as global variables.

## Memory allocation on the stack

When we declare a variable in C, we tell it to reserve space for the data of that variable to exist in memory, for example `int a = 10;` states that the integer `a` requires some memory space (4 bytes) to be allocated to it. Memory allocation is a term that refers to the process of making space for the storage of data. When we declare a variable of a type, we state that enough memory for that type must be allocated locally to store the data of the declared variable. This means that the allocation is local (e.g. when a variable occurs in a function and when the function returns a value, the memory for the variable is deallocated).

It should be clear that local variables cannot be accessed from outside functions. However, we have seen that we can use pointers to access the memory address of local variables. This practice often makes it easy to make mistakes when we attempt to use a pointer to reference an unallocated memory address. For

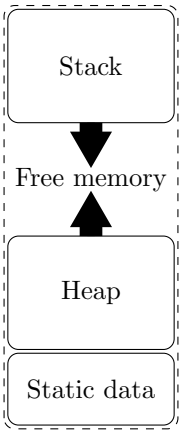



Figure 1:

Example Code 1:

```
#include<stdio.h>
int *plus(int a, int b);

int main() {
    int *p = plus(1,2);
    printf("%d\n", *p); // dereference return value to print an integer
    return 0;
}

int *plus(int a, int b) {
    int c = a + b;
    return &c; // return a ponter to locally declared variable c
}
```

demonstration purposes, consider the simple program in Example Code 1 with the function `plus()` which adds two numbers and returns a memory address for the result value.

The function `plus()` takes two integer arguments and returns a pointer to an integer. In the body of the function, the two integers are added up and the result is stored in a local variable `c`, where the function returns the memory address of `c`, which is assigned to the pointer value `p` in the `main()` function.

The problem with the code above is that the memory of `c` is deallocated once the function returns, which makes the variable `p` point to a memory address which is unallocated. The `printf` statement, which deferences `p`, following the pointer to the memory address, may fail. Thus the above code is *wrong*. When we explore the program, we can note that in `main()`, `p` waits for the result of the call to `plus()`, which sets `c`. Once `plus()` returns the value of `c`, the value of `p` references a variable declared in `plus()`, but all locally declared variables in `plus()` are deallocated once `plus()` returns. That means, by the time the `print()` is called and `p` is dereferenced, the memory address points to unallocated memory, and we cannot guarantee that the data at that memory address will be what we expect.

## Global Memory Allocation (the heap, `malloc()`, and `free()`)

The global memory region for a program is called the heap, which is a fragmented data structure where new allocations try and fit within unallocated regions. Whenever a program needs to allocate memory globally or in a dynamic way, that memory is allocated on the heap, which is shared across the entire program irrespective of function calls.

We can use global memory allocation on the heap to write functions that return a memory address and work properly (i.e. an allocation procedure that doesn't deallocate memory automatically when the function's `return` statement is executed).

In C, the `malloc()`, or memory allocator is a function that takes the number of bytes to be allocated as its argument, and returns a pointer to a memory region on the heap of the requested byte-size. Here is a code snippet to allocate memory to store an integer on the heap:

Example Code 2:

```
// Allocate sizeof(int) number of bytes
int *p = malloc(sizeof(int));
// malloc() returns a pointer of type void *.
// On assignment such pointers are automatically cast to pointers to other types as needed.
```

First, to allocate an integer on the heap, we have to know how big an integer is, i.e. the size of the integer, which we get it from the `sizeof` operator. Since an `int` is 4 bytes in size, `malloc()` will allocate 4 bytes of memory on the heap in which an integer can be stored. `malloc()` then returns the memory address of the newly allocated memory, which is assigned to `p` (if memory is not available, `malloc()` will return `NULL` instead). Since `malloc()` is a general purpose allocation tool, just allocating bytes which can be used to store data generally, it returns an pointer of type `void *`. This pointer is automatically cast to a type `int *` pointer on assignment.

