

Information, Calcul et Communication

Composante Pratique: Programmation C++

MOOC sem7 : pointeur (2) *le retour...*

Pointeur et structure : utiliser l'opérateur ->

Exemples d'usage de pointeurs

Mémoire centrale: pile (stack) et tas (heap)

Pointeur et allocation dynamique de mémoire

Opérateurs sur les pointeurs et accès aux champs d'une structure

Accès à un champ d'une **variable de type struct** avec **.**

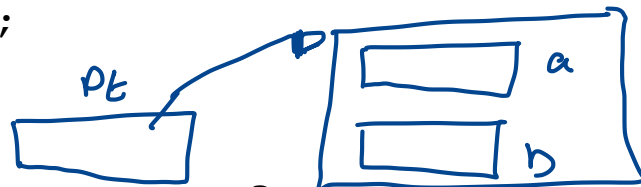
Avec un **pointeur sur une struct** on accède à un champ avec **->**

Indirection/déréférencement avec *****

Calcul de l'adresse d'une variable avec **&**

Or l'opérateur d'indirection ***** a une priorité plus faible
que l'opérateur d'accès à un champ d'une structure **.**

```
struct My_struct {string a; string b;};
My_struct s = {"AAA", "BBB"};
My_struct* pt = &s;
```



S.a *Δ *pt.a qui est Faux*

```
cout << (*pt).a << endl; // affiche: AAA
cout << pt->a << endl; // affiche: AAA
```

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a--	Suffix/postfix increment and decrement	Left-to-right
	type() type{}	Functional cast	
	a()	Function call	
	a[]	Subscript	
3	.	Member access	Right-to-left
	++a --a	Prefix increment and decrement	
	+a -a	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	C-style cast	
	*a	Indirection (dereference)	
	&a	Address-of	
	sizeof	Size-of ^[note 1]	
	co_await	await-expression (C++20)	
	new new[]	Dynamic memory allocation	
	delete delete[]	Dynamic memory deallocation	
4	.* ->*	Pointer-to-member	Left-to-right
5	a*b a/b a%b	Multiplication, division, and remainder	Left-to-right
6	a+b a-b	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	<=>	Three-way comparison operator (since C++20)	
9	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
10	== !=	For relational operators = and ≠ respectively	
11	&	Bitwise AND	
12	^	Bitwise XOR (exclusive or)	
13		Bitwise OR (inclusive or)	
14	&&	Logical AND	Right-to-left
15		Logical OR	
16	a?b:c	Ternary conditional ^[note 2]	
	throw	throw operator	
	co_yield	yield-expression (C++20)	
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Compound assignment by sum and difference	
	*= /= %=	Compound assignment by product, quotient, and remainder	
	<<= >>=	Compound assignment by bitwise left shift and right shift	
	&= ^= =	Compound assignment by bitwise AND, XOR, and OR	
17	,	Comma	Left-to-right

utiliser l'opérateur **->** avec un pointeur sur une structure pour la lisibilité du code

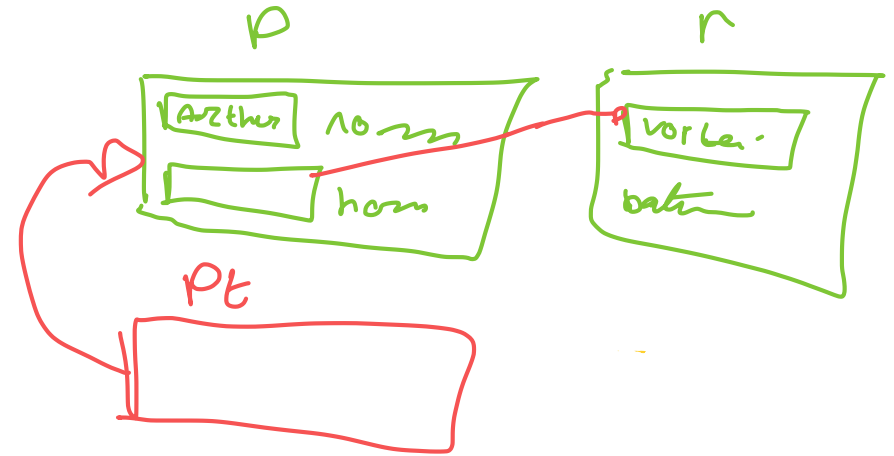
```
struct Residence {
    string batiment;
};
struct Personne {
    string nom;
    Residence *home;
};
int main()
{
    Residence    r = {"Vortex"};
    Personne     p = {"Arthur", &r};
    Personne *ptp = &p ;

    // 3 possibilités pour afficher
    // le champ batiment à partir de p :
    cout << (* (p.home) ).batiment << endl;

    cout <<      p.home->batiment << endl;

    cout <<  ptp->home->batiment << endl;

EPI cout <<  (* (*ptp) .home) .batiment << endl;
```



