Encadré par

M.Elyes OUTAY et M.Mehdi HANZOUTI

DOCUMENTATION C /GIT

Elaboré par

Ahmed MEMNI

Memniahmed20@gmail**.com**

June 2024

Introduction

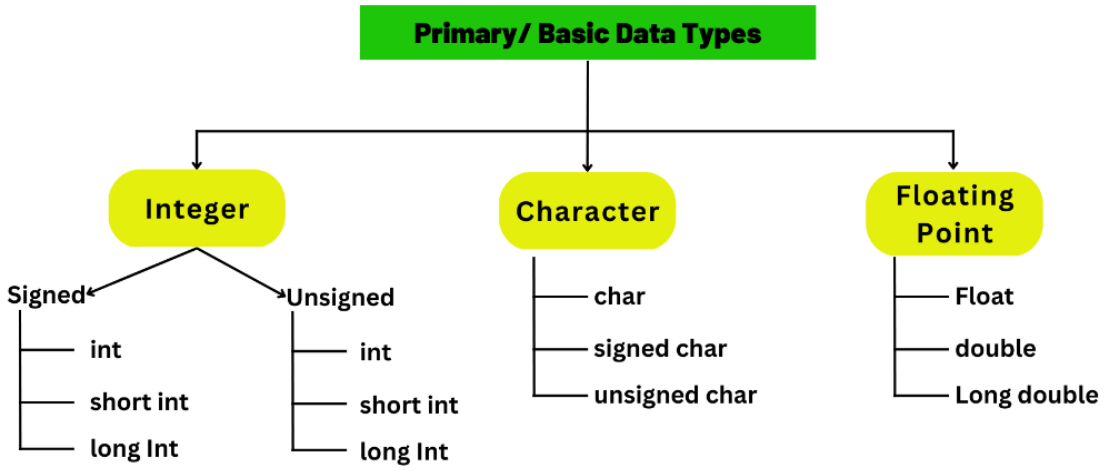
Au cours de mon parcours dans le monde de la programmation, j'ai eu l'opportunité de me plonger profondément dans le langage de programmation C, spécifiquement adapté aux systèmes embarqués. Le C embarqué est une compétence cruciale pour développer des logiciels qui interagissent directement avec le matériel, souvent avec des contraintes de ressources strictes. Contrairement à la programmation en C standard, le C embarqué nécessite une solide compréhension du matériel, des environnements d'exploitation en temps réel et des pratiques de codage efficaces pour répondre aux exigences des applications embarquées.

Ce rapport récapitule mon parcours d'apprentissage, mettant en lumière les concepts fondamentaux du C embarqué En maîtrisant ces techniques, je suis désormais mieux équipé pour contribuer au développement de systèmes embarqués sophistiqués qui forment l'épine dorsale de la technologie moderne.

DATA TYPES IN C :

En C, les types de données sont catégorisés en trois groupes principaux : les types primitifs, les types dérivés et les types définis par l'utilisateur. Chaque catégorie sert à des fins spécifiques et offre différentes capacités pour manipuler et stocker des données.

PRIMARY DATA TYPES :

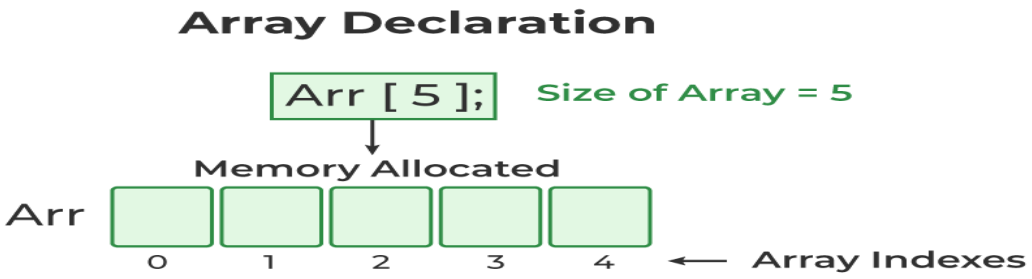


Le langage C offre une variété de types de données primaires, chacun servant à des fins différentes et offrant différentes plages de valeurs et tailles de stockage. Ces types de données primaires constituent les éléments de base pour des structures de données et des opérations plus complexes en programmation C. Ils peuvent être catégorisés en types signés et non signés, selon qu'ils peuvent représenter des valeurs négatives ou non. Voici un tableau récapitulatif des types de données primaires en C, incluant leurs tailles et leurs plages de valeurs.

