Chapter 7: dynamic allocation

1 warm-up: scanf and static allocation of memory

In the previous chapter, you have explored how to manipulate data stored in files, including standard input and output streams (stdin and stdout). In particular, you have used the functions fgets (that reads a line from the standard input) and printf that writes a string to the standard output.

Exercise Take this program again, and modify it so that it now receives the 10 elements of the array interactively from the user, at runtime. You will use `strtol` (see K&R §B.5 or `man strtol`) to convert the input string provided by the user into an integer.
You can now feed T's values to your program by entering them one by one at the prompt.
If you feel comfortable here, you can also use the `echo` command and the "pipe" ('|') operator seen in chapter 6.

```
echo 1 2 3 4 5 6 7 8 9 19 | ./array
```

2 Dynamic allocation of memory

2.1 The stack, the heap: where are my variables

Whenever you execute a program, the system allocates memory for the corresponding process (see chapter 5). This memory is organized in several parts and the details depend on the operating system and executable file format. The format used in Linux is called ELF, for Executable and Linkable Format[1].
We won't go into the details of executable formats but concentrate on where variables of your C program are stored. From a programmer's perspective, data is allocated in (ie accessed from) three different places:

- registers, which are located inside the CPU;
- the stack, which typically contains local variables, ie variables that are declared from within a C function;
- the heap, which contains global variables.

The use of registers and stack is completely transparent for the program. It's the **compiler**'s job to decide:

- How the variables are copied (or not) from memory to registers. This depends on architectural details as well as performance compromises.
- How the stack is used exactly.

These decisions rely on the **calling convention** that describes how local variables are allocated on the stack as well as how parameters are passed.
Note: This is the topic of chapters 7 "Programming with subroutines" and 8 "The execution stack" of our course on Assembly Programming.
One important point to grasp here is that both registers and the stack are limited in space and that **the programmer doesn't have any control on them**.

The C standard library hence provides the programmer with functions to allocate data within the **heap**. The heap is a larger piece of memory within the running program's address space, that is managed by the operating system, but with some controls available to the programmer. All the handling of details

[1] NB: it's also used in android, BSD and other Unix, several gaming consoles OSes, etc., see `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

(which we completely overlook here) is left to the operating system's Memory Management, but the programmer can ask for the system to **allocate** (ie reserve) arbitrarily large chunks of data.
This is done through the `malloc` function of the standard library. For how to use `malloc`, please see the man page: `man malloc`. For details about the implementation of `malloc`, see K&R §8.7.
The interface of malloc is :

```
void *malloc(size_t size);
```

- It takes a special integer (actually a `size_t`) parameter through which the programmer specifies the number of **bytes** she wants allocated.
- It returns the address of the memory that it allocates for the programmer.

Note: this address is a pointer to "void" (undetermined type). So for using it, you should cast it to a type suitable for your program.

Example To dynamically allocate an `s`-elements array of integers, where `s` is a properly initialized **`int`** variable, one can use:

```
int *T = (int *)malloc(10*sizeof(int));
```

Note : the **`sizeof()`** (see K&R §6.3) gives the number of bytes occupied by an expression (including pre-defined types). In this case, it returns `4` as integers are encoded on 4 bytes.

Exercise Modify your array-printing program so that the array `T` is no longer allocated statically. Your program should:

- ask the user for an integer `s`
- allocate the array `T` with `s` elements.

Note: `T` doesn't have to be a global variable anymore. This will essentially depend on how you will organize your code.

