RUSH02 – ROSETTA STONE

Description détaillée de chaque fonction + Audit du code

rush-02.c>

int is_space(int a)

Objectif : Cette fonction vérifie si le caractère donné est un caractère "d'espacement" (whitespace).

Paramètre :

int a : Le code ASCII du caractère à tester.

Fonctionnement :

Elle compare le caractère a à une liste de codes ASCII prédéfinis :

9 : Tabulation horizontale (\t)

10 : Saut de ligne (\n)

11 : Tabulation verticale

12 : Saut de page

13 : Retour chariot

32 : Espace

Si a correspond à l'un de ces caractères, la fonction retourne 1 (vrai).

Sinon, elle retourne 0 (faux).

Valeur de retour :

1 si le caractère est un espacement.

0 dans le cas contraire.

int atoi_char(const char *str, char *gnrltab)

Objectif : Cette fonction semble être une tentative de créer une version personnalisée de la fonction atoi (qui convertit une chaîne de caractères en entier). Elle a un double rôle : ignorer les espaces de début et copier les chiffres valides dans un autre tableau.

Paramètres :

const char *str : La chaîne de caractères à analyser (le nombre d'entrée).

char *gnrltab : Un tableau de caractères (char*) destiné à stocker les chiffres du nombre une fois les espaces et les signes de début ignorés.

Fonctionnement :

La fonction parcourt la chaîne str pour ignorer tous les caractères d'espacement au début, en utilisant la fonction is_space.

Elle vérifie la présence d'un signe + et l'ignore s'il est trouvé.

Si elle rencontre un signe -, elle considère cela comme une erreur, affiche "Error" sur la sortie standard et s'arrête.

Ensuite, elle entre dans une boucle qui continue tant qu'elle trouve des chiffres ('0' à '9').

À chaque itération, elle copie le chiffre trouvé de str vers gnrltab.

Valeur de retour : La fonction est déclarée comme retournant un int, mais dans le code fourni, elle ne retourne explicitement de valeur que lorsqu'elle écrit une erreur. Son objectif principal semble être de remplir gnrltab par effet de bord.

int stock_name(char *filename)

Objectif : Le but de cette fonction est probablement de stocker le nom d'un fichier. Cependant, son implémentation est incorrecte et provoquerait une erreur de segmentation à l'exécution.

Paramètre :

char *filename : La chaîne de caractères contenant le nom du fichier.

Fonctionnement (tel qu'écrit) :

char **file; : Déclare un pointeur sur un pointeur de caractère (file), mais sans l'initialiser. Il pointe donc vers une adresse mémoire aléatoire.

**file = *filename; : Tente de déréférencer deux fois ce pointeur non initialisé pour y assigner le premier caractère de filename. Ceci est une erreur critique.

Valeur de retour : La fonction ne retourne aucune valeur, bien qu'elle soit déclarée comme retournant un int.

int main(int argc, char **argv)

Objectif : C'est la fonction principale, le point d'entrée du programme. Son rôle est de gérer les arguments fournis en ligne de commande.

Paramètres :

int argc : Le nombre d'arguments passés au programme (le nom du programme lui-même est le premier argument).

char **argv : Un tableau de chaînes de caractères, où chaque chaîne est un argument.

Fonctionnement :

gnrltab = malloc(sizeof(char) * 40); : Alloue de la mémoire pour un tableau de 40 caractères. C'est probablement là que le nombre à convertir doit être stocké après "nettoyage".

if (argc == 2) : Si le programme a été lancé avec un seul argument (ex: ./name 123), argc vaut 2. Le code appelle alors atoi(argv[1]). L'appel à atoi ici ne sert à rien car le résultat n'est ni stocké ni utilisé. L'intention était probablement d'appeler atoi_char.

if (argc == 3) : Si le programme a été lancé avec deux arguments (ex: ./name numbers.dict 456), argc vaut 3. Le code appelle atoi(argv[2]). Encore une fois, le résultat de atoi est ignoré.

free(gnrltab); : Libère la mémoire qui a été allouée dynamiquement pour gnrltab, ce qui est une bonne pratique pour éviter les fuites de mémoire.

return (0); : Termine le programme et renvoie un code de succès (0) au système d'exploitation.

