```
// cette fonction affiche une zone mémoire à l'écran :
// 3 colonnes, séparées par un espace :
// - l'adresse en hexadécimal du premier caractère de la ligne (par ft_print_hex_addr_first_char)
// - le contenu en hexadécimal avec un espace tous les 2 caractères, complété par des espaces
// si nécessaire (par ft_print_str_hexa_content et ft_print_hex_char)
// - le contenu en caractères imprimable (ft_print_str_printable_content)
// les caractères non imprimables sont remplacés par des points "."
// chaque ligne doit gérer 16 caractères (voir ft_print_memory)
// si la size est égale à 0, rien ne sera affiché

#include <unistd.h>

void    ft_print_hex_addr_first_char(void *current_addr);
void    ft_print_str_hexa_content(unsigned char *current_addr, unsigned int current_size);
void    ft_print_hex_char(unsigned char c);
void    ft_print_str_printable_content(unsigned char *current_addr, unsigned int current_size);

// cette fonction gère l'affichage des 3 colonnes par bloc de 16 caractères
// ainsi que l'espacement entre les différentes colonnes
// (": " après la première colonne, " " après la deuxième et "\n" après la troisième)
// elle prend en paramètre un pointeur vers le début du bloc mémoire à afficher, addr
// et le nombre d'octets à afficher, size (de type unsigned int,
// pour s'assurer qu'elle soit toujours positive)
// elle renvoie le pointeur addr passé en paramètre (sans modification)

// REMARQUE : addr est un pointeur générique (de type void)
// cela permet à la fonction de fonctionner avec des données de n'importe que type
// (pas seulement des chaines de caractères,
// mais aussi des tableaux d'entiers ou de flottants par exemple),
// et de pouvoir traiter les données différemment

// il faudra convertir ce pointeur en un type utilisable
// pour sa manipulation (en type numérique pour des opérations mathématiques
// (arithmétique des pointeurs ou traitement numérique des données),
// en type char pour des opérations sur les caractères ...)
// (voir plus bas)

void    *ft_print_memory(void *addr, unsigned int size)
```

```
{
        // offset, de type unsigned int, sert de compteur pour parcourir le bloc mémoire
        // il correspond au nombre d'octets déjà affichés
        // il sera initialisé à 0, puis sera incrémenté de 16 à la fin de la chaque boucle
        // (car on souhaite afficher 16 octets sur chaque ligne)
        // ainsi, la lecture du bloc à partir de l'adresse fournie se fera
        // à partir de l'indice 0, puis 16
        // etc

        unsigned int    offset;

        // on déclare un pointeur de type unsigned char *
        // ce pointeur stockera addr sous la forme d'un pointeur d'unsigned char
        // (et non pas un pointeur void *), pour pouvoir manipuler chaque octet individuellement
        // comme des caractères
        // cela permet au compilateur d'appliquer correctement l'arithmétique des pointeurs
        // comme le type void * n'a pas de taille définie, l'arithmétique des pointeurs est
        // incorrecte sans cette conversion :
        // addr + offset n'a pas de sens
        // en effet, l'ajout d'un unsigned int i à une variable de type pointeur ad
        // signifie de se déplacer, à partir de l'adresse pointée par ad, de i fois la taille
        // du type d'élément sur lequel ad pointe
        // or, addr est de type void *
        // et la taille d'un void n'est pas définie
        // alors que la taille d'un unsigned int est de 1 octet
        // avec ptr + offset, et offset a 16 par exemple :
        // on se déplacera de 16 * 1 octets (donc 16 octets)
        // on pourra ainsi se déplacer caractère par caractère
        // (ici, de 16 caractères en 16 caractères)

        unsigned char *ptr;

        // offset est initialisé à 0 pour que le traitement et l'affichage du bloc mémoire
        // démarre à l'indice 0 de l'adresse fournie
        offset = 0;

        // on convertit addr en unsigned char *
        // voir plus haut
```

```
ptr = (unsigned char *)addr;


// ATTENTION : lors du passage de ptr en paramètre de la fonction
// ft_print_hex_addr_first_char, celui-ci sera converti à nouveau en void *
// cela est nécessaire pour convertir à nouveau l'adresse
// (en unsigned long long, voir plus bas) pour la manipuler correctement
// ft_print_str_hexa_content et ft_print_str_printable_content doivent manipuler
// l'adresse en unsigned char *, donc cette conversion ici (dans ft_print_memory)
// permet d'éviter la conversion dans ces fonctions

// ATTENTION : offset est initialisé à 0
