Licence Informatique et Vidéoludisme

Programmation avancée
de C à C++

Farès Belhadj
Revekka Kyriakoglou

Cours et exercices
14 novembre 2023

Plan

- Présentation
- Révisions
  - Préambule
  - Notions de base
  - Exercices de révision
- Du côté de la mémoire
  - La rue de la RAM
- Structures de données dynamiques
  - La liste chaînée
- Arbres binaires
- La généricité en C
- Généricité par préprocesseur
- Quelques éléments de C++
- La STL et les templates

Présentation générale

(les points cités ci-après ne sont pas nécessairement abordés dans l'ordre)

- Rapides révisions de notions abordées en *Programmation Impérative*.
- Maîtriser les concepts avancés de la programmation C : renforcement pointeurs, allocation mémoire et organisation multi-fichiers ; structures de données efficaces ; mesure de performances.
- Apprendre à utiliser une sélection d'outils : `make`, utilisation/création de bibliothèques, `gnuplot`, `GraphViz`, `gprof`, `valgrind`, ...
- La généricité en C (usage des `void *`).
- Utilisation poussée des macros (*C preprocessor*).
- Introduction au C++ sans paradigme objet.
- Utilisation de la STL (C++).

Préambule 1/3

Saisir ce code, sans faire un copier/coller, dans un éditeur convenable ayant au minimum une coloration syntaxique et une capacité à l'indenter correctement.

```c
/* Pour utiliser printf */
#include <stdio.h>
/* Fonction principale du programme */
int main(int argc, char ** argv) {
  int i;
  printf("Ce programme se nomme << %s >>\n", argv[0]);
  for(i = 1; i < argc; ++i)
    printf("Son paramètre %d est << %s >>\n", i, argv[i]);
  return 0; /* Pourquoi 0 ? Peut-être remplacé par EXIT_SUCCESS */
}
```

Code source 2.1 – Premier code du semestre `print_argv.c`

Puis le compiler

```
$ gcc -Wall -Wextra print_argv.c -o print_argv
```

Puis, si tout va bien, l'exécuter

```
$ ./print_argv "Hello World"
Ce programme se nomme << ./print_argv >>
Son paramètre 1 est << Hello World >>
```

Des questions ?

Préambule 2/3

Pour le précédent code, avez-vous créé un répertoire spécifique pour le tester ?

Disons que votre répertoire de travail soit `work`, un bonne option de travail aurait été d'avoir un répertoire : `~/work/cours/ProgIAvancee/Seance01/print_argv/`


Au fait, le `~/` est la référence vers votre *home directory* (racine de votre compte utilisateur

Préambule 3/3

Qu'est censé faire le code ci-après ?
Le fait-il vraiment ?

```c
#include <stdio.h>
void swap(int a, int b) {
  int c;
  c = a; /* c sauvegarde la valeur stockée dans a */
  a = b; /* a prend la valeur stockée dans b */
  b = c; /* b prend la valeur stockée dans c, soit le a initial */
}
int main(void) {
  int a = 7, b = 42;
  /* affichage de a et b avant l'échange */
  printf("a = %d, b = %d\n", a, b);
  /* échange (?) à l'aide de swap */
  swap(a, b);
  /* affichage de a et b après l'échange */
  printf("a = %d, b = %d\n", a, b);
  return 0;
}
```

Code source 2.2 – Problème simple ? swap.c

Quelqu'un·e souhaite apporter des modifications ?

Notions de base – les variables 1/6

- Qu'est-ce qu'une variable ?
- Comment déclarer une variable ?
- Quels types de variable et quel impact sur la mémoire ?
- Comment affecter une **valeur** à une **variable** ?

Notions de base – les variables 2/6

- Qu'est-ce qu'une variable (pour nous)?
  - « **symbole** qui associe un **nom** à une **valeur** » 
    (ça n'est pas plutôt le rôle d'une constante?)
  - Une variable est un symbole, identifié par son nom, qui associe ce nom à zone mémoire. La taille et l'emplacment de cette dernière diffère en fonction du type – ou *type specifier* – et de l'emplacement où a été déclarée la variable. Certains « *specifiers* » supplémentaires, appelées *qualifiers*, peuvent modifier l'emplacement et la nature de la variable.
- Comment déclarer une variable?
- Quels types de variable et quel impact sur la mémoire?
- Comment affecter une **valeur** à une **variable**?

Variable https://fr.wikipedia.org/wiki/Variable_(informatique)
Qualifier https://en.wikipedia.org/wiki/Type_qualifier

Notions de base – les variables 3/6

- Qu'est-ce qu'une variable ?
- Comment déclarer une variable ?

```
[qualifier1 [qualifier2 [...]]] <type> [* [* [...]]] nom_de_la_variable;
```

Exemples :

```
float moyenne;
static int compteur;
char * ptr;
```

Notez qu'un nom de variable bien choisi vaut mieux que plusieurs lignes de commentaires. Notez aussi qu'il n'y a aucune garantie (en C) qu'une variable dynamique (*i.e.* non statique) soit initialisée à zéro. À l'inverse, une variable ayant un *qualifier* `static` est mise à zéro (ou `NULL` si pointeur) et mise dans le *BSS segment* (voir aussi *Data segment*) du binaire ou éxécutable généré.

- Quels types de variable et quel impact sur la mémoire ?
- Comment affecter une **valeur** à une **variable** ?

BSS Segment https://en.wikipedia.org/wiki/.bss
Data Segment https://en.wikipedia.org/wiki/Data_segment

→ Plus tard, nous ferons un usage et une analyse poussés de ces notions.

Notions de base – les variables 4/6

- Qu'est-ce qu'une variable ?
- Comment déclarer une variable ?
- Quels types de variable et quel impact sur la mémoire ?
  1. **Bien retenir** que le type d'une variable a pour principal effet de déterminer le nombre de bits que cette dernière occupe en mémoire.
  2. L'autre utilité du type est liée à **l'interprétation** qui est faite de ce type ; principalement par le compilateur mais parfois par des fonctions comment `printf`. Nous donnerons une attention particulière au fait qu'une valeur affectée à une variable corresponde à son type.

     Arrêtez de penser qu'on stocke littéralement des caractères en RAM !

     ```
     char byte1 = 'A', byte2 = 65;
     /* byte1 et byte2 contiennent strictement la même valeur c-à-d (65)10 ou (01000001)2 */
     ```
- Comment affecter une **valeur** à une **variable** ?

La table ASCII `https://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange`
est une table d'indirection (LookUpTable) utilisée par le compilateur ou autres fonctions traitant des entrées/sorties

Notions de base – les variables 5/6

- Qu'est-ce qu'une variable ?
- Comment déclarer une variable ?
- Quels types de variable et quel impact sur la mémoire ?
- Comment affecter une **valeur** à une **variable** ?
  - À l'aide de l'opérateur affectation « = », lors de la déclaration de la variable ou plus tard, **ou bien**, par passage de paramètres (*i.e.* valable pour les arguments d'une fonction). Il y a d'autres moyens tels que `memset`/`memcpy` ou autres manipulations passant par des pointeurs : à voir plus tard.
  - Dans tous les cas il faut bien distinguer ce qu'est une **l-value** et une **r-value** ; **l** et **r** faisant respectivement référence à *left* et *right*, et ceci est lié à l'emplacement du symbole par rapport à l'opérateur d'affectation « = ». Ainsi, une **l-value** est un contenant, comme par exemple une boîte, et une **r-value** fait référence à une donnée, ou une information, qui correspond *in fine* à une valeur.

Cette notion est fondamentale, assurons-nous qu'elle soit maîtrisée par tou·te·s !

↓

Notions de base – les variables 6/6

- **(SUITE 1)** Comment affecter une **valeur** à une **variable** ?

  Insistons sur la **l-value** et la **r-value** :

  l-value : est un symbole identifiant une zone mémoire permettant le stockage d'une donnée. Ce symbole peut donc fournir au compilateur une information sur :
  - l'emplacement de la zone, soit l'adresse du premier octet de cette zone ;
  - la taille et le type de la donnée que nous pouvons y mettre, cela sert à vérifier la cohérence de l'affectation, car *in fine* ce que nous y mettons est toujours un packet de 0 et de 1 → du binaire.

  La **l-value** ne peut donc être une constante (comme `7`, `42`, `0x7ffee9239a68`, `...`), une fonction ou une expression (comme `(a + 1)`, `(a * b)`, `...`). Nous excluons de cette restriction les situations où nous ferons usage de **l'opérateur de déréférencement « * »**.

  r-value : est une donnée (au sens une ou plusieurs valeurs) qui peut être extraite à partir de symboles : une variable (ex. `nb_elements`), une constante (ex. `42` ou `mon_approximation_de_pi`), un retour de fonction (ex. `pgcd(p, q)`) ou le résultat d'une quelconque expression (ex. `((a + b) / c)`).

```c
#include <stdio.h>
int carre(int x) {
  return x * x; /* r-value, le contenu de x est récupéré pour le calcul */
}
int main(void) {
  int a, b, c, * p; /* a, b et c sont des conteneurs d'int, et p est
                       un conteneur d'adresse de int */
  a = 7; /* la valeur 7 (111 en binaire) est affectée à la zone
            mémoire "labélisée a" (l-value) */
  b = carre(a); /* a est une r-value, donc la r-value renvoyée par
                   carre(7) -> 49 est affectée à la zone mémoire
                   "labélisée b" (l-value) */
  c = (b - a); /* la différence entre le contenu extrait de b et celui
                  de a est affectée à la l-value c */
  p = &a; /* p (l-value) stocke l'adresse du conteneur "labellisé a" */
  /* &p = NULL; impossible, l'adresse de p est constante, indice en
     mémoire du conteneur, il est fixe et ne peut etre une l-value */
  /* a + 2 = 11; impossiblr, (a + 2) est une expression qui vaut, pas
     de sens d'affecter une valeur à un nombre */
  *p = 9; /* p stocke l'adresse de a, *p est physiquement l'entité qui
             se trouve à l'adresse stockée dans p, donc *p est a,
             ainsi 9 est affectée à la case mémoire dont l'adresse est
             stockée dans p, c-à-d la case labellisée a */
  *(p + 1) = carre(a); /* syntaxiquement correct mais dangeureux, p
                          contient l'adresse de a => (p + 1) est l'@
                          du int voisin (à sa droite en RAM), *(p + 1)
                          est donc physiquement ce voisin de droite,
                          on écrit dedans le carré de a */
  printf("a = %d, b = %d, c = %d, p = %p (&a = %p)\n", a, b, c, p, &a);
  return 0;
}
```

Code source 2.3 – Concepts de **l&r-values** `lvaluervalue.c`, à tester

Notions de base – les fonctions (en C) 1/4

- Qu'est-ce qu'une fonction ?
- Déclarer ET définir une fonction ?
- Quelques *qualifiers* et leurs impacts ?

Notions de base – les fonctions (en C) 2/4

- Qu'est-ce qu'une fonction ?

  Dans un programme, une fonction peut être résumée à une séquence d'instructions regroupées pour réaliser une tâche (généralement actions ou calculs) particulière, dans un cadre délimité [1] ou restreint, lié à ses paramètres, s'ils existent. Une fonction peut donc être appelée (ou utilisée) plusieurs fois au sein du programme, ou, dans le cas spécifique de l'écriture d'une bibliothèque de fonctions, être réservée à un usage ultérieur par d'autres programmes utilisant la dite bibliothèque.
- Déclarer ET définir une fonction ?
- Quelques *qualifiers* et leurs impacts ?

1. Voir néanmoins le cas particulier de la fonction à effet de bord : https://fr.wikipedia.org/wiki/Effet_de_bord_(informatique)

Notions de base – les fonctions (en C) 3/4

- Qu'est-ce qu'une fonction ?
- Déclarer ET définir une fonction ?

Déclarer une fonction consiste à donner un prototype, le plus complet possible, permettant de la qualifier [2], de donner son type de retour et d'informer sur ses paramètres. En donnant, à la suite, le type de chaque paramètre, nous déduisons leur nombre. Si le premier paramètre est `void` cela indique que la fonction n'en a pas. Il est important de déclarer une fonction avant de la définir ; la déclaration se fait en haut du fichier C quand les fonctions sont statiques et dans un fichier d'en-têtes (fichier `.h`) quand les fonctions sont externes. Ci-après, une forme générale d'un prototype :

```
[qualifier1 [...]] <type> [* [...]] nom_de_la_fonction(<void> | <type1>[, <type2>[...]]);
```

Définir une fonction consiste à donner « ***dans les grandes lignes*** [3] » son prototype, directement (à la place du «;») suivi du corps de la fonction entre accolades. Si le type de retour de la fonction est autre que `void`, alors le corps de la fonction doit avoir un retour (*i.e.* `return ...;`) correspondant à ce type dans toutes les situations rencontrées (tous les branchements du code). Ci-après, une forme générale d'une définition :

```
<type> [* [...]] nom_de_la_fonction(<void> | <type1 nom1>[, <type2 nom2>[...]]) {
   /* les instructions qui constituent le corps de la fonction */
   /* ne pas oublier un return lié au type dans toute situation */
}
```

- Quelques *qualifiers* et leurs impacts ?

---

2. Définir ses particularités : sans être exhaustifs, citons `extern` contre `static`, exportable au sein d'une bibliothèque ou non, ou encore *inlinable* (cf. `inline`) ou non (les deux derniers points sont compilateur-spécifiques).

3. Si la fonction est correctement prototypée (déclarée) avant sa définition, remettre les *qualifiers* est optionnel, voire non-productif. Par contre, dans le cas de la définition de fonction, il est nécessaire de nommer les paramètres (en plus des types).

Notions de base – les fonctions (en C) 4/4

- Qu'est-ce qu'une fonction ?
- Déclarer ET définir une fonction ?
- Quelques *qualifiers* et leurs impacts ?

  Le principal couple de *qualifiers* à connaître est : `static` ou `extern`. Le premier indique que la fonction a une portée limitée au fichier (à partir de sa déclaration) alors que le second lui permet d'avoir une portée au delà du fichier (*i.e.*le fichier `.c`), évidemment si son prototype est donné (généralement dans le fichier *header* `.h` inclus via la directive de préprocesseur `#include`).

   La vidéo `https://youtu.be/Q_XliWuKsKw` peut aider à comprendre la structuration multi-fichiers d'un programme et les concepts de `static` et `extern`. Prenez le temps de la visualiser en entier.

Notions de base – Informations utiles 1/5

Type	Occupation mémoire	Plage de valeurs
char	1 octet	-128 à 127
unsigned char	1 octet	0 à 255
int	2 ou 4 octets	Selon l'architecture
unsigned int	2 ou 4 octets	Selon l'architecture
short	2 octets	-32768 à 32767
unsigned short	2 octets	0 à 65535
long	4 octets	-2147483648 à 2147483647
unsigned long	4 octets	0 à 4294967295
long long	8 octets	-2^{63} à $2^{63}-1$
unsigned long long	8 octets	0 à $2^{64}-1$
Type à virgule flottante		
float	4 octets	3.4×10^{-38} à 3.4×10^{38} (IEEE 754)
double	8 octets	1.7×10^{-308} à 1.7×10^{308} (IEEE 754)
long double	10 octets	3.4×10^{-4932} à 3.4×10^{4932} (IEEE 754)

Table – Types de données standards (C89).

En pratique, les types de données évoluent et changent en fonction de l'architecture. La norme C99 spécifie de nouveaux types dont la taille est fixe :

Nouveaux type dans `stdint.h` https://en.wikibooks.org/wiki/C_Programming/stdint.h

Notions de base – Informations utiles 2/5

Priorité	Opérateur	Description	Exemple
0	`()`	appel de fonction, associativité	`foo();` `a = (b + c) * d;`
	`[]`	indexation	`int tab[3];` `tab[0] = tab[1] = tab[2] = 0;`
	`.`	nommage d'un champ	`obj.cdr = NULL;`
	`->`	nommage indirect de champ	`pt->cdr = NULL;`
1	`!`	négation	`!a` est vraie si `a` est fausse
	`~`	complément à 1	`a & (~a)` → `0`
	`-`	opposé	
	`++`	incrémentation	`i++; ++i;`
	`--`	décrémentation	`i--; --i;`
	`&`	adresse	`int i, *pt;` `pt = &i;`
	`*`	valeur indirecte	`*pt = 0; /* donne i = 0 */`
	`(type_de_donnée)`	force le type (cast)	`int i = (int)1.5;`
	`sizeof`	taille en octets	`s = sizeof i;`
2	`*`	Multiplication	
	`/`	Division	
	`%`	Modulo	
3	`+`	Addition	
	`-`	Soustraction	
4	`<<`	Décalage à gauche	
	`>>`	Décalage à droite	

Table – Table des priorités des opérateurs C/C++ (1/2)

Notions de base – Informations utiles 3/5

5	<	Strictement inférieur	
	<=	Inférieur ou égal	
	>	Strictement supérieur	
	>=	Supérieur ou égal	
6	==	Egal	
	!=	Différent	
7	&	“et” binaire	
8	^	“ou” exclusif binaire	
9	|	“ou” binaire	
10	&&	“et” logique	
11	||	“ou” logique	
12	?:	conditionnelle	`c = (a < b) ? a : b;`
13	= *= /= %= += -= ^= &= <<= >>= |=	Affectations	
14	,	Séquence	

Table – Table des priorités des opérateurs C/C++ (2/2)

Notions de base – Informations utiles 4/5

Format	Type de donnée	
%d	Entier	`short, int`
%i	Entier	`short, int`
%u	Entier non signé	`unsigned short, unsigned int`
%zu	taille	`size_t`
%o	Entier octal	`short, int`
%x, %X	Entier hexadécimal	`short, int`
%l, %ld, %li, %lu, %lo, %lx, %lX	Entier long	`long`
%lld, %lli, %llu, %llo, %llx, %llX	Entier long 64 bits	`long long`
%c	Caractère ASCII	`char, unsigned char`
%f, %F, %g, %G	Flottant	`float, double`
%e, %E	Flottant (exponentiel)	`float, double`
%Lf, %LF, %Lg, %LG, %Le, %LE	long double	`long double`
%s	Chaîne de caractères	`char *`
%p	Pointeur	`(type) *`

Table – Formats pris en charge par la famille de fonctions `printf`

Attention ! L'abus d'utilisation de fonctions « `print` » est dangereux pour la suite de votre cursus. Il est important de faire strictement ce qui est attendu de vous, exemple : si on vous demande d'écrire la fonction `foo` qui calcule et retourne « *bidule* », il **n'est pas attendu** d'y mettre de l'affichage (*i.e.*du `print`).