En résumé, ce fichier rush-02.c est responsable de la logique de démarrage du programme. Il analyse les arguments de la ligne de commande pour déterminer quel nombre convertir (et éventuellement quel dictionnaire utiliser), mais les fonctions qu'il contient sont soit incomplètes, soit incorrectes dans l'état actuel. Il sert de "squelette" pour lancer le processus de conversion qui est détaillé dans les autres fichiers.

conditioners_bridge.c>

Comme son nom l'indique, ce fichier sert de "pont". Il orchestre la conversion du nombre, en faisant le lien entre le découpage en paquets (fait dans number_conditioner.c) et la décomposition de chaque paquet (faite dans dict_packs_conditioner.c), tout en y ajoutant la logique des échelles (mille, million, etc.).

void scale_code_emitter(int pack, int scale_idx, int *outputs, int *count)

Objectif : Cette fonction a un seul rôle : ajouter le "jeton d'échelle" (mille, million...) à la liste des jetons à imprimer. Elle ne fait rien si le paquet de nombres est nul.

Paramètres :

int pack : La valeur numérique du paquet de trois chiffres (ex: 234). La fonction l'utilise pour vérifier qu'elle n'est pas égale à 0. On ne dit pas "zéro million".

int scale_idx : L'index de l'échelle. C'est la partie la plus importante.

0 = pas d'échelle (pour les nombres de 0 à 999).

1 = mille.

2 = million.

3 = milliard, etc.

int *outputs : Le tableau qui contient la liste finale des jetons numériques. Cette fonction y ajoute un nouvel élément.

int *count : Un pointeur vers le compteur du nombre total de jetons. La fonction incrémente cette valeur après avoir ajouté son jeton.

Fonctionnement :

Elle vérifie d'abord deux conditions : l'index de l'échelle doit être supérieur à 0 (on ne dit pas "cent vingt-trois unité") ET le paquet ne doit pas valoir 0.

Si les conditions sont remplies, elle ajoute un jeton au tableau outputs. Le jeton est négatif (-scale_idx). C'est une astuce pour différencier les jetons d'échelle (ex: -1 pour mille) des jetons de nombres (ex: 100 pour cent).

Enfin, elle incrémente *count pour indiquer qu'un jeton de plus a été ajouté.

Valeur de retour : void (rien), car elle modifie directement le tableau outputs via son pointeur.

int ft_transmitter(int *packs, int pmax, int *outputs, int max_outputs)

Objectif : C'est le chef d'orchestre de la traduction. Il parcourt tous les paquets de nombres (du plus grand au plus petit) et appelle les fonctions nécessaires pour générer la séquence complète de jetons.

Paramètres :

int *packs : Le tableau contenant les paquets de nombres (ex: pour 1,234,567, ce tableau serait {567, 234, 1}).

int pmax : Le nombre total de paquets dans le tableau.

int *outputs : Le tableau final où tous les jetons seront stockés.

int max_outputs : La taille maximale du tableau outputs pour éviter les dépassements.

Fonctionnement :

Il initialise un compteur de jetons count à 0.

Il commence une boucle qui itère à l'envers, depuis le dernier paquet (pmax - 1) jusqu'au premier. C'est crucial car on doit traiter "un million" avant "deux cent trente-quatre mille".

Pour chaque paquet :

Si le paquet n'est pas nul (packs[i] != 0), il procède en deux étapes :

Il appelle packsofthree_emitter(packs[i], ...) (la fonction de dict_packs_conditioner.c). Cette fonction décompose le nombre du paquet (ex: 234) en jetons de base (2, 100, 30, 4).

Il appelle scale_code_emitter(packs[i], i, ...) pour ajouter le jeton d'échelle correspondant à la position du paquet (ex: -1 pour "mille", -2 pour "million").

Valeur de retour : Le nombre total de jetons (count) qui ont été générés et placés dans le tableau outputs.

int nbr_from_str(char *str_nbr, int *outputs, int max_outputs)

Objectif : C'est la fonction principale et le point d'entrée de tout le processus de conversion. Elle prend en charge la chaîne de caractères brute et la transforme en une liste finale de jetons numériques prête à être imprimée.