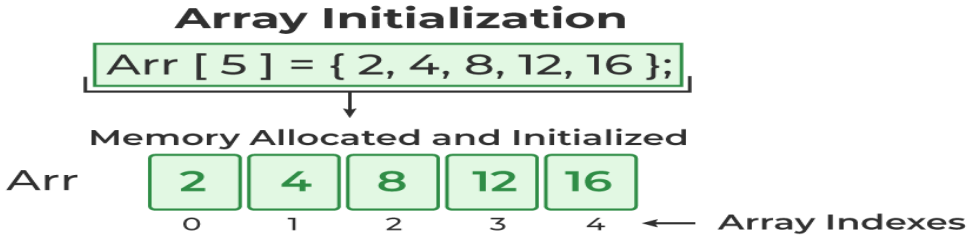
Data Types	Memory Size	Range	Format Specifier
char	1 byte	-128 to 127	%c
signed char	1 byte	-128 to 127	%c
unsigned char	1 byte	0 to 255	%c
int	2 byte	-32,768 to 32,767	%d, %i
signed int	2 byte	-32,768 to 32,767	%d, %i
unsigned int	2 byte	0 to 65,535	%u
short int	2 byte	-32,768 to 32,767	%hd
signed short int	2 byte	-32,768 to 32,767	%hd
unsigned short int	2 byte	0 to 65,535	%hu
long int	4 byte	-2,147,483,648 to 2,147,483,647	%ld, %li
signed long int	4 byte	-2,147,483,648 to 2,147,483,647	%lld
long long int	8 byte	-(2^63) to (2^63)-1	%ld, %li
unsigned long long int	8 byte	0 to 18,446,744,073,709,551,615	%llu
unsigned long int	4 byte	0 to 4,294,967,295	%lu
float	4 byte	1.2E-38 to 3.4E+38	%f
double	8 byte	1.7E-308 to 1.7E+308	%lf
long double	10 byte	3.4E-4932 to 1.1E+4932	%Lf

DERIVED DATA TYPES :

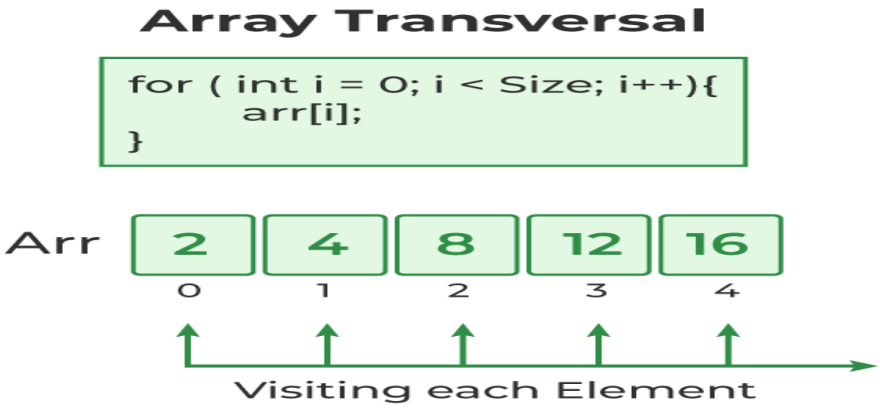
Un tableau en C est une collection contiguë de variables du même type, indexée par un nombre entier.