Notions de base – Informations utiles 5/5

Séquence d'échappement	Action
\n	Nouvelle ligne (`new line`)
\t	Tabulation horizontale
\v	Tabulation verticale
\b	Retour d'un caractère arrière (backspace)
\r	Retour chariot (`carriage return`)
\f	Saut de page (`form feed`)
\a	Signal sonore (`alarm`)
\'	Affiche une apostrophe
\"	Affiche un guillemet
\\	Affiche un `Backslash`
\ddd	Affiche les codes ASCII en octale
\xddd	Affiche les codes ASCII en hexadécimale

Table – Séquences d'échappement dans les chaînes de caractères

Exercices de révision 1/6

Faire en sorte que la fonction swap **échange** les contenus de variables (*i.e.* conteneurs) passées en paramètre, puis modifier l'appel.

```c
#include <stdio.h>
void swap(int a, int b) {
  int c;
  c = a; /* c sauvegarde la valeur stockée dans a */
  a = b; /* a prend la valeur stockée dans b */
  b = c; /* b prend la valeur stockée dans c, soit le a initial */
}
int main(void) {
  int a = 7, b = 42;
  /* affichage de a et b avant l'échange */
  printf("a = %d, b = %d\n", a, b);
  /* échange (?) à l'aide de swap */
  swap(a, b);
  /* affichage de a et b après l'échange */
  printf("a = %d, b = %d\n", a, b);
  return 0;
}
```

Code source 2.4 – À corriger (swap.c)

Exercices de révision 2/6

Vous arrivez à quelque chose d'équivalent à ça concernant `swap` ?

```c
void swap(int * pa, int * pb) {
  int c;
  c = *pa; /* c sauvegarde la valeur stockée dans le conteneur pointée
          par pa (*pa est "ce vers quoi pointe pa") */
  *pa = *pb; /* le conteneur pointé par pa prend la valeur stockée
          dans le conteneur pointée par pb */
  *pb = c; /* le conteneur pointé par pb prend la valeur stockée dans c,
        soit le *pa initial */
}
```

Code source 2.5 – Nouvelle fonction (`swap`)

Comment appeler `swap` ? (c'est à dire par quoi remplacer `swap(a, b);` dans le `main` de `swap.c`)

Exercices de révision 3/6

Pour le code donné ci-après :

```
char x = 7, y = 42;
x = (x ^ y);
y = (x ^ y);
x = (x ^ y);
```

1. Représentez (dessinez) chaque contenu de variable en binaire sur 6 bits (*i.e.* pour chacune, avoir six cases et y mettre soit des 0 soit des 1).
2. Réalisez les opérations ci-dessus, ne pas oublier de mettre à jour les contenus de vos variables après chaque XOR.
3. En déduire quelle est la “fonction” de ces trois instructions utilisant le XOR.
4. Modifiez `swap.c` pour en avoir une dernière version capable de se passer de la variable temporaire `c`.

Exercices de révision 4/6

Pour le code donné ci-après :

```c
#include <stdio.h>
static void d2b(unsigned char n) {
  /* posez le calcul pour en comprendre le sens */
  unsigned char mask = (1 << (((sizeof n) << 3) - 1));
  for( ; mask ; mask >>= 1) {
    if(n & mask)
      putchar('1');
    else
      putchar('0');
  }
  putchar('\n');
}
int main(void) {
  d2b(214);
  return 0;
}
```

1. Calculez à la main la valeur initiale de `mask` (**Spoiler Alert**, la bonne réponse est $1 \times 10^2 + 2 \times 10^1 + 8 \times 10^0$).
2. Faire tourner l'exemple à la main (donc `d2c(214)`).
3. En déduire l'utilité de la fonction `d2c`.

Exercices de révision 5/6

Questions faciles

Écrire l'ensemble des fonctions [4] dans le même fichier C et les tester [5] dans le `main`.

1. Écrire la fonction dont le prototype est **`static int ivmax(int * t, int n);`** qui retourne **l'indice de la plus grande valeur** contenue dans le tableau `t` passé en argument.
2. Écrire la fonction dont le prototype est **`static float somme(float * t, int n);`** qui retourne **la somme** des `n` éléments du tableau `t` passé en argument.
3. Écrire la fonction dont le prototype est **`static float pp(float * t, int n);`** qui retourne **le plus petit** des `n` éléments du tableau `t` passé en argument.
4. Écrire la fonction récursive dont le prototype est **`static static long long puissance_r(int x, unsigned int n);`** qui retourne x^n.
5. Écrire la fonction itérative dont le prototype est **`long long puissance_i(int x, unsigned int n)`** qui retourne x^n.

---

4. De préférence déclarer en haut du fichier et définir plus bas, de manière à ce que la première fonction définie soit la fonction `main`.

5. N'attendez pas de tout écrire pour commencer à tester : pour chaque fonction, une fois écrite, ajouter les instructions dans le `main` qui permettent de la tester.

Exercices de révision 6/6

Allons un peu plus loin. Écrire l'ensemble des fonctions [6] dans le même fichier C et les tester [7] dans le `main`.

1. Écrire la fonction **`static inv_i(int t[], int n);`** qui inverse l'ordre des éléments dans `t` (`n` étant le nombre d'éléments de `t`).
2. Écrire la fonction **`static inv_s(char * t);`** qui inverse l'ordre des éléments dans la chaîne de caractères `t`.

   
   Pour information, nous considérons qu'une chaîne de caractères est bien formée si et seulement si le pointeur sur la chaîne n'est pas `NULL` et que la chaîne se termine par le caractère `'\0' == 0`.
3. Réécrire votre propre version de la fonction `strcpy` [8] en la déclarant **`static char * my_strcpy(char * dest, const char * src);`** elle copie la chaîne pointée par `src` dans la zone mémoire pointée par `dest`.
4. Réécrire votre propre version de la fonction `strdup` [9] en la déclarant **`static char * my_strdup(const char * s);`** elle renvoie un pointeur sur une nouvelle chaîne de caractères qui est dupliquée depuis `s`.

---

6. De préférence déclarer en haut du fichier et définir plus bas, de manière à ce que la première fonction définie soit la fonction `main`.
7. N'attendez pas de tout écrire pour commencer à tester : pour chaque fonction, une fois écrite, ajouter les instructions dans le `main` qui permettent de la tester.
8. Faire `man strcpy` dans votre terminal pour obtenir le manuel de la fonction.
9. Faire `man strdup` dans votre terminal pour obtenir le manuel de la fonction.

Devoir Maison 01 – DM01

A rendre exclusivement via moodle. Faire attention aux dates limites liées à votre groupe.

- Écrire un programme (`main` + fonction(s)) pour le calcul **rapide** [10] de la puissance : x^n avec x et n entiers.
- Écrire en `C` une implémentation du Crible d'Ératosthène. Ce programme prendra en entrée `N`, en argument de l'exécutable. `N` est considéré comme la borne supérieure des nombres premiers à trouver. Le programme doit afficher (à l'aide de `printf`) tous les nombres premiers trouvés (`N` peut en faire partie s'il est premier, il sera ainsi le dernier à être affiché). Merci de vous inspirer de l'algorithme décrit sur la page correspondante sur Wikipédia :

  « L'algorithme procède par élimination : il s'agit de supprimer d'une table des entiers de 2 à N tous les multiples d'un entier. En supprimant tous les multiples, à la fin il ne restera que les entiers qui ne sont multiples d'aucun entier, et qui sont donc les nombres premiers.
  On commence par rayer les multiples de 2, puis à chaque fois on raye les multiples du plus petit entier restant. On peut s'arrêter lorsque le carré du plus petit entier restant est supérieur au plus grand entier restant, car dans ce cas, tous les non-premiers ont déjà été rayés précédemment.
  À la fin du processus, tous les entiers qui n'ont pas été rayés sont les nombres premiers inférieurs à N. »

L'ensemble du travail doit être réalisé dans un même dossier (ex. `PA_DM01`) qui contiendra un fichier C par question et un fichier `Makefile` qui se chargera de compiler les deux programmes. Ce dossier doit être **archivé** (de préférence en `tgz`, `tar.gz` ou `tar.bz2`) et **soumis** sur le Moodle du cours (aucun binaire n'est admis dans l'archive).

La vidéo `https://youtu.be/EOFg3zr7DXY` vous aidera à écrire votre `Makefile`, prenez le temps de la visualiser en entier.

---

10. Ce n'est donc pas une des versions vues précédemment dans la première série d'exercices.

La rue de la RAM 1/17

```c
#include <stdio.h>
int main(void) {
  int i;
  char une_maison = 97;
  char * une_adresse = &une_maison;
  char des_maisons[] = {'A', 'A' + 1, 67, 'Z' - 22};
  char * un_ptr_libre = NULL;
  printf("Affichage des adresses :\n"
     "\t@une_maison = %p\n"  "\t@une_adresse = %p\n"
     "\t@des_maisons = %p\n" "\t@un_ptr_libre = %p\n",
     &une_maison, &une_adresse, &des_maisons, &un_ptr_libre);
  printf("Affichage des valeurs en hexa castées en entier :\n"
     "\tune_maison = %x\n"    "\tune_adresse = %llx\n"
     "\tdes_maisons = %llx\n" "\tun_ptr_libre = %llx\n",
     une_maison, (long long)une_adresse,
     (long long)des_maisons, (long long)un_ptr_libre);
  printf("Affichage des char comme caractères :\n"
     "\tune_maison = '%c'\n", une_maison);
  for(i = 0; i < (int)(sizeof des_maisons / sizeof * des_maisons); ++i)
    printf("\tdes_maisons[%d] = '%c'\n", i, des_maisons[i]);
  return 0;
}
```

Code source 3.1 – Commençons par un code affichant des adresses de conteneurs ainsi que leur contenu (`mem_basics.c`)

Le copier/coller est susceptible de générer des erreurs difficiles à repérer, veuillez bien indenter et lire attentivement la sortie du compilateur.

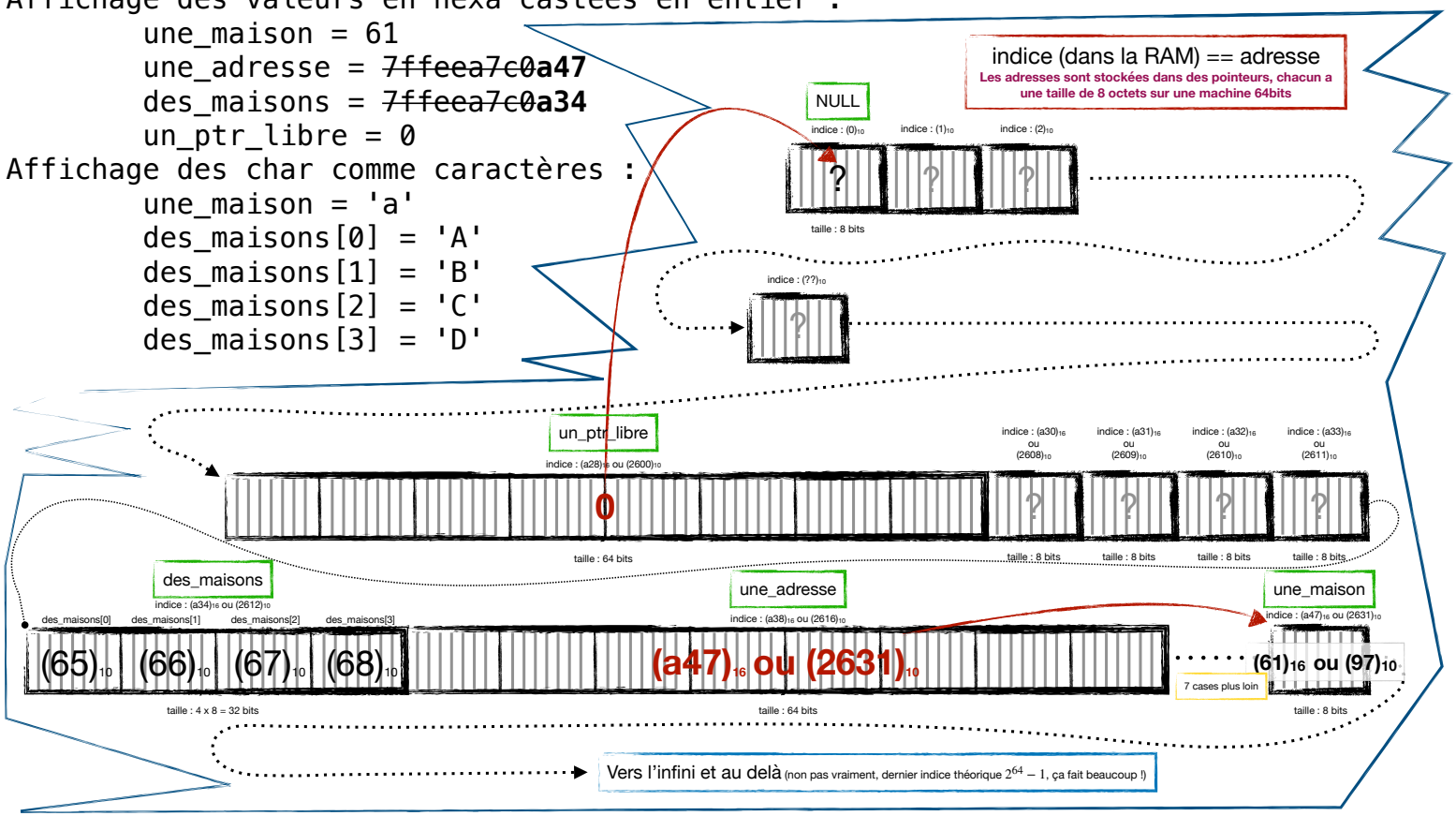
La rue de la RAM 2/17

Après compilation et exécution du code :

```
$ gcc -Wall -Wextra mem_basics.c -o mem_basics && ./mem_basics
Affichage des adresses :
        @une_maison = 0x7ffeea7c0a47
        @une_adresse = 0x7ffeea7c0a38
        @des_maisons = 0x7ffeea7c0a34
        @un_ptr_libre = 0x7ffeea7c0a28
Affichage des valeurs en hexa castées en entier :
        une_maison = 61
        une_adresse = 7ffeea7c0a47
        des_maisons = 7ffeea7c0a34
        un_ptr_libre = 0
Affichage des char comme caractères :
        une_maison = 'a'
        des_maisons[0] = 'A'
        des_maisons[1] = 'B'
        des_maisons[2] = 'C'
        des_maisons[3] = 'D'
```

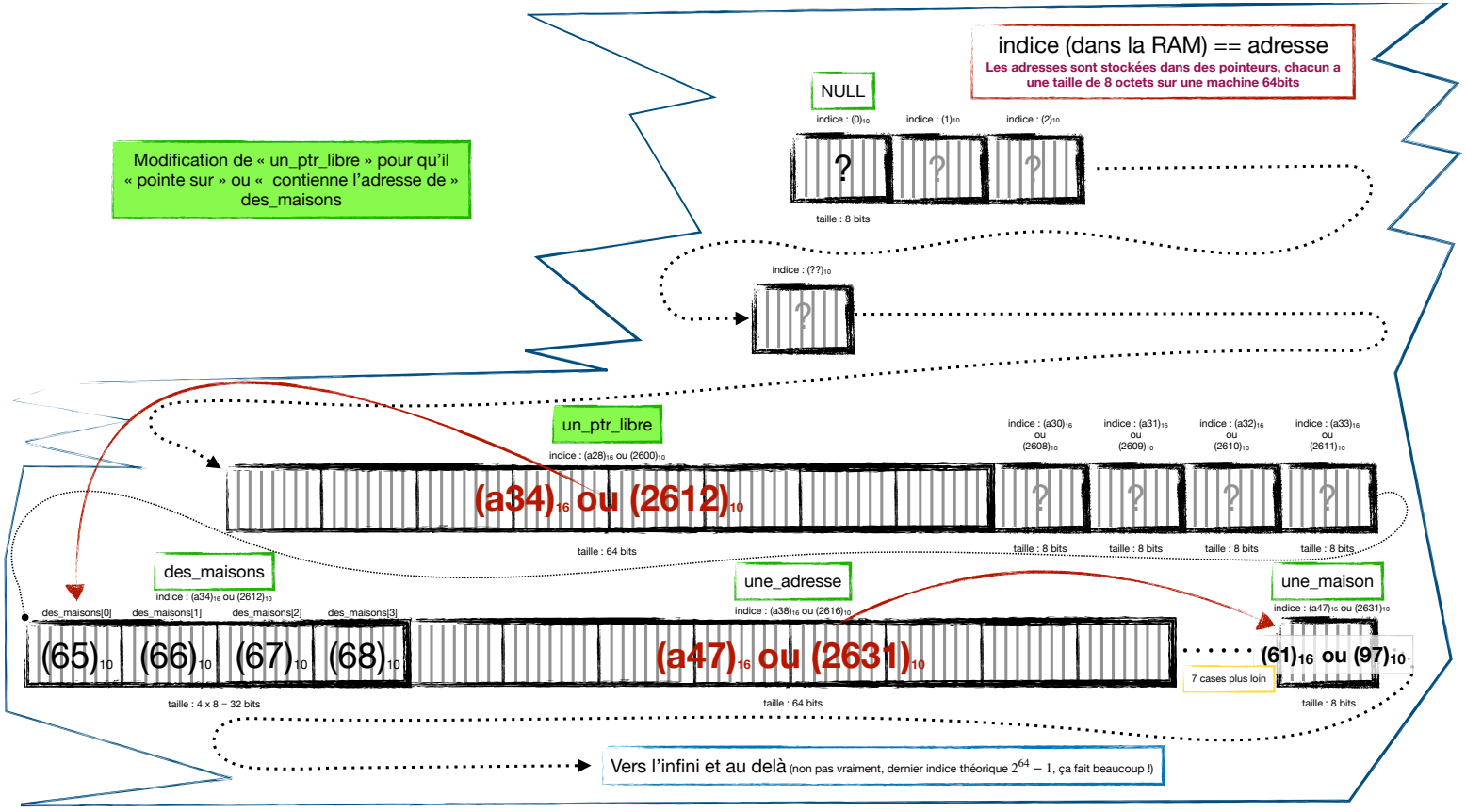
La rue de la RAM 3/17

```
$ gcc -Wall -Wextra mem_basics.c -o mem_basics && ./mem_basics
Affichage des adresses :
        @une_maison = 0x7ffeea7c0a47
        @une_adresse = 0x7ffeea7c0a38
        @des_maisons = 0x7ffeea7c0a34
        @un_ptr_libre = 0x7ffeea7c0a28
Affichage des valeurs en hexa castées en entier :
        une_maison = 61
        une_adresse = 7ffeea7c0a47
        des_maisons = 7ffeea7c0a34
        un_ptr_libre = 0
Affichage des char comme caractères :
        une_maison = 'a'
        des_maisons[0] = 'A'
        des_maisons[1] = 'B'
        des_maisons[2] = 'C'
        des_maisons[3] = 'D'
```



La rue de la RAM 4/17

Exercice :
en n'utilisant que `un_ptr_libre` et `des_maisons`, ajoutez une boucle affichant les caractères relatifs au contenu de `des_maisons`. La compilation ne doit provoquer aucun *warning*, même avec les options de compilation `-Wall -Wextra`



