Fonctions de ma libft

William

25/10/2025

Table des matières

1 Fonctions de la libc

1.1 isalpha

→ Prototype : `int ft_isalpha(int c)`

→ Check whether the character is alphabetic.

```
int ft_isalpha(int c)
{
 if ((c >= 65 && c <= 90) || (c >= 97 && c <= 122))
  return (1);
 return (0);
}
```

1.2 isdigit

→ Prototype : `int ft_isdigit(int c)`

→ Check whether the character is numeric.

```
int ft_isdigit(int c)
{
 if (c >= '0' && c <= '9')
  return (1);
 return (0);
}
```

1.3 isalnum

→ Prototype : `int ft_isalnum(int c)`

→ Check whether the caracter is alphabetic or numeric.

```
int ft_isalnum(int c)
{
 if ((c >= 65 && c <= 90) || (c >= 97 && c <= 122) || (c >= '0' && c <= '9'))
  return (1);
 return (0);
}
```

1.4 isascii

→ Prototype : `int ft_isascii(int c)`

→ Check whether the character is in the ASCII table.

```
int ft_isascii(int c)
{
 if (c >= 0 && c <= 127)
  return (1);
 return (0);
}
```

1.5 isprint

→ Check whether the character is printable.

→ Printable characters in the ASCII table start from 32 to 126.

```
int ft_isprint(int c)
{
 if (c >= 32 && c <= 126)
  return (1);
 return (0);
}
```

1.6 Rappel pour toute les fonctions mémoires

— Dans toute les fonctions mémories, comme memchr ou bzero etc.... Il en faut pas mettrre de garde fou car on considère que les pointeurs passés sont valides.

— Si le man ne précise aucun garde fou on n'en met pas !

1.7 memset

→ Prototype : `void *ft_memset(void *ptr, int c, size_t n)`

→ Fill the n first octet/bytes of the memory area pointed by ptr with the constant byte c.

→ We converted the pointer to unsigned char to be able to scan the memory bytes.

→ This function does not check if the pointer is NULL.

```
void    *ft_memset(void *ptr, int c, size_t n)
{
        size_t          i;
        unsigned char   *p;

        p = (unsigned char *)ptr;
        i = 0;
        while (i < n)
        {
                p[i] = (unsigned char)c;
                i++;
        }
        return (ptr);
}
```

1.8 bzero

→ Prototype : `void ft_bzero(void *s, size_t n)`

→ Fill n first bytes with the value 0 in the area pointed by s.

```
void    ft_bzero(void *s, size_t n)
{
    size_t          i;
    unsigned char   *p;

    i = 0;
    p = (unsigned char *)s;
    if (!n)
        return ;
    if (!s)
        return ;
    while (i < n)
    {
        p[i] = 0;
        i++;
    }
}
```

1.9 memcpy

→ Prototype : `void * ft_memcpy(void *destination, const void *source, size_t n)`

→ Copy the N first bytes of src to the N first bytes of dest.

→ If src == dest, we return dest !

```
void * ft_memcpy(void *destination, const void *source, size_t n)
{
    unsigned char *dest;
    unsigned char *src;
    size_t i;

    dest = (unsigned char *)destination;
    src = (unsigned char *)source;
    i = 0;
    if (source == destination)
        return destination;
    while (i < n)
    {
        dest[i] = src[i];
        i++;
    }
    return destination;
}
```

1.10 memmove

→ Prototype : `void *ft_memmove(void *destination, const void *source, size_t size)`

→ Copy n bytes of the source to the destination unlike memcopy, it checks for overlap conditions.

→ If the overlap is done from the front we will copy the information from the right to the left.

