Libft

# Part 1 - Libc functions

## ft\_isalpha.c

ㆍisalpha() function checks whether a character is an alphabet(a to z and A to Z) or not.

ㆍIf a character passed to isalpha() is an alpahabet, it returns a non-zero integer, if not it returns 0. ㆍPrototype : **int ft\_isalpha(int c);**

## ft\_isdigit.c

ㆍisdigit() function checks whether a character is numeric character(0 - 9) or not. ㆍPrototype : **int ft\_isdigit(int c);**

## ft\_isalnum.c

ㆍisalnum() function checks whether the argument passed is an alphanumeric character(alphabet or numeric) or not.

ㆍPrototype : **int ft\_isalnum(int c);**

## ft\_isascii.c

ㆍisascii() function checks whether the argument passed is a 7-bit ascii character or not. ㆍPrototype : **int ft\_isascii(int c);**

## ft\_isprint.c

ㆍisprint() function checks whether a character is a printable character or not. ㆍPrintable character : ascii 32 ~ 126.

ㆍPrototype : **int ft\_isprint(int c);**

## ft\_strlen.c

ㆍstrlen() function takes a string as an argument and returns its length.

ㆍstrlen() function doesn’t count the null character \0 while calculating the length. ㆍPrototype **size\_t ft\_strlen(const char \*s);**

ㆍParameters - *s* : The string whose length to be counted. ㆍReturn value The counted length.

* size\_t : size\_t can store the maximum size of a theoretically possible object of any type(including array). size\_t is an unsigned data type.

## ft\_memset.c

ㆍmemset() function sets memory to a given value.

ㆍmemset() function fills *length* bytes starting at *dst* with the value *c*. ㆍPrototype **void \*ft\_memset(void \*dst, int c, size\_t length);** ㆍParameters - *dst* : A pointer to the memory that you want to set.

* *c* : The value you want to store in each byte.

This value passed as an int, but the function fills the block of memory using the unsigned char conversion of this value.

* *length* : The number of bytes to set.

ㆍReturn value A pointer to the destination buffer. (That is, the value of *dst.*)

* why unsigned char? : Signed char can use some of internal bits as a sign bits, but unsigned char is not allowed.

## ft\_bzero.c

ㆍbzero() function fills the first *n* bytes of the object pointed to by *dst* with zero(NULL) bytes. ㆍPrototype **void ft\_bzero(void \*dst, size\_t n);**

ㆍParameters - *dst* : An existing object that you want to fill with zero.

- *n* : The number of bytes to fill. ㆍReturn value None.

→ use memset instead of bzero.

## ft\_memcpy.c

ㆍmemcpy() function copies bytes from one buffer to another.

ㆍmemcpy() function copies *length* bytes from buffer pointed to by *src* into the buffer pointed to by *dst*. ㆍPrototype **void \*ft\_memcpy(void \*dst, const void \*src, size\_t length);**

ㆍParameters - *dst* : A pointer to where you want the function to copy the data.

* *src* : A pointer to the buffer that you want to copy data from
* *length* : The number of bytes to copy.

ㆍReturn value The pointer to the destination where the data is being copied.

→ null : copy up to *n - 1* and put the null at the end.

## ft\_memmove.c

ㆍmemmove() function copies bytes from one buffer to another, handling overlapping memory correctly. ㆍmemmove() function copies *length* bytes from the buffer pointed to by *src* to the buffer pointed to by *dst*.

Copying of overlapping regions is handled safely. Use memcpy() for greater speed when copying buffers that don’t overlap.

## ㆍPrototype void \*ft\_memmove(void \*dst, const void \*src, size\_t length);

ㆍParameters - *dst* : A pointer to where you want the function to copy the data.

* *src* : A pointer to the buffer that you want to copy data from.
* *length* : The number of bytes to copy.

ㆍReturn value A pointer to the destination buffer. (Tthat is, the value of *dst*.)

* overlap : When performing a copy within an array, and when the start address of *src* is before the start address of *dst*, there is possibility of overlap. To solve the problem, copy each byte from the back. (The last address of *src* → The last address of *dst*.)

## ft\_strlcpy.c

ㆍstrlcpy() copies up to *dstsize - 1* characters from *src* to *dst*.

ㆍstrlcpy() takes full size of the destination buffer and guarantees NUL-termination if *dstsize* is not 0. (Room for NULL should be included in *dstsize*.)

