Introduction au langage C

Gilles Louppe

Mars 2013

Bases

Hello World

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

- La fonction `main` est toujours le point de départ d'un programme.
- `printf` est une fonction pour afficher quelque chose sur la sortie standard, dont la déclaration figure dans l'en-tête `stdio.h`.
- `\n` dénote un retour chariot.
- La valeur de retour est rendue au système d'exploitation. Par convention, 0 signifie succès.
- Chaque instruction est terminée par un point-virgule.

Variables

```
int a = 1, b, c;
float e;
```

- Toute variable doit être déclarée en spécifiant son type.
- Une variable peut être initialisée au moment de sa déclaration.

Types primitifs

`bool`	true ou false (ISO-C99, avec stdbool.h)
`char`	caractère signé
`int`	entier signé
`size_t`	entier non-signé représentant une taille ou un indice
`float`	nombre réel (précision simple)
`double`	nombre réel (précision double)

Typage

- Le langage C est un langage faiblement typé :
  Une valeur est convertie implicitement vers le type adéquat.
- Le typage d'une variable est statique :
  Le type d'une variable est déterminé à la compilation. Il ne peut changer pendant l'exécution.

Conversion

```
int a = 3, b = 2;

// Pas de conversion
a / b;                          // 1

// Conversion implicite
float c = 2.0;
a / c;                          // 1.5

// Conversion explicite
(float) a / 2;                  // 1.5

// Et ici ??
(float) (a / b);                  // ???
```

Opérateurs

(par ordre de précédence)

postfixe	`[] . -> expr++ expr--`
préfixe	`++expr --expr +expr -expr ~ !`
	`&expr *expr sizeof (type)expr`
multiplicatifs	`* / %`
additifs	`+ -`
décalages	`<< >>`
comparaisons	`< > <= >=`
égalité	`== !=`
ET binaire	`&`
OU exclusif binaire	`^`
OU binaire	`|`
ET logique	`&&`
OU logique	`||`
conditionnel	`?:`
affectations	`= += -= *= /= %= <<= >>= &= ^= |=`

if

```
if (a < b) {
    ...
}
```

```
if (a < b) {
    ...
} else {
    ...
}
```

- Le gardien d'une instruction `if` est une expression scalaire.
- Le corps du `if` est exécuté si et seulement cette expression est différente de zéro.

while

```
while (a < b) {
    ...
}

do {
    ...
} while (a < b);
```

for

```
for (int i = 0; i < N; i++) {
    ...
}
```

break, continue

- L'instruction `break` permet de quitter la boucle courante.
- L'instruction `continue` permet de passer à l'itération suivante, sans exécuter le restant de l'itération courante.

```
#include <stdbool.h>

bool stop = false;
bool pass = true;

while (true) {
    ...
    if (stop) {
        break;
    }
    if (pass) {
        continue;
    }
    ...
}
```

Tableaux (1)

```
int a[5];
a[0] = 1;
a[4] = 42;
a[-1] = 10; // Bug!
a[5] = -5;  // Bug!

int[3] b = {9, 3, 6};

int mat[3][4]; // Tableau multidimensionel
```

- Un tableau est un type de données indexable contenant des éléments du même type.
- Les éléments sont indexés à partir de 0 et jusqu'à N-1.

Tableaux (2)

```
// BubbleSort
for (size_t i = 0; i < length; i++) {
    for (size_t j = i; j < length; j++) {
        if (array[i] > array[j]) {
            int tmp = array[i];
            array[i] = array[j];
            array[j] = tmp;
        }
    }
}
```

Structures

```
// Définition d'une structure
struct Complex_t {
    double real, imaginary;
};

// Définition d'un nouveau type
typedef struct Complex_t {
    double real, imaginary;
} Complex;

// Utilisation
struct Complex_t a;
Complex b;
b.real = 1.0;
b.imaginary = -5;
```

- Une structure est un type de données composé, dont les éléments peuvent être de types différent.
- Les éléments sont accessibles par leurs noms.