→ For the others situations, we will copy from the left to the right.

```
void    *ft_memmove(void *destination, const void *source, size_t size)
{
    unsigned char       *dest;
    const unsigned char *src;
    size_t              i;

    dest = (unsigned char *)destination;
    src = (unsigned char *)source;
    if (!dest && !src)
        return (NULL);
    if (!size)
        return (dest);
    if (src < dest && dest < (src + size))
    {
        i = size;
        while (i-- > 0)
            dest[i] = src[i];
        return (dest);
    }
    i = 0;
    while (i < size)
    {
        dest[i] = src[i];
        i++;
    }
    return (dest);
}
```

1.11 memchr

→ Prototype : `void *ft_memchr(const void*s, int c, size_t n)`
→ Scan the n first bytes to find the first occurence c.
→ If c is found, we return the adress of the first occurence else we return NULL.

```
#include "/home/w/Bureau/libft/include/libft.h"

void    *ft_memchr(const void*s, int c, size_t n)
{
    unsigned char   *search;
    size_t          i;

    search = (unsigned char *)s;
    i = 0;
    if (!search)
        return (NULL);
    while (i < n)
    {
        if (*search == (unsigned char)c)
            return (search);
        i++;
        search++;
    }
    return (NULL);
}
```

1.12 memcmp

→ Prototype : `int ft_memcmp(const void *s1, const void *s2, size_t n)`

→ Compare the first n bytes of the memory area of s1 and s2.

```
int ft_memcmp(const void *s1, const void *s2, size_t n)
{
    size_t i = 0;
    unsigned char *uno = (unsigned char *)s1;
    unsigned char *deux = (unsigned char *)s2;

    while (i < n)
    {
        if (uno[i] != deux[i])
            return (uno[i] - deux[i]);
        i++;
    }
    return 0;
}
```

1.13 strlen

→ Prototype : `int ft_memcmp(const void *s1, const void *s2, size_t n)`

→ Return the number of character of the string.

```
int ft_memcmp(const void *s1, const void *s2, size_t n)
{
    size_t          i;
    unsigned char   *uno;
    unsigned char   *deux;

    i = 0;
    uno = (unsigned char *)s1;
    deux = (unsigned char *)s2;
    while (i < n)
    {
        if (uno[i] != deux[i])
            return (uno[i] - deux[i]);
        i++;
    }
    return (0);
}
```

1.14 strchr

→ Prototype : `char *ft_strchr(const char *s, int c)`

→ Return the first occurence of the character c in the string s.

```
/* Le caractère '\0' est pris en compte */
char    *ft_strchr(const char *s, int c)
{
    while (*s)
    {
        if (*s == (char)c)
            return ((char *)s);
        s++;
    }
    if (*s == (char)c)
        return ((char *)s);
    return (NULL);
}
```

1.15 strrchr

→ Prototype : `char *ft_strrchr(const char *s, int c)`

→ Return the last occurence of the character c in the string s.

```
/*
**  Ne pas oublier d'inclure le caractère de fin de chaîne
**  Dans la comparaison acvec c
*/
char    *ft_strrchr(const char *s, int c)
{
    char    *last_occurence;
    int     find;

    find = 0;
    while (*s)
    {
        if (*s == (char)c)
        {
            last_occurence = (char *)s;
            find++;
        }
        s++;
    }
    if (*s == (char)c)
        return ((char *)s);
    if (find)
        return (last_occurence);
    return (NULL);
}
```

1.16 strncmp

→ Prototype : `int ft_strncmp(const char *s1, const char *s2, size_t n)`

→ Compare the n first character of s1 and s2.

```
/*  Ne pas oublier la comparaison avec le caractère de fin */
int ft_strncmp(const char *s1, const char *s2, size_t n)
{
    size_t  i;

    i = 0;
    while (i < n)
    {
        if (s1[i] != s2[i])
            return ((int)s1[i] - (int)s2[i]);
        if (s1[1] == '\0' && s2[1] == '\0')
            break ;
        i++;
    }
    return (0);
}
```

1.17 strnstr

→ Prototype : `char *ft_strnstr(const char *big, const char *little, size_t len)`

→ Search the first occurence of the string little in the string big.

```
char    *ft_strnstr(const char *big, const char *little, size_t len)
{
    size_t  i;
    size_t  j;

    i = 0;
    if (!(*little))
        return ((char *)big);
    while (i < len && big[i])
    {
        j = 0;
        if (big[i] == little[j])
        {
            while ((i + j < len) && big[i + j]
                && little[j] && little[j] == big[i + j])
                j++;
            if (little[j] == '\0')
                return ((char *)big + i);
        }
        i++;
    }
    return (NULL);
}
```