## ㆍPrototype size\_t ft\_strlcpy(char \*dst, const char \*src, size\_t dstsize);

ㆍParameters - *dst* : A pointer to the destination string.

* *src* : A pointer to the source string.
* *dstsize* : The size of the destination buffer. ㆍReturn value The total length of the string. (The length of *src*.)

## ft\_strlcat.c

ㆍstrlcat() appends *src* to end of *dst* at most *‘dstsize - strlen(dst) - 1’* characters.

ㆍstrlcat() guarantees NUL-termination unless *dstsize* is 0 or original *dst* string was longer than *dstsize*. (Room for NULL should be included in *dstsize*.)

## ㆍPrototype size\_t ft\_strlcat(char \*dst, const char \*src, size\_t dstsize);

ㆍParameters - *dst* : A pointer to the destination string.

* *src* : A pointer to the source string.
* *dstsize* : The size of the destination buffer.

ㆍReturn value The total length of the string. (The initial length of *dst* plus the length of *src*.)

## ft\_toupper.c

ㆍtoupper() function converts a character to uppercase. ㆍPrototype **int ft\_toupper(int c);**

## ft\_tolower.c

ㆍtolower() function converts a character to lowercase. ㆍPrototype **int ft\_tolower(int c);**

## ft\_strchr.c

ㆍstrchr() function finds the first occurrence of *c*(converted to a char) in the string pointed to by *s*. The terminating NUL character is considered to be part of the string.

ㆍPrototype **char \*ft\_strchr(const char \*s, int c);**

ㆍParameters - *s* : The string that you want to search.

* *c* : The character that you’re looking for.

ㆍReturn value A pointer to the located character, or NULL if *c* doesn’t occur in the string.

## ft\_strrchr.c

ㆍstrrchr() function finds the last occurrence of *c*(converted to a char) in the string pointed to by *s*. The termination NUL character is considered to be part of the string.

ㆍPrototype **char \*ft\_strrchr(const char \*s, int c);**

ㆍParameters - *s* : The string that you want to search.

* *c* : The character that you’re looking for.

ㆍReturn value A pointer to the located character, or NULL if *c* doesn’t occur in the string.

## ft\_strncmp.c

ㆍstrncmp() function compares up to *n* characters from the strings pointed to by *s1* and *s2*. ㆍPrototype **int ft\_strncmp(const char \*s1, const char \*s2, size\_t n);** ㆍParameters - *s1, s2* : The strings that you want to compare.

* *n* : The maximum number of characters that you want to compare. ㆍReturn value - return value < 0 : *s1* is less than *s2*.
* return value = 0 : *s1* is equal to *s2*.
* return value > 0 : *s1* is greater than *s2*.

## ft\_memchr.c

ㆍmemchr() function locates the first occurrence of value(converted to an unsigned char) in the first

*num* bytes of the buffer pointed to by *ptr*.

## ㆍPrototype void \*ft\_memchr(void \*ptr, int value, size\_t num);

ㆍParameters - *ptr* : The buffer that you want to find.

* *value* : The character that you’re looking for.
* *num* : The number of bytes to search in the buffer.

ㆍReturn value A pointer to the located character, or NULL if *value* couldn’t be found.

## ft\_memcmp.c

ㆍmemcmp() function compares *num* bytes of the buffer pointed to by *ptr1* to the buffer pointed to by *ptr2*.

## ㆍPrototype int ft\_memcmp(const void \*ptr1, const void \*ptr2, size\_t num);

ㆍParameters - *ptr1, ptr2* : Pointers to the buffer that you want to compare.

* *num* : The number of bytes that you want to compare. ㆍReturn value - return value < 0 : The object pointed to by *ptr1*

is less than the object pointed to by *ptr2*.

* return value = 0 : The object pointed to by *ptr1*

is equal to the object pointed to by *ptr2*.

* return value > 0 : The object pointed to by *ptr1*

is greater than the object pointed to by *ptr2*.

## ft\_strnstr.c

ㆍstrnstr() function locates the first occurrence of the NUL-terminated string find in the string *src*, where not more than *length*, characters are searched. Characters that appear after \0 character are not searched.

## ㆍPrototype char \*ft\_strnstr(const char \*src, const char \*find, size\_t length);

ㆍParameters - *src* : The string that you want to search.

