

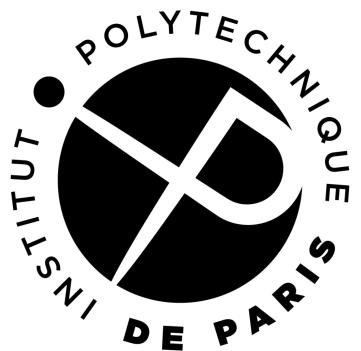
Memory management

Bachelor of Science - École polytechnique

gael.thomas@inria.fr

Key concepts

- An array behaves like a pointer
- A local array or structure is allocated within the frame
- A caller receives
 - A pointer to an array for an array
 - A copy of a structure for a structure
- To copy an array or string: `memcpy`, `strncpy`, `sizeof`
- Dynamic memory management: `malloc` and `free`



Array, structure and call frame

Array, structure and call frame

- A local array or structure is allocated within a call frame
 - (i.e., not allocated if the function is not called)

```
void f() {
    struct strange s = {
        .c: 'a',
        .i: 3,
        .f: 3.14 };
    int t[3] = { 7, 8, 9 };
}

int main(int argc, char* argv[]) {
    f();
    return 0;
}
```

```
struct strange {
    char c;
    int i;
    float f;
};
```



Array, structure and call frame

- A local array or structure is allocated within a call frame
 - (i.e., not allocated if the function is not called)

```
void f() {  
    struct strange s = {  
        .c: 'a',  
        .i: 3,  
        .f: 3.14 };  
    int t[3] = { 7, 8, 9 };  
}
```

```
→ int main(int argc, char* argv[]) {  
    f();  
    return 0;  
}
```

argc	1
argv	a value
	frame of main

Array, structure and call frame

- A local array or structure is allocated within a call frame
 - The memory is allocated when the function is called

```
void f() {  
    struct strange s = {  
        .c: 'a',  
        .i: 3,  
        .f: 3.14 };  
    int t[3] = { 7, 8, 9 };  
}  
  
int main(int argc, char* argv[]) {  
    f();  
    return 0;  
}
```

argc	1		
argv	a value		
frame of main			
s	'a'	3	3.14
t	7	8	9
frame of f			

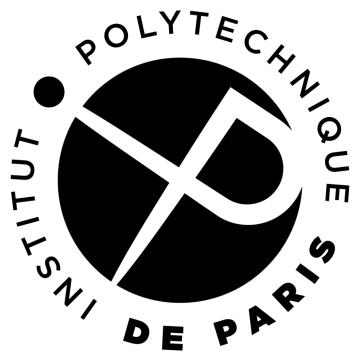
Array, structure and call frame

- A local array or structure is allocated within a call frame
 - The memory is allocated when the function is called
 - And released at the end of the function

```
void f() {  
    struct strange s = {  
        .c: 'a',  
        .i: 3,  
        .f: 3.14 };  
    int t[3] = { 7, 8, 9 };  
}  
  
int main(int argc, char* argv[]) {  
    f();  
    return 0;  
}
```

argc	1
argv	a value
	frame of main





Structure and function parameters

Structure and function parameters

- When a parameter is a structure, the argument is fully copied
 - Can take time if the structure is large

```
void fct(struct strange y) {  
}  
  
void main(int argc, char* argv[]) {  
    struct strange x = {  
        .c: 'a',  
        .i: 3,  
        .f: 3.14  
    };  
    fct(x);  
    return 0;  
}
```

argc	1		
argv	a value		
x	'a'	3	3.14
frame of main			
y	'a'	3	3.14
frame of fct			

Structure and function parameters

- When a parameter is a structure, the argument is fully copied
 - Can take time if the structure is large
 - A modification in the caller modifies the copy, not the original

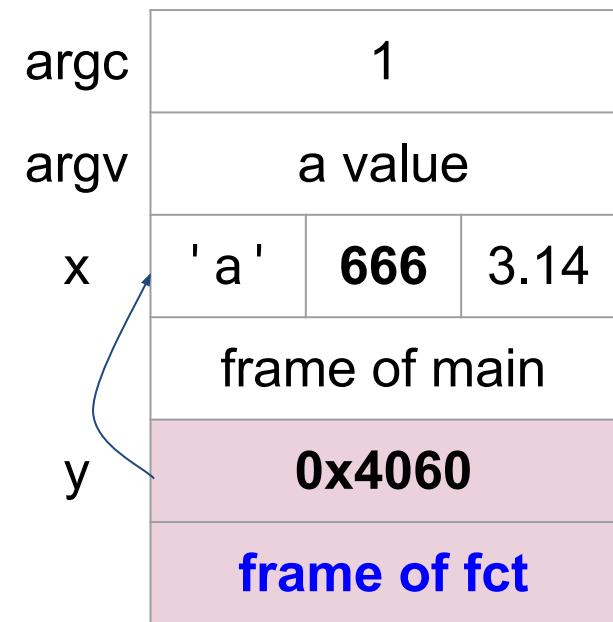
```
void fct(struct strange y) {  
    y.i = 666;  
}  
  
void main(int argc, char* argv[]) {  
    struct strange x = {  
        .c: 'a',  
        .i: 3,  
        .f: 3.14  
    };  
    fct(x);  
    return 0;  
}
```