// parenthèses obligatoires

// OK ; associativité gauche -> droite

// OK ; associativité gauche -> droite

// écriture correcte mais très peu lisible

Les 2 sortes d'usage de `const` avec des pointeurs

Question
Examen

- Règle: 1) `const` s'applique au type qui précède
2) s'il n'y en a pas, `const` s'applique au type qui suit

```
int a(0), b(0), c(0);  
int const *ptr1(&a);  
const int* ptr2(&b);  
const int *ptr3(&c);
```

Le compilateur signale une erreur si on cherche à modifier la valeur `int` de la variable pointée (a, b ou c) *à l'aide du pointeur* (resp. ptr1, ptr2 ou ptr3).

Par contre on peut modifier la valeur des pointeurs.

Ex: `ptr1 = ptr2 ; // ptr1 pointe sur b`

```
int d(0);  
int* const ptr4(&d);
```

Le compilateur signale une erreur si on cherche à modifier la valeur **du pointeur ptr4**.

Par contre on peut modifier la valeur pointée par ptr4

Ex: `*ptr4 = 99 ; // modifie la valeur de d`

```
int e(0);  
const int* const ptr5(&e);
```

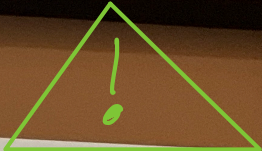
Le compilateur signale une erreur si on cherche à modifier la valeur `int` de `e` *à l'aide de ptr5* ou si on cherche à modifier la valeur du pointeur **ptr5**.

Quizz1: What are the types of the declared variables a, b, c and d ?
 (that compiles successfully):

`int a;`

`int* b, c, &d(a);` c => `int* b`
`int c`
`int* &d(a)`

	int	pointeur sur un int	référence sur un int	référence sur un pointeur sur un int
A	a et c	b et d		
B	a	b, c et d		
C	a	b et c		d
D	a et c	b	d	
E	a et c	b		d
F	a	b et c	d	



Analyse des déclarations

```
int a;  
int* b, c, &d(a);
```

Le symbole * se rapporte seulement à **b** ;
il ne s'applique pas au reste des déclarations sur la ligne

```
int *b, c, &d(a);
```

Cette écriture montre plus clairement que * est seulement associé à la déclaration de b

Bonne Pratique !

```
int a(0);  
int *b(&a), c(0), &d(a);
```

Usage de pointeurs pour des structures de données complexes : arbre

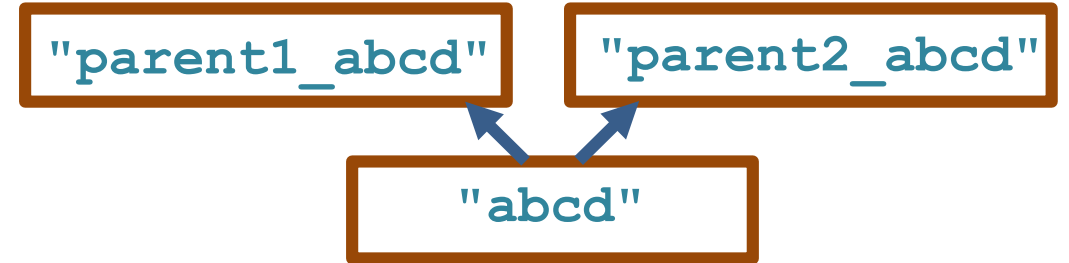
But: construire l'arbre généalogique d'une personne

Outil: définir une structure **Nœud_ADN** qui contient le nom de la personne et un champ décrivant chacun de ses deux parents biologiques.

Idée: utiliser la même structure **Nœud_ADN** pour représenter chaque parent. Permet de représenter l'arbre de manière homogène.

Problème: **une structure ne peut pas se contenir elle-même.**
Car le compilateur ne peut pas calculer l'espace mémoire nécessaire pour une variable de ce type.