1.18 toupper

→ Prototype : `int ft_toupper(int c)`

→ Replace the lower character into upper character.

```
int ft_toupper(int c)
{
    if ((unsigned char)c >= 'a' && (unsigned char)c <= 'z')
        return (c - 32);
    return (c);
}
```

1.19 tolower

→ Prototype : `int ft_tolower(int c)`

→ Replace the upper character into lower character.

```
int ft_tolower(int c)
{
    if ((unsigned char)c >= 'A' && (unsigned char)c <= 'Z')
        return (c + 32);
    return (c);
}
```

1.20 atoi

→ Prototype : `int ft_atoi(const char *s)`

→ Convert a string into an integer.

```
int ft_atoi(const char *s)
{
    int n;
    int sign;

    n = 0;
    sign = 1;
    while ((*s >= 9 && *s <= 13) || *s == 32)
        s++;
    if (*s == '-' || *s == '+')
    {
        if (*s == '-')
            sign = -1;
        s++;
        if (*s == '-' || *s == '+')
            return (0);
    }
    while (*s >= '0' && *s <= '9')
    {
        n = n * 10 + (*s - '0');
        s++;
    }
    return (n * sign);
}
```

1.21 calloc

→ Prototype : `void *ft_calloc(size_t nmemb, size_t size)`

→ The calloc() function allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero.

```
void *ft_calloc(size_t nmemb, size_t size)
{
 unsigned char *ptr;
 size_t   i;

 i = 0;
 if (nmemb > SIZE_MAX / size)
  return (NULL);
 if (nmemb == 0 || size == 0)
 {
  ptr = malloc(1);
  if (!ptr)
   return (NULL);
  return (ptr);
 }
 ptr = malloc(nmemb * size);
 if (!ptr)
  return (NULL);
 while (i < nmemb * size)
  ptr[i++] = 0;
 return ((void *)ptr);
}
```

1.22 strdup

→ Prototype : `char *ft_strdup(const char *s)`

→ Copy the string with malloc in a new char*.

```
char    *ft_strdup(const char *s)
{
    unsigned int    size;
    unsigned int    i;
    char            *copy;

    size = ft_strlen(s);
    copy = malloc(size + 1);
    if (!malloc)
         reutrn NULL;
    i = 0;
    while (i < size)
    {
        copy[i] = s[i];
        i++;
    }
    copy[i] = '\0';
    return (copy);
}
```

2 Fonctions supplémentaires

2.1 substr

→ Prototype : `char *ft_substr(char const *s, unsigned int start, size_t len)`

→ Return wether the position of the substring (if it exists) or NULL.

```
char    *ft_substr(char const *s, unsigned int start, size_t len)
{
    unsigned int    i;
    char            *copy;

    i = 0;
    if (start >= ft_strlen(s))
    {
        copy = malloc(1);
        if (!copy)
            return (NULL);
        *copy = '\0';
        return (copy);
    }
    if (start + (unsigned int)len > ft_strlen(s))
        len = ft_strlen(s) - start;
    copy = malloc(sizeof(char) * (len + 1));
    if (!copy || !s)
        return (NULL);
    while (i < len)
    {
        copy[i] = s[start + i];
        i++;
    }
    copy[i] = '\0';
    return (copy);
}
```

2.2 strjoin

→ Prototype : `char *ft_strjoin(char const *s1, char const *s2)`

→ Create a string composed of the string s1 concatenate with the string s2.

```
char    *remplissage(char const *s1, char const *s2)
{
    unsigned int    i;
    char            *copie;

    i = 0;
    copie = malloc((sizeof(char) * (ft_strlen(s1) + ft_strlen(s2))) + 1);
    if (!copie)
        return (NULL);
    i = 0;
    while (*s1)
    {
        copie[i++] = *s1;
        s1++;
    }
    while (*s2)
    {
        copie[i++] = *s2;
        s2++;
    }
    copie[i] = '\0';
    return (copie);
}

char    *ft_strjoin(char const *s1, char const *s2)
```

```
{
    char    *copie;

    if (!s1 || !s2)
        return (NULL);
    if (*s1 == '\0' && *s2 == '\0')
    {
        copie = malloc(1);
        if (!copie)
            return (NULL);
        copie[0] = '\0';
        return (copie);
    }
    copie = remplissage(s1, s2);
    return (copie);
}
```