argc	1		
argv	a value		
x	'a'	3	3.14
frame of main			
y	'a'	666	3.14
frame of fct			

Structure and function parameters

- To avoid the copy or to modify the structure in the caller
 - Use a pointer!

```
void fct(struct strange* y) {  
    (*y).i = 666;  
}  
  
void main(int argc, char* argv[]) {  
    struct strange x = {  
        .c: 'a',  
        .i: 3,  
        .f: 3.14  
    };  
    fct(&x);  
    return 0;  
}
```



The arrow operator: ->

- To avoid the copy or to modify the structure in the caller
 - Use a pointer!

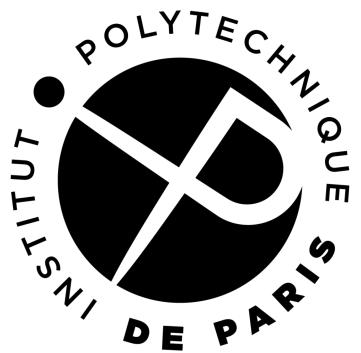
```
void fct(struct strange* y) {  
    y->i = 666;  
}
```

argc	1
argv	a value

Accessing a field of a structure through a pointer is so common that the C language has a specific operator for that:

the -> operator

$$y->i \Leftrightarrow (*y).i$$



Array and pointers

Array and pointer

- We can access an array through a pointer
 - $p = \text{tab} \Rightarrow p$ points to the first element of the array
 - Here, we suppose that tab is allocated at $0x4060$
 - Note that we don't need the $\&$ operator in this case!

```
void fct(int x) {  
    int tab[3];  
    int* p = tab;  
    for(int i=0; i<3; i++) {  
        *(p + i) = x + i;  
    }  
}  
  
int main(int argc, char* argv[]) {  
    fct(7);  
    return 0;  
}
```



	...
	frame of main
x	7
p	0x4060
tab	
i	
	frame of fct

Array and pointer

- We can access an array through a pointer
 - $p + i$ computes a pointers to the i^{th} element of the array
 - In details, $0x4060 + i*4$ since p points to an `int` (4 bytes)

```
void fct(int x) {  
    int tab[3];  
    int* p = tab;  
    for(int i=0; i<3; i++) {  
        *(p + i) = x + i;  
    }  
  
    int main(int argc, char* argv[]) {  
        fct(7);  
        return 0;  
    }
```

		...
		frame of main
x	7	
p	0x4060	
tab	7	8
i	1	
		frame of fct

Array and pointer

- In fact, a variable with the type array **behaves like** a pointer!
 - That's why we don't need the & operator
 - The difference between a pointer and an array is that the C language allocates the space for the array for an array, but allocates the space for a pointer for a pointer

```
void fct(int x) {  
    int tab[3];  
    for(int i=0; i<3; i++) {  
        *(tab + i) = x + i;  
    }  
}  
  
int main(int argc, char* argv[]) {  
    fct(7);  
    return 0;  
}
```



		...
		frame of main
x	7	
tab	7	8
i	1	
		frame of fct

Array and pointer

- In fact, a variable with the type array **behaves like** a pointer!
- And we can use a pointer as an array!
- $*(p + i)$ and $p[i]$ are totally equivalent

```
void fct(int x) {  
    int tab[3];  
    int* p = tab;  
    for(int i=0; i<3; i++) {  
        p[i] = x + i;  
    }  
  
    int main(int argc, char* argv[]) {  
        fct(7);  
        return 0;  
    }
```

