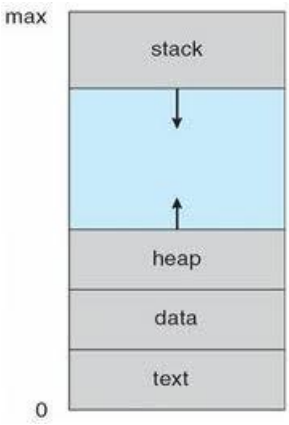
Memory

Floral is fairly low-level when it comes to memory. You have the choice of static and dynamic allocation, and in the latter case manual deallocation is necessitated.

Every process is split into multiple sections: a text section for the code, various data sections, a stack and a heap. The stack grows downwards while the heap grows up.



> The stack and the heap, image from stackoverflow. A note - 0 and max are placeholder values. The actual memory addresses are decided by the OS.

Static Allocation

In basic Floral programs, the stack will probably be the most used area of allocation. Local variables and constants are initialized on the stack in the current frame.

```
struct Complex {
    Complex(r: Float, i: Float): real = r, imag = i {}

exposed get:
    var real, imag: Float;
};

func main(): Int {
    var i = 1; // allocated on stack
    let char = 'f'; // also on stack
    let num = Complex(0, 1); // same here
    let ptr = &i; // pointer to variable on the stack is also allocated on the stack
}
```

Allocating on the stack is a very cheap operation, involving one line of assembly code: `sub rsp, N` where `N` is the number of bytes you want to allocate.

In the example above, the assembly code generated would allocate enough space to store all the variables/constants. In this case, `sizeof(Int) + sizeof(Char) + sizeof(Complex) + sizeof(&Int) == 41`. However - `alignof(Char) == 4` so you would expect `sub rsp, 44` and the corresponding `add rsp, 44` at the end of the frame.

Dynamic Allocation

The major issue with static allocation is that pointers to statically allocated objects are only valid within the current frame. On the contrary, dynamically allocated objects can be accessed from any frame with cost: heap allocation is slow and manual.

The following is a program which exposes the problem:

```
func ptrToInt(): &Int {
    // 2: new frame allocated
    var int = 67; // 3: integer allocated on stack
    return &int; // 4: returns pointer to location on stack
    // 5: frame pushed off stack - integer deallocates and pointer becomes invalid
}

func main(): Int {
    let pointer = ptrToInt(); // 1: call function
    print(*pointer, '\n'); // 6: ERROR! attempt to dereference dangling pointer
    return 0;
}
```

This can be easily mitigated with heap allocation. Floral provides two functions for heap (de)allocation: `alloc<T>(Type, UInt): &T` and `dealloc<T>(&T)`.

```
func ptrToInt(): &Int {
    var ptr = alloc(Int, 1);
    *ptr = 67;
    return ptr;
}

func main(): Int {
    let pointer = ptrToInt(); // obtain pointer
    print(*pointer, '\n'); // dereference is fine
    dealloc(pointer); // manual deallocation for dynamic memory
    return 0;
}
```

Under the hood, Floral uses the C functions `malloc` and `free`. These functions are available with the directive `using C;`.