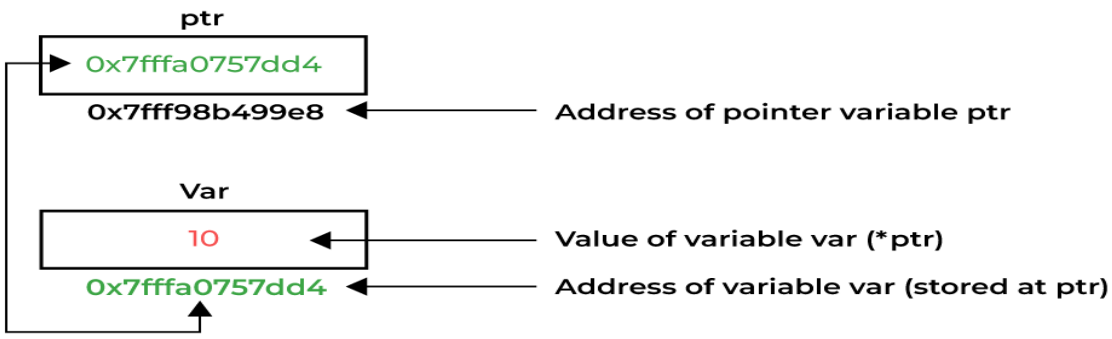
Les tableaux peuvent être initialisés lors de leur déclaration ou après.



Les tableaux peuvent être parcourus à l'aide de boucles, comme `for`



Les pointeurs sont l'un des composants essentiels du langage de programmation C. Un pointeur peut être utilisé pour stocker l'adresse mémoire d'autres variables, fonctions, ou même d'autres pointeurs. L'utilisation des pointeurs permet l'accès bas niveau à la mémoire, l'allocation dynamique de mémoire, et de nombreuses autres fonctionnalités en C.



Déclarer un pointeur en C en spécifiant le type de données qu'il pointe, suivi d'une `*` Un pointeur peut être initialisé avec l'adresse d'une variable existante à l'aide de l'opérateur d'adresse `&`.

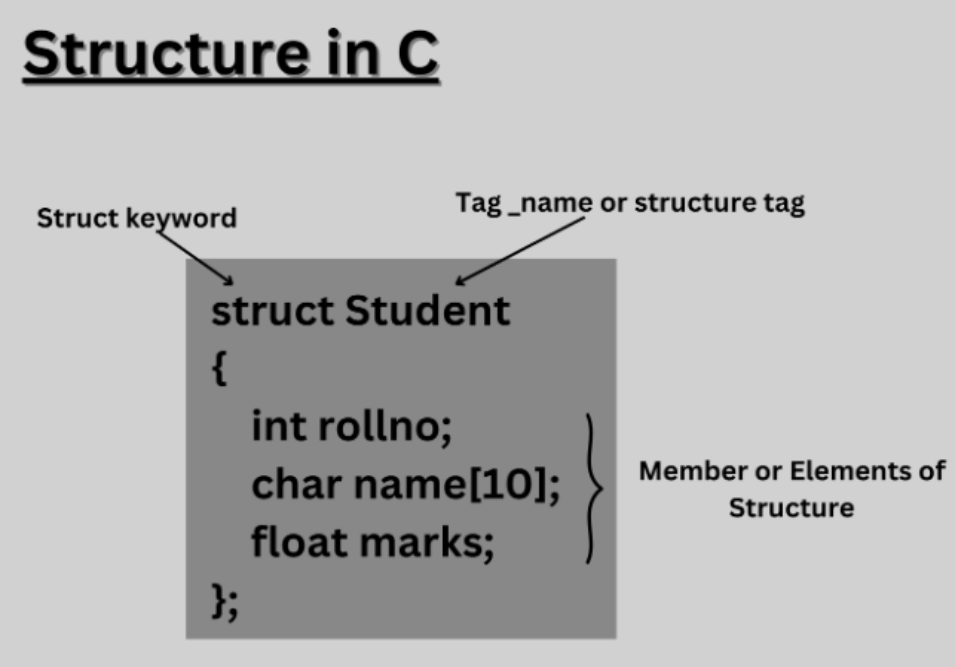
```
int var = 10;
int * ptr;
ptr = &var;
```

USER DEFINED TYPES :

STRUCTURE:

En C, les structures (struct) sont des types de données qui permettent de regrouper plusieurs variables de types différents sous un seul nom. Elles sont très utiles pour organiser et manipuler des données complexes de manière structurée.

