# C introduction part 4

2D arrays

# 2D arrays

Representation

- Stored with 1D arrays with additional overhead for checking bounds for each dimension

- Can represent with 1D array and manually keep track of indices

Memory allocation

- static

- [dynamic](dynamic)

# Example – static arrays

```c
#include <stdio.h>
#define NROW 3
#define NCOL 4

int idx(int i, int j) {
  return i*NCOL + j;
}

int main() {
  int arr1D[NROW*NCOL] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
  int arr2D[NROW][NCOL] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

  int i, j;

  for(i=0; i<NROW; i++) {
    for(j=0; j<NCOL; j++) {
      printf("row %d, col %d --> arr1D = %d, arr2D = %d\n", i, j, arr1D[idx(i, j)], arr2D[i][j]);
    }
  }

  return 0;
}
```

# Initializing 2D arrays

```c
int a[3][4] = {
    {0, 1, 2, 3} ,    /*  initializers for row indexed by 0 */
    {4, 5, 6, 7} ,    /*  initializers for row indexed by 1 */
    {8, 9, 10, 11}    /*  initializers for row indexed by 2 */
};
```

```c
int a[][4] = {
    {0, 1, 2, 3} ,    /*  initializers for row indexed by 0 */
    {4, 5, 6, 7} ,    /*  initializers for row indexed by 1 */
    {8, 9, 10, 11}    /*  initializers for row indexed by 2 */
};
```

# Example – dynamic arrays

2D array as 1D array (user responsible for tracking indices)

```
double *a = malloc(nrow * ncol * sizeof(double));
```

explicit 2D array - as with static arrays, omit first dimension in declaration

```
double (*a)[ncol] = malloc(nrow * ncol * sizeof(double));

double (*a)[ncol] = malloc(sizeof(double[nrow][ncol]);
```

note that the placement of parentheses is very important as *a[ncol] is an array of pointers rather than a pointer to an array

# When to use which

Static arrays

- if you know the size beforehand (or maximum size)
  - Use #define directive to declare array dimensions as constants (right after your #include statements)
  - Pass around variables that contain the actual number of columns and rows if different from size of array.

Dynamic arrays

- if you don't know the size when you compile the program (e.g., depends on input)
- you want to control the lifetime of the array

# Lifetime of the array
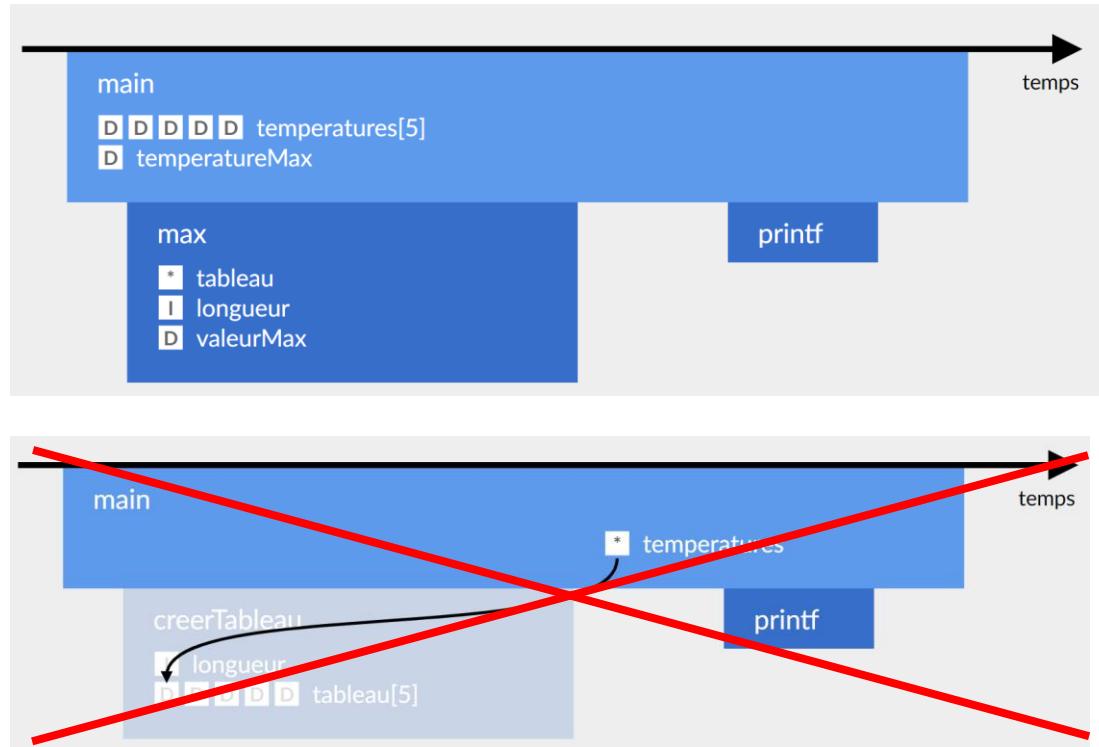


**Passing arrays or array data out of functions**

1. with *malloc.* Array exists until free() is called.

