C introduction part 3

pointers, static and dynamic memory allocation, and structures

Objectives

- Write functions that can accept and "return arrays"
  - Review of pointers
  - Static and dynamic arrays

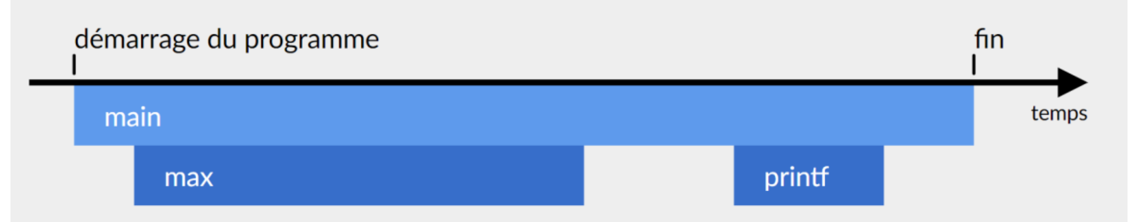
- Use *structures* to store and pass around data

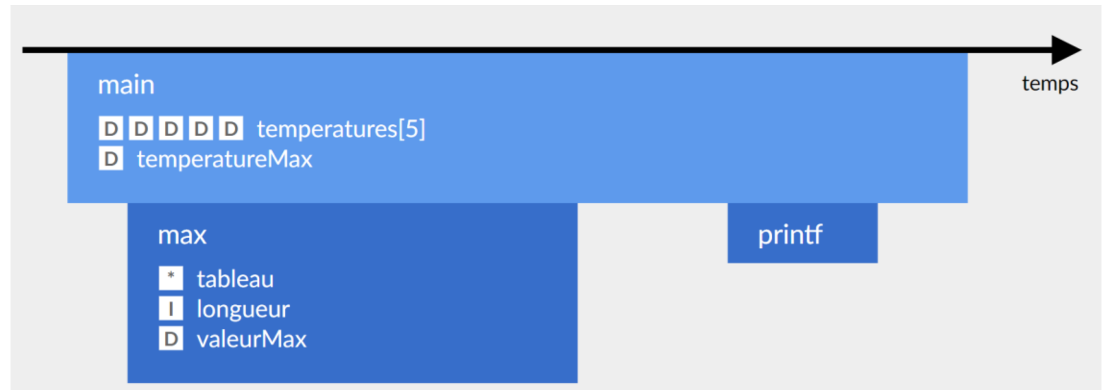
https://sieprog.ch/
https://stakahama.gitlab.io/sie-eng270/C_intro.html