Structures opaques

```
foo.h:
    typedef struct Foo_t Foo;

    Foo a;
    ...

foo.c:
    #include "foo.h"
    struct Foo_t {
        int a, b;
    };
```

Une structure opaque permet de cacher l'interface d'une structure, garantissant ainsi une plus grande flexibilité dans l'implémentation.

Fonctions (1)

- Chaque fonction doit être déclarée avant son utilisation.
- Chaque fonction prend zéro, un ou plusieurs arguments.
- Les arguments sont passés par valeur.
- Chaque fonction renvoie une valeur d'un type donné, ou `void`.
- Déclarez `static` toute fonction qui n'est utilisée que dans le fichier courant.

Fonctions (2)

```c
#include <stdio.h>

static int factorial(int n);

int main() {
    int value = 4;
    printf("%d! = %d\n", value, factorial(value));
    return 0;
}

static int factorial(int n) {
    int fac = 1;
    for (int i = 1; i <= n; i++) {
        fac *= i;
    }
    return fac;
}
```

Fonctions (3)

```
// Cette fonction ne fait rien!
void swap(int a, int b) {
    int t = a;
    a = b;
    b = t;
}
```

Entrées-sorties

```
#include <stdio.h>

int main() {
    int a, b;

    printf("Entrez une première valeur: ");
    scanf("%d", &a);
    printf("Entrez une seconde valeur: ");
    scanf("%d", &b);

    printf("%d + %d = %d\n", a, b, a + b);

    return 0;
}
```

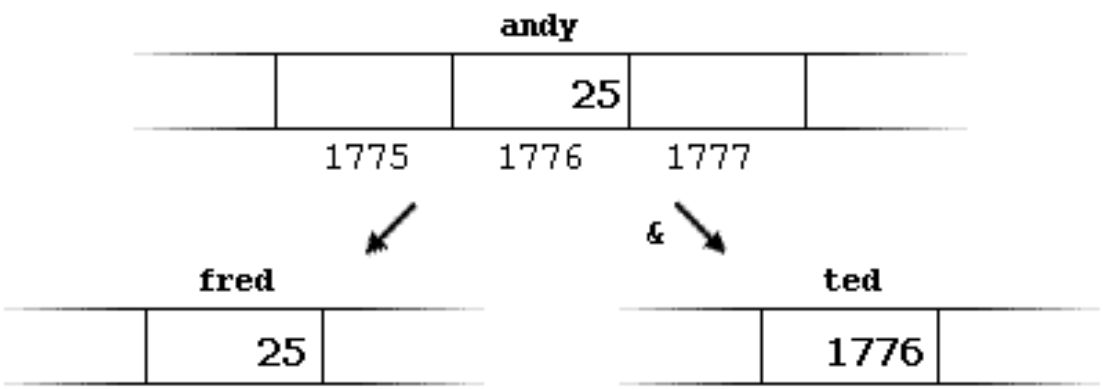
- `printf` prend comme premier argument une chaîne formatée. Les arguments suivant sont les valeurs affectées aux spécificateurs de format (c.f. http://en.wikipedia.org/wiki/Printf_format_string#Format_placeholders pour une specification complète).
- `scanf` permet d'entrer une valeur au clavier et de la stocker à l'adresse spécifiée. Attention : la gestion propre des erreurs est difficile !

Pointeurs

Variables et adresses

- L'identifiant d'une variable correspond à un emplacement mémoire, situé à une certaine adresse, contenant une valeur d'un certain type.
- Un pointeur est une variable dont la valeur est une adresse.
- Le type d'un pointeur est le type de la valeur pointée suivi de `*` (e.g., `int*` pour un pointeur vers un entier).