2.3 strlcat

- → Prototype : `size_t ft_strlcat(char *dest, const char *src, size_t size)`
- → Concatenate two strings.
- → The moulinette only check the case if (size == 0 && !dest) i have to return the size of src.

```
size_t  ft_strlcat(char *dest, const char *src, size_t size)
{
    size_t  dlen;
    size_t  slen;
    size_t  i;
    size_t  j;

    if (!dest && size == 0)
        return (ft_strlen(src));

    dlen = ft_strlen(dest);
    slen = ft_strlen(src);

    if (size <= dlen)
        return (size + slen);

    i = dlen;
    j = 0;
    while (src[j] && i < size - 1)
        dest[i++] = src[j++];
    dest[i] = '\0';
    return (dlen + slen);
}
```

2.4 strlcpy

→ : Prototype: `size_t ft_strlcpy(char *dst, const char *src, size_t size)`

→ size = the size of the buffer.

→ Pas de garde fou.

→ On retourne uniquement la taille de src.

```
size_t  ft_strlcpy(char *dst, const char *src, size_t size)
{
    size_t  i;

    i = 0;
    if (size > 0)
    {
        while (src[i] && i + 1 < size)
        {
            dst[i] = src[i];
            i++;
        }
        dst[i] = '\0';
    }
    while (src[i])
        i++;
    return (i);
}
```

2.5 strtrim

→ `char *ft_strtrim(char const *s1, char const *set)`

→ Remove all the set in the start and the end of the string.

```
int is_set(char c, const char *set)
{
    while (*set)
    {
        if (c == *set)
            return (1);
        set++;
    }
    return (0);
}
```

```
size_t  ft_strlcpy(char *dst, const char *src, size_t dstsize)
{
    size_t  i;

    i = 0;
    if (!src)
        return (0);
    if (!dstsize)
        return ((size_t)ft_strlen(src));
    while (src[i] && i < dstsize - 1)
    {
        dst[i] = src[i];
        i++;
    }
    dst[i] = '\0';
    return ((size_t)ft_strlen(src));
}

char    *ft_strtrim(char const *s1, char const *set)
{
    unsigned int    start;
    unsigned int    end;
    char            *new_word;

    if (!s1)
        return (NULL);
    if (!set)
        return (ft_strdup(s1));
    start = 0;
    while (s1[start] && is_set(s1[start], set))
        start++;
    end = ft_strlen(s1);
    if (start == end)
        return (ft_strdup(""));
    end--;
    while (end > start && is_set(s1[end], set))
        end--;
    new_word = malloc(sizeof(char) * (end - start + 2));
    if (!new_word)
        return (NULL);
    ft_strlcpy(new_word, s1 + start, end - start + 2);
    return (new_word);
}
```

2.6 strsplit

- Prototype : •
- Splits the src string into words separated by one or more of the characters in sep.

```
** Fonctionnelle
void all_clear(char **array)
{
 int i;

 if (!array)
  return ;
 i = 0;
 while (array[i])
  free(array[i++]);
 free(array);
 return ;
}
*/

// Fonctionnelle.
unsigned int is_separator(char c, char separator)
{
 if (c == separator)
  return (1);
 return (0);
}

// Fonctionnelle
unsigned int count_word(char const *s, char sep)
{
 unsigned int count;
 int    in_word;

 count = 0;
 in_word = 0;
 while (*s)
 {
  if (is_separator(*s, sep))
   in_word = 0;
  else if (!in_word)
  {
   in_word = 1;
   count++;
  }
  s++;
 }
 return (count);
}

// Fonctionnelle
char *ft_strndup(char *s, int n)
{
 char *copy;
 int  i;

 i = 0;
 if (*s == '\0' || !s)
 {
  copy = malloc(1);
  *copy = '\0';
  return (copy);
 }
 copy = malloc(sizeof(char) * (n + 1));
 if (!copy)
  return (NULL);
 while (i < n)
 {
  copy[i] = s[i];
  i++;
 }
 copy[i] = '\0';
 return (copy);
}

// Fonctionnelle
char **fill_array(char **array, char const *s,

unsigned int nb_word, char sep)
{
 unsigned int i;
 unsigned int len;

 i = 0;
 array = malloc(sizeof(char *) * (nb_word + 1));
 if (!array)
  return (NULL);
 while (*s && i < nb_word)
 {
  while (*s && is_separator(*s, sep))
   s++;
  len = 0;
  while (s[len] && !is_separator(s[len], sep))
```

```
   len++;
  if (len)
  {
   array[i] = ft_strndup((char *)s, len);
   if (!array[i++])
    return (NULL);
   s += len;
  }
 }
 array[i] = NULL;
 return (array);
}

// Fonctionnelle
char **ft_split(char const *s, char c)
{
 char **array;

 array = NULL;
 return (fill_array(array, s, count_word(s, c), c));
 if (!array)
  return (NULL);
 return (array);
}
```