2. *static* array declaration in function. Array exists until end of program.

3. modify value of global arrays passed as *pointers.* (If global arrays are created with malloc, then point 1 applies.)

# Stack vs. heap memory

- Static and dynamic arrays use a different portion of the memory
  - *heap* memory is created and reallocated during the program (dynamic arrays)
  - *stack* memory (not to be confused with the "call stack" which keeps track of program execution) keeps track of local variables

- "Extremely large" objects should not be created on the stack (can cause *stack overflow*).

- Not having enough heap memory at runtime can also cause the program to crash.



(see memoire-virtuelle)
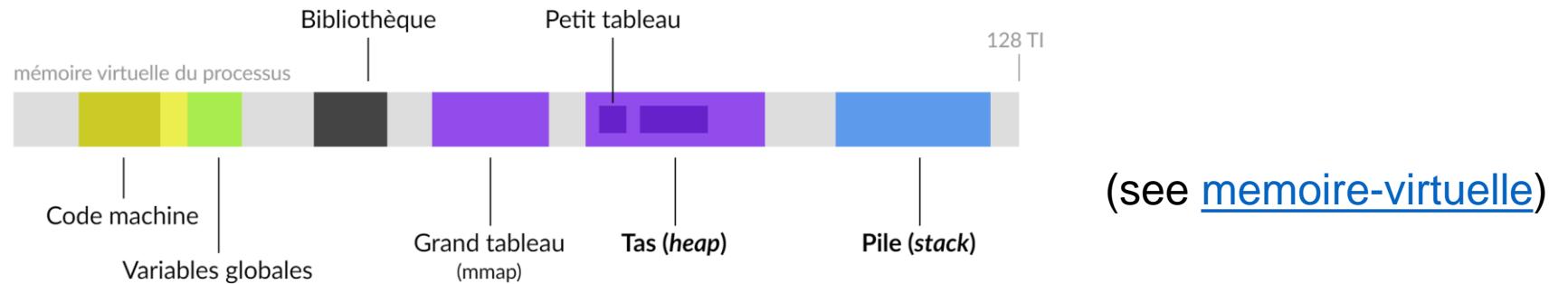
# Review

pass by reference vs. pass by value

```c
#include <stdio.h>

struct Example {
  char string[4];
  double x;
};

void fn(double a, double *b, char *s, double *arr, struct Example ex1, struct Example *ex2, struct Example *exarr) {

  a = 1.;
  *b = 1.;

  /* -------------------------------------------------------------------- */
  s[0] = 'a';
  s[1] = 'b';
  s[2] = 'c';
  s[3] = 0;

  /* -------------------------------------------------------------------- */
  arr[0] = 1.;
  arr[1] = 2.;

  /* -------------------------------------------------------------------- */
  char *localstring = "abcdef";

  /* -------------------------------------------------------------------- */
  for(int i=0; i<3; i++) {
    ex1.string[i] = localstring[i];
  }
  ex1.string[3] = 0;
  ex1.x = 1.;

  /* -------------------------------------------------------------------- */
  for(int i=0; i<3; i++) {
    (*ex2).string[i] = localstring[i];    // or ex2->string[i] = localstring[i];
  }
  (*ex2).string[3] = 0;
  (*ex2).x = 1.;              // or ex2->x = 1.

  /* -------------------------------------------------------------------- */
  for(int i=0; i<3; i++) {
    exarr[0].string[i] = localstring[i];
  }
  exarr[0].string[3] = 0;
  exarr[0].x = 1.;

  for(int i=0; i<3; i++) {
    exarr[1].string[i] = localstring[i+3];
  }
  exarr[1].string[3] = 0;
  exarr[1].x = 2.;

  /* -------------------------------------------------------------------- */
  return;
};
```

*Can you predict the output?*

```c
int main() {

  double a=0.;
  double b=0.;
  char string[4];
  double array[2];
  struct Example examp1;
  struct Example examp2;
  struct Example examparray[2];

  fn(a, &b, string, array, examp1, &examp2, examparray);

  printf("%f\n", a);
  printf("%f\n", b);
  printf("string = %s\n", string);
  printf("array[0] = %f, array[1] = %f\n", array[0], array[1]);
  printf("examp1.string = %s, examp1.x = %f\n", examp1.string, examp1.x);
  printf("examp2.string = %s, examp2.x = %f\n", examp2.string, examp2.x);
  printf("examparray[0].string = %s, examparray[0].x = %f\n", examparray[0].string, examparray[0].x);
  printf("examparray[1].string = %s, examparray[1].x = %f\n", examparray[1].string, examparray[1].x);

  return 0;
}
```