Unit 6
Streams, Dynamic Memory, and New Data Structures

Alberto Casagrande
Email: `acasagrande@units.it`

a.a. 2019/2020

In POSIX, every logical (e.g., file, stdin) and physical (e.g., mouses, disks) objects are handled through streams.

They are logical interfaces for reading from and writing to the associated objects.

In general, they must be opened before the usage and closed afterward.

stdio.h allows the usage of the following functions

fopen opens a stream

```
FILE *fopen (const char *name,
             const char *mode);
```

The mode specifies whether the file will be written ("w"),
read ("r"), or both of them ("r+").

See manpages for the full list.

fclose closes a stream

```
int fclose (FILE *fp);
```

Some Useful Functions on Files (Cont'd)

fgetc read a character from the stream

```
int fgetc (FILE *fp);
```

fputc write a character from the stream

```
int fputc (const int ch, FILE *fp);
```

High Level Stream Functions

Luckly, we also have high level stream functions fscanf and fprintf
. . . reminding you something?

fscanf reads data from an open input stream

```
int fscanf (FILE *fp, char *fmt, args...);
```

fprintf write data in an open output stream

```
int fprintf (FILE *fp, char *fmt, args...);
```

printf and scanf can be "emulated" by using stdin and stdout

```
fscanf ( stdin , "%d" , &a );

fprintf ( stdout , "We can emulate printf" );

fprintf ( stderr , "and output in stderr \n" );
```

stdin, stdout, and stderr need to be neither opened nor closed.

By the way...

C has very similar functions to deal with buffers: sprintf and sscanf

```c
int main(int argc, char *argv[]) {

  char buf[100]; int a;

  sscanf(argv[1], "%d", &a);
  sprintf(buf, "The first arg was %d\n", a);

  return 0;
}
```

A Simple Exercise

Example (Prime Numbers)

Write a function that takes a natural number *n* and returns the first *n* prime numbers.

A Simple Exercise

Example (Prime Numbers)

Write a function that takes a natural number *n* and returns the first *n* prime numbers.

Any problem?

A Simple Exercise

Example (Prime Numbers)

Write a function that takes a natural number *n* and returns the first *n* prime numbers.

Any problem? How to return the prime numbers?

A Simple Exercise

Any problem? How to return the prime numbers?

Using an array would be nice, but:

- its size is not known at compilation time
- if declared inside a function, it only exists inside it

A Naïve Solution

Over-estimate *n* and pass a pointer as a parameter.

```
void get_primes(unsigned int *output,
                const unsigned int n) {...}
...
unsigned int output[100000]; int n=10;

get_primes(output, n);
```

- when $n << 100000$, memory is "wasted"
- when $n > 100000$, other data are overwritten
- not really "readable"

We Need. . .

. . . a mechanism to:

- allocate the right amount of memory at runtime
- free the memory when it is no more required
- "resize" the allocated memory if needed

Pointers can be used to refer to this memory and can be returned.

We Need. . .

. . . a mechanism to:

- allocate the right amount of memory at runtime
- free the memory when it is no more required
- "resize" the allocated memory if needed

Pointers can be used to refer to this memory and can be returned.

We need **Dynamic memory handling**

Dynamic Memory

By including the header stdlib.h, we can use:

malloc allocates an uninitialized memory region having a given size. A pointer for it is returned.

```
void *malloc(const size_t buffer_size);
```

calloc allocates a 0-initialized memory region meant for an array.

```
void *calloc(const size_t num_elem,
             const size_t elem_size);
```

Dynamic Memory (Cont'd)

realloc resizes pre-allocated dynamic memory space

```
void *realloc(void *mem_ptr,
              const size_t new_total_size);
```

free deallocates pre-allocated dynamic memory space

```
void free(void *mem_ptr);
```

```c
unsigned int *get_primes(unsigned int n) {
    unsigned int *p

    p=(unsigned int *)calloc(n,
                sizeof(unsigned int));

    ... /* fill the allocated buffer
            by using prime numbers */

    return p;
}
```

The get_prime Exercise (Cont'd)

```c
...
unsigned int *primes = get_primes(1000);

for (unsigned int i=0; i<1000; i++)
    printf("Prime_number_#%u:_%u\n",
            i, primes[i]);

...

free(primes); /* when you don't need
                 the primes anymore */

...
```

A Different Exercise

Write a program to represent *n* students.