Paramètres :

char *str_nbr : Le nombre à convertir, sous forme de chaîne de caractères (ex: "1234567").

int *outputs : Le tableau vide qui sera rempli avec la séquence de jetons.

int max_outputs : La taille de ce tableau.

Fonctionnement :

Elle déclare un tableau local packs pour stocker les paquets de trois chiffres.

Elle appelle pack_filler(str_nbr, packs, &pmax) (la fonction de number_conditioner.c). Cette fonction lit str_nbr et remplit le tableau packs.

Elle gère un cas particulier important : le nombre zéro. Si le remplissage ne donne qu'un seul paquet (pmax == 1) et que ce paquet vaut 0, elle place simplement le jeton 0 dans outputs et retourne 1.

Pour tous les autres nombres, elle appelle ft_transmitter, lui passant les paquets qu'elle vient d'obtenir, pour faire le gros du travail.

Valeur de retour : Elle retourne le résultat de ft_transmitter, c'est-à-dire le nombre total de jetons créés. Elle retourne 0 si pack_filler a échoué.

number_conditioner.c>

Ce fichier a un rôle très précis et crucial : il prend le nombre brut sous forme de chaîne de caractères (ex: "1234567") et le transforme en une structure de données plus facile à manipuler pour le reste du programme : un tableau de nombres entiers, où chaque nombre représente un "paquet" de trois chiffres.

Voici le détail de chaque fonction.

int ft_by_ten_multiplier(int n)

Objectif : Cette fonction est un simple utilitaire mathématique. Elle calcule une puissance de 10. Par exemple, si vous lui donnez 3, elle calcule 10 * 10 * 10 et retourne 1000.

Paramètre :

int n : L'exposant à appliquer à 10 (le nombre de fois où l'on doit multiplier par 10).

Fonctionnement :

Elle initialise une variable result à 1.

Elle effectue une boucle n fois.

À chaque tour de boucle, elle multiplie result par 10.

Valeur de retour : Le résultat du calcul (10^n).

Note : Curieusement, cette fonction n'est pas utilisée dans le reste du code que tu as fourni. La logique de multiplication par 10 dans ft_pack_splitter est faite manuellement (multi = multi * 10). Cette fonction a peut-être été écrite au début du développement puis remplacée.

int ft_strlen(char *str_nbr)

Objectif : C'est une version personnalisée de la fonction standard strlen, qui calcule la longueur d'une chaîne de caractères. Elle a une particularité importante : elle intègre une limite de taille spécifique au projet.

Paramètre :

char *str_nbr : La chaîne de caractères dont on veut connaître la longueur.

Fonctionnement :

Elle parcourt la chaîne caractère par caractère, comptant le nombre de caractères jusqu'à atteindre le caractère de fin de chaîne (\0).

Pendant le comptage, elle vérifie si la longueur dépasse 39. 39 chiffres correspond au "décillion" dans le dictionnaire fourni. Si le nombre est plus grand, le programme considère que c'est une erreur.

Si la longueur dépasse 39, elle affiche le message "Dict Error\n" et retourne 0 pour signaler l'échec.

Valeur de retour :

La longueur de la chaîne si elle est inférieure ou égale à 39.

0 si la chaîne est trop longue.

int ft_pack_splitter(char *str_nbr, int *str_idx, int end_idx)

Objectif : C'est le cœur du découpage. Cette fonction extrait un seul "paquet" (un groupe de 1 à 3 chiffres) de la chaîne de caractères et le convertit en une valeur entière.

Paramètres :

char *str_nbr : La chaîne de caractères complète du nombre (ex: "1234567").

int *str_idx : Un pointeur vers l'index de lecture actuel dans la chaîne. La fonction lit la chaîne à l'envers (de droite à gauche) et modifie cet index au fur et à mesure.

int end_idx : L'index où la lecture de ce paquet doit s'arrêter.

Fonctionnement (exemple avec str_nbr="1234567", *str_idx=7, end_idx=4) :

La fonction va lire les caractères de l'index 6 à 4.

1ère boucle : Elle lit str_nbr[6] qui est '7'. Elle le convertit en entier (7) et l'ajoute à pack_val (qui vaut maintenant 7). L'index *str_idx passe à 6.

