

# [C] Kit de base



## Introduction

Le langage C est un langage de bas niveau qui doit être compilé avant d'être exécuté.

## Variables

En C, les variables ont 4 caractéristiques :

- Un **type** (int, float, char ou void)
- Un **identifiant** (un nom).
- Une **valeur** (correspondant au type).
- Une **adresse mémoire** (en hexadécimale).

Voici les différents types de variables avec leur taille associé en octet ainsi que leurs valeur minimum et maximum :

Type	Occupation mémoire	Valeur minimum	Valeur maximum
char	-	-	-
signed char	1 octet	-128	127
unsigned char	1 octet	0	255
int	2 octets / 4 octets	-32 768 / -2 147 483 648	32 767 / 2 147 483 647
unsigned int	2 octets / 4 octets	0 / 0	65 535 / 4 294 967 295
short	2 octets	-32 768	32 767
unsigned short	2 octets	0	65 535
long	4 octets	-2 147 483 648	2 147 483 647
unsigned long	4 octets	0	4 294 967 295
long long	8 octets	-9 223 372 036 854 780 000	9 223 372 036 854 780 000
unsigned long long	8 octets	0	18 446 744 073 709 600 000
float	4 octets	-3.4e38	3.4e38
double	8 octets	-1.7e308	1.7e308
long double	10 octets	-1.1e4932	1.1e4932

## Déclaration

- Entier :

```
int myInteger = 0;
```

- Flottant :

```
float myFloat = 26.34;
```

- Caractère :

```
char myChar = 'J';
```

```
// Avec le code ASCII (caractère 'A')
```

```
char myChar = 65;
```

- Chaîne de caractères (tableau de char terminant par '\0') :

```
char name[256]="Elhieroc";
```

- Tableau :

```
int monTableau[5]={1, 4, 7};
```

- Constante (en majuscule) :

```
const float PI = 3.14;
```

- Statique (la valeur reste en mémoire après avoir quittée la fonction appelante) :

```
static int maVariableStatique = 2;
```

- Préprocesseur (constante dont la valeur est remplacée par le compilateur lors de la compilation) :

```
#define LONGUEUR (15)
```

# Bibliothèques

- Standard :

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

- Standard Input/Output :

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

- String (pour les chaînes de caractères :

```
#include <string.h>
```

Assert (pour les vérifications) :

```
#include <assert.h>
```

# Fonctions

## Déclaration / Implémentation

En C, on distingue le prototype d'une fonction de son implémentation. Le prototype n'est pas indispensable mais utile lorsqu'on souhaite implémenter les fonctions après la fonction main par exemple.

- Prototypage :

```
float calculator(float op1, float op2);
```

- Implémentation :

```
float calculator(float op1, float op2){  
    return op1 + op2;  
}
```

## Type et valeur de retour

Chaque fonction a un type qui doit correspondre à sa valeur de retour :

```
int myFunc(){  
    return 3;  
}
```

Cette valeur de retour peut être stockée dans une variable :

```
myVar=myFunc(); // '3'
```

## Main

Tout programme en langage C présente la fonction main qui sera naturellement appelée lors de l'exécution du programme :

```
int main(){  
  
    return 0;  
}
```

## Printf

La fonction **printf()** permet d'afficher des éléments dans la sortie standard (souvent il s'agit de la console).

```
printf("Hello world !");
```

Elle prend en charge les flags pour afficher des variables :

```
printf("Mon entier : %d et mon flottant : %f \n", myInt, myFloat);
```

Remarque : On notera la présence du caractère '\n' pour marquer le retour à la ligne.

## Scanf

La fonction `scanf()` permet de récupérer des éléments de l'entrée standard (souvent il s'agit du clavier).

Ici, la valeur est transmise dans la variable *myInt* :

```
scanf("%d", &myInt);
```

Remarque : Il fonctionne avec les flags comme *printf*.