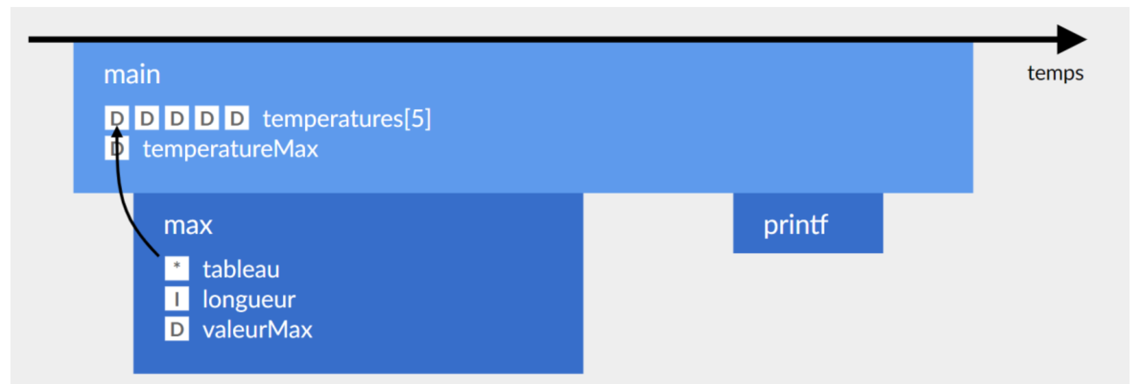
Call stack

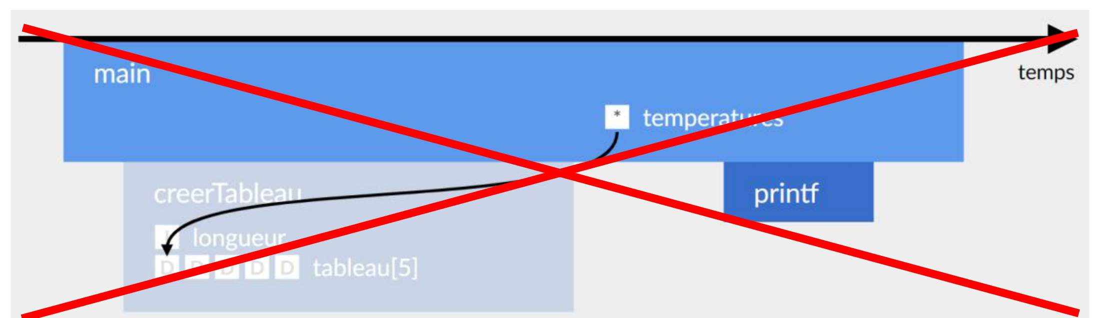
Data structure in memory that manages active functions and flow of execution

```c
#include <stdio.h>

double max(double * tableau, int longueur) {
    double valeurMax = tableau[0];
    for (int i = 1; i < longueur; i++) {
        if (tableau[i] > valeurMax) valeurMax = tableau[i];
    }
    return valeurMax;
}

int main(int argc, char * argv[]) {
    double temperatures[] = {24.2, 26.5, 27.4, 28.1, 26.9};
    double temperatureMax = max(temperatures, 5);
    printf("Température maximale: %0.1f\n", temperatureMax);
}
```









Three ways to pass arrays or array data out of functions

- use **static** memory allocation
  - (1) modify value of array passed by reference
  - (2) use *static* declaration and return pointer from function

- use **dynamic** memory allocation – (3) *malloc* in function

- declarations
  - **static**
    - define by size
    - define by values
    - define by size and values
  - **dynamic**
    - define by size

static (1) modify value of array passed by reference

```
#include <stdio.h>  // for printf
#include <stdlib.h> // for malloc
#include <math.h>   // for pow

#define NELEM 4

void squarearray(const int n, const int *arr_in, int *arr_out) {

  for(int i=0; i < n; i++) {
    arr_out[i] = pow(arr_in[i], 2);
  }

  return;
}

int main() {

  int myarr[NELEM] = {0, 1, 2, 3};
  int out[NELEM];

  squarearray(NELEM, myarr, out);

  for(int i=0; i < nelem; i++) {
    printf("original myarr[%d] = %d; squared out[%d]: %d\n", i, myarr[i], i, out[i]);
  }

  return 0;
}
```

```
original myarr[0] = 0; squared out[0]: 0
original myarr[1] = 1; squared out[1]: 1
original myarr[2] = 2; squared out[2]: 4
original myarr[3] = 3; squared out[3]: 9
```

static (2) use *static* declaration and return pointer from function

```
#include <stdio.h>  // for printf
#include <stdlib.h> // for malloc
#include <math.h>   // for pow

#define NELEM 4

int *squarearray(int *arr_in) {

  static int arr_out[NELEM];

  for(int i=0; i < NELEM; i++) {
    arr_out[i] = pow(arr_in[i], 2);
  }

  return arr_out;
}

int main() {

  int myarr[NELEM] = {0, 1, 2, 3};
  int *out;

  out = squarearray(myarr);

  for(int i=0; i < NELEM; i++) {
    printf("original myarr[%d] = %d; squared out[%d]: %d\n", i, myarr[i], i, out[i]);
  }

  return 0;
}
```

dynamic (3) malloc in function

```c
#include <stdio.h>  // for printf
#include <stdlib.h> // for malloc
#include <math.h>   // for pow

int *squarearray(size_t n, int *arr_in) {

  int *arr_out = malloc(sizeof(int) * n);

  for(int i=0; i < n; i++) {
    arr_out[i] = pow(arr_in[i], 2);
  }

  return arr_out;
}

int main() {

  int myarr[] = {0, 1, 2, 3};
  int *out;

  // assume we need to estimate the number of elements in the array
  size_t nelem = sizeof(myarr)/sizeof(myarr[0]);
  out = squarearray(nelem, myarr);

  for(int i=0; i < nelem; i++) {
    printf("original myarr[%d] = %d; squared out[%d]: %d\n", i, myarr[i], i, out[i]);
  }

  free(out);

  return 0;
}
```

you **must** use sizeof()
before you pass array to function
why?

answer:
1. you pass the location of the first element of the array but it doesn't know the location of the last element
2. if you query the size of the variable within the function, it will return the size of the pointer (memory address)