Pour définir une structure en C, utilisez le mot-clé `struct` suivi du nom de la structure et des membres qu'elle va contenir :



Une fois définie, une structure peut être utilisée pour déclarer des variables de ce type

```
int main()

{
struct Student st1, st2;
}
```

```
{
struct Student st1, st2;
st1.rollno = 7
strcpy(st1.name,"Akash");
st1.marks = 50
}
```

UNIONS

les unions (ou `union` en anglais) sont similaires aux structures (`struct`), mais avec une différence fondamentale : elles permettent de stocker différents types de données dans la même zone mémoire. Contrairement aux structures où chaque membre a sa propre zone mémoire distincte, tous les membres d'une union partagent la même zone mémoire

```
union union_name {
    datatype member1;
    datatype member2;
    ...
} var1, var2, ...;
```

Pour accéder aux membres d'une union, utilisez l'opérateur `.` (point) comme avec les structures :
La taille d'une union est déterminée par la taille de son membre le plus grand
Les unions sont utiles lorsque vous devez stocker différents types de données dans une même zone mémoire, mais où seul un type de données est utilisé à la fois

GLOBAL VARIABLE :

une variable globale est une variable définie en dehors de toutes les fonctions, ce qui lui confère une portée qui s'étend à l'ensemble du fichier source dans lequel elle est déclarée.

Pour définir une variable globale en C, vous la déclarez en dehors de toutes les fonctions, généralement au début du fichier source ou dans un fichier d'en-tête inclus par plusieurs fichiers source

```c
#include <stdio.h>

int globalVar = 100;

int main() {
    printf("Valeur de globalVar : %d\n", globalVar);

    return 0;
}
```

Une variable globale est visible et accessible depuis n'importe quelle fonction définie dans le même fichier source
La durée de vie d'une variable globale est tout au long de l'exécution du programme.

LOCAL VARIABLE :

une variable locale est une variable déclarée à l'intérieur d'une fonction ou d'un bloc spécifique, limitant sa portée à ce contexte particulier

```c
#include <stdio.h>

int main() {
    int localVar = 50;
    printf("Valeur de localVar : %d\n", localVar);
    return 0;
}
```

Une variable locale est accessible uniquement à l'intérieur de la fonction ou du bloc dans lequel elle est déclarée

La durée de vie d'une variable locale commence lorsque le programme entre dans le bloc où elle est déclarée et se termine lorsque le programme quitte ce bloc. Cela signifie que la mémoire allouée pour la variable est libérée à la sortie du bloc.

Les variables locales peuvent être initialisées lors de leur déclaration ou ultérieurement dans le bloc où elles sont définies.

BITWISE OPERATORS :

En C, les opérateurs bit à bit (bitwise operators) permettent de manipuler les bits individuels des variables entières. Voici les principaux opérateurs bit à bit disponibles :

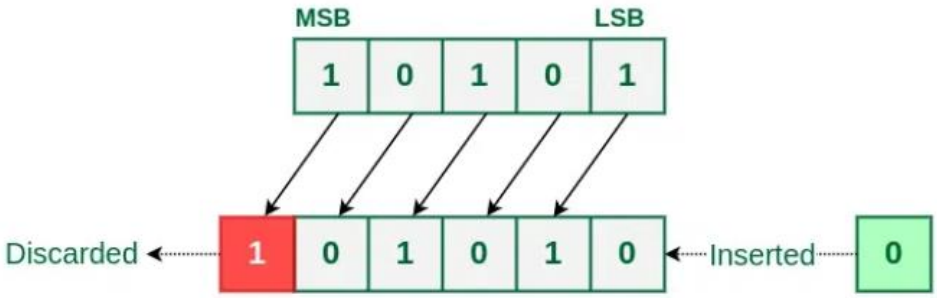
Operator	Name	Example	Result
&	Bitwise AND	6 & 3	2
|	Bitwise OR	10 | 10	10
^	Bitwise XOR	2^2	0
~	Bitwise 1's complement	~9	-10
<<	Left-Shift	10<<2	40
>>	Right-Shift	10>>2	2

1. ET Bit à Bit (`&`)** :
  - Syntaxe : `a & b`
  - Description : Effectue un ET logique bit à bit entre les bits de `a` et `b`. Le résultat est `1` uniquement si les deux bits correspondants sont `1`.