2ème boucle : Elle lit str_nbr[5] qui est '6'. Elle le convertit (6), le multiplie par 10 (60) et l'ajoute à pack_val (qui vaut maintenant 7 + 60 = 67). L'index *str_idx passe à 5.

3ème boucle : Elle lit str_nbr[4] qui est '5'. Elle le convertit (5), le multiplie par 100 (500) et l'ajoute à pack_val (qui vaut maintenant 67 + 500 = 567). L'index *str_idx passe à 4.

La boucle s'arrête car *str_idx a atteint end_idx.

Valeur de retour : La valeur numérique du paquet extrait (dans notre exemple, 567).

int pack_filler(char *str_nbr, int *packs, int *pmax)

Objectif : C'est la fonction principale de ce fichier. Elle orchestre tout le processus de découpage en appelant les autres fonctions pour transformer la chaîne de caractères entière en un tableau de paquets.

Paramètres :

char *str_nbr : Le nombre d'entrée sous forme de chaîne.

int *packs : Le tableau (qui est vide au début) qui sera rempli avec les valeurs numériques des paquets.

int *pmax : Un pointeur vers un entier. Cette fonction y écrira le nombre total de paquets qui ont été créés.

Fonctionnement (exemple avec str_nbr="1234567") :

Elle appelle ft_strlen pour obtenir la longueur (len = 7).

Elle initialise un index de lecture str_idx à 7 et un index pour le tableau packs à 0.

Elle entre dans une boucle qui continue tant qu'il reste des chiffres à lire (str_idx > 0).

1er tour : Elle appelle ft_pack_splitter pour lire les 3 derniers chiffres. ft_pack_splitter retourne 567. Elle stocke cette valeur : packs[0] = 567. L'index de lecture str_idx est maintenant à 4.

2ème tour : Elle appelle ft_pack_splitter pour lire les 3 chiffres suivants. La fonction retourne 234. Elle stocke : packs[1] = 234. L'index de lecture str_idx est maintenant à 1.

3ème tour : Elle appelle ft_pack_splitter pour lire le chiffre restant. La fonction retourne 1. Elle stocke : packs[2] = 1. L'index de lecture str_idx est maintenant à 0.

La boucle se termine. Elle met à jour la variable *pmax à 3 (car 3 paquets ont été créés).

Valeur de retour : Le nombre de paquets créés. Le résultat principal est stocké dans le tableau packs qui est passé par adresse : {567, 234, 1}.

dict_packs_conditioner.c>

Ce fichier est le spécialiste de la "micro-gestion". Son unique but est de prendre un seul paquet de trois chiffres (comme 567 ou 17 ou 8) et de le décomposer en une séquence de "jetons" (tokens) qui correspondent directement aux entrées du dictionnaire.

Par exemple, le nombre 567 n'existe pas dans le dictionnaire, mais 5, 100, 60 et 7 y sont. Ce fichier se charge de faire cette décomposition.

Voici le détail de chaque fonction.

void treat_hundreds(int pack, int *outputs, int *count)

Objectif : Gérer la partie "centaine" d'un nombre.

Paramètres :

int pack : Le paquet de 1 à 3 chiffres (ex: 567).

int *outputs : Le tableau où les jetons sont ajoutés.

int *count : Un pointeur vers le compteur du nombre total de jetons.

Fonctionnement :

Il calcule le chiffre des centaines en faisant une division entière : cent = pack / 100.

Si pack est 567, cent devient 5.

Si pack est 89, cent devient 0.

Si cent est plus grand que 0, cela signifie qu'il y a bien une centaine à traiter.

Il ajoute alors deux jetons au tableau outputs :

Le jeton pour le chiffre lui-même (ex: 5).

Le jeton pour le mot "cent" (qui est 100 dans le dictionnaire).

Il incrémente *count deux fois, une fois pour chaque jeton ajouté.

Exemple : Pour pack = 567, cette fonction ajoute les jetons {5, 100} à la liste. Pour pack = 89, elle ne fait rien.

void treat_tens(int pack, int *outputs, int *count)

Objectif : Gérer la partie "dizaine" d'un nombre (les deux derniers chiffres). C'est la partie la plus complexe à cause des cas particuliers comme "onze", "douze", etc.

Paramètres : Identiques aux autres fonctions.

Fonctionnement :

Il isole le chiffre des dizaines : dec = (pack / 10) % 10.