Solution: une structure peut contenir un lien (**pointeur**) vers la structure de chaque parent, initialisable avec la valeur **nullptr** si elle n'est pas connue.



```
struct Nœud_ADN{  
    string nom;  
    type-à-définir parent1;  
    type-à-définir parent2;  
};
```

```
struct Nœud_ADN{  
    string nom;  
    const Nœud_ADN parent1;  
    const Nœud_ADN parent2;  
};
```

```
struct Nœud_ADN{  
    string nom;  
    const Nœud_ADN *parent1=nullptr;  
    const Nœud_ADN *parent2=nullptr;  
};
```

Usage de pointeurs pour des structures de données complexes : réseau d'amis = graphe

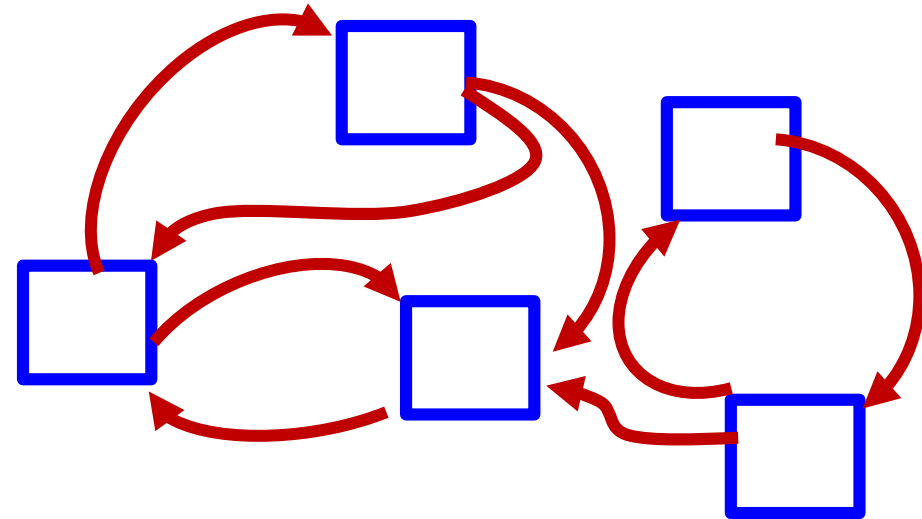
Besoin: définir une structure **Personne** qui contient le nom de la personne et *la liste de ses amis*.

Idée: utiliser la même structure **Personne** pour représenter un(e) ami(e). Permet de représenter un réseau d'amis de manière homogène.

Problème: une structure ne peut pas se contenir elle-même.

Solution: une structure peut contenir une liste de liens (pointeurs) vers d'autres structures.

```
struct Personne{
    string nom;
    // ... ? ...
};
```



```
struct Personne{
    string nom;
    vector<const Personne*> amis;
};
```

code liste liens.cc

Pointeur et tableau à-la-C de chaine-à-la-C : la vérité sur main()


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

↳ argument count *↳ argument value*

La fonction **main()** peut avoir deux paramètres :

- Un compteur entier **argc** valant au moins 1. *→ le nom de l'exécutable*
- Un **tableau à-la-C** **argv** de **argc** **pointeurs** vers des **chaînes à-la-C**

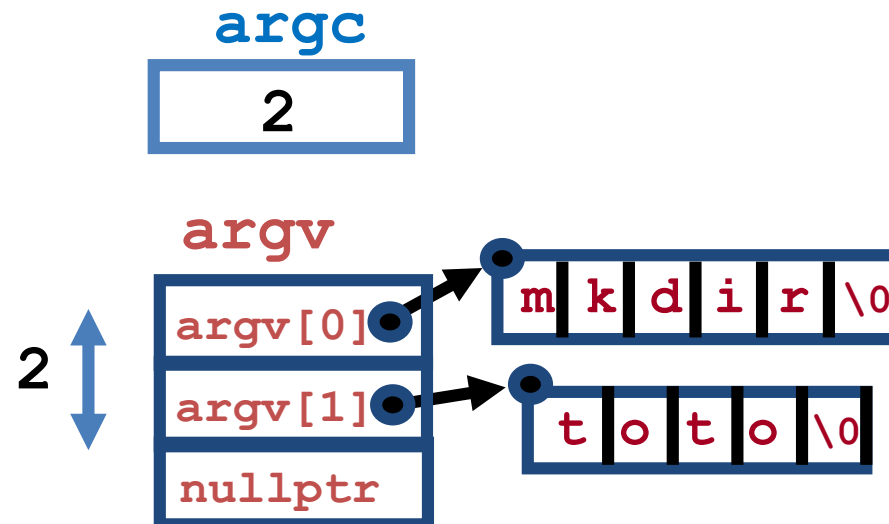
Permet de fournir un nombre variable de paramètres aux commandes Linux.