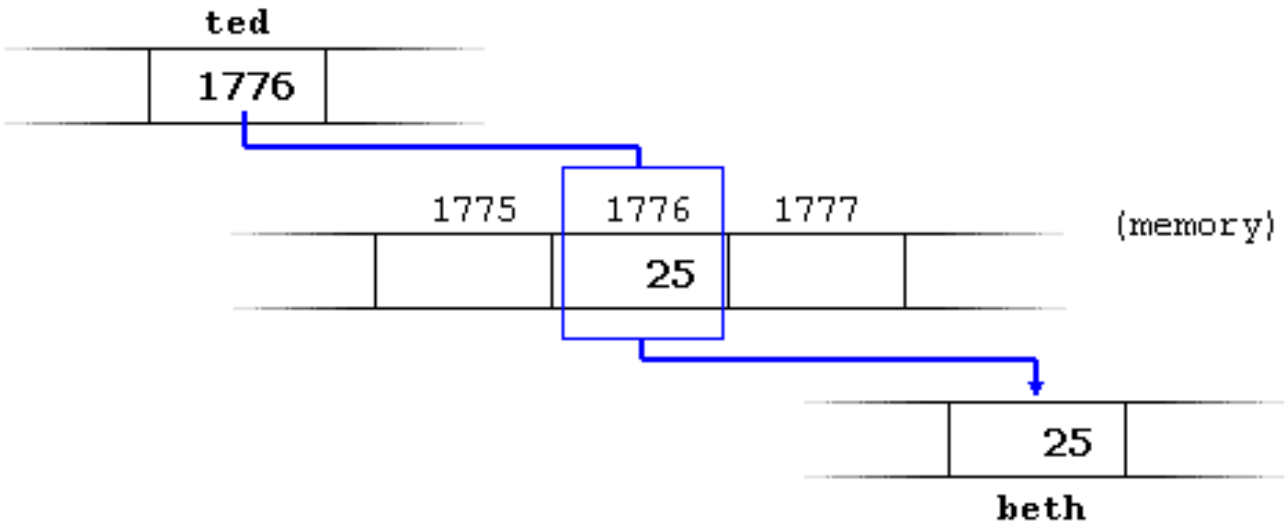
```
int andy = 25;
int fred = andy;
int* ted = &andy; // & dénote l'adresse de la variable andy
```



Déréférencement

L'opérateur * permet d'accéder à l'emplacement pointé par un pointeur.

```
int beth = *ted;
```

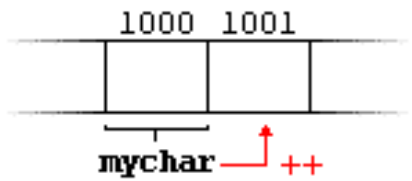


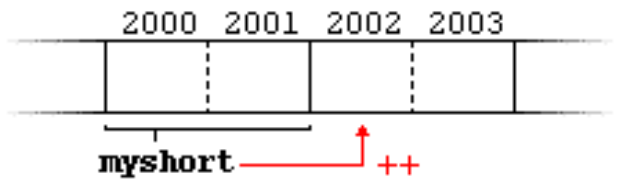
```
// Cette fonction fait quelque chose!
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```

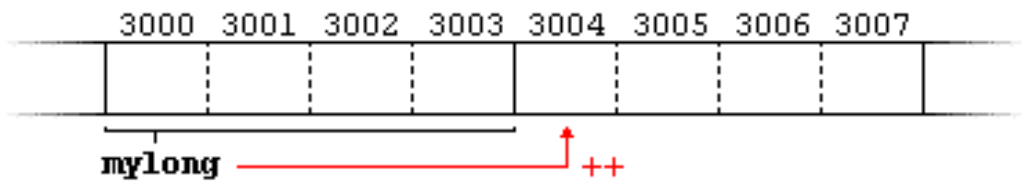
Arithmétique

- L'addition et la soustraction sont autorisées sur des pointeurs.
- $p + 1$ correspond à l'emplacement mémoire suivant p, du même type.
- $p + n$ correspond au n-ème emplacement mémoire après p, du même type.

```
char* mychar;
short* myshort;
long* mylong;
mychar++;        // mychar = mychar + 1 est équivalent
myshort++;
mylong++;
```







Tableaux

L'identifiant d'un tableau est équivalent à un pointeur pointant vers le premier élément de ce tableau.

```
int a[5];
int* p;

p = a;
a[0] = 10;
*p = 10;         // Ces deux expressions sont équivalentes
a[2] = 42;
*(p + 2) = 42;  // Ces deux expressions sont aussi équivalentes
```

Quel est le bug ?