Il isole le chiffre des unités : unit = pack % 10.

Cas 1 : Les nombres de 10 à 19. Si le chiffre des dizaines (dec) est 1, il faut créer un seul jeton pour le nombre entier (ex: 11 pour "onze", 17 pour "dix-sept"). Le jeton est calculé comme 10 + unit.

Cas 2 : Les autres dizaines. Si le chiffre des dizaines (dec) est de 2 ou plus, il faut créer un jeton pour la dizaine seule (ex: 20 pour "vingt", 80 pour "quatre-vingt"). Le jeton est calculé comme dec * 10.

Si un jeton est créé, il est ajouté au tableau outputs et *count est incrémenté.

Exemple : Pour pack = 567, dec est 6. La fonction ajoute le jeton {60}. Pour pack = 211, dec est 1. La fonction ajoute le jeton {11}. Pour pack = 305, dec est 0 et la fonction ne fait rien.

void treat_units(int pack, int *outputs, int *count)

Objectif : Gérer la partie "unité" d'un nombre.

Paramètres : Identiques aux autres fonctions.

Fonctionnement :

Il isole le chiffre des dizaines (dec) et des unités (unit).

Il ajoute le jeton de l'unité uniquement si deux conditions sont remplies :

Le chiffre des unités est supérieur à zéro (unit > 0). On n'écrit pas "zéro".

Le chiffre des dizaines n'est pas 1 (dec != 1). C'est la condition la plus importante. Si la dizaine était 1, le nombre a déjà été traité en entier par treat_tens (ex: pour "onze", on ne veut pas rajouter "un").

Si les conditions sont remplies, il ajoute le jeton de l'unité (ex: 7) au tableau outputs et incrémente *count.

Exemple : Pour pack = 567, dec est 6 et unit est 7. Les conditions sont remplies, la fonction ajoute le jeton {7}. Pour pack = 211, dec est 1. La condition dec != 1 est fausse, donc la fonction ne fait rien.

void packsofthree_emitter(int pack, int *outputs, int *count)

Objectif : C'est la fonction "chef d'orchestre" de ce fichier. Elle prend un paquet de trois chiffres et appelle les fonctions spécialisées dans le bon ordre pour le décomposer entièrement.

Paramètres : Identiques aux autres fonctions.

Fonctionnement :

D'abord, une sécurité : si le paquet vaut 0, la fonction ne fait rien et s'arrête immédiatement.

Puis, elle appelle les trois fonctions de traitement dans un ordre logique et immuable :

treat_hundreds(pack, ...)

treat_tens(pack, ...)

treat_units(pack, ...)

Exemple global : Si packsofthree_emitter est appelée avec pack = 567 :

treat_hundreds ajoute {5, 100}. La liste de jetons est {5, 100}.

treat_tens ajoute {60}. La liste est maintenant {5, 100, 60}.

treat_units ajoute {7}. La liste finale des jetons pour ce paquet est {5, 100, 60, 7}.

token_reader.c>

Ce fichier est le "traducteur" du projet. Son unique responsabilité est de prendre un jeton numérique (par exemple 100 ou -2 pour "million") et de rechercher dans le contenu du dictionnaire la ligne correspondante pour en extraire le mot en toutes lettres (par exemple "hundred" ou "million"). Il ne sait rien de la logique des nombres, il ne fait que de la recherche textuelle.

Voici le détail de chaque fonction.

int read_file(int *i, char *buffer)

Objectif : Lire un nombre au début d'une ligne dans le buffer. Le nom read_file est un peu trompeur, car il ne lit pas un fichier sur le disque, mais bien depuis un buffer (un tableau de caractères) déjà en mémoire.

Paramètres :

int *i : Un pointeur vers l'index de lecture actuel dans le buffer. La fonction va faire avancer cet index à mesure qu'elle lit les chiffres.

char *buffer : Le contenu complet du fichier dictionnaire.

Fonctionnement :

Il commence à la position *i et lit les caractères tant que ce sont des chiffres ('0' à '9').

Il les convertit et les assemble pour former un nombre entier. Par exemple, s'il lit les caractères '1', '0', '0', il construit la valeur 100.

Il met à jour l'index *i pour qu'il pointe juste après le dernier chiffre lu.