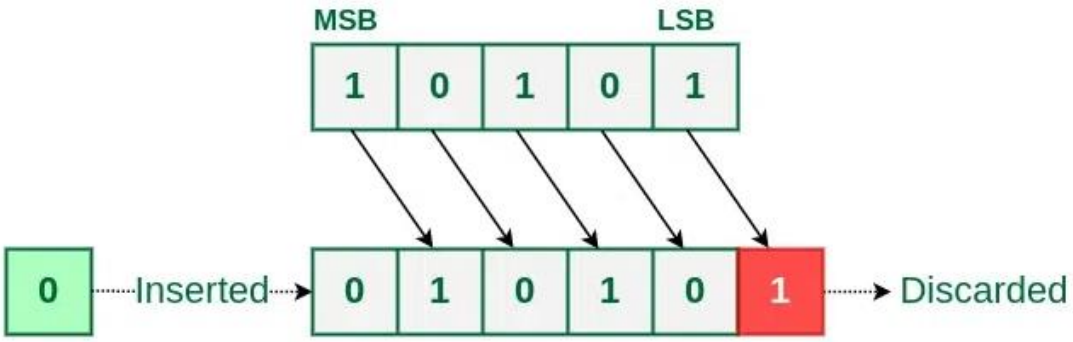
2. OU Bit à Bit (`|`)** :
  - Syntaxe : `a | b`
  - Description : Effectue un OU logique bit à bit entre les bits de `a` et `b`. Le résultat est `1` si au moins l'un des bits correspondants est `1`.

3. OU Exclusif (`^`)** :
  - Syntaxe : `a ^ b`
  - Description : Effectue un OU exclusif bit à bit entre les bits de `a` et `b`. Le résultat est `1` si exactement l'un des bits correspondants est `1`, mais pas les deux.

4. Complément à Un (`~`)** :
  - Syntaxe : `~a`
  - Description : Inverse tous les bits de `a`. Chaque `0` devient `1` et chaque `1` devient `0`.



Left Shift Operator



Right Shift Operator

CONST USAGES :

le mot-clé `const` est utilisé pour déclarer des constantes, c'est-à-dire des variables dont la valeur ne peut pas être modifiée une fois qu'elle a été initialisée

pointer modifiable d un variable constant

```
uint8_t const  *pData = (uint8_t*) 0x40000000;
```

pointeur constant d un variable modifiable

```
uint8_t *const pData = (uint8_t*) 0x40000000;
```

PRE-PROCESSOR DIRECTIVES :

Conditional Compilation:

Ces directives permettent d'inclure ou d'exclure des parties du code source basé sur des conditions prédéfinies lors de la compilation

```
#ifdef NEW_FEATURE
//Code block of new feature
#else
//code block for old feature
#endif
```

```
#ifdef  NEW_FEATURE
//Code block
#endif
```

```
#if 0

//code block

#endif
```

#define:

La directive de préprocesseur `#define` est utilisée pour définir des macros. Les macros sont des identificateurs définis par `#define` qui sont remplacés par leur valeur avant la compilation. Cela signifie que toutes les occurrences de l'identificateur seront remplacées par la valeur définie lors de la phase de prétraitement du code source.

On peut utiliser `#define` pour définir des constantes. Par exemple :

```
#define PI 3.14159
```

On peut également utiliser `#define` pour définir des fonctions comme des macros. Par exemple :

```
#define SQUARE(x) ((x) * (x))
```

Autre directives:

```
#error "No macros defined."
```

```
#warning "No macros defined."
```

GIT:

Git est un système de contrôle de version puissant, largement utilisé pour suivre les modifications du code source lors du développement de logiciels. Créé par Linus Torvalds en 2005, Git est devenu un outil essentiel pour les développeurs du monde entier. Comprendre Git peut considérablement améliorer votre efficacité en matière de codage et de collaboration.