We can now use `p` like a standard pointer as before. However, once we're done with `p`, we have to explicitly deallocate it. Unlike stack-based memory allocations, which are implicitly deallocated when functions return, there is no way for C to know when you are done using memory allocated on the heap. C does not track references like Python or Java, so it can't perform garbage collection; instead, you, the programmer, must indicate when you're done with the memory by *freeing*. The deallocation function is `free()`, which takes a pointer value as input and "frees" the referenced memory on the heap.

Example Code 3:

```
int *p = malloc(sizeof(int));
if (p == NULL) exit(1); // if malloc() failed then exit with an error
// do something with p
free(p); // deallocate p
```

With all of that, we can now complete the `plus()` program to properly return a pointer to the result; see Example Code 4.

Example Code 4:

```
#include<stdio.h>
#include<stdlib.h>
int *plus(int a, int b);

int main(){
    int *p = plus(1,2);
    // p now points to memory on the heap
    printf("%d\n",*p);
    free(p); //free allocated memory
    return 0;
}

int *plus(int a, int b){
    // allocate enough space for an int
    int *p = malloc(sizeof(int));
    if (p == NULL) exit(1); // if malloc() failed then exit with an error
    *p = a + b; // for an integer
    return p; // return pointer
}
```

## Memory Leaks and Other Memory Violations

In C (and C++), the programmer is responsible for memory management, which includes both the allocation and deallocation of memory. As a result, there are many mistakes that can be made, which is natural considering that all programmers make mistakes. One of the most common mistakes is a memory leak, where heap allocated memory is not freed. Consider the program in Example Code 5.

Example Code 5:

```
#include<stdlib.h>
int main(){
    int i, *p;
    for(i=0; i<100; i++){
        p = malloc(sizeof(int));
        if (p == NULL) exit(1);
        *p = i;
    }
    return 0;
}
```

At the `malloc()` on the fifth line in this code, the returned pointer to newly allocated memory is overwriting the previous value of `p`. No `free()` is occurring, and once the previous pointer value is overwritten, there is no way to free that memory. It is considered lost, and the above program has a memory leak. Memory leaks are very bad, and over time, can cause your program to fail.

Another common mistake is dereferencing a *dangling pointer*. A dangling pointer is when a pointer value once referenced allocated memory, but that memory has been deallocated. We've seen an example of this

already in the `plus()` program, but it can also occur for heap allocations.

Example Code 6:

```
#include<stdlib.h>
int main(){
    int *p = malloc(sizeof(int));
    if (p == NULL) exit(1);
    //... code
    free(p);
    //... code
    *p = 10;
    return 0;
}
```

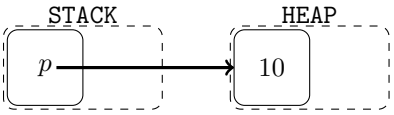
Once `p` has been freed, the memory referenced by `p`'s value can be reclaimed by other allocations. At the point where `p` is dereferenced for the assignment, it might be the case that you are actually overwriting memory for some other value, and corrupting your program. Once memory is freed, it should never be dereferenced.

Another, common mistake with memory allocation is a *double free*. The heap allocation functions maintain special data structures so that it is easy to find unallocated memory and reallocate it in later `malloc()` calls. If you call `free()` twice on a pointer, you will corrupt that process, result in a core dump or some other very scary error.

So far we only looked at the standard memory allocation situation where we need to allocate memory on the heap.

Example Code 7:

```
int *p = malloc(sizeof(int));
if (p == NULL) exit(1);
*p = 10;
```




We use `malloc()` to allocate enough memory to store an `int`, and we assign the address of that memory to the integer pointer `p`. Below Example Code 7, is the stack diagram of this allocation. The pointer `p` exists on the stack, but it now references the memory on the heap.

We now know that arrays and pointers are almost the same. If we have an integer pointer `p`, it can point to a single integer, or it can point to the start of a sequence of integers. A sequence of contiguous integers is an array. All we need is to allocate enough space to store all those integers, and `malloc()` can do that too.