La rue de la RAM 5/17

Les pointeurs : comment lire la déclaration ?
`char *** ptr = NULL;`

- **ptr** est une variable de type pointeur, elle existe physiquement en RAM (c'est un conteneur), elle contient donc une valeur, ici cette valeur est `((void*)0)`
- **char *** ptr** nous indique que `ptr` est un pointeur de pointeur de pointeur vers un `char`
- **char **** ptr** nous indique que `*ptr` est **potentiellement** un pointeur de pointeur vers un pointeur de `char`. Néanmoins, attention, `*ptr` n'est pas lisible car `ptr` ne peut être déréférencé (« devenir ce vers quoi il pointe ») car il pointe sur la zone d'indice mémoire 0. Ceci est valable pour tous les cas listés ci-dessous
- **char *** ptr** nous indique que `**ptr` est **potentiellement** un pointeur vers un pointeur de pointeur de `char` (non lisible aussi dans le cas présent)
- **char *** ptr** nous indique que `***ptr` est **potentiellement** un `char` (non lisible aussi dans le cas présent)

La rue de la RAM 6/17

Dessinez cet exemple puis faites l'exercice dont l'énoncé est en commentaire dans le code.

```c
#include <stdio.h>
int main(void) {
  short a = 42, * b = &a, ** c = &b, *** d = &c;
  printf("a = %d, son adresse est %p\n", a, &a);
  printf("b = %p, son adresse est %p\n", b, &b);
  printf("c = %p, son adresse est %p\n", c, &c);
  printf("d = %p, son adresse est %p\n", d, &d);
  /* EXERCICE ! (FACILE) */
  /* en n'utilisant que printf et d, affichez
   * le contenu de chaque variable d, c, b, et a
   */
  /* printf("%p, %p, %p, %d\n", ..............);
  return 0;
}
```

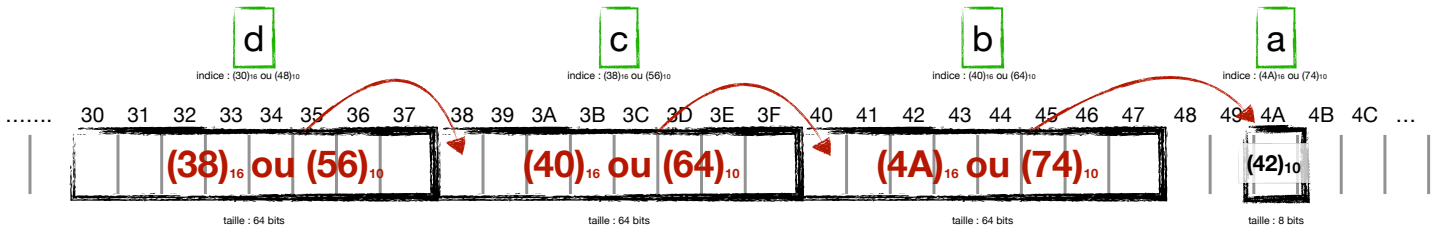
Code source 3.2 – Illustration d'utilisation de pointeur de pointeur de ... (mem_basics_02.c)

```
$ gcc -Wall -Wextra  mem_basics_02.c -o mem_basics_02 && ./mem_basics_02
a = 42, son adresse est 0x7ffee6fc8a4a
b = 0x7ffee6fc8a4a, son adresse est 0x7ffee6fc8a40
c = 0x7ffee6fc8a40, son adresse est 0x7ffee6fc8a38
d = 0x7ffee6fc8a38, son adresse est 0x7ffee6fc8a30
```

La rue de la RAM 7/17

État de la mémoire de l'exemple précédent :

```
$ gcc -Wall -Wextra  mem_basics_02.c -o mem_basics_02 && ./mem_basics_02
a = 42, son adresse est 0x7ffee6fc8a4a
b = 0x7ffee6fc8a4a, son adresse est 0x7ffee6fc8a40
c = 0x7ffee6fc8a40, son adresse est 0x7ffee6fc8a38
d = 0x7ffee6fc8a38, son adresse est 0x7ffee6fc8a30
```



La rue de la RAM 8/17

À faire :

Faire un schéma du contenu connu de mémoire liée à ce programme. Sans tester sur une machine, essayez de deviner quel affichage nous obtenons suite à son exécution. Enfin, testez sur machine pour vérifier votre proposition.

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
int main(void) {
  char str01[] = "Hello", str02[] = "world";
  char ** ptr = NULL;
  ptr = malloc(2 * sizeof *ptr);
  assert(ptr);
  ptr[0] = str01;
  ptr[1] = str02;
  printf("%c\n", *ptr[1]);
  printf("%c\n", (*ptr)[1]);
  free(ptr);
  return 0;
}
```

Code source 3.3 – Qu'affiche ce programme??? (`mem_basics_03.c`)

La rue de la RAM 9/17

Retour sur l'allocation dynamique de mémoire :

- Pour un pointeur **data** de n'importe quel type :
  `<type> * [*[*[...]]] data = NULL;`
- Demander à **malloc** [11] d'allouer une zone mémoire correspondant au nombre d'octets souhaités. Le nombre d'octets peut différer du nombre d'éléments qu'on imagine y mettre, par conséquent, pour **nb_elements** souhaités, il faudra allouer **nb_elements** $\times$ **la_taille_d_un_element**. Pour **nb_elements** donné, le *pattern* d'appel à **malloc** est toujours le même, le seul élément qui change est le nom du pointeur vers la nouvelle zone demandée :
  `data = malloc(nb_elements * sizeof *data);`
- Un test vérifiant que l'allocation est effective (résultat différent de `NULL`) est nécessaire. Pour faire simple, nous recommandons l'usage de **assert** (cf. le `man` et inclure `assert.h`) :
  `assert(data);`
- Utilisez votre mémoire allouée, sans jamais « perdre de vue » le pointeur vers le début de la zone.
- Libérez la mémoire allouée, à l'aide de **free**, quand vous n'en avez plus l'usage :
  `free(data);`
- Remettez le pointeur à `NULL` afin d'éviter de faire des « bêtises » avec :
  `data = NULL;`

11. Fonction de la bibliothèque standard **stdlib** (à inclure donc) qui alloue un nombre d'octets passés en argument et retourne un pointeur sans type (*i.e.* **void ***) vers la zone allouée. Voir **man malloc** pour lire en détail la documentation de cette fonction ainsi que celles des fonctions associées ou équivalentes telle que **calloc, realloc, free ...**

La rue de la RAM 10/17

Entraînement (1)

À faire ou à lire (solution dans le slide suivant) : créer une structure pour un point 3D [12], contenant trois champs flottants `x`, `y` et `z`. Proposer deux différentes fonctions pour remplir un tableau d'éléments de cette structure (`n`, le nombre d'éléments sera donné) avec des valeurs pseudo-aléatoires [13] comprises dans l'intervalle $[0, 1[$. L'une des fonctions de remplissage, **void initpoint3dv(point3d_t * p3darray, int n);**, remplit les **n** éléments de type **point3d_t**, l'autre, **void init3fv(float * triplefloatsarray, int n);**, remplit les $\mathbf{3} \times \mathbf{n}$ flottants du tableau. Pour deux tableaux statiques de **point3d_t**, de même taille et non initialisés, appeler pour chacun une fonction de remplissage différente. Comparer les deux tableaux à l'aide de la fonction `memcmp` (voir le *man*) et afficher le résultat de la comparaison.

---

12. Par exemple, cette structure pourra être nommée : `point3d_t`

13. Utilisez la fonction **rand()** de la `stdlib` (cf. **man 3 rand**). Nous utiliserons **srand** avec la même graine, avant chaque appel de la fonction de remplissage, afin d'avoir la même chaîne de nombres pseudo-aléatoires dans les deux cas.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct point3d point3d_t;
struct point3d {
  float x, y, z;
};
void initpoint3dv(point3d_t * p3darray, int n) {
  int i;
  for(i = 0; i < n; ++i) {
    p3darray[i].x = rand() / (RAND_MAX + 1.0f);
    p3darray[i].y = rand() / (RAND_MAX + 1.0f);
    p3darray[i].z = rand() / (RAND_MAX + 1.0f);
  }
}
void init3fv(float * triplefloatsarray, int n) {
  int i, _3n = 3 * n;
  for(i = 0; i < _3n; ++i)
    triplefloatsarray[i] = rand() / (RAND_MAX + 1.0f);
}
int main(void) {
  point3d_t A[100], B[100];
  srand(42); initpoint3dv(A, sizeof A / sizeof *A);
  srand(42); init3fv((float *)B, sizeof B / sizeof *B);
  if(memcmp(A, B, sizeof A) == 0)
    printf("même contenu\n");
  else
    printf("contenu différent\n");
  return 0;
}
```

Code source 3.4 – proposition de solution au slide précédent (`point3d.c`)

La rue de la RAM 11/17

Entraînement (2)

A faire en cours (TD à venir) : récupérer des lignes [14] de texte [15] sur l'entrée standard, les stocker dans un `char **` (le premier indice faisant référence au numéro de ligne), les réafficher en mettant les lettres en majuscule, sortir « proprement ».

14. Nous fixons une limite de 1024 caractères par ligne, le `'\0'` compris.
15. Je recommande l'utilisation de la fonction `fgets`.

La rue de la RAM 12/17

Lecture de code :

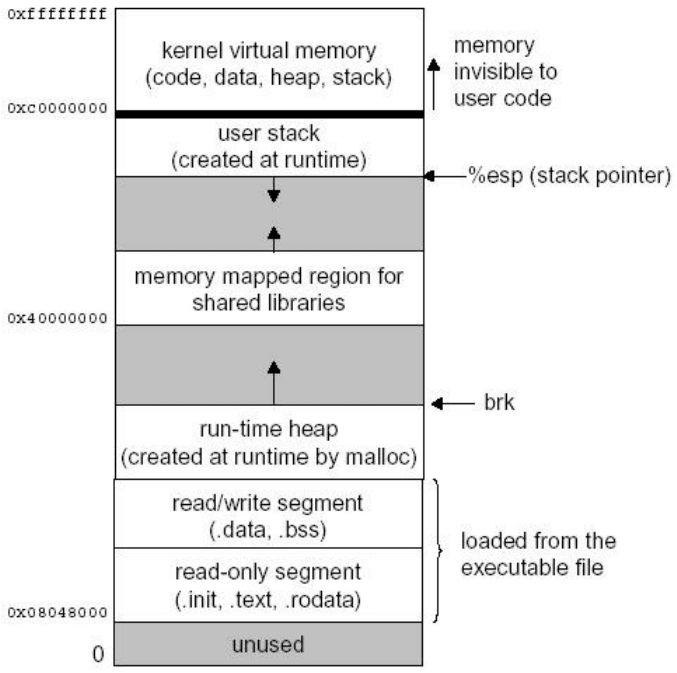
Que fait cette fonction ?

```
char * foo(char * d, const char * s) {
  char * retour = d;
  while( (*d++ = *s++) );
  return retour;
}
```

Pour vous entraîner à comprendre un code, testez-le. Ici, testez l'appel à **foo** en lui passant, comme premier argument, un pointeur vers un tableau de **char** suffisamment grand, et, comme second argument, une chaîne de caractères.
Proposez un **main** pour tester.

La rue de la RAM 13/17

Comment est utilisées la mémoire au sein d'un système d'exploitation ?



Cas d'un OS de type Unix (crédits Monjurul Karim – Source *Code based Buffer Overflow Detection Technology* – 2015)

.bss
en français

https://en.wikipedia.org/wiki/.bss
https://fr.wikipedia.org/wiki/Segment_BSS

La rue de la RAM 14/17

Expérimenter le placement en mémoire
Faire et tester sur un ou plusieurs OS, l'impression d'adresses :

- D'une variable locale *dynamique*, dans `main`;
- D'une variable locale *statique*, dans `main`;
- D'une variable locale *dynamique*, dans une fonction *externe* et *statique*;
- D'une variable locale *statique*, dans une fonction *externe* et *statique*;
- D'une allocation mémoire;
- D'une variable globale *externe*;
- D'une variable globale *statique*;
- D'une fonction *externe*;
- D'une fonction *statique*;
- D'une fonction de bibliothèque (par exemple `cos`);

Puis classer ces adresses.

La rue de la RAM 15/17

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h> /* pour cos */
static void st_print(char * what, const void * addr);
extern void foo(void);
static void bar(void);
extern int glo_var_e; /* déclaration */
int glo_var_e = 7;    /* définition  */
static int glo_var_s = 42;
int main(void) {
  int dy_var = 7;
  static int st_var = 42;
  const int const_var = 2;
  static const int st_const_var = 6;
  int * alloc = malloc(sizeof *alloc);
  st_print("variable locale dans main", &dy_var);
  st_print("variable locale statique dans main", &st_var);
  st_print("variable const dans main", &const_var);
  st_print("variable const statique dans main", &st_const_var);
  foo();
  bar();
  st_print("allocation avec malloc", alloc);
  st_print("variable globale externe", &glo_var_e);
  st_print("variable globale statique", &glo_var_s);
  st_print("fonction externe", foo);
  st_print("fonction statique", bar);
  st_print("fonction de bibliotheque (cos de la libm)", cos);
  free(alloc);
  return 0;
}
void st_print(char * what, const void * addr) {
  printf("@ d'une %50s\t: %16p (%llu)\n", what, addr, (unsigned long long)addr);
}
void foo(void) {
  int dy_var = 7;
  static int st_var = 42;
  st_print("variable locale dans foo (extern)", &dy_var);
  st_print("variable locale statique dans foo (extern)", &st_var);
}
void bar(void) {
  int dy_var = 7;
  static int st_var = 42;
  st_print("variable locale dans bar (static)", &dy_var);
  st_print("variable locale statique dans bar (static)", &st_var);
}
```

Code source 3.5 – Place en mémoire selon le type/variété (`memory_zones.c`)

La rue de la RAM 16/17

Darwin Kernel Version 20.2.0

```
$ gcc -Wall -Wextra -O0 memory_zones.c -o memory_zones -lm && ./memory_zones
@ d'une                          variable locale dans main        :   0x7ffee121da68 (140732675512936)
@ d'une                 variable locale statique dans main        :      0x10e9ea024 (4540244004)
@ d'une                           variable const dans main        :   0x7ffee121da64 (140732675512932)
@ d'une                  variable const statique dans main        :      0x10e9e5de0 (4540227040)
@ d'une                  variable locale dans foo (extern)        :   0x7ffee121da3c (140732675512892)
@ d'une         variable locale statique dans foo (extern)        :      0x10e9ea02c (4540244012)
@ d'une                  variable locale dans bar (static)        :   0x7ffee121da3c (140732675512892)
@ d'une         variable locale statique dans bar (static)        :      0x10e9ea030 (4540244016)
@ d'une                             allocation avec malloc        :   0x7faf7bc05960 (140391672207712)
@ d'une                           variable globale externe        :      0x10e9ea020 (4540244000)
@ d'une                          variable globale statique        :      0x10e9ea028 (4540244008)
@ d'une                                   fonction externe        :      0x10e9e5d20 (4540226848)
@ d'une                                  fonction statique        :      0x10e9e5d60 (4540226912)
@ d'une          fonction de bibliothèque (cos de la libm)        :   0x7fff2a468338 (140733902652216)
```

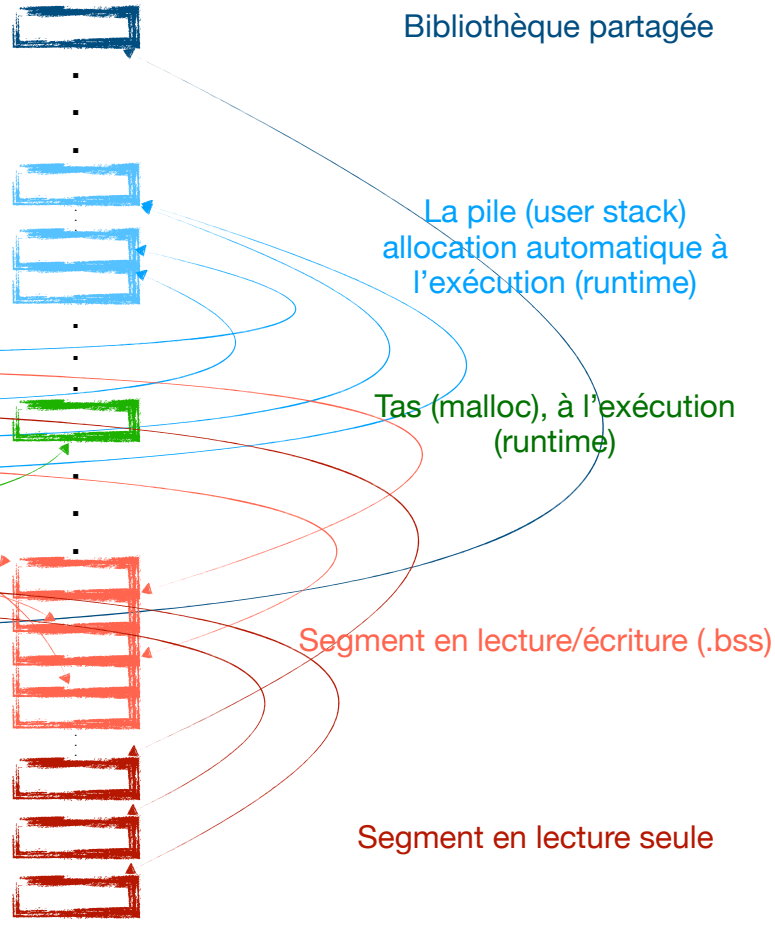
Linux 4.9

```
$ gcc -Wall -Wextra -O0 memory_zones.c -o memory_zones -lm && ./memory_zones
@ d'une                          variable locale dans main        :   0x7ffd7bbde8c8 (140726679496904)
@ d'une                 variable locale statique dans main        :   0x556104851018 (93875176017944)
@ d'une                           variable const dans main        :   0x7ffd7bbde8cc (140726679496908)
@ d'une                  variable const statique dans main        :   0x55610464fc1c (93875173915676)
@ d'une                  variable locale dans foo (extern)        :   0x7ffd7bbde8a4 (140726679496868)
@ d'une         variable locale statique dans foo (extern)        :   0x55610485101c (93875176017948)
@ d'une                  variable locale dans bar (static)        :   0x7ffd7bbde8a4 (140726679496868)
@ d'une         variable locale statique dans bar (static)        :   0x556104851020 (93875176017952)
@ d'une                             allocation avec malloc        :   0x556104852260 (93875176022624)
@ d'une                           variable globale externe        :   0x556104851010 (93875176017936)
@ d'une                          variable globale statique        :   0x556104851014 (93875176017940)
@ d'une                                   fonction externe        :   0x55610464f8f2 (93875173914866)
@ d'une                                  fonction statique        :   0x55610464f94d (93875173914957)
@ d'une          fonction de bibliotheque (cos de la libm)        :   0x7f1fcc2aaa60 (139774546061920)
```

La rue de la RAM 17/17

Visualisation des emplacements mémoire en fonction du type de donnée