Concept de base :

1. Dépôt (Repository):

Un dépôt Git est un espace de stockage pour votre projet, où Git garde une trace de toutes les modifications apportées aux fichiers au fil du temps.

2. Commit:

Un commit représente un instantané de votre projet à un moment donné. Chaque commit a un identifiant unique et inclut un message de commit décrivant les modifications.

3. Branche (Branch);

Les branches sont des chemins distincts de développement dans un projet. La branche principale par défaut est souvent appelée `master` ou `main`.

4. Fusion (Merge):

La fusion est le processus de combinaison de deux branches. Cela intègre les modifications d'une branche dans une autre.

5. Dépôt Distant (Remote):

Un dépôt distant est une version de votre dépôt stockée sur un serveur, permettant de collaborer avec d'autres développeurs.

6. Cloning:

Le clonage est une opération Git qui vous permet de copier un dépôt Git existant (qui peut être local ou distant) sur votre machine. Cette commande crée un nouveau répertoire contenant l'intégralité de l'historique du dépôt et tous ses fichiers à la version la plus récente. Voici une explication détaillée de la commande `git clone` et de son utilisation.

Commandes de Base de Git:

Configuration Initiale

```
memni@LAPTOP-AMOJIE2O MINGW64 ~
$ git config --global user.name "ahmed"

memni@LAPTOP-AMOJIE2O MINGW64 ~
$ git config --global user.email "memniahmed2O@gmail.com"
```

Initialisation d un repository

```
$ git init "stageete"
Initialized empty Git repository in C:/Users/memni/stageete/.git/
```

Pour cloner un dépôt existant depuis une URL

```
memni@LAPTOP-AMOJIE2O MINGW64 ~
$ git clone https://github.com/HASamer/gestionmaison.git
Cloning into 'gestionmaison'...
remote: Enumerating objects: 289, done.
remote: Counting objects: 100% (289/289), done.
remote: Compressing objects: 100% (146/146), done.
remote: Total 289 (delta 202), reused 224 (delta 137), pack-reused 0
Receiving objects: 100% (289/289), 985.90 KiB | 947.00 KiB/s, done.
Resolving deltas: 100% (202/202), done.
```

Ouvrir un Dépôt Local Git

```
cd chemin/vers/mon-projet
```

Vérifier l'état des fichiers :

```
memni@LAPTOP-AMOJIE2O MINGW64 ~/stageete (master)
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Ajouter des fichiers à l'index

```
git add fichier.txt
```

Afficher le histoire des modification

```
git log
```

Basculer vers la Branche Spécifique

```
memni@LAPTOP-AMOJIE2O MINGW64 ~/stageete (master)
$ git checkout 13f2d9afcbd4901d87bc251ee8ae6ed5034845a0
Note: switching to '13f2d9afcbd4901d87bc251ee8ae6ed5034845a0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.
```

Pour enregistrer les modifications ajoutées à l'index avec un message

```
memni@LAPTOP-AMOJIE2O MINGW64 ~/Documents/GitHub/gestionmaison (main)
$ git commit -m"enregistrement"
[main d9acfe2] enregistrement
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 js.css
```

Pour créer une nouvelle branche :

```
memni@LAPTOP-AMOJIE2O MINGW64 ~/Documents/GitHub/gestionmaison (main)
$ git branch new

memni@LAPTOP-AMOJIE2O MINGW64 ~/Documents/GitHub/gestionmaison (main)
$ git branch
* main
  new
```

Pour basculer vers une branche existante :

```
memni@LAPTOP-AMOJIE2O MINGW64 ~/Documents/GitHub/gestionmaison (main)
$ git checkout new
Switched to branch 'new'
```

Pour fusionner une branche dans la branche courante :

```
memni@LAPTOP-AMOJIE2O MINGW64 ~/stageete (master)
$ git merge me
Already up to date.
```

pour envoyer les modifications de votre dépôt local vers un dépôt distant.

`<remote>` : Le nom du dépôt distant (par défaut, `origin`).
`<branch>` : Le nom de la branche distante où vous voulez pousser vos modifications.

```
git push <remote> <branch>
```