Ex: **mkdir toto**  **main()** de la commande **mkdir** reçoit **2** pointeurs vers les chaînes à-la-C **mkdir** et **toto**

Comment utiliser **argv** ?

Chaque élément **argv[i]** du tableau **argv** pour **i** compris entre **0** et **argc-1** est un pointeur qui contient l'adresse d'une chaîne à-la-C :

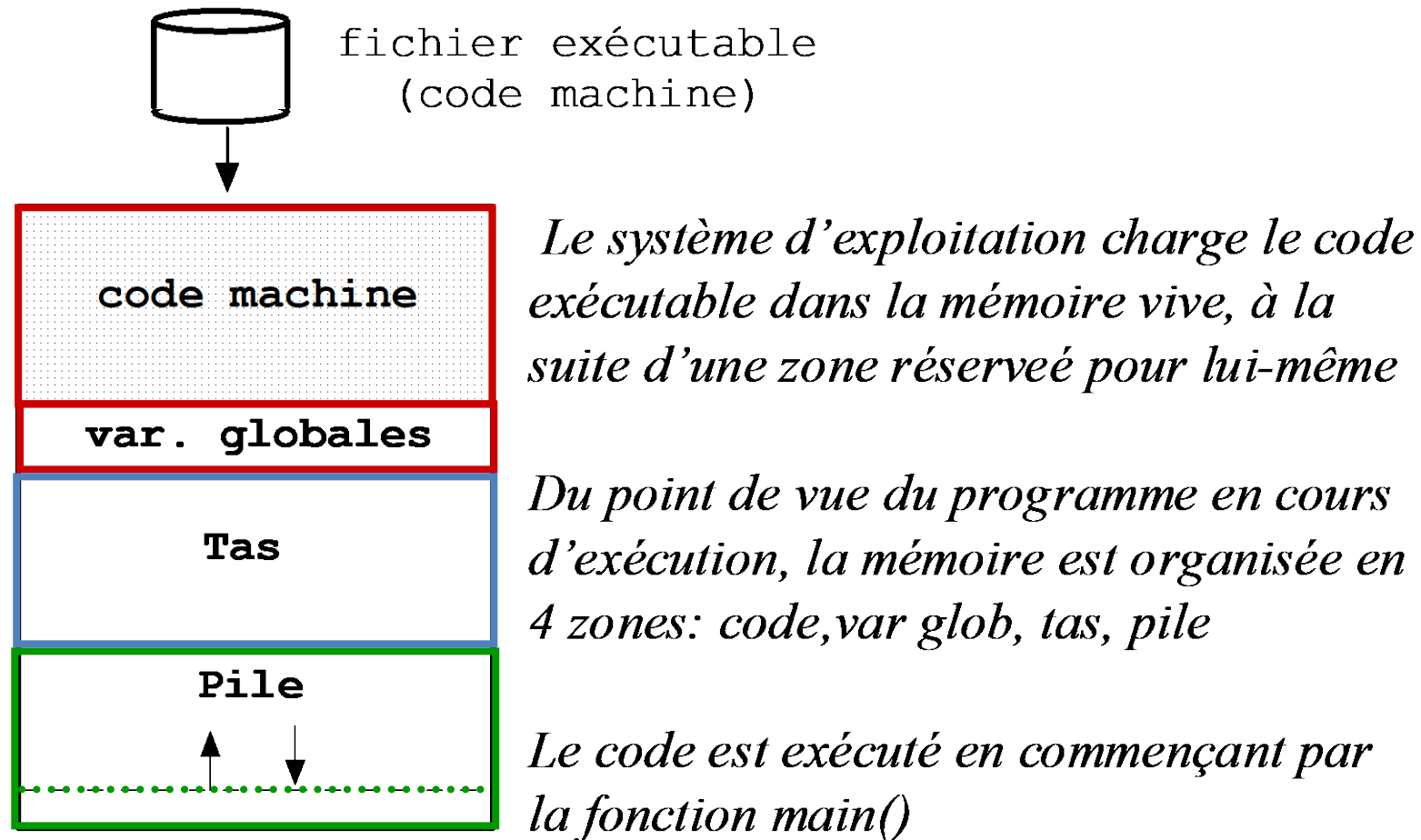
argv[0] pointe vers le nom de l'exécutable



