Le langage C: fiche synthétique

Source complète: `https://en.cppreference.com/w/c`
Compilateur sous Linux/Unix: `gcc`, `clang`

Commentaires
`/* .... */` multi-lignes ou `// ...` ligne simple (depuis C99)

Constantes

entiers: décimal: `12`, octal: `012`, hexadécimal: `0x12`, `0X12`

flottants: (décimal) `1.2`, `2.3e-4`, (hexadécimal) `0x1.2p3` (depuis C99)

Chaînes de caractères

`"..."`	caractères simples
`u8"..."`	caractères UTF8 (depuis C11)
`u"..."`	caractères sur 16bits/UTF16 (depuis C11)
`U"..."`	caractères sur 32bits/UTF32 (depuis C11)
`L"..."`	caractères larges

Quelques caractères spéciaux: guillemet `\"`, nouvelle ligne `\n`, tabulation `\t`, nul `\0`, backslash `\\`, ...

Mots-réservés

`auto`	`default`	`float`	`register`	`struct`	`volatile`
`break`	`do`	`for`	`return`	`swtich`	`while`
`case`	`double`	`goto`	`short`	`typedef`	
`char`	`else`	`if`	`signed`	`union`	
`const`	`enum`	`int`	`sizeof`	`unsigned`	
`continue`	`extern`	`long`	`static`	`void`	
`inline` (C99)	`restrict` (C99)				

Opérateurs communs

Affectation	In(de)crémentation	Arithmétique	Logique	Comparaison	Accès membre	Autres
		`+a`				
`a = b`		`-a`				
`a += b`		`a + b`				
`a -= b`		`a - b`				
`a *= b`		`a * b`		`a==b`	`a[b]`	`a(...)`
`a /= b`	`++a`	`a / b`	`!a`	`a!=b`	`*a`	`a,b`
`a %= b`	`--a`	`a % b`	`a && b`	`a<b`	`&a`	`a?b:c`
`a &= b`	`a++`	`~a`	`a || b`	`a>b`	`a->b`	`sizeof(a)`
`a |= b`	`a--`	`a & b`		`a<=b`	`a.b`	`_Alignof(a)` (C11)
`a ^= b`		`a | b`		`a>=b`		
`a <<= b`		`a^b`				
`a >>= b`		`a << b`				
		`a >> b`				

Types élémentaires classiques

Type vide: `void`

Caractères:

`char`: 8 bits, 1 octet, code 256 valeurs, signé ou non

`signed char`: 8 bits, caractères signés -128..127, (ASCII, 0..127)

`unsigned char`: 8 bits, caractères non signés 0..255, (ASCII étendu)

Entiers:

`short`/`short int`: au moins 16 bits (2 octets), souvent 16 bits

`int`: au moins 16 bits, souvent 32 bits (4 octets)

`long`/`long int`: au moins 32 bits, souvent 32 bits, dès fois 64 bits (8 octets)

`long long`/`long long int`: au moins 64 bits, souvent 64 bits

Versions signées (négatifs/positifs): `signed short`, `signed short int`, `signed`, `signed int`, `signed long`, `signed long long`

Versions non-signées (positifs): `unsigned short`, `unsigned short int`, `unsigned`, `unsigned int`, `unsigned long`, `unsigned long long`

Flottants:

`float` simple précision

`double` double précision

`long double` précision étendue

À la base en C, il n'existe pas de type booléen. Toute expression est dite fausse si l'évaluation de cette expression est nulle sinon elle est considérée comme vraie.

`0` est faux

`1` est vrai

`"ceci est faux"` est vrai

`1-1` est faux

Types énumérés

```
enum NomTypeEnum { CONSTANTE_1, CONSTANTE_2, .., CONSTANTE_N }
```

La constante `CONSTANTE_1` est de type `enum NomTypeEnum`.

Définition de type

`typedef un_type nouveau_nom;`

exemple 1: on définit un nouveau type `longueur` comme un entier non signé
`typedef unsigned int longueur;`

exemple 2: `typedef enum nomsemaine { LUN, MAR, MER, JEU, VEN, SAM, DIM} semaine;`
Le type `semaine` est un nouveau type enuméré et peut remplacer le type `enum nomsemaine`. Ainsi, `LUN` est une constante de type `semaine`.