* *find* : The string that you’re looking for.
* *length* : The *src* should not be more than *length*. ㆍReturn value - If *find* is an empty string : *src* is returned.
* If *find* occurs nowhere in *src* : NULL is returned.
* Otherwise a pointer to the first character of the first occurrence of find is returned.

## ft\_atoi.c

ㆍatoi() function converts the string pointed to by *str* to an int. ㆍPrototype **int ft\_atoi(const char \*str);** ㆍParameters - *str* : A pointer to the string to parse. ㆍReturn value The converted integer.

## ft\_calloc.c

ㆍcalloc() function allocates space from the heap for an array of *n* object, each of *size* bytes, and initializes them to 0.

ㆍUse free() or realloc() to free the block of memory.

ㆍBecause the malloc() implementation uses signed, 32-bit integers to represent the size internally, you can’t allocate more than 2GB in a single allocation. If the size is greater than 2GB, calloc() indicates an error of ENOMEM.

ㆍPrototype **void \*ft\_calloc(size\_t n, size\_t size);**

ㆍParameters - *n* : The number of array elements to allocate.

* *size* : The size, in bytes, of one array element.

ㆍReturn value A pointer to the start of the allocated memory, or NULL if an error occurred.

→ ENOMEM : Not enough memory.

* malloc ㆍmalloc() function allocates a buffer of size bytes. ㆍUse free() or realloc() to free the block of memory.

ㆍBecause the malloc() implementation use signed, 32-bit integers to represent the size

internally, you can’t allocate more than 2GB in a single allocation. If the size is greater than 2GB, malloc() indicates an error of ENOMEM.

ㆍPrototype **void \*malloc(size\_t size);**

ㆍParameters - *size* : The number of bytes to allocate.

ㆍReturn value A pointer to the start of the allocated memory, or NULL if an error occurred.

* free ㆍfree() function deallocates the memory block specified by *ptr*, which was previously returned by a call to calloc(), malloc(), realloc().

ㆍPrototype **void \*free(size\_t ptr);**

ㆍParameters - *ptr* : A pointer to the block of memory that you want to free.

It’s safe to call free() with a NULL pointer.

## ft\_strdup.c

ㆍstrdup() function creates a duplicate of the string pointed to by *src*, and returns a pinter to the new copy. ㆍstrdup() function allocates the memory for the new string by calling malloc(); it’s up to you to release the

memory by calling free.

ㆍPrototype **char \*ft\_strdup(const char \*src);**

ㆍParameters - *src* : The string that you want to copy.

ㆍReturn value A pointer to a copy of the string, or NULL if an error occurred.

# Part 2 - Additional functions

## ft\_substr.c

ㆍAllocates(with malloc(3)) and returns a substring from the string *s*. The substring begins at index *start* and is of maximum size *len*.

## ㆍPrototype char \*ft\_substr(char const \*s, unsigned int start, size\_t len);

ㆍParameters - *s* : The string from which to create the substring.

* *start* : The start index of the substring in the string *s*.
* *len* : The maximum length of the substring ㆍReturn value The substring. NULL if the allocation fails. ㆍExternal functs : malloc.
* exception - when s[0] : return 0;

- when the length of *s* is less than *start* : can’t make substring → return empty string.

* *char \** : Mutable pinter to a mutable character/string.
* *const char \** or *char const \** : Mutable pointer to an immutable character/string.

You cannot change the contents of the location(s) this pointer point to.

* *char \* const* : An immutable pointer. (It cannot point to any other location). But the contents of location at which it points are mutable.
* *const \* char const* : Immutable pointer to an immutable character/string.

## ft\_strjoin.c

ㆍAllocates(with malloc(3)) and returns a new string, which is the result of the concatenation of *s1* and *s2*. ㆍPrototype **char \*ft\_strjoin(char const \*s1, char const \*s2);**

ㆍParameters - *s1* : The prefix string.

* + *s2* : The suffix string.

ㆍReturn value The new string. NULL if the allocation fails. ㆍExternal functs : malloc.

* *char \* s = “”* : Empty string.
* *char \* s = NULL* : *s* pointer doesn’t point anything. (= NULL)

- when *return (s);* : - *char \* s = “”* : s[0] == ‘\0’.

- *char \* s = NULL* : segment fault.

## ft\_strtrim.c

ㆍAllocates(with malloc(3)) and returns a copy of *s1* with the characters specified in *set* removed from the beginning and the end of the string.

## ㆍPrototype char \*ft\_strtrim(char const \*s1, char const \*set);

ㆍParameters - *s1* : The string to be trimmed.