```
                variable locale dans main:   0x7ffee121da68 (140732675512936)
      variable locale statique dans main:      0x10e9ea024 (4540244004)
               variable const dans main:   0x7ffee121da64 (140732675512932)
     variable const statique dans main:      0x10e9e5de0 (4540227040)
      variable locale dans foo (extern):   0x7ffee121da3c (140732675512892)
variable locale statique dans foo (extern):      0x10e9ea02c (4540244012)
       variable locale dans bar (static):   0x7ffee121da3c (140732675512892)
variable locale statique dans bar (static):      0x10e9ea030 (4540244016)
                 allocation avec malloc:   0x7faf7bc05960 (140391672207712)
              variable globale externe:      0x10e9ea020 (4540244000)
             variable globale statique:      0x10e9ea028 (4540244008)
                    fonction externe:      0x10e9e5d20 (4540226848)
                   fonction statique:      0x10e9e5d60 (4540226912)
fonction de bibliothèque (cos de la libm):   0x7fff2a468338 (140733902652216)
```



La liste (simplement) chaînée 1/5

En résumé (rappel) : C'est une structure de données représentant une liste d'éléments – généralement de même type – chaînés entre eux deux à deux. Ainsi, par exemple pour une liste de 3 éléments, le premier a pour successeur le second – on dit aussi qu'il est chaîné au second, le second est chaîné au troisième et ce dernier n'est chaîné à rien [16]. De manière imagée, une liste chaînée peut faire penser à un train composé de ses wagons, par contre, la particularité de cette structure de donnée est qu'un élément n'est pas nécessairement contigu en mémoire avec son successeur direct; la position en mémoire de ce dernier peut se trouver plus loin, voire même avant. Pour se faire, un élément d'une liste chaînée, souvent appelé nœud de la liste, comporte deux parties : la première sert à stocker la donnée (liée au type de la liste [17]); la seconde est un lien vers l'élément suivant (le successeur), ce lien peut être fait à l'aide « d'un indice ou d'un pointeur vers ce successeur ». S'il n'y a pas de successeur, nous l'indiquerons par le biais d'un indice invalide (par exemple -1) ou un pointeur nul (*i.e.* `NULL`).

Remarques :

- La liste chaînée **EST incarnée** par le conteneur indiquant « l'indice de » ou « le pointeur vers » le premier nœud de la liste;
- La conséquence du point ci-dessus est que, au sein d'un nœud, la partie qu'on appelle successeur est elle-même une liste chaînée [18], et ceci est même vrai pour « la partie succsseur » du dernier nœud.

16. La liste chaînée circulaire est un cas particulier de la simple liste chaînée où le dernier élément chaîne le premier élément de la liste.

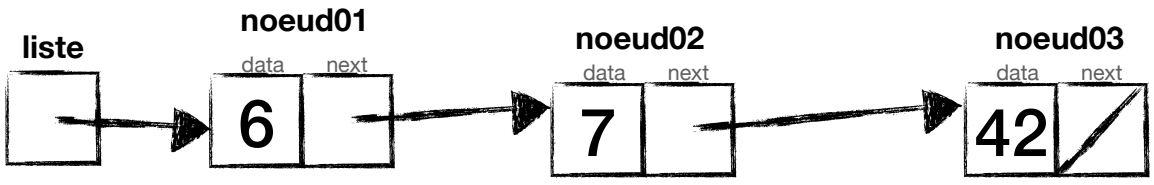
17. Nous pouvons donner comme exemple une liste chaînée d'entiers (donnée de type **int**) ou bien une liste chaînée de chaînes de caractères (donnée de type **char ***).

18. Elle est plus exactement une sous-liste chaînée.

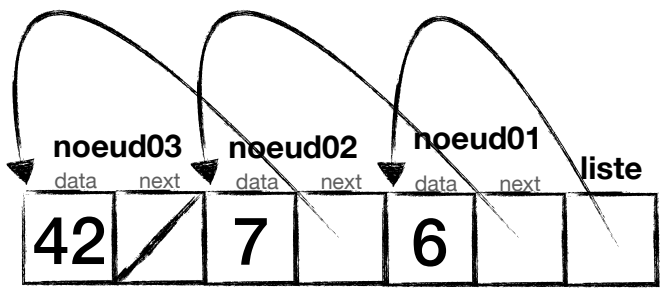
La liste chaînée 2/5

Fabriquer sa liste chaînée manuellement (pas pratique)

```c
#include <stdlib.h>
/* llnode_t pour linked list node (noeud de liste chaînée */
typedef struct llnode llnode_t;
struct llnode {
  int data; /* donnée stockée dans le noeud */
  struct llnode * next; /* pointeur vers le successeur */
};
int main(void) {
  llnode_t * liste = NULL;
  llnode_t noeud01, noeud02, noeud03;
  liste = &noeud01;
  noeud01.data =  6; noeud01.next = &noeud02;
  noeud02.data =  7; noeud02.next = &noeud03;
  noeud03.data = 42; noeud03.next = NULL;
  return 0;
}
```



En pratique, dans cet exemple (sur mon architecture) les vraies positions ressemblent plus à :



La liste chaînée 3/5

La même, manuellement, avec des `malloc` (encore moins pratique)

```c
#include <stdlib.h>
#include <assert.h>
/* llnode_t pour linked list node (noeud de liste chaînée */
typedef struct llnode llnode_t;
struct llnode {
  int data; /* donnée stockée dans le noeud */
  struct llnode * next; /* pointeur vers le successeur */
};
int main(void) {
  llnode_t * liste = NULL;
  liste = malloc(sizeof *liste); /* allocation du premier noeud */
  assert(liste);
  liste->data = 6;
  liste->next = malloc(sizeof *(liste->next)); /* allocation du deuxième noeud
                          les parenthèses ne sont pas obligatoires */
  assert(liste->next);
  liste->next->data = 7;
  liste->next->next = malloc(sizeof *(liste->next)); /* allocation du troisième
       noeud */
  assert(liste->next->next);
  liste->next->next->data = 42;
  liste->next->next->next = NULL;
  /* on libère (en sens inverse ... pourquoi donc ?) */
  free(liste->next->next);
  free(liste->next);
  free(liste);
  liste = NULL;
  return 0;
}
```

La liste chaînée 4/5

Exercice TD :

Nous allons écrire un ensemble de fonctions – de préférence sous la forme d'une bibliothèque – pour gérer des listes chaînées d'entiers :

- Reprendre la structure et le type de données utilisés dans le code de la précédente page ;
- Une fonction **llnode_t * ll_new_node(int data);** qui alloue un nouveau nœud, y stocke la valeur **data** transmise, met son successeur à `NULL` et retourne l'emplacement vers ce nœud ;
- En considérant qu'un pointeur vers un nœud est position représentant une liste (ou sous-liste) chaînée, écrire la fonction **void ll_insert(llnode_t ** pos, llnode_t * new_node);** qui insert l'unique nœud **new_node** à l'emplacement pointé par **pos** tout en chaînant le premier avec ce qui avait dans le second. Exemple : si la liste contenait (7, 42), en ajoutant le nœud contenant 6 à la position représentant le début de la liste, la liste devient (6, 7, 42).
- Écrire la fonction **main** qui utilise ces fonctions de création et d'insertion dans une liste pour créer la liste (6, 7, 42). Pour se faire commencez par insérer 42, puis 7, et enfin 6 ;
- Écrire et tester la fonction **void ll_print(llnode_t * list);** qui affiche [19] toutes les données (*i.e.* **data**) stockées en partant de **list** ;
- Écrire et tester la fonction **size_t ll_size(llnode_t * list);** qui retourne le nombre d'éléments (*i.e.* nœuds) **list**, c'est-à-dire le nombre de nœuds parcourus en partant du pointeur **list** ;
- Écrire et tester la fonction **void ll_clear(llnode_t ** plist);** qui libère tous les nœuds à partir du pointeur vers liste (ou sous-liste) **plist** (attention à mettre à jour la donnée pour qu'elle reste cohérente).

19. Cela peut être fait itérativement ou récursivement.

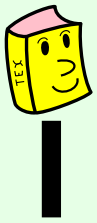
La liste chaînée 5/5

Exercice TD (suite) :

Nous allons un peu plus loin dans la bibliothèque de gestion de listes chaînées :

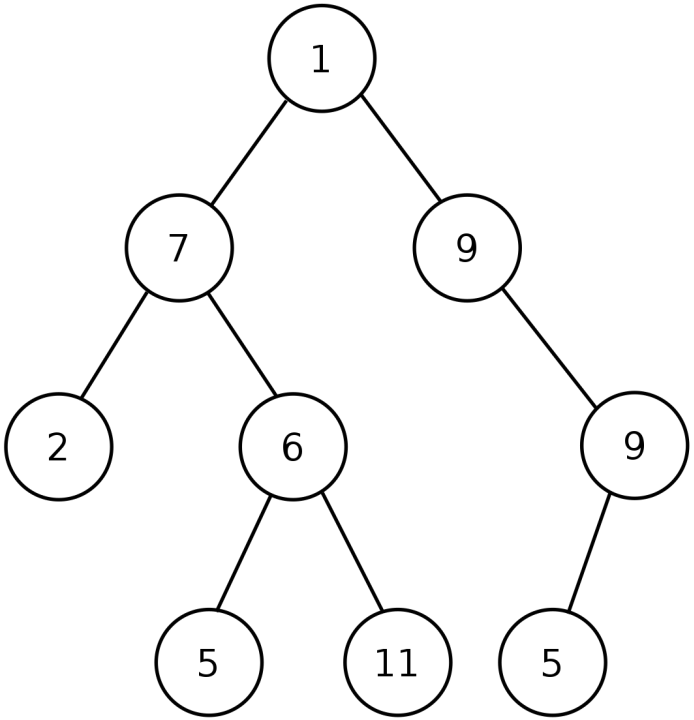
- Réfléchir à une solution pour ajouter rapidement un élément à la fin de la liste chaînée. Indice : nous pouvons modifier la fonction **ll_insert** pour qu'elle nous retourne une information utile pour ce cas là ;
- L'implémenter (le point ci-dessus) et le tester ;
- Généraliser cette solution en proposant une nouvelle structure de donnée pour représenter une **linkedlist** ;
- Proposer une solution complète qui permet d'insérer un élément dans une liste tout en garantissant que les éléments de la liste soient toujours triés dans l'ordre croissant ;
- BONUS : en utilisant les fonctions créées précédemment, proposer des fonctions **push_front, push_back, pop_front, pop_back** qui, chacune respectivement, insère au début, à la fin, retire et retour au début, à la fin.

Arbres binaires 1/9



Arbres binaires

Arbres dont chaque nœud a au plus deux fils, un fils gauche et/ou un fils droite.



Arbres binaires 2/9 – Terminologie

- **Nœud interne :** Un nœud qui possède au moins un fils.
- **Feuille :** Un nœud sans aucun fils.
- **Ascendants :** Le père d'un nœud et ses ascendants.
- **Descendants :** Les fils d'un nœud et leurs descendants (c-à-d nœuds du sous-arbre).
- **Chemin :** depuis un nœud vers un descendant.
- **Profondeur :** Nombre d'ascendants d'un nœud.
- **Hauteur :** d'un arbre vide = 0.
  D'un arbre avec une feuille = 1.
  Sinon, 1+hauteur maximale des sous-arbres.

Arbres binaires 3/9 – Questions

1. Quelle est la hauteur maximale d'un arbre binaire à n nœuds ?
2. Quel est le nombre maximal de nœuds au niveau k dans un arbre binaire ?
3. Quelle est la hauteur minimale d'un arbre binaire à n nœuds ?
4. Quel est le nombre de feuilles d'un arbre binaire complet à n nœuds ?

Arbres binaires 4/9

```c
typedef int element_t;
typedef struct bt_node bt_node_t;

struct bt_node {
  element_t data;
  struct bt_node * l_child;
  struct bt_node * r_child;
};
```

Un arbre binaire est représenté par un pointeur vers un `bt_node_t` :

- Si l'arbre est vide : on utilise la valeur NULL.
- Sinon le pointeur désigne une structure `bt_node_t` comprenant une étiquette (la donnée de la racine) et des pointeurs vers ses deux fils (`l_child` et `r_child`) qui peuvent être vides.

Arbres binaires 5/9

Si nous construisons un arbre n-aire avec $n > 2$, il serait possible de stocker un nombre plus large de fils :

- soit avec un vecteur ;
- soit avec une liste chainée.

Arbres binaires 6/9 – Fonction de construction

```c
bt_node_t * bt_new_tree(element_t elem, bt_node_t * l_child, bt_node_t * r_child) {
  bt_node_t * tree = malloc(sizeof *tree);
  assert(tree);
  tree->data    = elem;
  tree->l_child = l_child;
  tree->r_child = r_child;
  return tree;
}
/* n'est pas utilisée mais peut être utile */
bt_node_t * bt_new_node(element_t elem) {
  return bt_new_tree(elem, NULL, NULL);
}
/* Exemple d'usage */
int main(void) {
  bt_node_t * a = bt_new_tree(5, bt_new_tree(2, NULL, NULL), bt_new_tree(3, NULL,
       NULL));
  bt_node_t * b = bt_new_tree(8, NULL, a);
  return 0;
}
```

Arbres binaires 7/9 – Parcours sur les arbres


Pour effectuer une opération (ex. print, rechercher une étiquette), nous devons « visiter » tous ou une partie des nœuds.

L'ordre est important $\Rightarrow$ il existe trois parcours classiques :

- parcours préfixé : un nœud est visité avant ses descendants ;
- parcours suffixé : un nœud est visité après ses descendants ;
- parcours infixé (arbres binaires) : un nœud est visité, dans l'ordre, vers son sous-arbre gauche puis lui-même, puis vers son sous-arbre droit.

Dans les trois cas : les nœuds de n'importe quel sous-arbre sont visités consécutivement.

Arbres binaires 8/9 – Parcours Préfixe

```c
void bt_prefix_print(bt_node_t * tree) {
  if (tree == NULL)
    return;
  printf("%d␣", tree->data);
  bt_prefix_print(tree->l_child);
  bt_prefix_print(tree->r_child);
}
```

? Qu'imprimera-t-il pour l'exemple précédent ?

Arbres binaires 9/9 – Exercices

Écrire les fonctions :

1. `void bt_suffix_print(bt_node_t * a)` qui affiche les nœuds de l'arbre en ordre suffixe.
2. `void bt_infix_print(bt_node_t* a)` qui affiche les nœuds de l'arbre en ordre infixe. Quelle est la complexité des deux dernières fonctions, pour *n*, nombre de nœuds de l'arbres ?
3. `int bt_nb_leaves(bt_node_t * a)` qui compte le nombre de feuilles de l'arbre binaire.
4. `int bt_height(bt_node_t * a)` qui calcule la hauteur de l'arbre binaire.
5. `bt_node_t * bt_copy_tree(bt_node_t * a)` qui renvoie une copie de l'arbre binaire.
6. `bt_node_t * bt_free_tree(bt_node_t ** ptree)` qui libère l'ensemble de l'arbre binaire.

Arbre binaire de recherche 1/5

```
bt_node_t * bt_insert_node(bt_node_t ** pos, bt_node_t * new_node) {
  bt_node_t * previous = *pos;
  *pos = new_node;
  if(previous != NULL) {
    if(previous->data < new_node->data)
      new_node->l_child = previous;
    else
      new_node->r_child = previous;
  }
  return new_node;
}

bt_node_t ** bt_find_ordered_position(bt_node_t ** ptree, element_t elem) {
  if(*ptree == NULL)
    return ptree;
  if(elem < (*ptree)->data)
    return bt_find_ordered_position(&((*ptree)->l_child), elem);
  return bt_find_ordered_position(&((*ptree)->r_child), elem); /* else */
}
```

? Utilisez ces deux fonctions, ainsi que le fonction `bt_infix_print`, pour remplir un arbre vide de valeurs pseudo-aléatoires et l'afficher de manière à ce que les valeurs soient données dans un ordre croissant.

Arbre binaire de recherche 2/5

Sur le modèle du comparatif entre *liste chaînée* et *vecteur* fait précédemment [20] :

- Compléter le comparatif pour pouvoir réaliser une insertion ordonnée dans un vecteur [21] ;
- Ajouter l'arbre binaire sous la forme d'une bibliothèque (fichier .h et .c) et proposer, dans le fichier `test_them.c`, un test de performance équivalent à celui du vecteur, mais sans vérification de l'ordre des éléments ;
- Essayer d'ajouter une vérification de l'ordre des éléments insérés dans l'arbre binaire (*i.e.* avoir un ordre croissant selon un parcours infixé) ;
- Essayer d'améliorer l'insertion dans le vecteur en modifiant la recherche de position : modifier `find_ordered_pos_in_vec` pour qu'elle utilise une recherche dichotomique à la place d'une recherche naïve.

---

20. Archive du TP noté :`https://expreg.org/amsi/C/PA2223S1/dl/pa_linkedlist_n_vector-0.10.tgz`

21. Si vous n'avez pas réussi à compléter la partie concernant la liste chaînée, mettez-la de côté en supprimant son utilisation dans le fichier `test_them.c`.

Arbre binaire de recherche 3/5

Après l'ajout de l'arbre binaire dans le comparatif précédemment cité, nous obtenons les temps d'exécution suivants :

```
La liste est bien ordonnée, le temps passé en insertion de 10000  éléments est de :  0.109943 secondes
Le vecteur est bien ordonné, le temps passé en insertion de 10000  éléments est de :  0.032475 secondes
L'arbre binaire est bien ordonné, le temps passé en insertion de 10000  éléments est de :  0.001896 secondes

La liste est bien ordonnée, le temps passé en insertion de 30000  éléments est de :  2.070787 secondes
Le vecteur est bien ordonné, le temps passé en insertion de 30000  éléments est de :  0.286245 secondes
L'arbre binaire est bien ordonné, le temps passé en insertion de 30000  éléments est de :  0.007064 secondes