Variables

Toute variable utilisée doit être déclarée au préalable.

`nom_type nom_variable;` déclaration d'une variable non-initialisée (valeur inconnue)

`nom_type nom_variable = val;` déclaration d'une variable initialisée

En C89 (< C99), les déclarations de variables doivent toujours être faites en début de bloc (un bloc commence par une accollade {) avant toute autre instruction.

Variable globale: variable déclarée en dehors de toute fonction, consultable/modifiable partout dans le programme

Variable locale: variable déclarée dans une fonction, consultable/modifiable uniquement dans cette fonction

Une variable `var` locale de même nom qu'une variable globale `var` cache la variable globale, seule la variable locale est accessible dans cette focntion.

Fonctions

Déclaration de fonction: `type_retour nom_fonction(type_arg1 var_arg1,...);`

Définition de fonction:

```
type_retour nom_fonction(type_arg1 var_arg1,...)  {
 <declarations variables locales>
 <instructions>
```

}

`var_arg1, var_arg2,...` sont des variables locales à la fonction `nom_fonction`. Ces variables sont initialisées avec les valeurs mis en paramètre à chaque appel de fonction (passage de paramètres par copie).

Instruction `return <expression>`: stoppe l'exécution de la fonction et retourne la *valeur* de l'expression dont le type est `type_retour` (en général en fin de fonction).

Type retour `void`: le type vide, si le type retourné `type_retour` est `void` alors la fonction ne retourne rien (instruction `return` inutile) → c'est une procédure.

Appels de fonction:

`nom_fonction(val_arg1,...)` est une expression dont la valeur est celle retournée par la fonction (si pas `void`), le résultat de l'évaluation de `val_arg1` initialise la variable locale `var_arg1` (autrement dit `var_arg1 = val_arg1` etc)

exemple 1: `val = nom_fonction(val_arg1,...);` La valeur retournée (non `void`) est stockée dans la variable `val`.

exemple 2: `return nom_fonction(val_arg1,...);` retourne comme valeur de la fonction courante le résultat de l'appel à `nom_fonction`.

exemple 3: `nom_fonction2(nom_fonction(val_arg1,...))` appelle la fonction `nom_fonction2` avec pour paramètre le résultat de la fonction `nom_fonction(val_arg1,...)`.

Pour appeler une fonction, cette fonction doit avoir été déclarée auparavant (pas nécessairement définie).

Fonction principale `main`: tout programme C a une (et une seule) fonction principale qui sert de point d'entrée du programme:

```
int main() { ...  }

int main(void) { ...  }

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

`argc`: nombre de paramètres attendus sur la ligne de commandes lors de l'appel du programme

`argv`: tableau de paramètres de la ligne de commande: `argv[0]` nom du programme, `argv[1]` paramètre 1, `argv[2]` paramètre 2, ..., `argv[argc-1]` dernier paramètre.

La valeur du type `int` retournée par `main` est un code erreur analysé par le système d'exploitation à la fin de l'exécution du programme. La valeur `0` signifie que le programme n'a pas eu d'erreur lors de son exécution. Il n'est pas obligatoire de retourner un code erreur, mais conseillé (`return 0;` quand le programme s'arrête normalement.)

Expression/Instructions Toute instruction s'écrit dans une fonction et se termine par un `;`. En C, une instruction est considérée comme une expression qui a donc une valeur d'un certain type.

Structures de contrôle

Condition: `expr` est une expression qui retourne vrai ($\neq 0$) ou faux ($= 0$)

```
if (expr) instruction;

if (expr) { instruction1; instruction2; ...}

if (expr) instruction1 else instruction2;

if (expr) { instruction1; instruction2; ...} else { instruction3; ...}
```

Sélection de cas:
`expr` est une expression qui retourne une valeur dont le type est énuméré ou intégral (`char`,`int`...)
`constant_expri` est une expression constante (valeur d'un type énuméré ou un caractère (ex: `'a'`), un entier (ex: `1`)).