3 Structures

3.1 Structured variables

In C, you can declare structured variables. A structured variable, called a **`struct`** is one that is organized with several fields, that can themselves be of different types. See K&R §6.1 for details.
For example, the following *variable* `mystruct` is a variable with:

- a first integer field, named `rank`
- a second field, named `name` that is a pointer to characters.

```
struct mystruct {
  int rank;
  char *name;
};
```

Once properly initialized, the fields of this variable can be accessed, eg as follows:

```
printf("this is mystruct.rank = %d\n", mystruct.rank);
printf("this is mystruct.name = %s\n", mystruct.name);
```

3.2 Structured datatypes

Usually, one will define a new type for such variables. This is done using the `typedef` operator, see K&R §6.7. Following our example, we can write:

```
typedef struct {
  int rank;
  char *name;
} myType;
```

We can now declare several variables of the same type, which can, once properly initialized, be accessed in the same spirit:

```
myType var1;
myType var2;
// proper initialization occurs here ...
printf("this is var1.rank = %d\n", var1.rank);
printf("this is var2.name = %s\n", var2.name);
```

3.3 Dynamic allocation of structured variables

The most common way to use structured types, is to combine them with dynamic allocation.

The following program declares 2 variables `var1` and `var2` as pointers to `myType`. The corresponding memory regions are then allocated using `malloc` and the two pointers are initialized with corresponding addresses. Finally, the `name` field for each struct is `malloc`ed and initialized with strings obtained from the user through the `fgets` function.

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int rank;
  char *name;
} myType;

int main(){
  // pointers declaration
  myType *var1;
  myType *var2;
  // myType struct allocations (for the two pointers)
  var1 = (myType*)malloc(sizeof(myType));
  var2 = (myType*)malloc(sizeof(myType));
  // rank initialization
  var1->rank = 0;
  var2->rank = 1;

  // name allocation
  // nb: names can be 10 characters long, max
  var1->name = (char*)malloc(10*sizeof(char));
  var2->name = (char*)malloc(10*sizeof(char));
  // name definition
  printf("please give me a name for 1st variable: ");
  fgets(var1->name, 10, stdin);
  printf("please give me a name for 2nd variable: ");
  fgets(var2->name, 10, stdin);

  // printing var1 and var2
  printf("this is variable 1: rank = %d -- name = %s", var1->rank, var1->name);
  printf("this is variable 2: rank = %d -- name = %s", var2->rank, var2->name);
}
```

exercise Write a C program implementing a kind of phonebook, ie a program that:

- interactively asks for pairs of (name, phone number)
- adds each pair to an array `phonebook`
- does this for a fixed number of persons
- exists after printing the whole phonebook.

An example execution (with a phonebook size statically fixed to 3) outputs the following:

```
./phoneb
type a name: Asterix
type a number for ‘Asterix’: 0123456789
type a name: Obelix
type a number for ‘Obelix’: 9876543210
type a name: Idefix
type a number for ‘Idefix’: 1212
Asterix: 0123456789
Obelix: 9876543210
Idefix: 1212
```

You may want to use syscalls like `scanf`, `printf`, `malloc` and `strcpy`.

4 Recursive type definitions

Structures can be used to build arbitrarily complex data structures, trees, linked lists, graphs. A lot of these structures are inherently recursive: one element contains some specific value and also points to one or several elements of the same type.
To implement such structures in C, one uses recursive structures, also explained in K&R §6.7. The code below gives an example of a binary tree node structure, where each node has one integer `value` as well as a pointer to its left child and a pointer to its right child.

```
typedef struct tnode {
  int value;
  struct tnode *leftChild;
  struct tnode *rightChild;
} Treenode;
```

exercise Write a program that interactively builds a linked list of integers values. Each node in the list contains an `int val` field, as well as a pointer to the next element in the list. Your program should interactively add the user for new values to be added to the list. It should maintain the list sorted at all time.
Your program should behave like this:

```
./link
give me value: 12
12,
done!
give me value: 8
8,12,
done!
give me value: 9
8,9,12,
done!
give me value: 789
8,9,12,789,
done!
give me value: 1
1,8,9,12,789,
done!
give me value: 5
1,5,8,9,12,789,
done!
give me value: 10
1,5,8,9,10,12,789,
done!
give me value:
```