Valeur de retour : Le nombre entier qui a été lu (ex: 100).

int copy_word(int *i, char *buffer, int len, char *number)

Objectif : Copier la partie "mot" d'une ligne du dictionnaire. Par exemple, sur la ligne 100: hundred, cette fonction est appelée après que le 100: a été traité pour extraire "hundred".

Paramètres :

int *i : Le pointeur vers l'index, qui est maintenant positionné au début du mot.

char *buffer : Le contenu du dictionnaire.

int len : La longueur du buffer, pour ne pas lire au-delà.

char *number : Le tableau de destination où le mot sera copié (le nom du paramètre number est un peu confus, word_buffer serait plus clair).

Fonctionnement :

Il copie les caractères un par un depuis buffer vers number jusqu'à ce qu'il rencontre une fin de ligne (\n) ou la fin du buffer.

Il ajoute le caractère nul (\0) à la fin de number pour en faire une chaîne de caractères valide en C.

Valeur de retour : Retourne toujours 1, probablement pour signifier "succès".

int match_key_numeric(int token, char *buffer, int len, int *i)

Objectif : Vérifier si le début de la ligne actuelle correspond à un jeton numérique simple (positif ou nul).

Paramètres :

int token : Le jeton numérique que l'on recherche (ex: 20).

char *buffer, int len, int *i : Les paramètres habituels pour le parsing du buffer.

Fonctionnement :

Il utilise read_file pour lire le nombre au début de la ligne.

Il compare ce nombre lu au token recherché.

Si les deux sont identiques, il fait avancer l'index *i pour sauter les espaces et le caractère : qui suivent le nombre.

Il retourne 1 si ça correspond, 0 sinon.

int match_key_scale(int token, char *buffer, int len, int *i)

Objectif : C'est une fonction de correspondance spéciale pour les jetons d'échelle (mille, million...), qui sont représentés par des nombres négatifs. Elle ne vérifie pas la valeur exacte, mais la "forme" du nombre (un 1 suivi du bon nombre de zéros).

Paramètres :

int token : Le jeton d'échelle négatif (ex: -1 pour mille, -2 pour million).

Fonctionnement :

Il parcourt les chiffres au début de la ligne.

Il vérifie que le premier chiffre est un '1' et que tous les suivants sont des '0'.

Il compte le nombre total de chiffres (digits).

Il utilise la formule digits != 1 + 3 * (-token) pour vérifier si le nombre de zéros est correct.

Pour "mille" (token = -1), la formule donne 1 + 3*1 = 4 chiffres (1000).

Pour "million" (token = -2), la formule donne 1 + 3*2 = 7 chiffres (1000000).

Si la forme est correcte, il retourne 1, sinon 0.

int token_reader(int token, char *number, char *buffer, int len)

Objectif : C'est la fonction principale du fichier. Elle orchestre la recherche complète d'un jeton dans le dictionnaire.

Paramètres :

int token : Le jeton à trouver (peut être positif ou négatif).

char *number : Le buffer où stocker le mot trouvé.

char *buffer, int len : Le contenu du dictionnaire.

Fonctionnement :

Il parcourt tout le buffer du début à la fin, ligne par ligne.

Pour chaque ligne, il vérifie le type de jeton :

Si token >= 0, il appelle match_key_numeric.

Si token < 0, il appelle match_key_scale.

Si l'une de ces fonctions retourne 1 (une correspondance a été trouvée) :

Il appelle immédiatement copy_word pour extraire le mot correspondant.

Il retourne 1 pour indiquer que la traduction a réussi.

Si la ligne ne correspond pas, il avance son index jusqu'au début de la ligne suivante et recommence.

S'il arrive à la fin du buffer sans avoir trouvé de correspondance, il retourne 0 (échec).

printer.c>

Ce fichier joue le rôle de "maître d'œuvre final". Il ne fait aucun calcul. Sa seule mission est de prendre la liste de jetons numériques (qui est le produit final de tout le travail de conditionnement) et de l'utiliser pour générer la sortie textuelle lisible par un humain.

Voici le détail de chaque fonction.

int str_len(char *list)

Objectif : C'est une fonction utilitaire basique, une réimplémentation de la fonction standard strlen. Elle calcule le nombre de caractères dans une chaîne.