Every student must be associated to

- a name
- a family name
- a student ID
- his/her courses and grades

Write a program to represent *n* students.

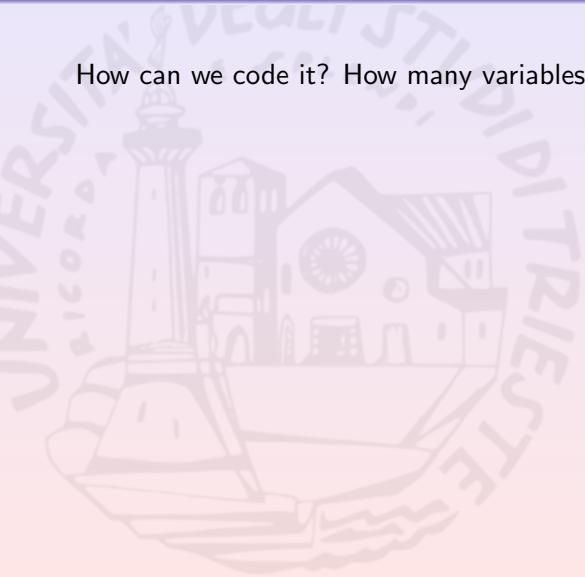Every student must be associated to

- a name
- a family name
- a student ID
- his/her courses and grades

We may want to:

- find the list of the student IDs corresponding to given a name
- find all those students having a given average grade
- . . .

A Different Exercise

How can we code it? How many variables/arrays do we need?

How can we code it? How many variables/arrays do we need?

- n strings to represent names
- n strings to represent family names
- n unsigned integer to represent IDs
- n string arrays for course IDs
- n integer arrays for grades

We can use arrays . . .

A Different Exercise

How can we code it? How many variables/arrays do we need?

- *n* strings to represent names
- *n* strings to represent family names
- *n* unsigned integer to represent IDs
- *n* string arrays for course IDs
- *n* integer arrays for grades

We can use arrays ... but we will have to handle the names-surnames-IDs-courses-grades relations by our own

So what?

Coding New Data Structures

We can define new data structures by using struct

```
struct <structure name> {
  <1st member type> <1st member name>;
  <2nd member type> <2nd member name>;
  ...
};
```

E.g.,

```
struct course_t {
   unsigned int ID;
   char grade;
};
```

How to "Use" New Data Structures?

Variables and arrays can have the new structure as "type".

E.g.,

```
struct course_t computer_programming;

struct course_t my_courses[34];

struct course_t *ptr = (struct course_t *)
           malloc(sizeof(struct course_t));
```

We can access members by using "."

"ptr->" is a shortcut for "(*ptr)."

E.g.,

```
computer_programming.ID = 15;

my_courses[5].grade = -1;

ptr->grade = 30;  /* ptr->grade is a shortcut fo
                      (*ptr).grade */
```

Back to The Student Example

A new data structure for students

```c
struct career_t {
    unsigned int num_of_courses;
    struct course_t *courses;
};

struct student_t {
  char name[100];
  char family_name[100];
  unsigned int ID;
  struct career_t career;
};
```

We can add a struct to the list of types by using `typedef`.

E.g.,

```
typedef struct course_t course_type;

typedef int generic_type;
```

Structs vs Types

Struct and type/function identificators are in different space name.

Thus, a struct can have the same name of a type or a function.

E.g.,

```
typedef struct course_t course_type;

void course_t() {...} /* admitted */

void course_type() {...} /* not admitted */
```

Where Define Structs and Types

In an header to share the definition itself

However, we need a way to elude multiple inclusion to avoid ...

```
In file included from struct-bad.c:4:
struct-bad.h:1:8: error: redefinition of
                        'struct struct_test'
 struct struct_test{
        ^~~~~~~~~~~
In file included from struct-bad.c:3:
struct-bad.h:1:8: note: originally defined here
 struct struct_test{
        ^~~~~~~~~~~
```

Use pre-processor directives in the header!!!

```
#ifndef __HEADER_NAME__
#define __HEADER_NAME__

... /* write here signatures and macros */

#endif
```

Coming soon. . .

- abstract vs concrete data structure
- recursion
- dynamic programming