Consider what is needed to allocate an array of a given size. For example, how many bytes would be needed to allocate an integer array of size 5? There are 4-bytes for each integer, and the array holds 5 integers: 20 bytes. To allocate the array, we just ask `malloc()` to allocate 20 bytes, and assign the result to an `int *` pointer, like in Example Code 8.

The result of the `malloc()` is 20 bytes of contiguous memory which is referenced by an integer pointer, which is the *same* as an array! We can even use the array indexing method, `[]`, to access and assign to the array.

Example Code 8:

```
int *p = malloc(5*sizeof(int));
if (p == NULL) exit(1);
p[0] = 10;
p[1] = 20;
//...
p[4] = 50;
```

## calloc()

Because allocating items as an array is so common, we have a special function for it.

Example Code 9:

```
int *p = calloc(5, sizeof(int));
```

While allocating arrays with `malloc()` is simple and effective, it can be problematic. First off, `malloc()` makes no guarantee that memory allocated will be clean — it can be anything. For example, consider this simple program:

Example Code 10:

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    // allocate a 20 byte array
    int *a =  malloc(20 * sizeof(int));
    if (a == NULL) exit(1);
    int i;
    for(i = 0; i < 20; i++){
            printf("%d\n",a[i]);
    }
    return 0;
}
```

What is the output of this program? We don't know. It's undefined. The allocated memory from `malloc()` can have any value, usually whatever value the memory used to have if it was previously allocated.

The second problem with using `malloc()` is that it is a multi-purpose allocation tool. It is generally designed to allocate memory of a given size that can be used for both arrays and other data types. This means that to allocate an array of the right size, you have to perform an arithmetic computation, like `20 * sizeof(int)`, which is non-intuitive and reduces the readability of code. It can also cause problems if the product of the numbers in such a multiplication are too large, in which case an integer overflow can occur.

To address these issues, there is a special purpose allocator that is a lot more effective for array allocation. It is `calloc()` or the *counting allocator*. Its usage is as follows.

`calloc()` takes two arguments, the number of items needed and the size of each item. For an array, that is the length of the array, and the size is the `sizeof()` the array type, which is `int` in this case. Not only

Example Code 11:

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

does `calloc()` make the array allocation more straightforward, it will also zero-out or clean the memory that is allocated. For example, this program will always print `0`'s.

Example Code 12:

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    //allocate an array of 20 ints
    int *a = calloc(20, sizeof(int));
    if (a == NULL) exit(1);
    int i;
    for(i = 0; i < 20; i++){
        printf("%d\n",a[i]); // 0 because calloc zeros out allocated memory
    }
    return 0;
}
```

## realloc()

The `realloc()` function in C (`void *realloc(void *ptr, size_t size)`) can be used to resize the memory block pointed to by a pointer `ptr`. The memory to be resized must have been previously allocated by `malloc()`, `calloc()` or `realloc()` and not yet freed with a call to `free()` or `realloc()`. Otherwise, the results are undefined. The reallocation is done either by:

(i) expanding or contracting the existing area pointed to by `ptr` and

(ii) allocating a new memory block of new size, copying the memory area with size equal the lesser of the new and the old sizes, and freeing the old block.

If there isn't enough memory, the old memory block is not freed and a `NULL` pointer is returned.

The pointer parameter `ptr` points to a memory block that was previously allocated with `malloc()`, `calloc()` or `realloc()` to be reallocated. If this parameter is set to `NULL` then a new block is allocated and a pointer to it is returned by the function. The `size` parameter is used to allocate a new size for the memory block.

Example Code 13:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main(){
    char *str;
    str = malloc(15); // initial memory allocation
    if (str == NULL) exit(1);
    strcpy(str, "C programming");
    printf("String = %s, Address =%p\n", str, &str);
    str = realloc(str, 25); // reallocating memory
    if (str == NULL) exit(1);
    strcat(str, " is fun");
    printf("string = %s, Address = %p\n", str, &str);
    free(str);
    return 0;
}
```