// avec la condition "while (offset < size)" et une size de 92 par exemple
// la boucle sera donc parcourue exactement 92 fois
while (offset < size)
{
        // on imprime l'adresse en hexadécimal du premier caractère de la ligne en cours
        // en passant à la fonction le point de départ du bloc mémoire
        ft_print_hex_addr_first_char(ptr + offset);

        // on sépare cette colonne de la suivante par ": "
        write(1, ": ", 2);

        // on imprime le contenu en hexadécimal de la ligne en cours
        // en passant à la fonction le point de départ du bloc mémoire
        // et la taille restante (pour empêcher à la fonction d'aller
        // au-delà de la mémoire fournie, dans le cas où il reste moins de 16 caractères)
        // car cette fonction affiche 16 caractères ou ce qu'il reste à afficher
        // selon le plus petit des deux
        // (voir plus bas)
        ft_print_str_hexa_content(ptr + offset, size - offset);

        // on sépare cette colonne de la suivante par " "
        write(1, " ", 1);

        // on imprime le contenu en caractères imprimable de la ligne en cours
        // en passant à la fonction le point de départ du bloc mémoire
        // et la taille restante (pour empêcher à la fonction d'aller
```

```
                // au-delà de la mémoire fournie, dans le cas où il reste moins de 16 caractères)
                // car cette fonction affiche 16 caractères ou ce qu'il reste à afficher
                // selon le plus petit des deux
                // (voir plus bas)
                ft_print_str_printable_content(ptr + offset, size - offset);

                // on passe à la ligne suivante (en retournant à la ligne)
                write(1, "\n", 1);

                // on incrémente l'offset de 16
                offset += 16;
        }
        // on retourne l'adresse du bloc mémoire
        return (addr);
}

// cette fonction affiche l'adresse en hexadécimal du premier caractère
// (autrement dit, du premier octet) de la ligne
// elle prend en paramètre l'adresse de ce caractère
// EN LE CONVERTISSANT EN VOID *
void    ft_print_hex_addr_first_char(void *current_addr)
{
        // ATTENTION : la valeur d'une adresse de type void * n'est pas directement accessible
        // il faut tout d'abord la convertir dans un type entier capable de contenir une adresse
        // les opérations pour en extraire les bits et les convertir en hexadécimal ne seraient
        // pas possible autrement
        // (en interne, cette adresse est stockée sous forme de bits)
        // il faut manipuler numériquement la valeur de cette adresse pour la convertir
        // en hexadécimal :


        // on déclare une variable de type unsigned long long
        // ce pointeur stockera current_addr sous la forme d'un unsigned long long
        // (et non pas un pointeur de type void *)
        // ce qui permet de s'assurer que l'adresse sera manipulée comme une valeur numérique
        // (pour pouvoir extraire, octet par octet, l'adresse mémoire du premier caractère,
        // puis les afficher à l'écran (après leur conversion en hexadécimal)
```

```
// REMARQUE :
// les octets sont des valeurs entières de 8 bits, entre 0 et 255
// en utilisant un type unsigned long, on dispose d'une assez grande plage de valeurs
// pour stocker et manipuler les octets extraits de l'adresse sur un système 32 bits :

// sur un système 32 bits, une adresse mémoire est représentée par 32 bits
// la valeur maximale (en unsigned) pour une adresse est donc
// 1111 1111 1111 1111 1111 1111 1111 1111
// (en binaire) (32 = 8 * 4)
// cela correspond à 4 294 967 295 (en décimal)
// la valeur maximale d'un unsigned long est aussi égale à 4 294 967 295

// ATTENTION : pour rendre la fonction utilisable sur un système 64 bits,
// il faut convertir en unsigned long long
// la même logique s'applique : la valeur maximale pour une adresse est
// 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
// (en binaire) (64 = 16 * 4)
// cela correspond à 18 446 744 073 709 551 615 (en décimal)
// la valeur maximale d'un unsigned long long est aussi égale à 18 446 744 073 709 551 615

// donc, pour pouvoir afficher toutes les adresses possibles en hexadécimal
// sur les systèmes 32 bit et 64 bits
// il faut convertir cette adresse en unsigned long long

// cette conversion permettra d'effectuer des opérations de décalage
// et des masques bit à bit
// pour extraire chaque octet (groupe de 4 bits) de l'adresse