2.7 itoa

→ Prototype : `char *ft_itoa(int nb)`

→ Convert string into int.

```
int count_nb(long int n)
{
    int count;

    count = (n <= 0);
    while (n)
    {
        count++;
        n /= 10;
    }
    return (count);
}

char    *ft_itoa(int nb)
{
    char        *digits;
    long int    n;
    int         size;

    n = (long)nb;
    size = count_nb(n);
    digits = malloc(sizeof(char) * (size + 1));
    if (!digits)
        return (NULL);
    digits[size--] = '\0';
    if (n < 0)
    {
        digits[0] = '-';
        n = -n;
    }
    while (size > 0)
    {
        digits[size--] = (n % 10 + '0');
        n = n / 10;
    }
    if (digits[0] != '-')
        digits[0] = (n % 10) + '0';
    return (digits);
}
```

2.8 strmapi

→ Prototype : `char *ft_strmapi(char const *s, char (*f)(unsigned int, char))`

→ Apply a given function to each character of a string.

```
char    *ft_strmapi(char const *s, char (*f)(unsigned int, char))
{
    int     size;
    int     i;
    char    *retour;

    size = ft_strlen(s);
    i = 0;
    retour = malloc(sizeof(char) * (size + 1));
    if (!retour)
        return (NULL);
    while (i < size)
    {
        retour[i] = f(i, s[i]);
        i++;
    }
    retour[i] = '\0';
    return (retour);
}
```

2.9 striteri

→ Prototype : `void ft_striteri(char *s, void (*f)(unsigned int, char *))`

→ Like strmapi but return anything, the modification are done directly in the str.

```
void    ft_striteri(char *s, void (*f)(unsigned int, char *))
{
    unsigned int    i;

    if (!s || !f)
        return ;
    i = 0;
    while (s[i])
    {
        f(i, &s[i]);
        i++;
    }
}
```

2.10 putchar_fd

→ Prototype : `void ft_putchar_fd(char c, int fd)`

→ Write a character in the file descriptor chosen.

```
void    ft_putchar_fd(char c, int fd)
{
    write(fd, &c, 1);
}
```

2.11 putstr_fd

→ Prototype : `void ft_putchar_fd(char c, int fd)`

→ Write a string in the file descriptor chosen.

```
void    ft_putstr_fd(char *str, int fd)
{
    if (!str)
        return ;
    while (*str)
    {
        write(fd, str, 1);
        str++;
    }
}
```

2.12 putendl_fd

→ Prototype : `void ft_putendl_fd(char *str, int fd)`

→ After write the string, write the end of line `\n`

```
void    ft_putendl_fd(char *str, int fd)
{
    if (!str)
        return ;
    while (*str)
    {
        write(fd, str, 1);
        str++;
    }
    write(fd, "\n", 1);
}
```

2.13 putnbr_fd

→ Prototype : `void ft_putnbr_fd(int n, int fd)`

→ Write the number in the file descriptor.

```
void    ft_putnbr_fd(int n, int fd)
{
    long int    nb;
    char        solo;
    int         suivant;

    nb = n;
    if (nb < 0)
    {
        nb = -nb;
        write(fd, "-", 1);
    }
    solo = (nb % 10) + '0';
    suivant = nb / 10;
    if (nb > 9)
        ft_putnbr_fd(suivant, fd);
    write(fd, &solo, 1);
}
```

3 Bonus — Fonctions sur les listes chaînées

3.1 lstnew

→ Prototype : `t_list *ft_lstnew(void *content)`

→ Create a new link of a list with is content.

```
t_list *ft_lstnew(void *content)
{
 t_list *elem;

 elem = malloc(sizeof(t_list));
 if (!elem)
  return (NULL);
 elem->next = NULL;
 elem->content = content;
 return (elem);
}
```

3.2 lstadd_front

→ Prototype : `void ft_lstadd_front(t_list **lst, t_list *new)`

→ Add a new link in the list by the front.