* + *set* : The reference set of characters to trim. ㆍReturn value The trimmed string. NULL if the allocation fails. ㆍExternal functs : malloc.

→ use strchr, substr.

ex) - *s1* = “ABCCBA”, *set* = “AB” → result : “CC”.

* *s1* = “ACCBACBA”, *set* = “AB” → result : “CCBAC”.
* *s1* = “Hello World!”, *set* = “Hlde”→ result : “o World!”.

## ft\_split.c

ㆍAllocates(with malloc(3)) and returns an array of strings obtained by splitting *s* using the character *c*

as a delimiter. The array must be ended by a NULL pointer. ㆍPrototype **char \*\*ft\_split(char const \*s, char c);** ㆍParameters - *s* : The string to be split.

- *c* : The delimiter character.

ㆍReturn value The array of new string resulting from the split. NULL if the allocation fails. ㆍExternal functs : malloc, free.

## ft\_itoa.c

ㆍAllocates(with malloc(3)) and returns a string representing the inter received as an arguments. Negative numbers must be handled.

ㆍPrototype **char \*ft\_itoa(int n);**

ㆍParameters - *n* : The integer to convert.

ㆍReturn value The string representing the integer. NULL if allocation fails. ㆍExternal functs : malloc.

## ft\_strmapi.c

ㆍApplies the function *f* to each character of the string *s* to create a new string(with malloc(3)) resulting from successive applications of *f*.

## ㆍPrototype char \*ft\_strmapi(char const \*s, char (\*f)(unsigned int, char));

ㆍParameters - *s* : The string on which to iterate.

- *(\*f)(unsigned int, char)* : The function to apply to each character. ㆍReturn value The string created from the successive applications of *f*. ㆍExternal functs : malloc.

* function pointer : A pointer that points to a funcion.

ㆍA function pointer points to executable code within memory.

Dereferencing the function pointer yields the referenced funcion, which can be invoked and passed arguments just as in a normal function call.

→ return\_type (\*pointer\_name)(parameter types);

ex) - A normal function with an int parameter and void return type.

## void fun(int a)

**{**

## printf(”Value of a is %d\n”, a);

**}**

- Use function pointer.

int main()

## {

**ft\_striteri.c**

## void (\*fun\_ptr)(int);

**fun\_ptr = &fun;** # fun\_ptr is a pointer to function fun().

**(fun\_ptr)(10);** # invoking fun() using fun\_ptr.

## return (0);

**}**

ㆍApplies the function *f* to each character of the string passed as argument, and passing its index as first argument. Each character is passed by address to *f* to be modified if necessary.

## ㆍPrototype void ft\_striteri(char \*s, void (\*f)(unsigned int, char \*));

ㆍParameters - *s* : The string on which to iterate.

- *(\*f)(unsinged int, char \*)* : The function to apply to each character. ㆍReturn value None.

ㆍExternal functs : None.

## ft\_putchar\_fd.c

ㆍOutputs the character *c* to the given file descriptor. ㆍPrototype **void ft\_putchar\_fd(char c, int fd);** ㆍParameters - *c* : The character to output.

- *fd* : The file descriptor on which to write. ㆍReturn value None.

ㆍExternal functs : write.

## ft\_putstr\_fd.c

ㆍOutputs the string *s* to the given file descriptor. ㆍPrototype **void ft\_putstr\_fd(char \*s, int fd);** ㆍParameters - *s* : The string to output.

- *fd* : The file descriptor on which to write.

ㆍReturn value None. ㆍExternal functs : write.

## ft\_putendl\_fd.c

ㆍOutputs the string *s* to the given file descriptor, followed by a newline. ㆍPrototype **void ft\_putendl\_fd(char \*s, int fd);**

ㆍParameters - *s* : The string to output.

- *fd* : The file descriptor on which to write. ㆍReturn value None.

ㆍExternal functs : write.

## ft\_putnbr\_fd.c

ㆍOutputs the integer *n* to the given fild descriptor. ㆍPrototype **void ft\_putnbr\_fd(char \*n, int fd);** ㆍParameters - *n* : The integer to output.

- *fd* : The file descriptor on which to write. ㆍReturn value None.

ㆍExternal functs : write.

# Bonus part

\* linked list

ㆍA linear collection of data elements whose order is not given by their physical placement in memory.

Instead, each element points to the next.