La liste est bien ordonnée, le temps passé en insertion de 100000 éléments est de : 28.448681 secondes
Le vecteur est bien ordonné, le temps passé en insertion de 100000 éléments est de :  3.212331 secondes
L'arbre binaire est bien ordonné, le temps passé en insertion de 100000 éléments est de :  0.026300 secondes
```

? À partir de ces résultats, pouvons-nous déduire une complexité par type de données utilisé ? Si oui, lesquelles ?

Arbre binaire de recherche 4/5

En reprenant l'ensemble des fonctions données concernant l'arbre binaire de recherche, et en ajoutant une fonction de calcul de hauteur de l'arbre ainsi que les fonctions mesure de temps (`initTimer` et `getElapsedTime`) présentent dans le comparatif précédent :

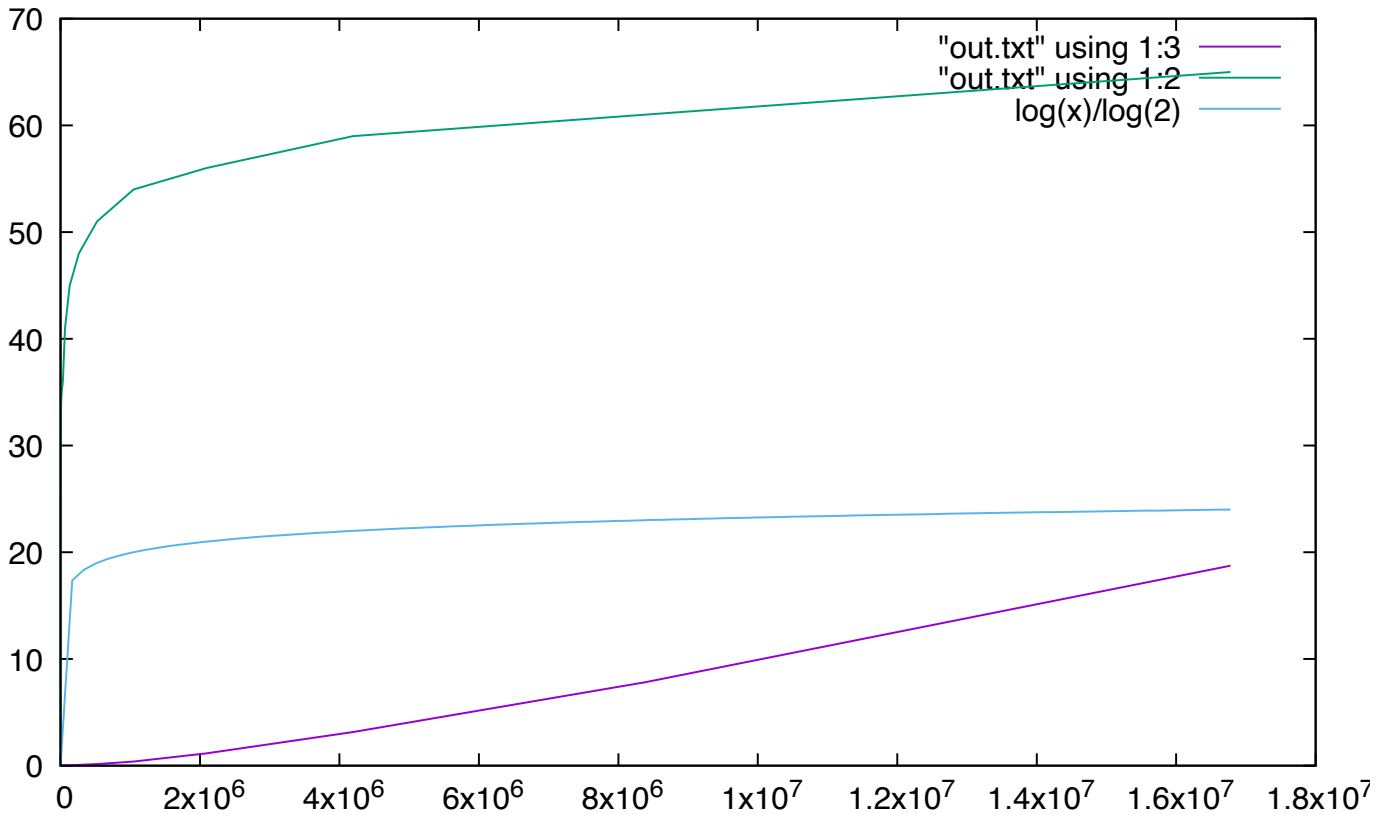
```c
static inline int bt_height(bt_node_t * tree) {
  int l, r;
  if(tree == NULL)
    return 0;
  l = 1 + bt_height(tree->l_child);
  r = 1 + bt_height(tree->r_child);
  return l > r ? l : r;
}

/* statistiques sur une insertion ordonnée de nombres
 * pseudo-aléaloires dans un arbre binaire */
/* Après compilation (gcc -Wall -Wextra -O3 bttest.c -o bttest),
   lancez le programme en redirigeant sa sortie dans un fichier:
   bttest > out.txt
   puis lancez gnuplot dans un terminal, et dedans tapez :
   plot "out.txt" using 1:3 with lines, "out.txt" using 1:2 with lines, log(x)/log(2)
*/
int main(void) {
  bt_node_t * tree = NULL, ** pos;
  int i, n = 1, nbi = 0 /* nombre d'insertions */;
  double time_acc = 0.0 /* accumulateur de temps */;
  while(n < (1 << 24)) {
    initTimer();
    for(i = 0; i < n; ++i) {
      int r = rand();
      pos = bt_find_ordered_position(&tree, r);
      bt_insert_node(pos, bt_new_node(r));
      ++nbi;
    }
    time_acc += getElapsedTime();
    printf("%8d\t%2d\t%f\n", nbi, bt_height(tree), time_acc);
    n <<= 1;
  }
  return 0;
}
```

Suivre les commentaires pour obtenir (à l'aide de gnuplot `http://www.gnuplot.info`) la figure ci-après.

Arbre binaire de recherche 5/5

Hauteur d'arbre et mesure de temps d'insertion en fonction du nombre d'éléments insérés (dans l'ordre). En bleu, la hauteur minimale théorique d'un arbre binaire en fonction du nombre d'éléments (atteignable si nous mettons en place un équilibrage d'arbre).

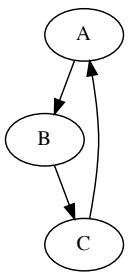


Visualisation d'arbres, graphes ... 1/3

Graphviz est un logiciel open source de visualisation de graphes : `https://graphviz.org`
Une fois cet outil installé sur votre système, pour le tester, voici un exemple simple représentant un graphe nommé G :

```
digraph G {
        A -> B
        B -> C
        C -> A
}
```

Sauvegardez ce code dans un fichier `g.dot`, puis tapez :
dot g.dot -Tpdf -o g.pdf
Une fois la ligne exécutée, ouvrez `g.pdf` pour visualiser le résultat.



Visualisation d'arbres, graphes ... 2/3

Pour vous aider dans l'exercice ci-dessous, voici une version de la fonction **bt_find_ordered_position** qui permet de savoir si l'élément dont nous cherchons la position est déjà dans l'arbre, et ne l'insère que s'il n'y est pas :

```
bt_node_t ** bt_find_ordered_position(bt_node_t ** ptree, element_t elem, int *
      pfound) {
  if(*ptree == NULL)
    return ptree;
  if(elem > (*ptree)->data)
    return bt_find_ordered_position(&((*ptree)->r_child), elem, pfound);
  if(elem == (*ptree)->data)
    *pfound = 1;
  return bt_find_ordered_position(&((*ptree)->l_child), elem, pfound);
}
```

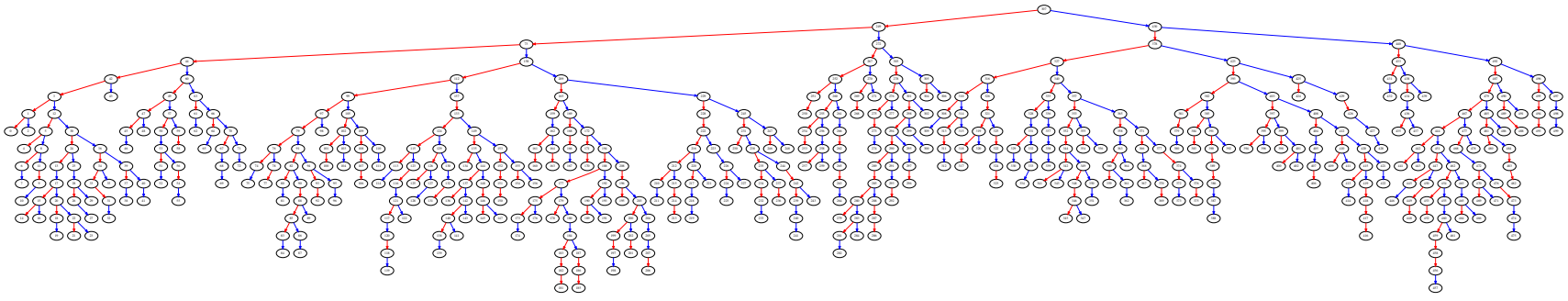
À utiliser de cette manière :

```
...
  int r = rand() % N, f = 0;
  pos = bt_find_ordered_position(&tree, r, &f);
  if(f == 0)
    bt_insert_node(pos, bt_new_node(r));
...
```

Exercice : préparez un programme d'insertion, de manière unique, d'entiers *random* dans un arbre binaire et modifiez la fonction **bt_infix_print** afin d'obtenir un *print* exploitable par *GraphViz/dot* pour produire une représentation graphique de l'arbre tel que celui de la page suivante.

Visualisation d'arbres, graphes ... 3/3

Représentation graphique d'un arbre binaire de recherche telle qu'attendu par l'exercice de la page précédente. Cet arbre contient 500 nœuds.



La généricité via **void *** 1/12

void * est le type utilisé quand on stocke une adresse vers une donnée pour laquelle “nous” [22] n’avons pas plus informations que l’adresse de l’octet de début. En pratique, le pointeur à lui seul ne permet pas de savoir avec quel type de donnée il est associé. Ainsi, un pointeur de type **void *** peut contenir une adresse vers une donnée de n’importe quel type ... le problème est qu’on en sait pas plus.

```
...
  void foo(void * ptr) {
    /* ptr est un pointeur vers ??? */
    /* si on en sait pas plus, on ne peut pas en faire grand chose */
    /* hormis peut-être tester s'il est NULL ... */
  }
...
```

La fonction **void free(void *ptr);** de la **stdlib** ne sait pas grand chose sur **ptr** mais elle connaît « sa mission » , marquer “libre” (libérer) la mémoire pointée par **ptr** et censée avoir été allouée par **malloc** [23].

---

22. Par “nous”, j’entends principalement le compilateur. Le programmeur peut, selon le contexte, en savoir plus sur ce qu’il manipule.

23. **malloc** et **free** sont “sœurs”, elles fonctionnent de paire.

La généricité via **void *** 2/12

La conséquence de ce qui a été dit précédemment est que nous ne pouvons pas *directement* déréférencer [24] un pointeur **void ***. Ceci ne fonctionne pas :

```
char chaine[] = "Mello␣Zorld␣!";
void * ptr = chaine; /* OK, no pb */
printf("%p␣=␣%p\n", ptr, chaine); /* OK, no pb */
/* remplacer le M par un H */
*ptr = 'H'; /* non OK, même juste le *ptr pose problème */
/* remplacer le Z par un W */
ptr[6] = 'W'; /* non OK, même juste le ptr[6] pose problème */
/* avancer jusqu'à la fin */
do {
  ptr++; /* non OK, on ne sait pas de combient d'octets on doit avancer */
} while(*ptr != '\0'); /* *ptr est non OK */
```

Donc, pour les exemples ci-dessus, le pointeur **void *** doit être *typé de force* (on le force à prendre un rôle, *i.e. typecast*) de sorte à pouvoir être utilisé de cette manière. Exemple, nous pouvons remplacer chaque occurrence de **ptr** par **((char *)ptr)**, ou bien le transvaser dans un pointeur non générique (ex. **char * cptr = ptr;**) et utiliser le non générique.

---

24. Il devient ce vers quoi il pointe, ou se déplace ou va chercher un ième élément à partir de là où il pointe ; tout cela est impossible sans donner plus d'informations au compilateur.

La généricité via **void *** 3/12

Nous donnons ici deux cas « caricaturaux » d'utilisation réussie de pointeurs **void *** :

```c
#include <stdio.h>
enum basic_type { /* on énumère quelques types basiques pour faire "propre" */
  TB_char = 0,
  TB_int,
  TB_double
};
static void affiche(void * ptr, enum basic_type type);
int main(void) {
  /* trois tableaux statiques de types basiques, construits sur le
   * même modèle, c-à-d se terminent tous par une dernière valeur
   * nulle. */
  char   tc[] = "Hello␣Zorld␣!";
  int    ti[] = { 1, 2, 3, 4, 6, 0 };
  double tf[] = { 1.41421, 2.71828, 3.19159, 0.0 };
  void * ptr;
  /* je corrige World */
  ptr = &tc[6]; *(char *)ptr = 'W';
  /* je corrige la suite d'entiers */
  ptr = &ti[0]; ((int *)ptr)[4] = 5;
  /* je corrige les constantes "réelles" */
  ptr = tf;     *((double *)ptr + 2) = 3.14159;
  /* affichage des valeurs des trois tableaux */
  affiche(tc, TB_char); affiche(ti, TB_int); affiche(tf, TB_double);
  return 0;
}
void affiche(void * ptr, enum basic_type type) {
  switch(type) {
  case TB_char:
    for(char * cptr = ptr; *cptr; ++cptr)
      printf("%c", *cptr);
    break;
  case TB_int:
    for(int * iptr = ptr; *iptr; ++iptr)
      printf("%d␣", *iptr);
    break;
  case TB_double:
    for(double * dptr = ptr; *dptr; ++dptr)
      printf("%f␣", *dptr);
    break;
  default:
    fprintf(stderr, "cas␣non␣géré␣par␣la␣fonction\n");
    break;
  }
  printf("\n");
}
```

Code source 6.1 – Exemple simple d'utilisation d'un pointeur **void ***

La généricité via **void *** 4/12

Quels autres usages possibles du void *?

La généricité permet aussi d'implémenter des algorithmes sur des données indépendamment du type de données. Par exemple, nous pouvons implémenter un algorithme de tri (tri par sélection, tri à bulles, tri par insertion, tri rapide, tri fusion, ...) pouvant trier n'importe quel type de données (`char`, `char *`, `int`, `float`, `...` et autres types définis par l'utilsateur).
La contrainte nécessaire ici est de savoir ordonner les éléments deux à deux, c'est-à-dire, pour un A et un B, savoir dire si $A < B$, $A > B$ ou encore si $A == B$; ceci peut être satisfait par une « fonction spécifique » à la donnée qui compare deux éléments et retourne respectivement un nombre négatif, positif ou nul pour une infériorité, supériorité ou égalité. L'autre nécessité et de connaître la taille de la donnée à trier, soit la taille en octets d'un élément ainsi que le nombre d'éléments.

Il existe par exemple une fonction de la librairie standard triant n'importe quelle donnée en utilisant le tri rapide ou *Quick Sort*.

```
void
qsort(  void *base, size_t nel, size_t width,
        int (*compar)(const void *, const void *)  );
```

où **base** est la donnée (tableau) à trier, **nel** est le nombre d'éléments, **width** est la place en octets qu'occupe un élément et **compar** est le pointeur vers une fonction capable [25] de comparer deux éléments de la donnée **base**.

Faire **man qsort** pour avoir le manuel complet de la fonction.

---

25. Elle en est capable car elle connaît le « vrai » type caché derrière le **void ***.

La généricité via **void *** 5/12

Utilisons la fonction qsort – exercice

Complétez l'exemple ci-après pour trier la donnée **data** puis vérifier que le tri a été effectué.

```
#include <stdio.h> /* pour printf */
#include <stdlib.h> /* pour srand/rand puis plus tard qsort */
#include <time.h> /* pour time */
/* fonction qui teste si base est dans l'ordre croissant */
static int is_in_ascending_order(double * base, int nel) {
  int i;
  for(i = 0; i < nel - 1; ++i)
    if(base[i] > base[i + 1])
      break;
  return (i < nel - 1) ? 0 : 1;
}
int main(void) {
  int i;
  double data[1024];
  /* une graine aléatoire différente à chaque exécution */
  /* attendre au minimum 1 seconde entre deux exécutions */
  srand(time(NULL));
  for(i = 0; i < (int)(sizeof data / sizeof *data); ++i)
    data[i] = rand() / (RAND_MAX + 1.0); /* aléa dans [0 ; 1[ */
  if( is_in_ascending_order(data, sizeof data / sizeof *data) )
    printf("La␣donnée␣est␣triée\n");
  else
    printf("La␣donnée␣n'est␣pas␣triée\n");
  return 0;
}
```

La généricité via **void *** 6/12

Utilisons la fonction qsort – solution

D'abord déclarer puis définir la fonction qui compare deux **void ***, sachant que ce sont des doubles :

```
static int compare_2_doubles(const void * ptr1, const void * ptr2) {
  /* la version courte et suffisante */
  /* return *(double *)ptr1 < *(double *)ptr2 ? -1 : 1; */

  /* version longue, étape par étape */
  const double * pa = ptr1;
  const double * pb = ptr2;
  double a = *pa;
  double b = *pb;
  if(a < b)
    return -1;
  else if(a > b)
    return 1;
  else
    return 0;
}
```

Puis ajouter l'appel à la fonction **qsort**, dans **main**, juste avant de tester (ou tester à nouveau) si le tableau est trié dans l'ordre croissant :

```
  qsort(data, sizeof data / sizeof *data, sizeof *data, compare_2_doubles);
  if( is_in_ascending_order(data, sizeof data / sizeof *data) )
    printf("La donnée est triée\n");
  else
    printf("La donnée n'est pas triée\n");
  return 0;
```

La généricité via **void *** 7/12

Exercice :
(la solution ne sera pas donnée)

Modifiez la fonction **is_in_ascending_order** pour la rendre générique puis **utilisez-la** avec `compare_2_doubles` pour qu'elle continue à donner la bonne réponse en fonction de la situation.

La généricité via **void *** 8/12

Écrire son propre algorithme générique :

Prenons un algorithme de tri, par exemple sur entiers, et transformons la fonction pour qu'elle devienne générique. Voici un code source d'un tri par sélection qui ne fonctionne que sur des entiers :

```
static void selection_sort_i(int * base, int nel) {
  int i, j, min;
  for(i = 0; i < nel - 1; ++i) {
    min = i;
    for(j = i + 1; j < nel; ++j)
      if(base[j] < base[min])
        min = j;
    if(i != min) {
      int tmp = base[i];
      base[i] = base[min];
      base[min] = tmp;
    }
  }
}
```

Modifiez cette fonction pour qu'elle devienne générique et testez-la. Commencez par lui donner un prototype équivalent à **qsort** de la bibliothèque standard puis corrigez le code en fonction des changements opérés sur les paramètres. Indice : **char *** (car avec on peut avancer d'octet en octet) et **memcpy** sont vos amis.

La généricité via **void *** 9/12

Écrire son propre algorithme générique : La Solution

En vidéo : https://expreg.org/amsi/C/Distanciel/rushs/PA_20221110_Genericite_1.mp4

```
static void selection_sort(void * base, size_t nel, size_t width,
                           int (*compar)(const void *, const void *)) {
  int i, j, min;
  char * p = base;
  char * tmp = malloc(width * sizeof *tmp);
  assert(tmp);
  for(i = 0; i < (int)nel - 1; ++i) {
    min = i;
    for(j = i + 1; j < (int)nel; ++j)
      if(compar(p + j * width, p + min * width) < 0)
        min = j;
    if(i != min) {
      memcpy(tmp, p + i * width, width);
      memcpy(p + i * width, p + min * width, width);
      memcpy(p + min * width, tmp, width);
    }
  }
  free(tmp);
}
```

Quatre choses à retenir : besoin d'un pointeur d'octets pour vous déplacer exactement de la taille souhaitée ; utilisation de la fonction de comparaison au lieu de comparer directement avec l'opérateur arithmétique ; possible besoin d'une mémoire tempon pour transvaser ; utiliser **memcpy** pour remplacer les affectations.

La généricité via **void *** 10/12

Exercice :
(la solution ne sera pas donnée)

Rendre la fonction **isort**, ci-après, générique. **isort** implémente un tri par insertion sur un tableau d'entiers.