```
switch(expr){
  case constant_expr1 :  instructions; break;
  case constant_expr2 :  instructions; break;
  ...
  default:  instructions;
}
```

Boucle *tant que*: `condition_entree_boucle` est une expression qui retourne vrai ($\neq 0$) ou faux ($= 0$). L'itération se poursuit tant que `condition_entree_boucle` est vraie.

```
while (condition_entree_boucle) {
  instruction1;
  ...
}
```

Boucle *répéter jusqu'à*: l'itération se poursuit tant que `condition_entree_boucle` est vraie

```
do {
  instruction1;
  ...
}
while(condition_entree_boucle)
```

Boucle *pour*: l'itération se poursuit tant que `condition_entree_boucle` est vraie

```
for (initialisations; condition_entree_boucle; instructions_increments) {
  instruction1;
```

```
  ...
}
```

Instruction `break`: arrête l'exécution de l'itération si présent dans le corps d'une boucle (`while`, `do while`, `for`) et passe à la suite.

Instruction `continue`: ignore les instructions qui suivent dans l'itération courante (`while`, `do while`, `for`) et passe directement aux tests de conditions de boucle et à l'éventuelle itération suivante.

Instruction `goto label`: saut inconditionel à la ligne dont l'étiquette est `label`. Étiqueter une ligne avec `label` se fait de la façon suivante: `label:;`

Directives du préprocesseur Avant la compilation effective d'un fichier source C, il y a une étape de précompilation sur ce fichier qui exécute des directives qui sont inclues dans le fichier source (manipulation de textes, insertion de fichiers, etc). Les directives ne font pas parties du langage C mais leur utilisation est nécessaire en pratique. Une directive peut être insérée à n'importe quel endroit dans le fichier source. Les directives sont écrites sur une seule ligne et débute par le caractère `#`.

`#define MOT` établit que le mot `MOT` est connu du préprocesseur dans la suite du fichier.

`#ifdef MOT` si `MOT` est connu du préprocesseur, inclure le code qui suit cette condition dans le code à effectivement compiler

`#ifndef MOT` si `MOT` n'est pas connu du préprocesseur, inclure le code qui suit cette condition dans le code à effectivement compiler

`#else` alternative à `#ifdef MOT`, `#ifndef MOT`, si la condition n'est pas verifiée inclure le code sous le `#else`.

`#endif` délimite la fin du bloc de code dont l'inclusion est conditioné par `#ifdef#ifndef#else`

`#include<fichier.h>`/ `#include"fichier.h"` inclure le contenu d'un fichier de déclaration (*header file*) à cet endroit précis dans le fichier source avant de compiler.

`#undef MOT` établit que le mot `MOT` n'est plus connu du préprocesseur dans la suite du fichier.

`#error "message"` Si le préprocesseur doit inclure cette directive dans le code à compiler, la compilation est avortée et le message est affiché.

`#line numero` indique au préprocesseur que la ligne de cette directive a pour numéro celui de la directive, et la ligne suivante aura pour numéro `numero + 1` et ainsi de suite.

`#pragma parametre` Indique au compilateur certaines options de compilation à activer pour compiler le fichier en question.

Macros du préprocesseur Outre les directives du préprocesseur, il peut également être utilisée pour définir des macros avec la directice `#define`.

`#define UNE_CONSTANTE 3` Le préprocesseur remplacera toute occurrence de `UNE_CONSTANTE` par la valeur `3` suite à cette macro. C'est le moyen de définir des constantes en C.

`#define AJOUTE(a,b) a+b` Le préprocesseur remplacera toute occurrence de `AJOUTE(x,y)` par l'expression `x+y` dans la suite.

Tableaux Structure de données contenant `N` éléments de même nature dont l'allocation en mémoire est contiguë.

Déclaration simple `type nom_tableau[N];`

→ Les éléments du tableau ne sont pas initialisés à la déclaration, juste alloués en mémoire.

Déclaration et initialisation `type nom_tableau[N] = { val_type0, val_type1,...,val_typeN_1};`

→ on peut omettre `N`: `type nom_tableau[] = { val_type0, val_type1,...,val_typeN_1};`

Premier élément du tableau: `nom_tableau[0]` est une variable de type `type` (consultable/modifiable)

Dernier élément du tableau: `nom_tableau[N-1]`

La taille `N` du tableau est fixée à la déclaration et ne peut pas être modifiée.

Parcours d'un tableau de taille `N`, soit `int i;` une variable déclarée auparavant