		...
	x	7
p	0x4060	
tab	7	8
i	1	
	frame of fct	

Array and function parameters

- If a parameter has the type array, the array is not copied
 - A pointer to the array is passed
 - We say that arrays are passed by pointer

```
void fct(int tab[]) {  
    for(int i=0; i<3; i++) {  
        tab[i] = -1;  
    }  
}  
  
int main(int argc, char* argv[]) {  
    int x[3] = { 4, 5, 6 };  
    fct(x);  
    return 0;  
}
```



argc	7		
argv	a value		
x	4	5	6
	frame of main		

Array and function parameters

- If a parameter has the type array, the array is not copied
 - tab receives a pointer to the array x (0x4060 for the example)

```
void fct(int tab[]) {  
    for(int i=0; i<3; i++) {  
        tab[i] = -1;  
    }  
}  
  
int main(int argc, char* argv[]) {  
    int x[3] = { 4, 5, 6 };  
    fct(x);  
    return 0;  
}
```

argc	7		
argv	a value		
x	4	5	6
	frame of main		
tab	0x4060		
i			
	frame of fct		

Array and function parameters

- If a parameter has the type array, the array is not copied
 - `tab[i] = -1` modifies the array `x`

```
void fct(int tab[]) {
    for(int i=0; i<3; i++) {
        tab[i] = -1;
    }
}

int main(int argc, char* argv[]) {
    int x[3] = { 4, 5, 6 };
    fct(x);
    return 0;
}
```



argc	7		
argv	a value		
x	-1	-1	6
	frame of main		
tab	0x4060		
i	1		
	frame of fct		

Array and function parameters

- There is no fundamental difference between
 - Declaring a parameter with an array type
 - Or with a pointer type

```
void fct(int tab[]) {  
    for(int i=0; i<3; i++) {  
        tab[i] = -1;  
    }  
}
```



```
void fct(int* tab) {  
    for(int i=0; i<3; i++) {  
        tab[i] = -1;  
    }  
}
```

Array and string

■ A string is not a string

- It's an array of characters (`char[]`)
- That ends with the integer `0`, which indicates the end
- And `char*` is a pointer to the first character of the string

```
int main(int argc, char* argv[]) {
    char* str = "hello";

    for(int i=0; str[i] != 0; i++) {
        printf("%c\n", str[i]);
    }
    // => h, e, l, l, o on each line
    return 0;
}
```

Array and string

- A string literal is a shortcut that
 - Creates a hidden global variable with the type `char[]`
 - Initialized with the characters of the string followed by `\0`
 - Replaces the literal by the hidden variable

```
int main(int argc, char* argv[]) {  
    char* str = "hello";
```

```
    ...
```



```
char hidden[] = { 'h', 'e', 'l', 'l', 'o', \0 };
```

```
int main(int argc, char* argv[]) {  
    char* str = hidden;
```

```
    ...
```

Array and copy

- `t1 = t0` does not copy the array `t0` into the array `t1`
 - The expression either fails or copies the pointers
- To copy an array
 - `memcpy(dst, src, n)`: copy `n` bytes from `src` to `dst`
 - `strncpy(dst, src, n)`: copy the string `src` into `dst`
 - Stop the copy when the end of `src` is found (value `0`)
 - Or when `n` bytes are copied (avoid bugs if `dst` is too small)
- To use `memcpy`, we have to know the size of the elements
 - `sizeof(type)`: give the size of `type`
 - `sizeof(*var) =>` give the size of the type of the value pointed by `var` (provided that `var` is a pointer)

Array and copy

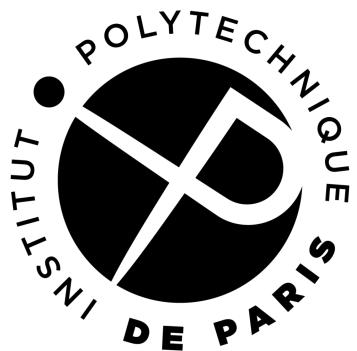
- To use `memcpy` and `strncpy`, you have to include `<string.h>`

```
#include <string.h>

void test() {
    int t0[3] = { 1, 2, 3 };
    int t1[3];
    memcpy(t1, t0, sizeof(*t0) * 3);
}
```

```
#include <string.h>

void test() {
    char* s0 = "hello";
    char s1[128];
    strncpy(s1, s0, 128);
    // copy 6 bytes
    // 5 for hello
    // + 1 for the final 0
}
```



