Chapter 9: Dynamic memory allocation

1 Warmup: text input/output with <stdio.h>

In this chapter we are going to write several small programs which operate on text in the form of character strings. Let's start by reviewing useful functions from the standard library.

Exercise 1 Implement a program `llen.c` which reads its standard input line by line, and counts characters on each line, like illustrated below. Use functions `fgets()` from `stdio.h` to read input, `strlen()` from `string.h` to count chars[1], and `printf()` to display the results.
Like in `minigrep` in chapter 7, use a line buffer i.e. a single character array of fixed size (e.g. 1kB) and assume that input will never overflow.

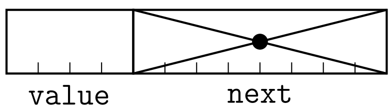
```
$ /bin/ls ~ | ./llen
 5 a.txt
 7 Desktop
 9 Documents
 9 Downloads
 5 Music
 8 Pictures
 6 Public
10 readme.txt
 8 some dir
 6 Videos
 4 Work
```

Exercise 2 The Unix `rev` command reads lines from its standard input and copies them to standard output, but reversing the order of characters in every line. Try it with e.g. `/bin/ls ~ | rev` then implement a `myrev.c` program which does the same, using the `fputc()` function to print individual characters.

2 Recursive data structures

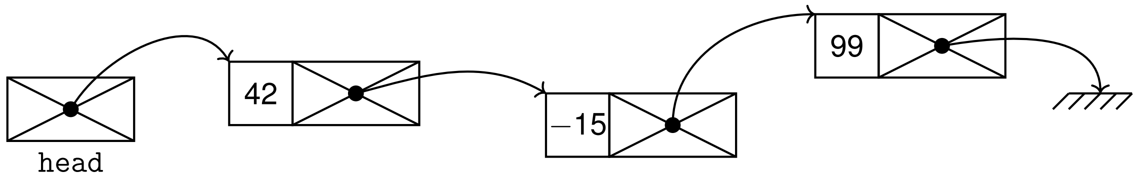
We learned in chapter 8 that the `struct` keyword can be used to define so-called **composite data types** aka **structures**. A structure type can have several fields with various types, either **scalar types** like `int`, `float`, `char` or **reference types** (i.e. pointers) like `int *`, `char *`, etc. Today we learn that it is also possible to define **recursive types** by having one (or more) field be a reference to another object of the same type:

```
struct node {
    int value;
    struct node *next;
};
```



[1] note that `strlen()` does count the `'\n'` character at the end of the line, but not the `'\0'` marking the end of the string.

As illustrated on the preceding page, every object of type "**struct** node" has two fields: an integer, and a pointer to another **struct** node. This makes it possible to build a **linked list** containing several nodes:



Remarks

- A linked list is very different from an array (cf chapter 5) because successive nodes are not necessarily close to each other in memory.
- We generally keep track of the **list head** using not a struct but a pointer to a struct, declared for example as `struct node *head;`
- The weird shape on the right (which looks like an electronics ground symbol) represents the **list end**, encoded as a **null pointer**. In other words, in our last node, `value` is 99 and `next` points to address zero. There is no universal convention for representing null pointers in a diagram, so choose your favourite.
- An **empty list** would have no nodes, i.e. `head = NULL`.
- The K&R describes pointers to structures in §6.4 and "self-referential structures" in §6.5. Go read those.

Exercise 3 Write a program `mylist.c` where you create the list illustrated above (all three nodes can be global variables, or local to `main()`), including the `head` pointer. Then walk the list in a **while** loop to print all values, like illustrated below:

```
$ ./mylist
42
-15
99
```

3 Dynamic allocation

The C language offers two ways of allocating memory space to store data: automatic and manual. **Automatic memory allocation** happens implicitely, everywhere we declare a program variable. So-called **global variables** are allocated just once, in a dedicated region of memory. So-called **local variables** are allocated on the execution stack (cf IST-ASM chapter 8) when entering a function, and are automatically deallocated when leaving the function.
But if we want to create a new list node at every iteration of e.g. a while loop, this is not enough: we want to **manually** create a new object in memory each time. This is called **dynamic allocation** of memory. The C language includes a function named `malloc()` which does just that: `malloc(N)` searches for a free block of size `N` bytes and returns its address.
Compared to automatic allocation, dynamic allocation takes more effort (because we must explicitely invoke a function) and offers slower performance (because of the execution time of the allocation algorithms) but it is a lot more flexible.

Exercise 4 Modify your programs from section 1 to use `malloc()` instead of automatic allocation.

- You will need to add `#include <stdlib.h>` near the top of your source files.

Exercise 5 To help with manual allocation, C provides the compile-time unary operator `sizeof()` that can be used to compute the size of any data type, including structures: `sizeof(typename)` evaluates to the number of bytes occupied by one object of that type. Write a small program to display[2] the memory size of types `int`, `char`, `float`, `double`, `int*`, `char*`, `float*`, `double*`, `char[100]`, `struct fraction` (from chap 8), and `struct node`. Try to guess the results before running your program, and ask us for help if anything seems confusing.

4 Putting it all together

Exercise 6 The unix `tac` command reads all lines from its standard input, then copies them to stdout but in reverse order.[3] Try it with e.g. `/bin/ls ~ | tac` then implement a `mytac.c` program which does the same. Store all lines in a linked list of `struct line` objects, where each struct contains a character array of fixed size. The idea is to repeatedly add new lines at the head of the list. Use function `memcpy()` from `string.h` to copy data from your line buffer into your newly created structs.

Exercise 7 Write a second version of `mytac` where the `struct line` does not contain a full array but only a character pointer, and use `malloc()` to allocate just the right amount of space for each line, as per `strlen()` of your line buffer.

Exercise 8 The unix `sort` command reads all lines from its standard input, then prints them to stdout but in lexicographic order according to ASCII encoding. Try it with `cat mytac.c | sort`. (If the sorting order seems to ignore leading whitespace, type `export LC_ALL=C` and try again.)

Then implement a `mysort.c` program which does the same. Use function `strcmp()` to compare strings. The idea is to always keep the linked list sorted, and insert each new line at the correct position.

Exercise 9 (optional) Modify your code from the previous exercise to sort by line length instead. The resulting tool is probably not useful in itself, but command `cat *.c | ./mysortbylen` produces quite nice-looking output.

[2] `sizeof()` evaluates to some exotic integer type which `printf("%d")` might not like. You will probably want to "convert" (aka typecast) that value into a proper `int` by writing something like `(int)sizeof(sometype)`.

[3] `tac` is the reverse of `cat`. hilarious, isn't it ?