Other use of dynamic memory allocation

Example directory structure

```
.
├── csv_example_files
│   ├── example_input.csv
│   └── example_output.csv
├── csv_example
└── csv_example.c
```

Contents of csv_example_files/example_input.csv

```
col1,col2,col3
0.0,0.1,0.2
0.3,0.4,0.5
```

csv_example.csv

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

#define NCOL 3

int main(int argc, char *argv[]) {
  // expected syntax:
  //   ./{executable} inputfile.csv outputfile.csv

  int c = EOF;
  char header[1024];
  int i, nlines;

  /* open files for reading and writing */
  FILE *fp = fopen(argv[1], "r");        // "csv_file_example.csv"
  FILE *fout = fopen(argv[2], "w");      // "csv_file_out.csv"

  /* count number of lines */
  nlines=1;
  while ((c=fgetc(fp)) != EOF) {
    if (c=='\n') {
      nlines++;
    }
  }
  printf("%d lines read\n", nlines);

  /* dynamically allocate array */
  double (*arr)[NCOL] = malloc((nlines-1)*NCOL*sizeof(double));

  /* read file  */
  fseek(fp, 0, SEEK_SET);
  fscanf(fp, "%s", header);
  i = 0;
  while(fscanf(fp, "%lf,%lf,%lf", &arr[i][0], &arr[i][1], &arr[i][2]) == 3) {
    i++;
  }

  /* output  */
  fprintf(fout, "col1_squared,col2_squared,col3_squared\n");
  for(i=0; i<nlines-1; i++) {
    fprintf(fout, "%lf,%lf,%lf\n", pow(arr[i][0],2), pow(arr[i][1],2), pow(arr[i][2],2));
  }

  /* deallocate array */
  free(arr);

  /* close files */
  fclose(fout);
  fclose(fp);

  /* exit function */
  return 0;
}
```

Summary - defining array size

- known at time of compilation
    - use array in global scope
    - use static arrays in functions

- determined at runtime
    - use malloc

Structures in C

Why?

- heterogeneous data
- refer to content by name (more understandable code)

Options:

- structure of array(s) – *pass by value*
- array of structures – *pass by reference*

```c
#include <stdio.h>

// Déclarer une structure
struct Sommet {
    char * nom;
    double latitude;
    double longitude;
    float altitude;
};

int main(int argc, char * argv[]) {
    // Créer une variable et remplir les champs
    struct Sommet cervin;
    cervin.nom = "Matterhorn";
    cervin.latitude = 45.97639;
    cervin.longitude = 7.65833;
    cervin.altitude = 4478;

    // ou créer et remplir en même temps
    struct Sommet montBlanc = {"Mont Blanc", 45.832778, 6.865, 4808};

    // Afficher
    printf("%s: %0.0f m alt.\n", cervin.nom, cervin.altitude);
    printf("%s: %0.0f m alt.\n", montBlanc.nom, montBlanc.altitude);
    return 0;
}
```

```c
struct Sommet sommets[10];

sommets[0].nom = "Matterhorn";
sommets[0].latitude = 45.97639;
sommets[0].longitude = 7.65833;
sommets[0].altitude = 4478;

sommets[1].nom = "Weisshorn";
sommets[1].latitude = 46.101667;
sommets[1].longitude = 7.716111;
sommets[1].altitude = 4506;

...
```

Additional syntactic sugar

when we pass a pointer to the structure (note: default is to pass by value)

```
#include <stdio.h>

// Déclarer une structure
struct Sommet {
    char * nom;
    double latitude;
    double longitude;
    float altitude;
};

void afficherSommet(struct Sommet sommet, char *units) {
  printf("%s: %0.0f %s alt.\n", sommet.nom, sommet.altitude, units);
}

void corrigerAltitude(struct Sommet * sommet) {
    // Convertir mètres en feet
    sommet->altitude = sommet->altitude / 0.3048;
}

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

  // declare
  struct Sommet cervin;

  // initialize with values
  cervin.nom = "Matterhorn";
  cervin.latitude = 45.97639;
  cervin.longitude = 7.65833;
  cervin.altitude = 4478;

  afficherSommet(cervin, "meters");

  corrigerAltitude(&cervin);

  afficherSommet(cervin, "feet");

  return 0;
}
```

Vu que *sommet* est un pointeur, on utilise `->` pour accéder aux valeurs de la structure. Ceci est une syntaxe abrégée pour:

```
(*sommet).altitude = (*sommet).altitude / 0.3048;
```