Mémoire centrale: pile (*stack*) et tas (*heap* ou *free store*)

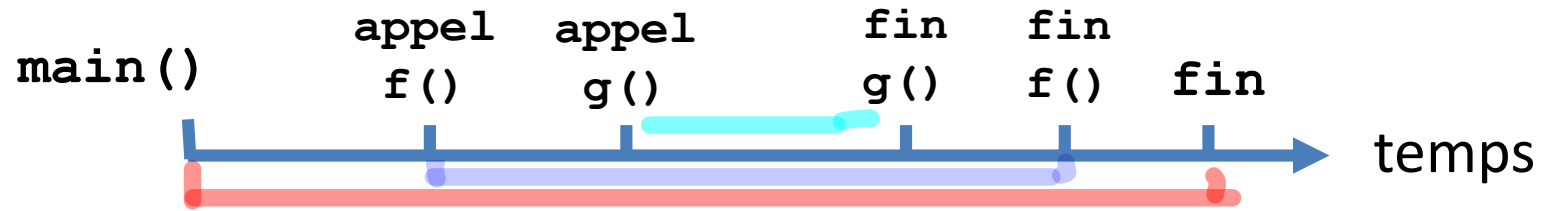
Si nécessaire revoir les cours des Topic5 et Topic6 qui illustrent le fonctionnement de la Pile

Organisation de la mémoire pour l'exécution d'un programme



Le tas (*heap/free store*) offre une durée de vie étendue

Durée de vie limitée sur la Pile = exécution de la fonction où la variable est déclarée



But de la zone mémoire «Tas»:

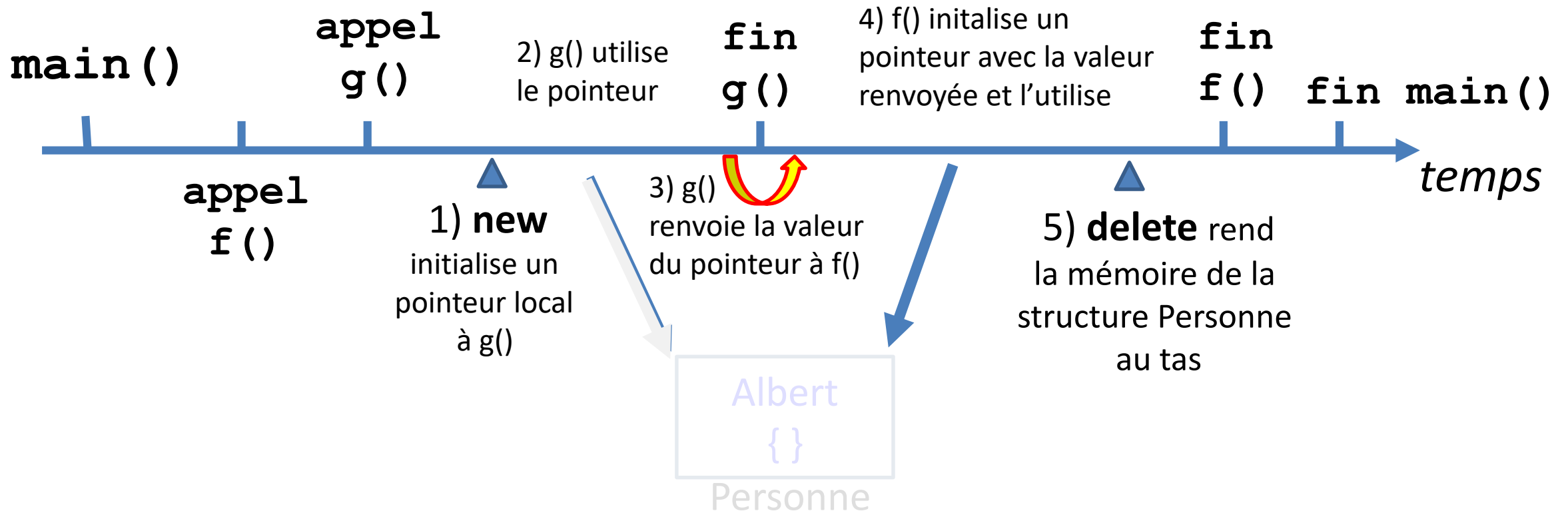
- Se libérer de la durée de vie limitée sur la pile
- Pouvoir **demander de la mémoire librement**, quand on en a besoin pendant l'exécution.
 - Éviter d'avoir à bloquer de manière statique, au moment de l'écriture du programme, de grandes zones de mémoire dans des array ou tableaux à-la-C.
- Pouvoir **libérer la mémoire quand elle n'est plus nécessaire**
- Outil: un pointeur pour mémoriser l'adresse renvoyée par **new**
la valeur du pointeur doit être transmise à l'extérieur d'une fonction qui se termine pour pouvoir continuer à utiliser la mémoire allouée dynamiquement