```
static void isort(int * t, int n) {
  int i, j, v;
  for(i = 1; i < n; i++) {
    v = t[(j = i)];
    while(j > 0 && t[j - 1] > v) {
      t[j] = t[j - 1];
      j--;
    }
    t[j] = v;
  }
}
```

La généricité via **void *** 11/12

Exercice :

(la solution ne sera pas donnée)

Rendre la fonction **merge_sort**, ci-après, générique. **merge_sort** implémente un tri fusion sur un tableau d'entiers, attention, ici `_merge[N]` est statique, faire en sorte qu'elle soit allouée dynamiquement.

```c
/* Farès Belhadj, 10/11/2022 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
/* pas très dynamique :/ */
#define N 100000
static int _merge[N];

static void sub_merge_sort(int * t, int g, int d) {
  int i, j, k, m;
  if(d > g) {
    m = (d + g) >> 1;
    sub_merge_sort(t, g, m);
    sub_merge_sort(t, m + 1, d);
    for(i = m; i >= g; --i)
      _merge[i] = t[i];
    for(j = m; j < d; ++j)
      _merge[d + m - j] = t[j + 1];
    for(k = g, i = g, j = d; k <= d; ++k)
      t[k] = (_merge[i] < _merge[j]) ? _merge[i++] : _merge[j--];
  }
}
void merge_sort(int * t, int n) {
  sub_merge_sort(t, 0, n - 1);
}
int main(void) {
  int t[N], i;
  srand(time(NULL));
  for(i = 0; i < N; ++i)
    t[i] = rand();
  merge_sort(t, N);
  return 0;
}
```

Code source 6.2 – Un tri par fusion à rendre générique

La généricité via **void *** 12/12

Problème :

(la solution sera donnée dans l'enregistrement de la séance du 11/11/2022)
`https://expreg.org/amsi/C/Distanciel/rushs/PA_20221110_Genericite_2.mp4`

- rendre générique la bibliothèque (incomplète) de gestion de vecteurs : prendre comme point de départ le code source téléchargeable à l'adresse `https://expreg.org/amsi/C/PA2223S1/dl/pa_vector-0.11.tgz`
  Nous proposons, dans cette version de généricité, de stocker simplement des pointeurs vers de la donnée à allouer, au lieu de stocker des éléments de taille fixe, cette taille fixe serait évidemment non définie à l'avance, elle serait fixée à la création du vecteur (autre option de généricité).
- modifier le test pour stocker, dans l'ordre croissant, tous les mots présents dans ce text : `https://expreg.org/amsi/C/PA2223S1/dl/une_ligne.h`.

Un début de généricité via préprocesseur 1/5

Il s'agit de refléchir à une possibilité de généricité passant par le biais des **macros** (*i.e.* la partie *preprocessing* de la compilation).

```c
#define MIN(a, b) ((a) < (b) ? (a) : (b))
int main(void) {
  int i = 1, j = 2, k;
  float x = 3.14159f, y = 2.71828f, z = 1.41421f, l, m, n;
  k = MIN(i, j);
  l = MIN(x, y);
  m = MIN(l, z);
  n = MIN(MIN(x, y), z);
  return 0;
}
```

Code source 7.1 – La macro **MIN** renvoie l'expression donnant le plus petit entre deux valeurs, sans aucune contrainte liée au typage (fichier **min01.c**).

En pré-compilant ce programme à l'aide de **gcc -E min01.c** nous obtenons le résultat du *preprocessing*[26] du code :

```c
int main(void) {
  int i = 1, j = 2, k;
  float x = 3.14159f, y = 2.71828f, z = 1.41421f, l, m, n;
  k = ((i) < (j) ? (i) : (j));
  l = ((x) < (y) ? (x) : (y));
  m = ((l) < (z) ? (l) : (z));
  n = ((((x) < (y) ? (x) : (y))) < (z) ? (((x) < (y) ? (x) : (y))) : (z));
  return 0;
}
```

Le calcul de **n** illustre une partie des défauts de cette méthode.

26. Il s'agit de l'exécution des macros; nous éviterons les directives **#include** car elles provoquent l'inclusion des fichiers liés et leurs dépendances.

Un début de généricité via préprocesseur 2/5

Allons un peu plus loin, avec le précédent problème de minimum, en utilisant la directive de concaténation **##** pour créer puis utiliser des fonctions locales (*i.e.* **static**) spécialisée par type.

```c
/* macro permettant de fabriquer une fonction min_XXX liée au type "type" */
#define MK_MIN(type) static type min_## type (type a, type b) { \
  return a < b ? a : b;\
}
/* macro permettant d'appeler la fonction min_XXX préalablement fabriquée */
#define MIN(type, a, b) min_## type(a, b)

MK_MIN(int);
MK_MIN(float);

int main(void) {
  int i = 1, j = 2, k;
  float x = 3.14159f, y = 2.71828f, z = 1.41421f, l, m, n;
  k = MIN(int, i, j);
  l = MIN(float, x, y);
  m = MIN(float, l, z);
  n = MIN(float, MIN(float, x, y), z);
  return 0;
}
```

Code source 7.2 – Une macro pour générer une fonction **min** par type et une autre pour l'utiliser (fichier **min02.c**).

Un début de généricité via préprocesseur 3/5

En pré-compilant ce programme à l'aide de **gcc -E min02.c** nous obtenons le résultat du *preprocessing* du code :

```
static int min_int (int a, int b) { return a < b ? a : b;};
static float min_float (float a, float b) { return a < b ? a : b;};

int main(void) {
  int i = 1, j = 2, k;
  float x = 3.14159f, y = 2.71828f, z = 1.41421f, l, m, n;
  k = min_int(i, j);
  l = min_float(x, y);
  m = min_float(l, z);
  n = min_float(min_float(x, y), z);
  return 0;
}
```

L'avantage de cette approche est que nous évitons le problème de répétition des instructions lié à la réécriture, l'inconvénient est de devoir prévoir l'usage d'un cas à l'avance et appeler une macro qui nous permet de gérer ce cas.

Un début de généricité via préprocesseur 4/5

Allons encore plus loin en proposant une fonction de tri par sélection adaptable au type.

```c
/* Farès Belhadj, 17/11/2022
 *
 * tentative de d'exemple de tri par sélection "générique" par
 * macro
 *
 * compilez avec l'option -g et utilisez le debugger (gdb ou lldb ou
 * ..) pour tester si les tris ont bien été effectués.
 */

/* macro permettant de fabriquer une fonction ssort_XXX liée au type "type" */
#define MK_SSORT(type) static void ssort_## type (type * t, int n) {    \
  int i, j, min;\
  for(i = 0; i < n - 1; i++) {\
    min = i;\
    for(j = i + 1; j < n; j++)\
      if(t[j] < t[min])\
    min = j;\
    if(i != min) {\
      type tmp = t[i];\
      t[i] = t[min];\
      t[min] = tmp;\
    }\
  }\
}
/* macro permettant d'appeler la fonction ssort_XXX préalablement fabriquée */
#define SSORT(type, t, n) ssort_## type(t, n)

MK_SSORT(int);
MK_SSORT(double);

int main(void) {
  int ti[] = { 7, 15, 4, 7, 90, 1, 5, 11, 66, 17 };
  double td[] = { 0.7, 0.15, 0.4, 0.7, 0.90, 0.1, 0.5, 0.11, 0.66, 0.17 };
  SSORT(int, ti, 10);
  SSORT(double, td, 10);
  /* mettre un breakpoint au niveau de l'instruction suivante pour
   * regarder si les tableaux sont bien triés */
  return 0;
}
```

Code source 7.3 – Macros pour créer et utiliser un tri par sélection adapté au type (fichier **ssort.c**).

Un début de généricité via préprocesseur 5/5

En pré-compilant ce programme à l'aide de **gcc -E ssort.c** nous obtenons le résultat du *preprocessing* du code :

```
static void ssort_int (int * t, int n) { int i, j, min; for(i = 0; i < n - 1; i++) {
      min = i; for(j = i + 1; j < n; j++) if(t[j] < t[min]) min = j; if(i != min) {
      int tmp = t[i]; t[i] = t[min]; t[min] = tmp; } }};
static void ssort_double (double * t, int n) { int i, j, min; for(i = 0; i < n - 1;
      i++) { min = i; for(j = i + 1; j < n; j++) if(t[j] < t[min]) min = j; if(i !=
      min) { double tmp = t[i]; t[i] = t[min]; t[min] = tmp; } }};

int main(void) {
  int ti[] = { 7, 15, 4, 7, 90, 1, 5, 11, 66, 17 };
  double td[] = { 0.7, 0.15, 0.4, 0.7, 0.90, 0.1, 0.5, 0.11, 0.66, 0.17 };
  ssort_int(ti, 10);
  ssort_double(td, 10);
  return 0;
}
```

L'inconvénient ici est que l'ordre est donné par l'opérateur $<$ et que nous n'avons, en C, pas de moyen de l'adapter (le surchargé) à n'importe quel type de donnée.

C++ / vu la semaine dernière 1/15

Un peu d'histoire (C++)

Le langage C++ a deux grands ancêtres :

- Simula, dont la première version a été conçue en 1967. C'est le premier langage qui introduit les principaux concepts de la programmation objet.
- Le langage C a été conçu en 1972 aux laboratoires Bell Labs.

Le concepteur de C++, Bjarne Stroustrup, qui travaillait également aux Bell Labs, désirait ajouter au langage C les classes de Simula. Après plusieurs versions préliminaires, le langage a trouvé une première forme stable en 1983, et a très rapidement connu un vif succès dans le monde industriel. Mais ce n'est que plus tard que le langage a trouvé sa forme définitive, confirmée par une norme.
C++ peut être considéré comme un successeur de C qui permet la programmation objet.

g++ -o <name-you-want-to-give> source.cpp

C++ / vu la semaine dernière 2/15

Using namespace std

Un **namespace** est une zone de déclaration d'identificateurs permettant au compilateur de résoudre les conflits de noms.

Si, pat exemple, deux développeurs définissent des fonctions avec le même nom, il y aura un conflit lors de l'utilisation de ces fonctions dans un programme. Les `namespace` permettent de résoudre ce problème en ajoutant un niveau supplémentaire aux identificateurs.

C++ / vu la semaine dernière 3/15

Exemple

```
#include<iostream>
using namespace std;
namespace  a{
  int x = 5;
  void f(void){
    cout<<"This␣is␣f()␣of␣a"<< endl;
  }
}
namespace b{
  int x = 10;
  void f(void){
    cout<<"This␣is␣f()␣of␣b"<< endl;
  }
}
int main(void){
  //call variable using scope resolution (::)
  cout<<a::x<<endl;
  //function call using scope resolution (::)
  a::f();
  cout<<b::x<<endl;
  b::f();
  return 0;
}
```

`std` : Les identificateurs de la bibliothèque standard C++ sont définis dans std

Si nous n'écrivons pas `using namespace std`, alors nous devrions écrire `std::cout`,

C++ / vu la semaine dernière 4/15

Memory allocation

Les fonctions `malloc` et `free`, en C, ainsi que les opérateurs du langage C++ **`new`** et **`delete`** permettent, respectivement d'allouer et désallouer la méloire sur le tas.

- Une expression comprenant l'opération `new` retourne un pointeur sur l'objet alloué.

```
int *p = new int;
int *tab = new int[20];
```

- C'est à nous de libérer la mémoire dynamique dont nous n'avons pas plus besoin.

```
delete p;
delete [] tab;
```

C++ / vu la semaine dernière 5/15

Exemple

```
#include<iostream> //header for standard input/output stream objects
#include<new> //header for functions used to manage dynamic storage

using namespace std;

int main(void){
  int *p = new int[5];
  for (int i = 0; i < 5; i++ ){
    *(p+i) = 0;
    cout << "p[" << i << "]=" << p[i] << endl;
  }
  delete []p;
  return 0;
}
```

C++ / vu la semaine dernière 6/15

void *

- Ils peuvent pointer sur une variable de n'importe quel type.
- On ne peut pas utiliser l'opératuer d'indirection * sur un pointeur void. Il faut d'abord le convertir en un pointeur d'un type donné.

```
int a = 1;
int b = 2;
int *p1 = &a;
void *p2 = &b;

p2 = p1;
p1 = p2; // ERROR
p1 = (int *)p2; //correct
cout << *p2; // ERROR
cout << *((int *)p2); //correct
```

C++ / vu la semaine dernière 7/15

Classes/Objects

- C++ est un langage de programmation orienté objet.
- Les attributes et les méthodes (attributes and methods) sont essentiellement des variables et des fonctions qui appartiennent à la classe. Elles sont souvent appelés "membres de la classe".

Création : Class

```
class Myclass{
  public:
    int my_Num;
    string my_String;
};
```

Création : Objet

Un objet est crée à partir d'une Classe.
- Pour créer un objet de "Myclass", indiquer le nom de la classe, suivi du nom de l'objet.
- Pour accéder aux attributes de la classe (my_Num et my_String) utilisez la syntaxe (.).

```
int main(void){
  Myclass my_obj; //create Object
  my_obj.my_Num = 30;
  my_obj.my_String = "Revekka";
  cout << my_obj.my_Num << "\n";
  cout << my_obj.my_String;
  return 0;}
```

C++ / vu la semaine dernière 8/15

Méthodes

Les **méthodes** sont des fonctions qui appartiennent à la classe. Il y a deux façons de definir les fonctions qui appartiennent à une classe :

- définition interne de la classe,
- définition à l'exterieur de la classe.

C++ / vu la semaine dernière 9/15

Méthodes

Les **méthodes** sont des fonctions qui appartiennent à la classe. Il y a deux façons de definir les fonctions qui appartiennent à une classe :

- définition interne de la classe,
- définition à l'exterieur de la classe.

Exemple : interne

```
class Myclass{
  public :
    void myMethod(void){
      cout << "Hello word";
    }
};
```

Exemple : externe

```
class Myclass{
  public :
    void myMethod(); // method/function declaration
};
void Myclass::myMethod(void){
  cout << "Hello word";
}
int main(void){
  Myclass myObj;
  myObj.myMethod();
  return 0;
}
```

C++ / vu la semaine dernière 10/15

Constructors

Un **constructeur** (constructor) en C++ est une méthode spéciale qui est automatiquement appelée lorqu'un objet d'une classe est crée.
Pour crée un constructeur, nous utilisons le même nom que la classe suivi de parenthèses ().
Exemple

```
class Student{
  public :
    string first_name;
    string last_name;
    int student_number;
    Student(string x, string y, int z){
      first_name = x;
      last_name = y;
      student_number = z;
    }
};

int main(void){
  Student S1("john", "Smith", 1456);
  Student S2("maria", "vergara", 1457);
  cout << S1.first_name << "␣" << S1.last_name << "␣n." << S1.student_number <<
       endl;
  cout << S2.first_name << "␣" << S2.last_name << "␣n." << S2.student_number <<
       endl;
  return 0;
}
```

C++ / vu la semaine dernière 11/15

Access Specifiers

- **public :** Les membres sont accessibles depuis l'extérieur de la classe.
- **private :** Les membres ne sont pas accessibles (ou visibles) de l'extérieur de la classe.
- **protected :** Les membres ne sont pas accessibles de l'extérieur de la classe, mais ils peuvent être accessibles dans les classes héritées.

Par defaut tous les membres d'une classe sont **private**!!!

C++ / vu la semaine dernière 12/15

Access Specifiers

- **public :** Les membres sont accessibles depuis l'extérieur de la classe.
- **private :** Les membres ne sont pas accessibles (ou visibles) de l'extérieur de la classe.
- **protected :** Les membres ne sont pas accessibles de l'extérieur de la classe, mais ils peuvent être accessibles dans les classes héritées.

Par defaut tous les membres d'une classe sont **private**!!!

Exemple : private and public

```
class Employee{
  public :
    int number;
  private :
    int salary;
};
int main(void){
  Employee e;
  e.number = 1345;
  e.salary = 50; // ERROR : salary is private
  return 0;
}
```

C++ / vu la semaine dernière 13/15

Encapsulation

Idée : d'assurer que les données "sensibles" sont cachées aux utilisateurs.

Comment : Declarer les variables/attributes de la classe comme **private**. Si on souhaite que d'autre personnes puissent lire ou modifier la valeur d'un membre 'private' de la classe, nous pouvons utiliser **public get** et **set**.

C++ / vu la semaine dernière 14/15

Encapsulation

Idée : d'assurer que les données "sensibles" sont cachées aux utilisateurs.

Comment : Declarer les variables/attributes de la classe comme **private**. Si on souhaite que d'autre personnes puissent lire ou modifier la valeur d'un membre 'private' de la classe, nous pouvons utiliser **public get** et **set**. **Exemple : private and public**

```
#include <iostream>

class Employee{
  private :
    int salary;
  public :
    void set_salary(int s){
      salary = s;
    }
    int get_salary(void){
      return salary;
    }
};
int main(void){
  Employee e;
  e.set_salary(70000);
  cout << e.get_salary();
  return 0;
}
```

C++ / vu la semaine dernière 15/15

Inheritance

Il est possible d'hériter des attributes et des méthodes d'une classe à une autre.

- **Derived class :** La classe qui hérite d'une autre classe.
- **Base class :** La classe dont in hérite.