# Commentaires

Commentaire sur une seule ligne :

```
// Mon commentaire
```

Commentaire sur plusieurs lignes :

```
/*  
    Mon commentaire  
    en plusieurs lignes  
*/
```

# Cast (transtypage)

- Passer d'un flottant à un entier :

```
int myInteger = (int)myFloat;
```

# Les flags

Pour certaines fonctions, il peut être nécessaire d'utiliser des flags (ex : `printf` ou `scanf`).

Voici la liste des flags les plus courants :

Flags	Types de variables
-------	--------------------

%d	int
%f	float
%c	char
%s	string

# Les conditions

## If

```
int niveau = 18;

if (niveau < 5 || niveau == 18)
{
    printf("Yop");
}
```

## If / Else

```
int niveau = 18;
if (niveau < 5 || niveau == 18)
{
    printf("Yop");
}
else
{
    printf("Hoho");
}
```

## Switch

```
int age = 18;
switch(age)
{
case 18:
    printf("Tu as la majorité !");
}
```

```
    break;
case 100:
    printf("Tu es centenaire !");
    break;
default:
    break;
}
```

## Ternaire

```
int age = 16;
(age == 15) ? printf("tu as 15 ans") : printf("Interdit pour toi");
```

# Les boucles

## While

```
int i = 0;
while (i<10){
    printf("Waou !\n");
    i++;
}
```

## For

```
int i;
for (i = 0 ; i<5 ; i++){
    printf("Ce message est affiché 5 fois\n");
}
```

- Parcourir un tableau :

```
int monTableau[5];
for (int i = 0 ; i < 5 ; i++)
{
    monTableau[i] = i*3;
}
```

# Tableaux

Les tableaux en langage C ne sont capables de stocker qu'un seul type de valeur et ont une longueur prédéfinie.

Par exemple, voici la déclaration d'un tableau de 5 entiers :

```
int mon_tableau[5] = {3, 5, 9, -6, -2};
```

On peut parcourir chaque éléments du tableau avec des boucles :

```
for (int i = 0 ; i < 5 ; i++)  
{  
    mon_tableau[i] = i*3;  
}
```

Remarque : La mention de l'identifiant d'un tableau dans le code fera référence à l'adresse du premier élément de celui-ci.

On peut aussi déclarer des tableaux double dimensions :

```
int tableau_deux_dimensions[5][3] = {  
    {4, 6, -1},  
    {-2, 6, 7},  
    {8, 5, 0},  
    {4, 3, 6},  
    {7, -9, 2}  
};
```

Pour parcourir un tableau à deux dimensions et afficher les valeurs contenues :

```
for (int i = 0 ; i < 5 ; i++)  
{  
    for (int j = 0 ; j < 3 ; j++){  
        printf("[%d]", tableau_deux_dimension[i][j]);  
    }  
    printf("\n");  
}
```



# Pointeurs

Un pointeur est une variable qui stocke une adresse mémoire.

## Déclaration

```
int nb1 = 5;  
int *pt_nb1 = &nb1;
```

## Utilisation

- Le caractère **&** placé devant l'identifiant de la variable fait référence à l'adresse de celle-ci.
- Le caractère **\*** placé devant l'identifiant de la variable fait référence à la valeur stockée dans la variable derrière le pointeur.

```
void exchange(float*nombreA, float*nombreB)  
{  
    float tmp = *nombreB;  
    *nombreB = *nombreA;  
    *nombreA = tmp;  
}  
  
int main(){  
  
    float nombreA = 6.8;  
    float nombreB = 9.45;  
  
    printf("A = %.2f | B = %.2f\n", nombreA, nombreB);  
    exchange(&nombreA, &nombreB);  
    printf("A = %.2f | B = %.2f", nombreA, nombreB);  
  
    return 0;  
}
```

## Malloc

La fonction **malloc()** permet d'allouer de l'espace mémoire pour une variable.

Elle prend en paramètre la taille désirée du bloc mémoire en octet et renvoie l'adresse de ce bloc mémoire.

## Free

La fonction **free()** permet de libérer l'espace mémoire pris par une variable.

Elle prend en paramètre l'adresse du bloc mémoire de la variable à libérer et retourne *NULL*.