Le tas (heap) offre une durée de vie étendue (2)

code alloc_dyn_simple.cc

But de la zone mémoire «Tas»:

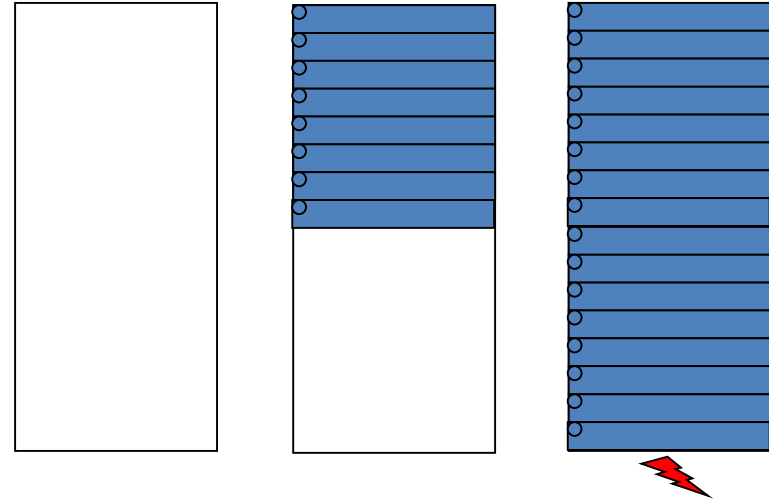
- Se libérer de la durée de vie limitée sur la pile
- Outil: un pointeur pour mémoriser/transmettre l'adresse renvoyée par **new**



Le bon usage de l'allocation dynamique

Ne pas oublier de **libérer** la mémoire allouée avec **delete** quand on n'en a plus besoin

Sinon on crée une *fuite de mémoire (memory leak)* qui fait planter le programme lorsqu'il n'y a plus de place dans le Tas (heap).



**L'usage des pointeurs à-la-C avec l'allocation dynamique
présente beaucoup de risques
demande une grande rigueur**

Il faut au moins comprendre le code écrit avec ces pointeurs à-la-C

**Par manque de temps le cours ne développe pas d'exercices sur les smart pointers au sem1
et aussi parce qu'on dispose de l'outil puissant de vector**

vector est une bonne alternative à l'allocation dynamique avec des pointeurs à-la-C

vector permet **d'ajuster dynamiquement la quantité de mémoire utilisée**

S'il est déclaré comme variable locale, un **vector** existe seulement pendant la durée d'exécution de la fonction où il est déclaré.

Pour augmenter sa durée de vie, une approche simple est de:

- Déclarer le **vector** dans une fonction de haut niveau
 - Ex : dans `main()`
- Passer le **vector** par référence aux fonctions appelées par `main()`
 - Passer une référence **const** si le **vector** ne doit pas être modifié dans les fonctions

Annexe: ordre de grandeur du coût calcul de l'allocation-libération de mémoire

Les opérations d'allocation dynamique de mémoire avec **new** et de libération avec **delete** ne sont pas gratuites en termes de coût calcul.

Par exemple, voici la comparaison de l'estimation (en ns) de quelques opérations en langage C/C++ sur un laptop avec un processeur CORE i7 en 2018 (*valeurs dépendantes du processeur*):

<code>k = i+j</code>	2 ns	
<code>k = i*j</code>	2 ns	
<code>k = i/j</code>	3 ns	
<code>f = sqrt(i+j)</code>	5 ns	
<code>f = sin(i+j)</code>	103 ns	
<code>free(malloc(16))</code>	26 ns	⇔ new de 16 octets suivi par delete
<code>free(malloc(100))</code>	26 ns	⇔ new de 100 octets suivi par delete
<code>free(malloc(2000))</code>	38 ns	⇔ new de 2000 octets suivi par delete

temps obtenus avec `timemod.c` de John Bentley dans son livre «Programming pearls» 1999.