// (voir plus bas) pour les passer à ft_print_hex_char (voir plus bas) (voir ex11)
unsigned long long      addr_first_char;

// shift_value sera initialisé pour pointer sur le premier octet à extraire
// (le plus significatif)
// il sera décrémenté pour permettre d'accéder à l'octet suivant
// (voir plus bas)
int             shift_value;

// current_byte contiendra l'octet courant, après avoir effectué le décalage de bits
// et appliqué le masque pour l'extraire
```

```
// (voir plus bas)
unsigned char   current_byte;

// on convertit current_addr en unsigned long long (voir plus haut)
addr_first_char = (unsigned long long)current_addr;

// exemple :
// 214 748 212
// 0000 0000 0000 0000 0000 0000 0000 0000 0000 1100 1100 1100 1100 1100 0011 0100
// 0    0    0    0    0    0    0    0    0    12   12   12   12   12   3    4
// 0    0    0    0    0    0    0    0    0    C    C    C    C    C    3    4

// pour convertir cette valeur de décimal en hexadécimal, il faut extraire chaque nibble
// (groupe de 8 bits, 2 * 4 bits) en partant du plus significatif (le plus à gauche)
// jusqu'au moins significatif (le plus à droite)

// pour cela, on peut effectuer un décalage de bits puis extraire chacun de ceux-ci

// on calcule le nombre de bits nécessaires pour positionner le curseur
// sur le premier octet significatif (tout à gauche)

// l'opérateur sizeof renvoie la taille en octets du type de la variable donnée
// ici, on calcule la taille du type d'addr_first_char,
// donc la taille d'un unsigned long long
// en général, sur les systèmes 32 bits et 64 bits,
// la taille d'un unsigned long long est de 8 octets (8 * 8 bits),
// mais cela peut dépendre du système
// sizeof(addr_first_char) correspond donc au nombre d'octets
// que prend la variable
// cela correspond à 8 octets

// chaque octet contient lui-même 8 bits
// on multiplie donc le nombre d'octets que prend addr_first_char
// par 8 pour obtenir
// le nombre de bits qu'elle contient au total
// 8 * 8 = 64
// la variable prend donc 64 bits (voir exemple)
```

```
// REMARQUE : le type unsigned long long est toujours représenté par 8 octets !
// même si l'on est sur un système 32 bits, et que l'adresse donc prend 32 bits au maximum
// la variable qui stocke cette adresse, addr_first_char, prend 64 bits en mémoire !
// (elle sera alors complétée par 32 bits de valeur 0 à gauche,
// pour une adresse de 32 bits) (voir exemple plus haut)

// on soustrait 8 à 64, car on veut extraire 1 octet de cette adresse (et 1 octet = 8 bits)

// ainsi, le décalage à effectuer sera de 56 bits (voir plus bas)

shift_value = (sizeof(addr_first_char) * 8) - 8;

// tant que tous les octets n'ont pas été traités
// (le décalage initial est de 56 bits, puis il est réduit de 8 bits à chaque itération)
// il y aura 8 décalages au total pour parcourir chaque octet de la variable de 64 bits
// (au bout du 7éme décalage, shift_value vaudra 0 (7 * -8 = -56 => 56 - 56 = 0),
// une dernière itération sera effectuée avant que shift_value ne soit plus supérieur ou
// égal à 0 (il sera alors égal à -8))
while (shift_value >= 0)
{
        // on décale addr_first_char de 56 bits vers la droite
        // avec l'opérateur >> (opérateur de décalage de bits à droite)
        // (un décalage de n bits d'un nombre (en binaire) ajoute n bits de valeur 0
        // à gauche de ce nombre)

        // ainsi, lors de la première itération de la boucle,
        // les 8 bits les plus significatifs (les plus à gauche)
        // deviendront les 8 bits les moins significatifs (les plus à droite) :
        // 0000 0000 0000 0000 0000 0000 0000 0000 0000 1100 1100 1100 1100 1100 0011 0100
        // ____ ____

        // => 56

        // 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
        //                                                                   ____ ____

        // puis  :
```

```
// 0000 0000 0000 0000 0000 0000 0000 0000 0000 1100 1100 1100 1100 1100 0011 0100
// ____ ____

// => 48

// 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
//                                                                 ____ ____


// REMARQUE : on part toujours du début de l'adresse, mais le nombre de bits
// décalés diminue chaque itération

// puis on extrait les 8 derniers bits à droite (après le décalage)
// pour cela, on utilise l'instruction & (0xFF) :

// & est l'opérateur binaire ET (bit à bit)
// il retourne le résultat de l'opération AND sur les bits correspondants
// des 2 opérandes (addr_first_char >> shift_value) et (0xFF)
// 0 & 0 => 0