# Structures

Les structures sont des variables complexes dont les caractéristiques sont fixées par le développeur.

Elles sont l'ancêtre de l'objet et présente donc de grandes similitudes.

Voici la syntaxe de la structure :

```
struct <LABEL> {  
    <TYPE_1> <CHAMP_1>;  
    <TYPE_2> <CHAMP_2>;  
    <TYPE_3> <CHAMP_3>;  
} <INSTANCE_1>, <INSTANCE_2>;
```

Remarque : On peut aussi placer le mot clé **typedef** devant la structure pour définir un nouveau type de variable qui sera reconnu par le compilateur.

On peut accéder à un champs de la manière suivante :

```
struct Player {  
    char username[12];  
    int hp;  
    float mana;  
} elieroc;  
  
elieroc.mana = 4;
```

Et pour accéder aux pointeurs, c'est le même procédé mais il faut remplacer le "." par une flèche "->".

Voici un petit jeu montrant l'intérêt et un exemple d'utilisation des structures :

```

#include <stdio.h>
#include <string.h>
#include "main_header.h"

typedef struct Player {
    char username[12];
    int hp;
    float mana;
    unsigned int alive : 1; // Le ": 1" permet de spécifier que la variable alive n'est codé que sur 1 bit.
} Player;

void change_username(Player *p){
    printf("Saisir le nouveau pseudo : ");
    scanf("%s", &p->username);
}

void heal(Player *p, int number){
    p->hp+=number;
}

void damage(Player *p, int number){
    p->hp-=number;
    if (p->hp <= 0){
        p->alive = 0;
    }
}

int main(){

    Player Elieroc = {"Elieroc", 150, 200, 1};

    change_username(&Elieroc);
    printf("Un ange vous a miraculeusement soigné !\n");
    heal(&Elieroc, 20);
    printf("Un ennemi vous attaque !\n");

    damage(&Elieroc, 160);
    if (Elieroc.alive == 0){
        printf("On a un homme à terre !\n");
    }
}

```

```
}  
else{  
    printf("Battez vous soldat !\n");  
}  
  
printf("Player name : %s\n", Elieroc.username);  
printf("HP : %d && Mana : %.0f\n", Elieroc.hp, Elieroc.mana);  
  
return 0;  
}
```

# Énumérations

Les énumérations ont pour particularité d'être descriptives.

Elles servent généralement à décrire des états.

Dans l'exemple suivant, nous souhaitons mettre en place des flags pour décrire les états possibles d'une application en adossant à chaque état une valeur binaire unique :

```
typedef enum e_appFlagsMasks{  
    [ST_APP_ALL_CLEARED]= 0x00000000,  
    [ST_APP_INIT_FAILED]= 0x00000001,  
    [ST_APP_SDL_INITIATED]= 0x00000002,  
    [ST_APP_RUNNING]= 0x00000004,  
    [ST_APP_REFRESH_ASKED]= 0x00000008,  
    [ST_APP_PAUSED]= 0x00000010  
}t_appFlagsMasks;
```

Remarques : Nous pourrions faire avec des variables simples mais l'intérêt ici est d'économiser de la mémoire en stockant tous ces états sur un seul octet.

# Masques

Il est possible d'appliquer des masques pour tester l'état d'un flag.

On le fait avec des masques qui appliquent des opérations logiques :

```
#define mBitsSet(f,m) ((f)|=(m))
#define mBitsClr(f,m) ((f)&=~(m))
#define mBitsTgl(f,m) ((f)^=(m))
#define mBitsMsk(f,m) ((f)&(m))
#define mlsBitsSet(f,m) ((f)&(m))
#define mlsBitsClr(f,m) (((~(f))&(m))&(m))
```

Ces masques prennent la syntaxe d'une fonction où on donne en premier "paramètre" la variable et en second l'état à tester :

```
mBitsSet(pApp->uStatus,ST_APP_INIT_FAILED);
```

---

Revision #17

Created 27 July 2023 09:41:12 by Elieroc

Updated 27 December 2023 08:59:26 by Elieroc