```
#include <stdio.h>
int main() {
    int s[4], t[4];
    for (int i = 0; i <= 4; i++) {
        s[i] = t[i] = i;
    }
    printf("i:s:t\n");
    for (int i = 0; i <= 4; i++) {
        printf("%d:%d:%d\n", i, s[i], t[i]);
    }
    return 0;
}
```

```
i:s:t
0:4:0
1:1:1
2:2:2
3:3:3
4:4:4
```

Allocation / Désallocation

- Un bloc mémoire peut être alloué dynamiquement avec la fonction `malloc`.
- Tout bloc alloué dynamiquement doit être libéré explicitement avec la fonction `free`.

```
int* p = (int*) malloc(sizeof(int)); // Alloue un bloc de la taille d'un int
if (!p) {                            // Toujours vérifier le succès de malloc
    printf("Error");
    return 1;
}
free(p);                             // On libère le bloc

int* q = (int*) malloc(10 * sizeof(int)); // Alloue un bloc pour 10 int
q[0] = 42;
free(q);

Complex* i = (Complex*) malloc(sizeof(Complex));
i->real = 10;                        // Equivalent à (*i).real = 10
free(i);
```

NULL

- La valeur `NULL` corresond à l'adresse 0.
- Un pointeur `NULL` correspond à un pointeur ne pointant vers rien.

Fuites mémoire

- Un bloc mémoire non-libéré conduit à des fuites mémoires.
- Utilisez l'outil **valgrind** pour vérifier que vos programmes n'ont pas de fuites mémoires.

Pointeur de fonction

```
type_return (*func)(type_arg1, type_arg2);
```

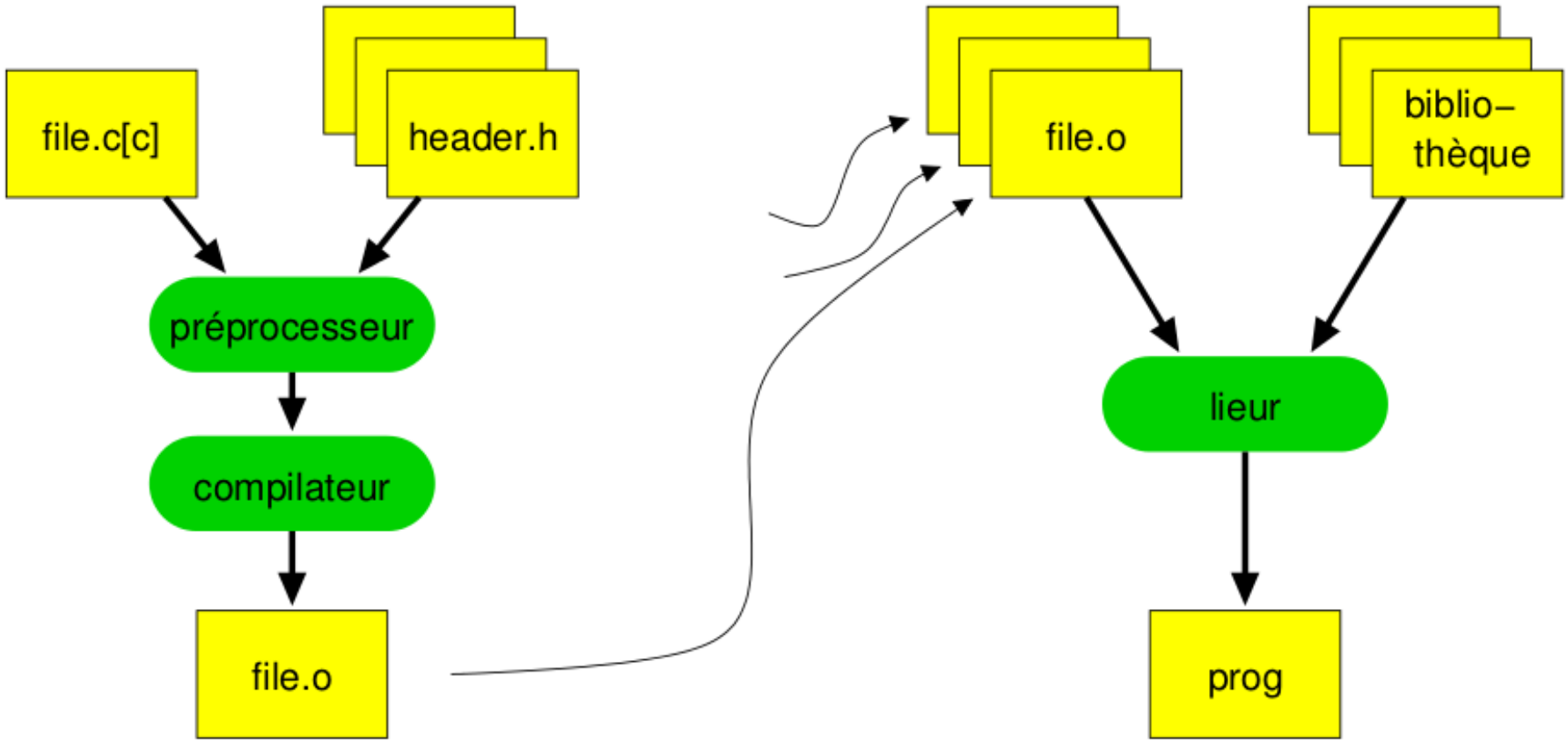
```
int foo(int a) {
    ...
}