Heap and dynamic memory management

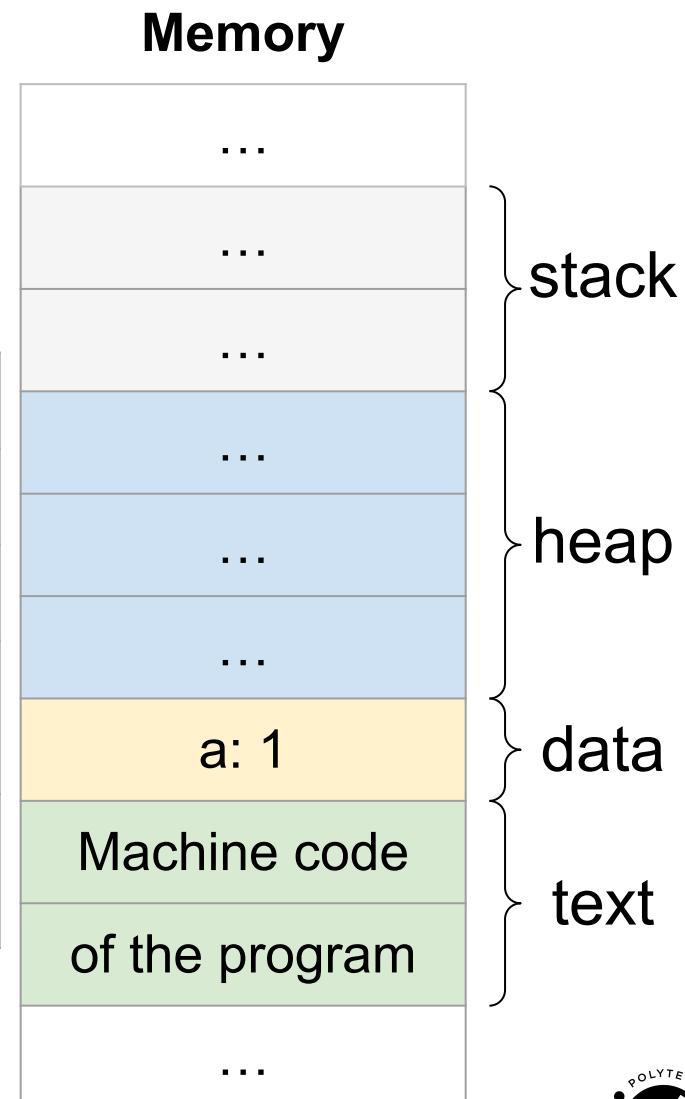
Why dynamic allocation

- Sometimes, the lifetime of a data structure does not match the lifetime of a function, for example:
 - The objects owned by a character in a video game
 - The user accounts of a web server
 - ...
- Why the lifetime cannot match in these cases
 - The order of frame allocations gives the order of release
 - f call g => allocate frame f, allocate frame g, free frame g, free frame f
 - But for many objects, allocating and freeing them can have any order
 - find a potion, find a sword, drink the potion but keep the sword

The heap

- The heap is a fourth segment used to **explicitly** and **dynamically** allocate and free memory

	Used for	Read/Write	Data lifetime
Stack	Frames	R/W	Dynamic
Heap	Dynamic data	R/W	Dynamic
Data	Global variables	R/W	Whole execution
Text	Code	R	Whole execution



Dynamic memory allocation

- `void* malloc(size_t n)`
 - `size_t`: a type used to represent a size (`unsigned long int`)
 - `n`: the number of bytes to allocate
 - `void*`: returns a pointer to an unknown type

```
struct pokemon {
    char name[256];
    int health;
};

int main(int argc, char* argv[]) {
    struct pokemon* p = malloc(sizeof(*p));
    strncpy(p->name, "Pikachu", 256);
    p->health = 78;
    return 0;
}
```

Memory allocation and array

- To allocate an array, simply allocates N times a data structure

```
struct pokemon {
    char name[256];
    int health;
};

int main(int argc, char** argv) {
    // allocate 42 pokemons
    struct pokemon* p = malloc(42 * sizeof(*p));
    // allocate 666 int
    int* q = malloc(666 * sizeof(*q));
    q[33] = -1;
    return 0;
}
```

Free the memory of a data structure

- `void free(void* ptr)`
 - `ptr` has to have been allocated by `malloc`
(otherwise the application crashes at runtime)

```
struct pokemon {  
    char name[256];  
    int health;  
};  
  
int main(int argc, char** argv) {  
    struct pokemon* p = malloc(42 * sizeof(*p));  
    int* q = malloc(666 * sizeof(*q));  
  
    free(p);  
    free(q);  
  
    return 0;  
}
```

Key concepts

- An array behaves like a pointer
- A local array or structure is allocated within the frame
- A caller receives
 - A pointer to an array for an array
 - A copy of a structure for a structure
- To copy an array or string: `memcpy`, `strncpy`, `sizeof`
- Dynamic memory management: `malloc` and `free`