ㆍIt is a data structure consisting of a collection of nodes which together represent a sequence. ㆍIn its most basic form, each node contains : data, and a refrence(= a link) to the next node in the

sequence.

\* ‘t\_list’ struct

## typedef struct s\_list

**{**

## void \*content;

**struct s\_list \*next;**

## } t\_list;

ㆍcontent : The data contained in the element. The *void \** allows to store any kind of data. ㆍnext : The next element’s address or NULL if it’s the last element.

## ft\_lstnew.c

ㆍAllocates(with malloc(3)) and returns a new elements. The variable *content* is initialized with the value of the parameter *content*.

ㆍPrototype **t\_list \*ft\_lstnew(void \*content);**

ㆍParameters - *content* : The content to create the new element with. ㆍReturn value The new element.

ㆍExternal functs : malloc.

## ft\_lstadd\_front.c

ㆍAdds the element *new* at the beginning of the list.

## ㆍPrototype void ft\_lstadd\_front(t\_list \*\*lst, t\_list \*new);

ㆍParameters - *lst* : The address of a pointer to the first link of a list.

- *new* : The address of a pointer to the element to be added to the list. ㆍReturn value None.

ㆍExternal functs : None.

## ft\_lstsize.c

ㆍCounts the number of elements in a list. ㆍPrototype **int ft\_lstsize(t\_list \*lst);** ㆍParameters - *lst* : The beginning of the list. ㆍReturn value Length of the list.

ㆍExternal functs : None.

→ when ‘*next = NULL’* : when it’s the last element.

## ft\_lstlast.c

ㆍReturns the last element of the list. ㆍPrototype **t\_list ft\_lstlast(t\_list lst);** ㆍParameters - *lst* : The beginning of the list. ㆍReturn value Last element of the list. ㆍExternal functs : None.

## ft\_lstadd\_back.c

ㆍAdds element *new* at the end of the list.

## ㆍPrototype void ft\_lstadd\_back(t\_list \*\*lst, t\_list \*new);

ㆍParameters - *lst* : The address of a pointer to the first of a list.

- *new* : The address of a pointer to the element to be added to the list. ㆍReturn value None.

ㆍExternal functs : None.

## ft\_lstdelone.c

ㆍTakes as a parameter an element and frees the memory of the element’s content using the function del given as a parameter and free the element. The memory of *next* must not be freed.

## ㆍPrototype void ft\_lstdelone(t\_list lst, void (\*del)(void \*));

ㆍParameters - *lst* : The element to free.

- *(\*del)(void \*)* : The address of the function used to delete the content. ㆍReturn value None.

ㆍExternal functs : None.

→ free the content first, and then free lst.

ㆍcontent variable contains only the address,

for this reason, you need to delete the value pointed by the address. ㆍAfter *free(lst)*, it is impossible to access with *lst->content.*

So free the content first, and then free lst.

## ft\_lstclear.c

ㆍDeletes and frees the given elements and every successor of element, using the function *del* and free(3).

Finally, the pointer to the list must be set to NULL.

## ㆍPrototype void ft\_lstclear(t\_list \*\*lst, void (\*del)(void \*));

ㆍParameters - *lst* : The address of a pointer to an element.

- *(\*del)(void \*)* : The address of the function used to delete the content of the element. ㆍReturn value None.

ㆍExternal functs : free.

## ft\_lstiter.c

ㆍIterates the list *lst* and applies the function *f* to the content of each element. ㆍPrototype **void ft\_lstiter(t\_list \*lst, void (\*f)(void \*));**

ㆍParameters - *lst* : The address of a pointer to an element.

- *(\*f)(void \*)* : The address of the function used to iterate on the list. ㆍReturn value None.

ㆍExternal functs : None.

## ft\_lstmap.c

ㆍIterates the list *lst* and applies the function *f* to the content of each element. Creats a new list resulting of the successive applications of the function *f*. The *del* function is used to delete the content of an element if needed.

## ㆍPrototype t\_list \*ft\_lstmap(t\_list \*lst, void \*(\*f)(void \*), void (\*del)(void \*);

ㆍParameters - *lst* : The address of a pointer to an element.

* *(\*f)(void \*)* : The address of the function used to iterate on the list.
* *(\*del)(void \*)* : The address of the function used to delete the content of an element if needed.

ㆍReturn value The new list. NULL if the allocation fails. ㆍExternal functs : malloc, free.