```
for(i=0; i < N; ++i) { ...instructions sur nom_tableau[i]...  }
```

```
i=0; while(i<N) { ...instructions sur nom_tableau[i]...; ++i; }
```

```
i=0; do { ...instructions sur nom_tableau[i]...; ++i; } while(i<=N);
```

Chaînes de caractères simples

Une chaîne de caractères simples est un tableau de caractères (`char chaine[N]`) dont le dernier caractère est `\0`. Par exemple, la chaîne `"un chat"` est stockée dans un tableau de 8 caractères, le caractère à l'indice 0 est `u`, celui à l'indice 6 est `t` et celui à l'indice 7 est `\0`. Voiçi deux façons équivalentes de déclarer une variable contenant une chaine de caractères:

```
char chat1[] = "un chat";
```

```
char chat2[] = { 'u','n',' ','c','h','a','t','\0'};
```

Manipulation simple:

```
char chou[] = chat1; chou[5]='o'; chou[6]='u';
```

Structures
Un type structuré est un type qui est composé de plusieurs champs, chaque champ a son propre type. Les champs d'une variable structurée sont alloués en séquence du premier au dernier champ (allocation contiguë)

Déclaration d'un type structuré:

```
struct nomStruct
{
  type1 champ1;
  type2 champ2;
  ...
  typeN champN;
};
```

Le type ainsi défini est `struct nomStruct`. Si le nom `nomStruct` n'est pas présent, on dit que le type de la structure est anonyme. En général, on nomme un type structuré pour un usage plus flexible avec `typedef`. Exemple:

```
typedef struct _date{
  unsigned char heure;
  unsigned char minute;
  unsigned char jour;
  semaine jour_semaine;
  unsigned char mois;
  signed int annee;
} date;
```

Déclaration d'une variable de type structuré sans initialisation:

```
struct _date madate;
date madate; /* equivalent à la ligne du dessus mais plus court */
```

Déclaration d'une variable de type structuré avec initialisation:

```
date madate = { 15,42,25,12,SAM,12,2021 };
```

Consultation/Modification d'un champ de variable structurée:

```
date.heure=2;
date.jour_semaine=VEN;
```

Taille mémoire d'une variable structurée: somme des tailles des champs.

La taille de `date` en bits est:
8 (heure) + 8 (minute) + 8 (jour) + 32 (jour_semaine) + 8 (mois) + 32 (annee).

Unions
Un type union est un type qui est composé de plusieurs champs, chaque champ a son propre type. Une variable de type union n'a qu'un seul champ actif à la fois.
Déclaration d'un type union:

```
union nomUnion
{
  type1 champ1;
  type2 champ2;
  ...
  typeN champN;
};
```

Le type ainsi défini est `union nomUnion`. Si le nom `nomUnion` n'est pas présent, on dit que le type d'union est anonyme. En général, on nomme un type union pour un usage plus flexible avec `typedef`. Exemple:

```
typedef union _chiffre{
  int valeur;
  char text[7];
} chiffre;
```

Déclaration d'une variable de type union sans initialisation:

```
union _chiffre monchiffre;
chiffre monchiffre; /* equivalent à la ligne du dessus mais plus court */
```

Déclaration d'une variable de type union avec initialisation:

```
chiffre monchiffre = { 8 }; /* membre actif:  valeur */
chiffre monchiffre2 = { "huit" }; /* membre actif:  text */
chiffre monchiffre3 = { .text="sept" }; /* membre actif:  text */
chiffre monchiffre4 = { .valeur=4 };/* membre actif:  valeur */
```

Consultation/Modification d'un champ de variable union:

```
monchiffre.valeur=2; /* membre actif:  valeur */
monchiffre.text="quatre"; /* membre actif:  text */
```

Taille mémoire d'une variable union: maximum des tailles des champs.

La taille de `monchiffre` en bits est: max(valeur,text) = max(32,7*8) = 56.

Pointeurs

Chaque variable, chaque fonction d'un programme en exécution est stockée dans une zone mémoire qui lui est propre et adaptée à sa taille. (un caractère nécessite 1 octet, un entier 4 octets, pour une fonction cela dépend de son nombre d'instructions dans la fonction, pour une structure son nombre et la taille de ses champs, etc). Chaque zone mémoire a sa propre adresse qui permet à l'ordinateur de consulter/modifier le contenu de cette zone mémoire. L'adresse d'une zone mémoire est un nombre positif entier de 64 bits (sur les machines courantes).

Comment récupérer l'adresse d'une variable `var`, d'une fonction `fonc`:

```
&var
&fonc
```

L'adresse étant une valeur, on peut la stocker dans une variable. Une variable qui stocke des adresses est de type *pointeur*.

Déclaration d'un pointeur sur des variables de type `un_type` sans initialisation:

```
un_type* pointeur;
```

Exemples de pointeurs:

```
int* pEntier;

char* pCaractere;

date* pDate;

chiffre* pChiffre;
```

Un pointeur nul est un pointeur qui ne stocke l'adresse d'aucune variable.

Déclaration d'un pointeur sur des variables de type `un_type` initialisé à nul:

```
un_type* pointeur = NULL;
```

Vérifier qu'un pointeur est nul/pas nul:

```
if(ptr==NULL) { ....  }

if(ptr!=NULL) { ....  }
```

Stocker l'adresse d'une variable `var` de type `var_type` dans un pointeur:

```
var_type * ptrVar = &var;
```

Déréférencement de pointeurs
Soit `ptrVar` un pointeur contenant l'adresse d'une variable `var` de type `var_type`:

```
var_type * ptrVar = &var;
```

On dit que `ptrVar` *référence* (*pointe sur*) la variable `var`. On peut accéder à la variable `var` par déréférencement du pointeur `ptrVar`.

Déréférencement de pointeur:

```
*ptrVar
```

Exemple:

```
int var1 = 2;

int var2 = 4;

int * pVar = &var1; /* pVar refere var1 */

*pVar = 3; /* var1 vaut 3 */

pVar = &var2; /* pVar refere maintenant var2 */

var1 = *pVar; /* var1 vaut 4 (la valeur de var2) */
```

Pointeurs de type structuré/type union

Soit le type structuré `date` (voir structures)

```
date uneDate = { 12,34,25,SAM,12,2021 };

date * pDate = &uneDate;
```

Déréférencement de pointeur version 1:

```
int heure = (*ptrVar).heure;

semaine jour = (*ptrVar).jour;

(*ptrVar).minute = 12;
```

Déréférencement de pointeur version 2 (équivalent et plus lisible):

```
int heure = ptrVar->heure;

semaine jour = ptrVar->jour;

ptrVar->minute = 12;
```

Fonctionne également avec les types union:

```
chiffre unChiffre = { 1 };

chiffre * pChiffre = &unChiffre;

pChiffre->valeur = 2;

pChiffre->text = "deux";
```

Pointeur de pointeur
Un pointeur est une variable qui stocke l'adresse d'une variable. Donc un pointeur peut très bien stocker l'adresse d'un pointeur.

Déclaration d'un pointeur de pointeur de type `var_type`: `var_type** pointeur_de_pointeur;`
Exemple:

```
int var1 = 3;

int var2 = 4;

int * pVar1 = &var1;

int * pVar2 = &var2;

int ** pVar = &pVar1;

(**pVar) = 1; /* double dereferencement:  var1 = 1 */

pVar = &pVar2;

(**pVar) = 1; /* double dereferencement:  var2 = 1 */
```

Arithmétique de pointeurs et tableaux
Soit un type `var_type`, voiçi un tableau de M éléments de ce type:

```
var_type tableau[M];
```

Soit un pointeur `pVar` de type `var_type*`:

```
var_type *pVar;
```

Un tableau est une zone de mémoire contenant M variables de type `var_type` de facon contiguë. L'adresse du tableau est l'adresse de son premier élément:

```
pVar = &tableau[0]; /* addresse du tableau */
```

Avec des opérations arithmétiques, on peut explorer le tableau avec le pointeur:

```
pVar += 1; /* pVar pointe maintenant sur tableau[1] */

pVar += 2; /* pVar pointe maintenant sur tableau[3] */

pVar = pVar - 3; /* pVar revient sur tableau[0] */

pVar = pVar + M-1; /* pVar pointe sur le dernier élément du tableau */
```

Noter que lorsque l'on déclare un tableau comme suit:

```
var_type tableau[M];
```

On déclare implicitement que `tableau` est un pointeur de type `var_type *`, autrement dit `tableau == &tableau[0]`, `tableau+1 == &tableau[1]`, etc.

Une chaine de caractère étant un tableau particulier:

```
char chat[] = "un chat";
```

`chat` est en fait un pointeur sur des caractères (`char *`). On peut ainsi écrire:

```
char * chat = "un chat";
```

Pointeurs de fonction
Un pointeur peut stocker l'adresse d'une fonction. Le type du pointeur dépend la signature de la fonction pointée.
Exemple:

Soit la déclaration de fonction: `unsigned f1(int i, char c);`

La signature de `f1` est `(int,char)->unsigned`

Soit la déclaration de fonction: `unsigned f2(int i, char c);`

La signature de `f2` est la même que celle de `f1`.

Déclaration d'un pointeur `pf` pour les fonctions de signature `(int,char)->unsigned`

```
unsigned (*pf)(int,char);
```

Utilisation de `pf`

```
pf = &f1;

pf(2,'a'); /* appel de f1(2,'a') */

pf = &f2;

pf(2,'a'); /* appel de f2(2,'a') */
```

Entrées/Sorties

Nécessite d'inclure la bibliothèque standard du C dans le fichier source:

```
#include <stdio.h>
```

Écrire sur la sortie standard:

`int putchar(char c);` écrit le caractère `c` sur l'entrée standard, retourne `EOF` si problème.

Exemples:

```
putchar('A');
```

```
char c = 'a'; putchar(c);
```

```
int printf(const char *format, ...);
```

Exemples:

```
printf("Bonjour\n"); /* Bonjour */
```

```
printf("Bonjour %s\n", "Tony"); /* Bonjour Tony */
```

```
printf("%d bonjours %s%c %d\n", 3, "valent mieux qu", 'e', 1); /* 3 bonjours valent mieux
  que 1 */
```

Lire sur l'entrée standard:

`char getchar();` Lit un caractère sur l'entrée standard.

Exemples:

`char c = getchar();` lit un caractère et le stock dans `c`

`do { c = getchar(); } while(c!=EOF);` lit des caractères jusqu'au dernier (`EOF`: *end of file*)

```
int scanf(const char *format, ...);
```

Exemples:

```
int entier; scanf("%d",&entier); /* attends un entier */
```

```
char caractere; scanf("%c",&caractere); /* attends un caractère */
```

```
char nom[30]; scanf("Votre nom ?  %s",nom); /* attends un nom de moins de 30 caracteres */
```

Gestion dynamique de la mémoire

Nécessite d'inclure la bibliothèque standard du C dans le fichier source:

```
#include <stdlib.h>
```

Allocation dynamique de la mémoire (allocation dans le tas).

```
void * malloc(size_t taille)
```

→ alloue dans le tas un bloc de mémoire qui est de `taille` octets et renvoie un pointeur avec l'addresse du début de bloc.

Exemples:

```
int * vingtEntiers = (int *) malloc(20 * sizeof(int));
```

→ `sizeof(int)` retourne la taille en octet d'un entier (4 octets en général) donc `malloc` alloue 20 * 4 = 80 octets, assez pour stocker un tableau de 20 entiers. `(int *)` sert à convertir le pointeur de type `void *` retourné par `malloc` en un pointeur de type `int *`.

```
date * trenteDates = (date *) malloc(30 * sizeof(date));
```

→ allocation d'un tableau de 30 `date`.

Libération de la mémoire: **toute mémoire allouée doit être libérée avant la fin du programme !!** Sinon c'est la **fuite de mémoire**, et tous les problèmes d'instabilités qui vont avec.

```
void free(void *ptr);
```

Exemples:

```
free(vingtEntiers);
```

```
free(trenteDates);
```