Paramètre :

char *list : Un pointeur vers la chaîne de caractères dont on veut mesurer la longueur.

Fonctionnement :

Elle initialise un compteur i à 0.

Elle parcourt la chaîne caractère par caractère (list[i]) jusqu'à ce qu'elle trouve le caractère de fin de chaîne (\0).

À chaque caractère, elle incrémente le compteur i.

Valeur de retour : Le nombre total de caractères dans la chaîne (sans compter le \0 final).

int read_dict(char *buffer, int cap)

Objectif : Lire l'intégralité du fichier dictionnaire (numbers.dict) et charger son contenu en mémoire dans un grand tableau de caractères (un "buffer").

Paramètres :

char *buffer : Le tableau de caractères qui va recevoir le contenu du fichier.

int cap : La capacité (taille maximale) du buffer, pour éviter d'écrire au-delà de la mémoire allouée.

Fonctionnement :

open("numbers.dict", O_RDONLY) : Tente d'ouvrir le fichier en mode lecture seule (READ ONLY).

Si l'ouverture échoue (parce que le fichier n'existe pas ou qu'on n'a pas les droits), open retourne une valeur négative et la fonction renvoie 0 pour signaler une erreur.

read(fd, buffer, cap - 1) : Si l'ouverture réussit, elle lit le contenu du fichier et le copie dans buffer. Elle lit au maximum cap - 1 octets pour se garder une place pour le caractère de fin de chaîne.

Si la lecture échoue ou si le fichier est vide, read retourne 0 ou moins. La fonction ferme alors le fichier et renvoie 0.

buffer[len] = '\0'; : C'est une étape cruciale. Elle ajoute le caractère nul à la fin du contenu lu pour transformer le buffer en une chaîne de caractères valide en langage C.

close(fd) : Elle ferme le fichier pour libérer les ressources système.

Valeur de retour : Le nombre d'octets lus, ou 0 en cas d'erreur.

void print_tokens(int *token, int size, int i)

Objectif : C'est la fonction principale et la plus importante de ce fichier. Elle prend la liste finale des jetons, les traduit un par un en mots, et les imprime à l'écran dans le bon ordre.

Paramètres :

int *token : Le tableau contenant la séquence complète des jetons numériques (ex: pour "123", ce serait {1, 100, 20, 3}).

int size : Le nombre total de jetons dans le tableau token.

int i : Ce paramètre est ici inutile. Bien qu'il soit dans la signature de la fonction, il est immédiatement réinitialisé à 0 à l'intérieur. C'est probablement un vestige d'une version précédente du code.

Fonctionnement :

Elle déclare deux grands tableaux locaux : buffer pour contenir tout le dictionnaire, et number pour stocker le mot traduit d'un seul jeton.

Elle appelle read_dict pour charger le dictionnaire en mémoire. Si cela échoue, la fonction s'arrête prématurément.

Elle initialise une variable first = 1. C'est une astuce classique pour gérer l'espacement : on ne met pas d'espace avant le premier mot, mais on en met un avant tous les autres.

Elle entre dans une boucle qui parcourt chaque jeton du tableau token.

Dans la boucle (pour chaque jeton) :
a. Elle appelle token_reader(token[i], number, buffer, len). C'est l'appel clé : elle demande au "traducteur" (token_reader.c) de trouver le mot correspondant au jeton token[i] et de le copier dans le tableau number.
b. Si token_reader réussit (retourne vrai) :
i. Elle vérifie si ce n'est pas le premier mot (!first). Si c'est le cas, elle écrit un espace " ".
ii. Elle écrit le mot qui vient d'être trouvé : write(1, number, str_len(number)).
iii. Elle met first à 0, pour que tous les mots suivants soient précédés d'un espace.

Après la boucle : Elle écrit un caractère de saut de ligne (\n) pour terminer proprement l'affichage.

Valeur de retour : void (rien), car son travail est d'écrire directement sur la sortie standard (l'écran).