```
class <derived_class_name> : <access-specifier> <base_class_name> {//body}
```


Pour hériter d'une classe, nous utilisons le symbole **:**.
Une classe peut hériter de plusiers classes separées par comma (,).
Exemple : private and public

```
class Name: public Employee{
  public:
    string name = "smith";
};

int main(void){
  Name nm;
  nm.set_salary(70000);
  cout << nm.get_salary();
  return 0;
}
```

C++ / vous n'aimez pas les pointeurs ?

Passez aux références (c'est une façon cachée d'utiliser les pointeurs)

```
#include <iostream>
void swap(int &a, int &b) { //attend pointeurs mais n'oblige pas à l'écrire
  a = a ^ b;
  b = a ^ b;
  a = a ^ b;
}
void testAddressAndValue(int &uneVariable) {//elle se comporte comme l'originale
  //le std:: est obligatoire si pas de "using namespace std;"
  std::cout << "l'adresse " << &uneVariable << " contient " << uneVariable <<
       std::endl;
}
int main(void) {
  int x = 7, y = 42;
  //vérifions les adresses ici
  std::cout << "x est à l'adresse " << &x << std::endl;
  std::cout << "y est à l'adresse " << &y << std::endl;
  //maintenant dans les fonctions avec référence
  testAddressAndValue(x); // même pas besoin de spécifier l' << adresse >> avec &
  testAddressAndValue(y);
  swap(x, y);
  testAddressAndValue(x);
  testAddressAndValue(y);
  return 0;
}
```

Code source 8.1 – Utilisation des références en lieu et place des pointeurs (fichier **ref.cpp**).

```
$ g++ -Wall -Wextra ref.cpp -o ref && ./ref
x est à l'adresse 0x7ffeea4c19d8
y est à l'adresse 0x7ffeea4c19d4
l'adresse 0x7ffeea4c19d8 contient 7
l'adresse 0x7ffeea4c19d4 contient 42
l'adresse 0x7ffeea4c19d8 contient 42
l'adresse 0x7ffeea4c19d4 contient 7
```

Encore un peu d'entraînement en C++ 0/7

Les exemples donnés ci-après, illustrent progressivement certains aspects de la programmation orientée objet en C++.

Il est recommandé de les réécrire un par un (sans les commentaires), puis de les tester l'un après l'autre en suivant l'ordre poo01.cpp, poo02.cpp, ...

Pour chaque code (ils sont indépendants les uns des autres), afin de compiler (générer) et exécuter il faut :

```
*************************************************
* Exemple pour poo01.cpp, pour les autres exemples *
* remplacer poo01 par poo02, poo03, ....          *
*************************************************

(faire)
g++ -Wall -Wextra poo01.cpp -o poo01
(si pas d'erreurs faire)
./poo01
```

Encore un peu d'entraînement en C++ 1/7

Bien lire les commentaires avant de recopier

```
//pour le pointeur NULL
#include <cstddef>

//on déclare une classe A, vide
//(le nom A n'est pas terrible, on fera mieux plus tard)
class A {
};

int main(void) {
  //ici a est une instance de la classe d'objets A
  A a;
  //ici pa est un pointeur vers une instance de la classe A
  //par contre, pa ne pointe sur rien au début (NULL)
  A *pa = nullptr;
  //on peut faire pointer pa vers a (pa reçoit adresse de a)
  pa = &a;
  //sinon on instancie dynamiquement une nouvelle instance de A
  //et on fait en sorte que pa pointe dessus
  pa = new A();
  //avant de finir le programme, il faut s'assurer qu'on libère
  //toute instance créée dynamiquepent (celle pointée par pa)
  //les autres (comme a) sont libérées automatiquement.
  //libérer = dire que la mémoire utilisée par l'instance dynamique
  //est libre
  delete pa;
  return 0;
}
```

Code source 8.2 – C++ en *crescendo* (fichier **poo01.cpp**).

Encore un peu d'entraînement en C++ 2/7

Bien lire les commentaires avant de recopier

```cpp
//pour le pointeur NULL
#include <cstddef>
//pour std::cout
#include <iostream>

//on déclare une classe A, vide
//(le nom A n'est pas terrible, on fera mieux plus tard)
class A {
  //ce mot clé, déclare qu'à partir de là, tout ce qui est
  //ajouté est public, c'est à dire qu'on peut y accéder
  //en dehors de la classe elle-même. Par défaut, tout ce qu'on
  //créé est privé (private) et ne peut donc être appelé en dehors
  //de la classe.
public:
  //on appelle ça un constructeur de l'instance
  //il est appelé au moment où on créé l'instance
  //quand il n'y en a pas, un constructure par défaut
  //vide (se charge au minimum d'allouer la mémoire) est appelé
  A(void) {
    std::cout << "Coucou, je suis une instance de la classe A\n";
  }
  //on appelle ça un destructeur,
  //il est appelé au moment où on delete l'instance
  //quand il n'y en a pas, un destructure par défaut
  //vide (se charge au minimum de libérer la mémoire) est appelé
  ~A(void) {
    std::cout << "Adieu :(\n";
  }
};

int main(void) {
  //ici a est une instance de la classe d'objets A
  A a;
  //ici pa est un pointeur vers une instance de la classe A
  //par contre, pa ne pointe sur rien au début (NULL)
  A *pa = nullptr;
  //on peut faire pointer pa vers a (pa reçoit adresse de a)
  pa = &a;
  //sinon on instancie dynamiquement une nouvelle instance de A
  //et on fait en sorte que pa pointe dessus
  pa = new A();
  //avant de finir le programme, il faut s'assurer qu'on libère
  //toute instance créée dynamiquepent (celle pointée par pa)
  //les autres (comme a) sont libérées automatiquement.
  //libérer = dire que la mémoire utilisée par l'instance dynamique
  //est libre
  delete pa;
  return 0;
}
```

Code source 8.3 – C++ en *crescendo* (fichier **poo02.cpp**).

Encore un peu d'entraînement en C++ 3/7

Bien lire les commentaires avant de recopier

```cpp
//pour le pointeur NULL
#include <cstddef>
//pour std::cout
#include <iostream>

//on déclare une classe A, vide
//(le nom A n'est pas terrible, on fera mieux plus tard)
class A {
  //ce mot clé, déclare qu'à partir de là, tout ce qui est
  //ajouté est public, c'est à dire qu'on peut y accéder
  //en dehors de la classe elle-même. Par défaut, tout ce qu'on
  //créé est privé (private) et ne peut donc être appelé en dehors
  //de la classe.
public:
  //ceci est une propriété, ici elle est appelée propriété
  //d'instance, ça veut dire que chaque objet (instance) de la classe
  //ici présente possède sa propre propriété "monNumero".
  //remarquez qu'elle est déclarée après le mot clé "public"
  //donc elle est publique et on peut y accéder depuis "dehors"
  int monNumero;
  //on appelle ça un constructeur de l'instance
  //il est appelé au moment où on créé l'instance
  //quand il n'y en a pas, un constructure par défaut
  //vide (se charge au minimum d'allouer la mémoire) est appelé
  //
  //remarquez qu'on a ajouté un paramètre obligatoire pour permettre
  //la création d'une instance de cette classe
  //
  //on peut aussi laisser le constructeur sans paramètre,
  //comme ça on aurait deux façons justes de créer des
  //instances de cette classe: LE FAIRE ET LE TESTER
  //
  //si le paramètre (ici numero) avait le même nom
  //que la propriété, comment faire pour les différencier ?
  //MONTRER "this"
  A(int monNumero) {
    //je modifie mon propre monNumero
    this->monNumero = monNumero;
    std::cout << "Coucou,_je_suis_l'instance_n#" << monNumero <<"_de_la_classe_A\n";
  }
  //on appelle ça un destructeur,
  //il est appelé au moment où on delete l'instance
  //quand il n'y en a pas, un destructure par défaut
  //vide (se charge au minimum de libérer la mémoire) est appelé
  ~A(void) {
    std::cout << "l'instance_n#" << monNumero << "_dit:_Adieu_:'(\n";
  }
};

int main(void) {
  //ici a est une instance de la classe d'objets A
  //on a ajouté un paramètre à la déclaration car
  //le constructeur nous y oblige
  A a(1);
  //ici pa est un pointeur vers une instance de la classe A
  //par contre, pa ne pointe sur rien au début (NULL)
  A *pa = nullptr;
  //on peut faire pointer pa vers a (pa reçoit adresse de a)
  pa = &a;
  //comme la propriété "monNumero" est publique, je peux
  //la modifier (la corrompre) depuis là où je veux, sans
  //aucun contrôle de la par de la classe ou de l'objet
  a.monNumero = 42; //depuis l'objet lui-même avec l'opérateur "."
  pa->monNumero = 666; //depuis le pointeur vers l'objet avec l'opérateur ->
                       //une propriété publique "c'est le mal !!!" :D

  //sinon on instancie dynamiquement une nouvelle instance de A
  //et on fait en sorte que pa pointe dessus.
  //ici aussi un paramètre entier est devenu obligatoire
  pa = new A(2);
  //avant de finir le programme, il faut s'assurer qu'on libère
  //toute instance créée dynamiquepent (celle pointée par pa)
  //les autres (comme a) sont libérées automatiquement.
  //libérer = dire que la mémoire utilisée par l'instance dynamique
  //est libre
  delete pa;
  return 0;
}
```

Code source 8.4 – C++ en *crescendo* (fichier **poo03.cpp**).

Encore un peu d'entraînement en C++ 4/7

Bien lire les commentaires avant de recopier

```cpp
//pour le pointeur NULL
#include <cstddef>
//pour std::cout
#include <iostream>

//on déclare une classe A, vide
//(le nom A n'est pas terrible, on fera mieux plus tard)
class A {
  //après ce mot clé et avant un autre qui changerait
  //la portée de des éléments (propriétés ou méthodes)
  //ces derniers seront considérés privés (pour plus
  //tard, voir aussi protected)
private:
  //ceci est une propriété, ici elle est appelée propriété
  //d'instance, ça veut dire que chaque objet (instance) de la classe
  //ici présente possède sa propre propriété "monNumero".
  //remarquez qu'elle est déclarée après le mot clé "private"
  //donc elle n'est pas accessible en dehors de l'instance elle-même
  int monNumero;
  //ce mot clé, déclare qu'à partir de là, tout ce qui est
  //ajouté est public, c'est à dire qu'on peut y accéder
  //en dehors de la classe elle-même. Par défaut, tout ce qu'on
  //créé est privé (private) et ne peut donc être appelé en dehors
  //de la classe.
public:
  //on appelle ça un constructeur de l'instance
  //il est appelé au moment où on créé l'instance
  //quand il n'y en a pas, un constructure par défaut
  //vide (se charge au minimum d'allouer la mémoire) est appelé
  //
  //remarquez qu'on a ajouté un paramètre obligatoire pour permettre
  //la création d'une instance de cette classe
  A(int monNumero) {
    //je modifie mon propre monNumero
    this->monNumero = monNumero;
    std::cout << "Coucou,_je_suis_l'instance_n#" << monNumero <<"_de_la_classe_A\n";
  }
  //on appelle ça un destructeur,
  //il est appelé au moment où on delete l'instance
  //quand il n'y en a pas, un destructure par défaut
  //vide (se charge au minimum de libérer la mémoire) est appelé
  ~A(void) {
    std::cout << "l'instance_n#" << monNumero << "_dit:_Adieu_:'(\n";
  }
};

int main(void) {
  //ici a est une instance de la classe d'objets A
  //on a ajouté un paramètre à la déclaration car
  //le constructeur nous y oblige
  A a(1);
  //ici pa est un pointeur vers une instance de la classe A
  //par contre, pa ne pointe sur rien au début (NULL)
  A *pa = nullptr;
  //on peut faire pointer pa vers a (pa reçoit adresse de a)
  pa = &a;
  /*** ce qui suit n'est plus vrai car on a changé la portée
       de la propriété monNumero, elle est maintenant privée

       //comme la propriété "monNumero" est publique, je peux
       //la modifier (la corrompre) depuis là où je veux, sans
       //aucun contrôle de la par de la classe ou de l'objet
       a.monNumero = 42; //depuis l'objet lui-même avec l'opérateur "."
       pa->monNumero = 666; //depuis le pointeur vers l'objet avec l'opérateur ->
       //une propriété publique "c'est le mal !!!" :D
       ****/
  //sinon on instancie dynamiquement une nouvelle instance de A
  //et on fait en sorte que pa pointe dessus.
  //ici aussi un paramètre entier est devenu obligatoire
  pa = new A(2);
  //avant de finir le programme, il faut s'assurer qu'on libère
  //toute instance créée dynamiquepent (celle pointée par pa)
  //les autres (comme a) sont libérées automatiquement.
  //libérer = dire que la mémoire utilisée par l'instance dynamique
  //est libre
  delete pa;
  return 0;
}
```

Code source 8.5 – C++ en *crescendo* (fichier **poo04.cpp**).

Encore un peu d'entraînement en C++ 5/7

Bien lire les commentaires avant de recopier

```cpp
//pour le pointeur NULL
#include <cstddef>
//pour std::cout
#include <iostream>

//on déclare une classe A, vide
//(le nom A n'est pas terrible, on fera mieux plus tard)
class A {
  //après ce mot clé et avant un autre qui changerait
  //la portée de des éléments (propriétés ou méthodes)
  //ces derniers seront considérés privés (pour plus
  //tard, voir aussi protected)
private:
  //ceci est une propriété, ici elle est appelée propriété
  //d'instance, ça veut dire que chaque objet (instance) de la classe
  //ici présente possède sa propre propriété "monNumero".
  //remarquez qu'elle est déclarée après le mot clé "private"
  //donc elle n'est pas accessible en dehors de l'instance elle-même
  int monNumero;
  //ceci est une propriété de classe, elle n'est pas liée à une instance
  //en particulier, mais à toute la classe (structure) en général
  //elle est privée
  static int compteur;
  //ce mot clé, déclare qu'à partir de là, tout ce qui est
  //ajouté est public, c'est à dire qu'on peut y accéder
  //en dehors de la classe elle-même. Par défaut, tout ce qu'on
  //créé est privé (private) et ne peut donc être appelé en dehors
  //de la classe.
public:
  //on appelle ça un constructeur de l'instance
  //il est appelé au moment où on créé l'instance
  //quand il n'y en a pas, un constructure par défaut
  //vide (se charge au minimum d'allouer la mémoire) est appelé
  //
  //remarquez qu'on a ajouté un paramètre obligatoire pour permettre
  //la création d'une instance de cette classe
  A(int monNumero) {
    //je modifie mon propre monNumero
    this->monNumero = monNumero;
    std::cout << "Coucou,_je_suis_l'instance_n#" << monNumero <<"_de_la_classe_A\n";
    compteur++;
  }
  //on appelle ça un destructeur,
  //il est appelé au moment où on delete l'instance
  //quand il n'y en a pas, un destructure par défaut
  //vide (se charge au minimum de libérer la mémoire) est appelé
  ~A(void) {
    std::cout << "l'instance_n#" << monNumero << "_dit:_Adieu_:'(\n";
    compteur--;
  }
  //ceci est une méthode de classe, elle n'est pas liée à une instance
  //en particulier, mais plutôt à toute la classe (la structure)
  //elle est publique, donc accessible depuis dehors
  static void combienSontElles(void) {
    std::cout << "Elles_sont_" << compteur << "_instances\n";
  }
};
//on doit créer et initialiser les propriétés de classes, juste à
//l'extérieur de la classe.
int A::compteur = 0;

int main(void) {
  //combien sont elles ?
  A::combienSontElles();
  //ici a est une instance de la classe d'objets A
  //on a ajouté un paramètre à la déclaration car
  //le constructeur nous y oblige
  A a(1);
  //combien sont elles ?
  A::combienSontElles();
  //ici pa est un pointeur vers une instance de la classe A
  //par contre, pa ne pointe sur rien au début (NULL)
  A *pa = nullptr;
  //on peut faire pointer pa vers a (pa reçoit adresse de a)
  pa = &a;
  //combien sont elles ?
  A::combienSontElles();
  /*** ce qui suit n'est plus vrai car on a changé la portée
       de la propriété monNumero, elle est maintenant privée

      //comme la propriété "monNumero" est publique, je peux
      //la modifier (la corrompre) depuis là où je veux, sans
      //aucun contrôle de la par de la classe ou de l'objet
      a.monNumero = 42; //depuis l'objet lui-même avec l'opérateur "."
      pa->monNumero = 666; //depuis le pointeur vers l'objet avec l'opérateur ->
      //une propriété publique "c'est le mal !!!" :D
      ****/
  //sinon on instancie dynamiquement une nouvelle instance de A
  //et on fait en sorte que pa pointe dessus.
  //ici aussi un paramètre entier est devenu obligatoire
  pa = new A(2);
  //combien sont elles ?
  A::combienSontElles();
  //avant de finir le programme, il faut s'assurer qu'on libère
  //toute instance créée dynamiquement (celle pointée par pa)
  //les autres (comme a) sont libérées automatiquement.
  //libérer = dire que la mémoire utilisée par l'instance dynamique
  //est libre
  delete pa;
  //combien sont elles ?
  A::combienSontElles();
  return 0;
}
```

Code source 8.6 – C++ en *crescendo* (fichier **poo05.cpp**).

Encore un peu d'entraînement en C++ 6/7

Bien lire les commentaires avant de recopier

```cpp
//pour le pointeur NULL
#include <cstddef>
//pour std::cout
#include <iostream>

//on déclare une classe A, vide
//(le nom A n'est pas terrible, on fera mieux plus tard)
class A {
  //après ce mot clé et avant un autre qui changerait
  //la portée de des éléments (propriétés ou méthodes)
  //ces derniers seront considérés privés (pour plus
  //tard, voir aussi protected)
private:
  //ceci est une propriété, ici elle est appelée propriété
  //d'instance, ça veut dire que chaque objet (instance) de la classe
  //ici présente possède sa propre propriété "monNumero".
  //remarquez qu'elle est déclarée après le mot clé "private"
  //donc elle n'est pas accessible en dehors de l'instance elle-même
  int monNumero;
  //ceci est une propriété de classe, elle n'est pas liée à une instance
  //en particulier, mais à toute la classe (structure) en général
  //elle est privée
  static int compteur;
  //ce mot clé, déclare qu'à partir de là, tout ce qui est
  //ajouté est public, c'est à dire qu'on peut y accéder
  //en dehors de la classe elle-même. Par défaut, tout ce qu'on
  //créé est privé (private) et ne peut donc être appelé en dehors
  //de la classe.
public:
  //on appelle ça un constructeur de l'instance
  //il est appelé au moment où on créé l'instance
  //quand il n'y en a pas, un constructure par défaut
  //vide (se charge au minimum d'allouer la mémoire) est appelé
  //
  //remarquez que le numéro de l'instance est automatique
  A(void) {
    this->monNumero = ++compteur;
    std::cout << "Coucou,_je_suis_l'instance_n#" << monNumero <<"_de_la_classe_A\n";
  }
  //on appelle ça un destructeur,
  //il est appelé au moment où on delete l'instance
  //quand il n'y en a pas, un destructure par défaut
  //vide (se charge au minimum de libérer la mémoire) est appelé
  ~A(void) {
    std::cout << "l'instance_n#" << monNumero << "_dit:_Adieu_:'(\n";
    --compteur;
  }
  //ceci est une méthode de classe, elle n'est pas liée à une instance
  //en particulier, mais plutôt à toute la classe (la structure)
  //elle est publique, donc accessible depuis dehors
  static void combienSontElles(void) {
    std::cout << "Elles_sont_" << compteur << "_instances\n";
  }
};
//on doit créer et initialiser les propriétés de classes, juste à
//l'extérieur de la classe.
int A::compteur = 0;

int main(void) {
  //combien sont elles ?
  A::combienSontElles();
  //ici a est une instance de la classe d'objets A
  A a;
  //combien sont elles ?
  A::combienSontElles();
  //ici pa est un pointeur vers une instance de la classe A
  //par contre, pa ne pointe sur rien au début (NULL)
  A *pa = nullptr;
  //on peut faire pointer pa vers a (pa reçoit adresse de a)
  pa = &a;
  //combien sont elles ?
  A::combienSontElles();
  /*** ce qui suit n'est plus vrai car on a changé la portée
       de la propriété monNumero, elle est maintenant privée