// 1 & 0 => 0
// 0 & 1 => 0
// 1 & 1 => 1

// exemple :
// 1100 & 1010 :

// 1100
// &&&&
// 1010
// ----
// 1000

// 0xFF correspond à 1111 1111 en binaire
// car :
// 0x signifie que le nombre est exprimé en notation hexadécimale (en base 16)
// F vaut 15 en hexadécimal
// donc 1111 en binaire
// donc 0xFF vaut 1111 1111 en binaire
```

```
                // (0xFF) est appelé un masque de bits : il permet d'extraire des bits
                // d'une valeur donnée :

                // exemple : résultat de (addr_first_char >> shift_value) & (0xFF)
                // pour (addr_first_char >> shift_value) valant
                // 0000 0000 0000 0000 0000 0000 0000 0000 0000 1100 1100 1100 1100 1100 0011 0100

                // 0000 0000 0000 0000 0000 0000 0000 0000 0000 1100 1100 1100 1100 1100 0011 0100
                // &&&& &&&& &&&& &&&& &&&& &&&& &&&& &&&& &&&& &&&& &&&& &&&& &&&& &&&& &&&& &&&&
                // 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111
                // ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ---- ----
                // 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011 0100

                // cette opération permet bien d'extraire les 8 derniers bits (le dernier octet)
                current_byte = (addr_first_char >> shift_value) & (0xFF);

                // on passe ces 8 bits à ft_print_hex_char
                // qui convertit un octet en son équivalent hexadécimal
                // en affichant les deux chiffres hexadécimaux correspondant
                // (voir ex11)
                ft_print_hex_char(current_byte);

                // on décrémente shift_value de 8, pour extraire les 8 prochains bits à la prochaine itération
                shift_value -= 8;
        }
}


void    ft_print_str_hexa_content(unsigned char *current_addr, unsigned int current_size)
{
        unsigned int    i;

        i = 0;

        // la boucle sera exécutée 16 fois
        // pour afficher chacun des caractères en hexadécimal
        // (les 16 caractères suivant l'adresse current_addr)
```

```
	while (i < 16)
	{
		// s'il reste des caractères restant à imprimer sur la ligne
		// (si i est encore dans la plage de caractères à imprimer)
		if (i < current_size)

			// on convertit le caractère en hexadécimal pour l'afficher
			// (par son passage à ft_print_hex_char)
			ft_print_hex_char(current_addr[i]);

		// sinon (si i a dépassé current_size)
		// (si il n'y a plus de données à afficher)
		else
			// on écrit 2 espaces
			// (pour maintenir l'alignement et la structure de la ligne)
			// (le contenu en hexadécimal doit être
			// complété avec des espaces si nécessaire)
			write(1, "  ", 2);

		// après chaque 2ème caractère hexadécimal (ou espace)
		// on ajoute un espace
		// (le contenu en hexadécimal doit avoir avec un espace tous les deux caractères)
		// (i % 2 == 1) est vrai lorsque i est impair
		// donc un espace sera ajouté lorsque i vaudra 1, 3, 5, 7, 9, 11, 13 et 15

		if (i % 2 == 1)
			write(1, " ", 1);

		// on incrémente i
		i++;
	}

// affiche un caractère sous sa forme hexadécimale
// est utilisé par ft_print_hex_addr_first_char et ft_print_str_hexa_content
```

```
// (voir ex11)
void    ft_print_hex_char(unsigned char c)
{
        char    *hex;

        hex = "0123456789abcdef";
        write(1, &hex[c / 16], 1);
        write(1, &hex[c % 16], 1);
}

// affiche le contenu en caractères imprimables
// en remplaçant les caractères non imprimables par des "."
// par blocs de 16
// (sauf s'il ne reste plus de caractères à afficher
// (condition vérifiée par i < current_size))
void    ft_print_str_printable_content(unsigned char *current_addr, unsigned int current_size)
{
        unsigned int    i;

        i = 0;

        while (i < 16 && i < current_size)
        {
                if (current_addr[i] >= 32 && current_addr[i] <= 126)
                {
                        write(1, &current_addr[i], 1);
                }
                else
                        write(1, ".", 1);
                i++;
        }
}

// EXEMPLE :
// ATTENTION A PASSER UNE SIZE + 1 (POUR INCLURE \0)

#include "ft_print_memory.h"
#include "ft_strcpy.h"
```

```c
#include "ft_strlen.h"

int     main(void)
{
        char            str[92];
        unsigned int    size;

        ft_strcpy(str, "Bonjour les aminches\t\n\tc  est fou\ttout\tce qu on peut faire
avec\t\n\tprint_memory\n\n\n\tlol\nlol\n ");
        size = ft_strlen(str) + 1;
        ft_print_memory(str, size);
        return (0);
}
```