Audit du code>

Fichier	Statut	Problèmes Majeurs	Comment corriger
Makefile	**À corriger**	Référence à un fichier manquant (dict.caller.c).	Supprimer la référence.
rush-02.c	**Cassé**	La fonction main est vide et ne connecte rien. stock_name provoque un crash. atoi_char a un bug.	Réécrire main pour appeler nbr_from_str puis print_tokens. Supprimer stock_name.
conditioners_bridge.c	**Fonctionnel**	Aucun bug interne. Sert de pont logique.	Rien à corriger. main doit l'appeler.
number_conditioner.c	**Fonctionnel**	Semble correct.	Rien à corriger, supprimer eventuellement ft_by_ten_multiplier
dict_packs_conditioner.c	**Fonctionnel**	Semble correct.	Rien à corriger.
token_reader.c	**Fonctionnel**	Semble correct.	Rien à corriger.
printer.c	**Fonctionnel**	Logique saine, mais un paramètre i inutile.	Rien de critique, mais print_tokens peut être nettoyée.

Analyse Globale : Le Problème de Connexion

Le principal défaut n'est pas tant un bug dans une fonction isolée, mais une absence totale de flux de données dans le programme principal. Chaque "département" (conditionnement, lecture du dictionnaire, impression) fonctionne bien en interne, mais le "directeur" (la fonction main) ne leur transmet aucune information.

Le flux logique correct devrait être le suivant :

main (dans rush-02.c) reçoit le nombre en chaîne de caractères.

main appelle nbr_from_str (dans conditioners_bridge.c) pour transformer cette chaîne en une liste de jetons numériques (ex: {1, 100, 20, 3}).

main prend cette liste de jetons et la passe à print_tokens (dans printer.c), qui se charge de la traduction et de l'affichage.

Audit Détaillé Fichier par Fichier>

1. Makefile

Problème : La ligne SRC contient un fichier non fourni : dict.caller.c. Le projet ne compilera pas en l'état car ce fichier est manquant.

Solution : Le supprimer de la liste SRC si ce fichier n'est pas utilisé.

2. rush-02.c (Le Fichier le Plus Problématique)

main :

Problème Grave : Les appels à atoi(argv[1]) et atoi(argv[2]) sont inutiles. La fonction atoi retourne une valeur qui est immédiatement jetée. Le nombre fourni par l'utilisateur n'est jamais stocké ni utilisé.

Problème : Il n'y a aucun appel aux fonctions de conditioners_bridge.c ou printer.c. C'est le cœur du problème de déconnexion.

Problème : Le malloc pour gnrltab est suivi d'un free, mais gnrltab n'est jamais réellement utilisé de manière productive dans main.

atoi_char :

Problème/Bug Subtil : La ligne gnrltab[i] = str[i]; est incorrecte. i est l'index dans la chaîne source str, qui peut avoir des espaces au début. Si str est " 123", la fonction copiera '1' dans gnrltab[3], laissant les trois premiers caractères de gnrltab non initialisés. Il faut un deuxième index pour la destination qui commence à 0.

Problème : La fonction est déclarée comme retournant un int mais n'a pas de return sur tous les chemins de code.

stock_name :

Problème Critique : Cette fonction est cassée. char **file; déclare un pointeur non initialisé. La ligne **file = *filename; tente d'écrire dans une zone mémoire aléatoire. C'est un crash (segmentation fault) garanti si cette fonction est appelée. Elle doit être supprimée ou entièrement réécrite.

3. conditioners_bridge.c

Qualité : Ce module est bien conçu. Il prend une chaîne de caractères en entrée (nbr_from_str) et produit correctement une liste de jetons entiers (outputs).

"Problème" : Ce n'est pas un bug, mais une question d'architecture. La fonction nbr_from_str retourne un int (le nombre de jetons), mais modifie surtout le tableau outputs passé en paramètre. C'est la sortie de ce module. La fonction qui l'appelle (main) est responsable de récupérer ce tableau et de le passer à l'étape suivante.

4. printer.c

Qualité : Ce module est également bien conçu et fonctionnel. Il fait ce qu'on attend de lui.

"Problème" (Architectural) : La fonction print_tokens est le point d'arrivée de la logique. Elle attend un tableau de jetons et sa taille, exactement ce que produit conditioners_bridge.c.

Défaut Mineur : Dans print_tokens, le paramètre int i est inutile car il est immédiatement réaffecté à 0. C'est du code "mort" qui peut être nettoyé.