       //comme la propriété "monNumero" est publique, je peux
       //la modifier (la corrompre) depuis là où je veux, sans
       //aucun contrôle de la par de la classe ou de l'objet
       a.monNumero = 42; //depuis l'objet lui-même avec l'opérateur "."
       pa->monNumero = 666; //depuis le pointeur vers l'objet avec l'opérateur ->
       //une propriété publique "c'est le mal !!!" :D
       ****/
  //sinon on instancie dynamiquement une nouvelle instance de A
  //et on fait en sorte que pa pointe dessus.
  pa = new A();
  //combien sont elles ?
  A::combienSontElles();
  //avant de finir le programme, il faut s'assurer qu'on libère
  //toute instance créée dynamiquepent (celle pointée par pa)
  //les autres (comme a) sont libérées automatiquement.
  //libérer = dire que la mémoire utilisée par l'instance dynamique
  //est libre
  delete pa;
  //combien sont elles ?
  A::combienSontElles();
  return 0;
}
```

Code source 8.7 – C++ en *crescendo* (fichier **poo06.cpp**).

Encore un peu d'entraînement en C++ 7/7

Bien lire les commentaires avant de recopier

```
//pour le pointeur NULL
#include <cstddef>
//pour std::cout
#include <iostream>

//on déclare une classe A, vide
//(le nom A n'est pas terrible, on fera mieux plus tard)
class A {
  //après ce mot clé et avant un autre qui changerait
  //la portée de des éléments (propriétés ou méthodes)
  //ces derniers seront considérés privés (pour plus
  //tard, voir aussi protected)
private:
  //ceci est une propriété, ici elle est appelée propriété
  //d'instance, ça veut dire que chaque objet (instance) de la classe
  //ici présente possède sa propre propriété "monNumero".
  //remarquez qu'elle est déclarée aprés le mot clé "private"
  //donc elle n'est pas accessible en dehors de l'instance elle-même
  int monNumero;
  //ceci est une propriété de classe, elle n'est pas liée à une instance
  //en particulier, mais à toute la classe (structure) en général
  //elle est privée
  static int compteur;
  //ce mot clé, déclare qu'à partir de là, tout ce qui est
  //ajouté est public, c'est à dire qu'on peut y accéder
  //en dehors de la classe elle-même. Par défaut, tout ce qu'on
  //créé est privé (private) et ne peut donc être appelé en dehors
  //de la classe.
public:
  //on appelle ça un constructeur de l'instance
  //il est appelé au moment où on créé l'instance
  //quand il n'y en a pas, un constructure par défaut
  //vide (se charge au minimum d'allouer la mémoire) est appelé
  //
  //remarquez que le numéro de l'instance est automatique
  A(void) {
    this->monNumero = ++compteur;
    std::cout << "Coucou, je suis l'instance n#" << monNumero <<" de la classe A\n";
  }
  //on appelle ça un destructeur,
  //il est appelé au moment où on delete l'instance
  //quand il n'y en a pas, un destructure par défaut
  //vide (se charge au minimum de libérer la mémoire) est appelé
  ~A(void) {
    std::cout << "l'instance n#" << monNumero << " dit: Adieu :'(\n";
    --compteur;
  }
  //ceci est une méthode de classe, elle n'est pas liée à une instance
  //en particulier, mais plutôt à toute la classe (la structure)
  //elle est publique, donc accessible depuis dehors
  static void combienSontElles(void) {
    std::cout << "Elles sont " << compteur << " instances\n";
  }
};
//on doit créer et initialiser les propriétés de classes, juste à
//l'extérieur de la classe.
int A::compteur = 0;

int main(void) {
  //combien sont elles ?
  A::combienSontElles();
  //ici a est un tableau d'instances de la classe d'objets A
  A a[50];
  //combien sont elles ?
  A::combienSontElles();
  //ici pa est un pointeur vers une instance de la classe A
  //par contre, pa ne pointe sur rien au début (NULL)
  A *pa = nullptr;
  //on peut faire pointer pa vers a (pa reçoit adresse de a)
  pa = a;
  //combien sont elles ?
  A::combienSontElles();
  //sinon on instancie dynamiquement d'autres nouvelles instances de A
  //et on fait en sorte que pa pointe dessus.
  //
  //TODO par exemple avec vector !???
  return 0;
}
```

Code source 8.8 – C++ en *crescendo* (fichier **poo07.cpp**).

La surcharge de fonctions en C++ 1/2

En C++, il est possible de donner le même nom de fonction à plusieurs « versions » de « cette fonction » à partir du moment où il y a une variation dans : le type et/ou le nombre de paramètres (le type de retour s'il est différent ne constitue pas une différence suffisante pour que la surcharge soit prise en compte).

```
int   min(int   a, int   b) {
  return a < b ? a : b;
}
float min(float f, float g) {
  return f < g ? f : g;
}
float min(float f, float g, float h) {
  return f < g ? (f < h ? f : h) : (g < h ? g : h);
}
int main(void) {
  int a = 2, b = 3, c;
  c = min(a, b);     //donnera 2
  float d = 1.1f, e = 2.2f, f;
  f = min(d, e);     //donnera 1.1f
  float g = 3.3f, h = 4.4f, i = 0.1f, k;
  k = min(g, h, i); //donnera 0.1f
  return 0;
}
```

Code source 8.9 – Plusieurs fonctions portant le nom **min** (fichier **surcharge_min.cpp**).

La surcharge de fonctions en C++ 2/2

Nous pouvons comprendre comment le compilateur C++ fait pour éviter la confusion entre les différentes fonctions, une fois le code compilé. Pour cela nous pouvons analyser les symboles générés dans le binaire. Gardez en tête que le C++ est un langage fortement typé par rapport au C, et la raison est principalement liée à cette possibilité de surcharge de fonction.

```
$ g++ -Wall -Wextra surcharge_min.cpp -o surcharge && nm surcharge
0000000100003e10 T __Z3minff
0000000100003e60 T __Z3minfff
0000000100003de0 T __Z3minii
0000000100000000 T __mh_execute_header
0000000100003f00 T _main
                 U dyld_stub_binder
```

La surcharge d'opérateur en C++ (très pratique) 1/2

En C++, il est possible aussi possible de surcharger un opérateur soit pour lui affecté un nouveau comportement soit pour lui en donner un.

```
#include <iostream>
typedef struct vec2 vec2;
struct vec2 {
  float x, y;
  //permet de définir l'opération '+'
  //entre deux vec2
  vec2 operator+(vec2 v) {
    vec2 r;
    r.x = this->x + v.x; //le this-> est facultatif
    r.y = this->y + v.y; //le this-> est facultatif
    return r;
  }
};
int main(void) {
  vec2 a = {1.0f, 2.0f}, b = {3.0f, 4.0f};
  vec2 res = a + b;
  std::cout << "résultat = (" << res.x << ", " << res.y << ")\n";
  return 0;
}
```

Code source 8.10 – Surcharge de l'opérateur **'+'** pour une structure de vecteur bidimensionnel (fichier **surcharge_op.cpp**).

La surcharge d'opérateur en C++ (très pratique) 2/2

Allons plus loin en créant un comportement « produit scalaire » ($\vec{u} \cdot \vec{v}$) pour l'opérateur **'*'** mais aussi pouvoir multiplier un vecteur par un scalaire (un nombre flottant).

Notez que nous ne pouvons utiliser l'opérateur **'.'** pour exprimer un produit scalaire car cet opérateur est nécessaire pour accéder aux champs de la structure.

```
#include <iostream>
typedef struct vec2 vec2;
struct vec2 {
  float x, y;
  vec2 operator+(vec2 v) { // somme de deux vec2
    vec2 r;
    r.x = x + v.x;
    r.y = y + v.y;
    return r;
  }
  // notez le type de retour
  float operator*(vec2 v) { // produit scalaire de deux vec2
    return x * v.x + y * v.y;
  }
  // Attention cette multiplication ne fonctionne que dans un sens
  vec2 operator*(float k) { // renvoie (u * k)
    vec2 r = { k * x, k * y };
    return r;
  }
};
// Celle-là la rend fonctionnelle dans l'autre sens
// Vous remarquerez que nous sommes en dehors de la structure
vec2 operator*(float k, vec2 u) { // renvoie (k * u)
  return u * k;
}
int main(void) {
  vec2 a  = {1.0f, 2.0f}, b = {3.0f, 4.0f};
  vec2 ka  = 0.5f * a; // donne (0.5f, 1.0f)
  float s  = ka *  b; // donne (0.5f * 3.0f + 1.0f * 4.0f) = 5.5f
  std::cout << "résultat = " << s << "\n";
  return 0;
}
```

Code source 8.11 – Surcharge de l'opérateur **'*'** pour un produit scalaire et la multiplication d'un vecteur par un nombre (fichier **surcharge_op_more.cpp**).

À vous de jouer !

Après ce que nous venons d'apprendre,

1. Proposez une méthode permettant de calculer la longueur d'un vecteur bidimensionnel **vec2** (appelons-la `length`).
2. Cherchez et trouver un moyen de surcharger la fonction **min** (vue avant) pour qu'elle renvoie la **référence** vers le plus petits des deux vecteurs bidimensionnels passés en paramètre (via des **références**) aussi.
3. Au lieu d'utiliser une fonction (question précédente) proposez un opérateur.
4. Trouver un moyen concis et élégant d'afficher un élément de la structure **vec2**.

Tous vos ajouts devront être testés.

La STL

Prendre le temps de découvrir quelques fonctionnalités de la STL « *Standard Template Library* » , en particulier les bibliothèques de conteneurs. Pour cela visitez la page `https://en.cppreference.com/`

Essayons le conteneur **vector**

Voici un exemple où nous remplissons deux vecteurs (au sens tableau de valeurs), l'un avec des entiers, l'autre avec des flottants.

```
// Pour utiliser le conteneur vector de la STL
#include <vector>
#include <iostream>
using namespace std;
// pour time, srand et rand
#include <time.h>
#include <stdlib.h>
int main(void) {
  vector<int> vecDInts;
  vector<float> vecDeFloats;
  int n = 10;
  //remplir les deux vecteurs
  for(int i = 0; i < n; ++i) {
    vecDInts.push_back(rand() % n);
    vecDeFloats.push_back( (float)n * rand() / (RAND_MAX + 1.0f));
  }
  //afficher les éléments de vecDInts (méthode "classique")
  for(int i = 0; i < (int)vecDInts.size(); ++i) {
    cout << vecDInts.at(i) << "␣";
  }
  cout << endl;
  //afficher les éléments de vecDInts (avec les iterators)
  for(vector<int>::iterator it = vecDInts.begin(); it != vecDInts.end(); ++it) {
    cout << *it << "␣";
  }
  cout << endl;
  // une écriture encore plus concise, on l'utilise ici pour vecDeFloats
  // c'est du C++11 (on dirait presque du python
  for(float f : vecDeFloats) {
    cout << f << "␣";
  }
  cout << endl;
  //on peut effacer avant de sortir
  vecDInts.clear();
  vecDeFloats.clear();
  return 0;
}
```

Code source 9.1 – Utilisation du conteneur **vector** de la STL (fichier **vector_int_float.cpp**).

À vous de jouer !

Après ce que nous venons d'apprendre,

1. Proposez un exemple comparable au précédent mais utilisant le conteneur **forward_list**.
2. Une fois vos listes remplies de valeurs aléatoires, affichez-les une fois, appeler la méthode **sort** pour chacune et affichez-les une seconde fois.
3. Faire en sorte de réaliser le même processus (remplissage pseudo-aléatoire, affichage, tri, affichage) pour « notre » structure de vecteur bidimensionnel **vec2** proposée précédemment.

vector_t (en C) VS **forward_list**

Après avoir réalisé les points de la précédente page, nous allons :

1. Reprendre le code testant les performances du **vector_t** codé en C pour les comparer aux performances d'une **forward_list** de la STL/C++. Téléchargez le code :`https://expreg.org/amsi/C/PA2223S1/dl/pa_vector_int-0.12.tgz`
2. Modifions la structure du projet pour pouvoir utiliser à la fois du C et du C++ :
   - 2.1 Renommer **test.c** en **test.cpp** (et changer le petit commentaire dans le haut du fichier pour aller avec ce qu'on souhaite faire);
   - 2.2 Modifier quelques lignes du **Makefile**
     - **(+)** Ajoutons une constante **CXX = g++** après la ligne **CC = gcc**
     - **(U)** Modifions **PROGNAME** tel que **PROGNAME = vector_VS_forward_list**
     - **(U)** Pour la constante **SOURCES** remplaçons **test.c** par **test.cpp**
     - **(U+)** Renommons la constante **OBJ** en **OBJT** et ajoutons la ligne : **OBJ = $(OBJT:.cpp=.o)**
     - **(U)** Sur la ligne **$(CC) $(OBJ) $(LDFLAGS) -o $(PROGNAME)** remplaçons **CC** par **CXX**
     - **(+)** Après la cible **%.o: %.c** et sa commande, ajoutons la nouvelle cible et sa commande :

```
%.o: %.cpp
	$(CXX) $(CPPFLAGS) $(CFLAGS) -c $< -o $@
```

3. Le projet est prêt pour du C et du C++, compilons pour tester que tout est OK.
4. Dans **test.cpp** Créons la fonction (déclaration et définition) **test_forward_list** qui fait la même chose que **test_vector** mais pour une **forward_list**. Ajoutons son appel dans **main** et testons l'ensemble.

vector_t (en C) VS **forward_list** (FIN 1/2)

```
void test_forward_list(void) {
  double el;
  int i, data;
  forward_list<int> fl;
  /* on utilise la même graine (seed) de suite pseudo-aléatoire */
  srand(42);
  initTimer();
  for(i = 0; i < N; ++i) {
    data = rand() % N;
    /* deux cas particuliers où l'insertion se fait en FRONT */
    if(fl.empty() || data < fl.front()) {
      fl.push_front(data);
      continue;
    }
    /* sinon faire le parcours avec deux itérateurs :
     un devant, l'autre derrière, on utilise le second pour INSERT_AFTER */
    forward_list<int>::iterator ptr = fl.begin(), optr = ptr;
    for(; ptr != fl.end() ; optr = ptr, ++ptr) {
      if(data < *ptr)
        break;
    }
    fl.insert_after(optr, data);
  }
  el = getElapsedTime();
  /* Il nous reste à : */
```

vector_t (en C) VS **forward_list** (FIN 2/2)

```
  /* tester l'ordre et afficher le résultat */
  bool triee = true;
  forward_list<int>::iterator ptr = fl.begin(), optr = ptr;
  for(; ptr != fl.end(); optr = ptr, ++ptr) {
    if(*ptr < *optr) {
        triee = false;
        break;
    }
  }
  if(triee)
    printf("La forward_list est bien ordonnée, le temps passé en insertion de %d é
        léments est de : %f secondes\n", N, el);
  else
    printf("La forward_list n'est pas bien ordonnée, le temps passé en insertion de
        %d éléments est de : %f secondes\n", N, el);
}
```

Les *Templates* en C++

Le *Template*[27] est une classe (ou structure) ou une méthode (ou une fonction) qui permet de formaliser un concept ou une fonctionnalité (algorithme) en faisant abstraction du (ou des) type(s) de données traitées.

Son fonctionnement est proche de « la généricité par macros » précédemment expérimentée en C.

Nous en présentons un application pour des fonctions ou méthodes ainsi que pour des structures (les conteneurs de la STL sont des structures en *Templates* contenant des méthodes en *Templates*).

27. https://en.cppreference.com/w/cpp/language/templates

Syntaxe pour un *Template* de fonction ou méthode

Une forme générale de *Template* de fonction ou méthode peut ressembler à :

```
template <class T> le_type_de_retour_de_la_fonction nom_fct (
    plusieurs_paramètres_typés_au_besoin_ou_T
    ) {
      Le_corps_de_la_fonction_exploitant_T
}
```

Ainsi, par exemple, pour une fonction calculant le minimum entre deux entiers :

```
int min(int a, int b) {
  return (a < b) ? a : b;
}
```

On peut la réécrire sous la forme *Template* ainsi :

```
template <class T> T min(T a, T b) {
  return (a < b) ? a : b;
}

int main(void) {
  int i = 4, j = 2, k = min<int>(i, j);
  float x = 3.14f, y = 1.41f, z = min<float>(x, y);
  (void)k; // pour taire le warning "unused variable 'k'"
  (void)z; // pour taire le warning "unused variable 'z'"
  return 0;
}
```

Code source 9.2 – Minimum entre deux valeurs en *Template* (fichier **min.cpp**).

Écrire une fonction en *Template*

Exercice : réécrire le tri par insertion fait en C (fonction **void isort(int * t, int n);**) de manière à le rendre générique par le biais des *Templates* C++.

Syntaxe pour un *Template* de structure ou classe

Une forme générale de *Template* de structure ou classe peut ressembler à :

```
template <class T> struct_OU_class nom_qu_on_lui_donne {
  // ses attributs et/ou méthodes
  // dont certaines utilisent le type T
  //
  // D'autres peuvent être template d'un autre type ...
};
```

Ainsi, par exemple, pour deux structures, l'une pour stocker des points 2d à coordonnées entières, l'autre pour des points 2d à coordonnées flottantes :

```
struct point2di {
  int x, y;
};
struct point2df {
  float x, y;
};
```

On peut les unifier (et aller au delà) sous la forme *Template* ainsi :

```
template <class T> struct point2d {
  T x, y;
};

int main(void) {
  point2d<int>   unPoint2dACoordonneesEntieres  = { 1, 2 };
  point2d<float> unPoint2dACoordonneesFlottantes = { 0.5f, 3.2f };
  (void)unPoint2dACoordonneesEntieres; // vous savez pourquoi
  (void)unPoint2dACoordonneesFlottantes; // vous savez pourquoi
  return 0;
}
```

Code source 9.3 – Structure de point 2d en *Template* (fichier **point2d.cpp**).

Enrichire la structure *Template* **point2d**

Exercice : ajouter la gestion de plusieurs opérateurs $(+, -, *, <$ ...) au sein de la structure *Template* **point2d**.

Licence **I**nformatique et **V**idéoludisme
Université Paris 8