→ Important rappel :

```
int a = 42;

int *p = &a;     // p contient l'adresse de a
int **pp = &p;   // pp contient l'adresse de p

a = valeur 42.
&a = adresse de la variable a.
p = adresse de la varaible a.
*p = Valeur pointée par a donc 42.
pp = Adresse de p.
*pp = Le contenu de p donc l'adresse de a.
**pp  = le contenu de a


void ft_lstadd_front(t_list **lst, t_list *new)
{
 if (!lst || !new)
```

```
     return ;
    new->next = *lst;
    *lst = new;
   }
```

3.3 lst_size

→ Prototype : `int ft_lstsize(t_list *lst)`

→ Count the number of link in the list.

```
int ft_lstsize(t_list *lst)
{
    if (lst == NULL)
        return (0);
    return (1 + ft_lstsize(lst->next));
}
```

3.4 lst_last

→ Prototype : `int ft_lstsize(t_list *lst)`

→ Get the last link of the list.

```
t_list  *ft_lstlast(t_list *lst)
{
    if (!lst)
        return (NULL);
    while (lst->next != NULL)
        lst = lst->next;
    return (lst);
}
```

3.5 lstadd_back

→ Prototype : `void ft_lstadd_back(t_list **lst, t_list *new)`

→ Add a link in a list by the end.

```
void    ft_lstadd_back(t_list **lst, t_list *new)
{
    t_list  *copy;

    if (!new || !lst)
        return ;
    if (*lst == NULL)
    {
        *lst = new;
        return ;
    }
    copy = *lst;
    while (copy->next != NULL)
        copy = copy->next;
    copy->next = new;
}
```

3.6 lstdelone

→ Prototype : `void ft_lstclear(t_list **lst, void (*del)(void *))`

→ Remove a link of the list.

```
void    ft_lstclear(t_list **lst, void (*del)(void *))
{
    t_list  *nettoyeur;

    if (!lst || !del)
        return ;
    while (*lst)
    {
        nettoyeur = (*lst)->next;
        del((*lst)->content);
        free(*lst);
        *lst = nettoyeur;
    }
    *lst = NULL;
}
```

3.7 lstclear

→ Prototype : `void ft_lstclear(t_list **lst, void (*del)(void *))`

→ Clear all the list

```
void    ft_lstclear(t_list **lst, void (*del)(void *))
{
    t_list  *nettoyeur;

    if (!lst || !del)
        return ;
    while (*lst)
    {
        nettoyeur = (*lst)->next;
        del((*lst)->content);
        free(*lst);
        *lst = nettoyeur;
    }
    *lst = NULL;
}
```

3.8 lstiter

→ Prototype : `void ft_lstiter(t_list *lst, void (*f)(void *))`

→ Apply a function f to all the link of the list.

```
void    ft_lstiter(t_list *lst, void (*f)(void *))
{
    if (!lst)
        return;
    while(lst)
    {
        f(lst->content);
        lst = lst->next;
    }
}
```

3.9 lstmap

— Prototype :

— Create a new list that is a copy of the original but with the content modified by a function to each link (without modifying the original).

```
#include "/home/w/Bureau/libft/include/libft.h"

t_list *ft_lstmap(t_list *lst, void *(*f)(void *), void (*del)(void *))
{
 t_list *new_list;
 t_list *new_maillon;

 new_list = NULL;
 if (!lst || !f || !del)
  return (NULL);
 while (lst)
 {
  new_maillon = ft_lstnew(f(lst->content));
  if (!new_maillon)
  {
   ft_lstclear(&new_list, del);
   return (NULL);
  }
  ft_lstadd_back(&new_list, new_maillon);
  lst = lst->next;
 }
 return (new_list);
}
```