int (*bar)(int) = foo;

bar(42); // Call foo(a) through bar
```

Compilation

Compilation



GCC

Compilation :

```
gcc main.c --std=c99 -pedantic -Wall -Wextra -Wmissing-prototypes -o main
```

`main.c`	Fichier(s) à compiler
`--std=c99`	Spécifie la norme C99
`-pedantic`	Application stricte de la norme C99
`-Wall`	Affiche (presque) tous les warnings
`-Wextra`	Affiche d'autres warnings
`-Wmissing-prototypes`	Affiche un warning pour les prototypes non définis
`-o main`	Nom du binaire compilé

Exécution :

```
./main
```

Modularité (1)

- Pour des raisons de modularité, de réusabilité et de lisibilité, il est important d'organiser le code source d'un programme en plusieurs modules.
- Chaque module est (idéalement)
  - déclaré dans un fichier d'en-têtes (.h)
  - défini dans un fichier source (.c)
- Un module est (idéalement) lui-même découpé en plusieurs fonctions implémentant chacune une routine bien spécifiée. Evitez de produire du code monolithique !

Modularité (2)

Sort.h:

```
void sort(int* array, int length);
```

Sort.c:

```
#include "Sort.h"
void sort(int* array, int length) {
    for (size_t i = 0; i < length; i++)
        for (size_t j = i; j < length; j++)
            if (array[i] > array[j]) {
                int tmp = array[i];
                array[i] = array[j];
                array[j] = tmp;
            }
}
```

main.c:

```
#include "Sort.h"
int main() {
    ...
    sort(array, length);
    ...
}
```

Inclusion gardée (1)

Chaque fichier inclus avec la commande `#include` est copié par le préprocesseur à l'endroit où la commande est invoquée.

```
a.h:
    struct Complex_t {
        float real, imaginary;
    };

b.h:
    #include "a.h"
    ...

main.c:
    #include "a.h"
    #include "b.h"

    int main() {
        struct Complex_t i;
        ...
    }
```

Quel est le problème ?

Inclusion gardée (2)

La structure `Complex_t` est définie deux fois !

a.h:

```
#ifndef _A_H_                    // Inclusion gardée
#define _A_H_
struct Complex_t {
    float real, imaginary;
};
#endif
```

b.h:

```
#include "a.h"
...
```

main.c:

```
#include "a.h"
#include "b.h"

int main() {
    struct Complex_t i;
    ...
}
```

Librairies standards

`#include <stdio.h>`	Fonctions d'entrées-sorties `printf`, `scanf`, `fopen`, `fclose`, ...
`#include <stddef.h>`	Définitions standards `NULL`, `size_t`, ...
`#include <stdlib.h>`	Librairie standard `malloc`, `free`, `rand`, ...
`#include <math.h>`	Fonctions mathématiques `fabs`, `sqrt`, `exp`, `sin`, `cos`, ...
...	

Robustesse

Contrat d'une fonction

- Votre fonction fonctionne t-elle correctement ?
- ... même dans les cas particuliers ?

Exemple :

```
void sort(int* array, size_t length);
...
sort(array, 10); // array est t-il effectivement trié?
sort(array, 1);
sort(array, 0);
sort(NULL, -10); // Que se passe t-il?
```

Robustesse

- En présence de tableaux, assurez-vous de ne pas sortir des bornes.
- Sauf explicitement mentionné, ne faites jamais confiance aux données entrées par l'utilisateur.
- Quand vous utilisez une fonction, pensez toujours à gérer les erreurs éventuelles (e.g., `malloc`).
- Libérez toujours un bloc mémoire alloué dynamiquement.
- Fermez toujours un fichier.

Style

Gneh ?

```
#include "stdio.h"
#define e 3
#define g (e/e)
#define h ((g+e)/2)
#define f (e-g-h)
#define j (e*e-g)
#define k (j-h)
#define l(x) tab2[x]/h
#define m(n,a) ((n&(a))==(a))

long tab1[]={ 989L,5L,26L,0L,88319L,123L,0L,9367L };
int tab2[]={ 4,6,10,14,22,26,34,38,46,58,62,74,82,86 };

main(m1,s) char *s; {
    int a,b,c,d,o[k],n=(int)s;
    if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b); printf(b); }
    else switch(m1-=h){
    case f:
        a=(b=(c=(d=g)<<g)<<g)<<g;
        return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
    case h:
        for(a=f;a<j;++a)if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
    case g:
        if(n<h)return(g);
        if(n<j){n-=g;c='D';o[f]=h;o[g]=f;}
        else{c='\r'-'\b';n-=j-g;o[f]=o[g]=g;}
        if((b=n)>=e)for(b=g<<g;b<n;++b)o[b]=o[b-h]+o[b-g]+c;
        return(o[b-g]%n+k-h);
    default:
        if(m1-=e) main(m1-g+e+h,s+g); else *(s+g)=f;
        for(*s=a=f;a<e;) *s=(*s<<e)|main(h+a++,(char *)m1);
    }
}
```

Lisibilité (1)

- 40 à 80% de la vie d'un programme correspond à de la maintenance.
- Rares sont les programmes qui sont maintenus à long terme par leurs auteurs originaux.
- Ecrire un code propre, lisible et clair est indispensable, pour vous-même et pour les autres.

Lisibilité (2)

Choisissez un style et **adhérez y** !

	Suggestions :
Indentation	2 ou 4 espaces par niveau
Position des accolades	Même ligne pour {, à la ligne pour }
Espaces, lignes vides	`1 + 1`, `methode()`
Minuscules, majuscules	`MY_CONST`, `MyClasse`, `myVar`, `myFunction()`
Noms des variables	Les plus significatifs possible
Langue	Anglais

http://www.montefiore.ulg.ac.be/~piater/Cours/Coding-Style/Coding-Style.pdf

Documentation

Ni trop, ni trop peu.

- Utilisez des commentaires inline (//) pour documenter les parties non-triviales de votre code.
- Documentez chaque déclaration de fonction en spécifiant ce que fait la fonction, ce que sont les arguments et quelle est la valeur de retour.

```
/* ------------------------------------------------------------------------ *
 * Create a sorted array of integers (from 0 to length-1).
 *
 * The array must later be deleted by calling free().
 *
 * PARAMETERS
 * length       Number of elements in the array (pre-condition: 0 < length)
 *
 * RETURN
 * array        A new array of integers, or NULL in case of error
 * ------------------------------------------------------------------------ */
int* createSortedArray(size_t length);
```

Références

Références

- Pour les fonctions, consultez les pages **man** sous UNIX (e.g., **man 3 printf**).
- **Le langage C, Norme ANSI** (Brian W. Kernighan, Denis M. Ritchie, 2004)
- **Langage C** (Claude Delannoy, 2008)

- Ces slides sont librement adaptés du cours **Initiation au langage C** (Justus H. Piater, 2005-2006).
- Chapitre Pointeurs : http://www.cplusplus.com/doc/tutorial/pointers/