Mkdir :

Pour créer une commande simple en C qui simule le comportement de la commande `mkdir`, vous pouvez utiliser la fonction système `mkdir` de la bibliothèque `sys/stat.h`. Voici un exemple de code :

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

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <directory>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // Crée le répertoire avec les permissions 0777 (rwxrwxrwx)
    if (mkdir(argv[1], 0777) == -1) {
        perror("mkdir");
        exit(EXIT_FAILURE);
    }

    printf("Répertoire créé avec succès : %s\n", argv[1]);

    return EXIT_SUCCESS;
}
```

Cet exemple suppose que vous souhaitez créer un seul répertoire à la fois. Si vous souhaitez étendre cela pour prendre en charge la création de plusieurs répertoires à la fois (comme `mkdir dir1 dir2 dir3`), vous devrez ajouter une boucle dans le code pour traiter chaque argument.

N'oubliez pas de compiler le code avec un compilateur C tel que `gcc` :

```bash
gcc votre_code.c -o votre_executable
```

Ensuite, vous pouvez exécuter le programme en utilisant :

```bash
./votre_executable nom_du_repertoire
```

Pwd

La commande `pwd` (print working directory) affiche le répertoire de travail actuel. Pour simuler cela en C, vous pouvez utiliser la fonction `getcwd` de la bibliothèque `unistd.h`. Voici un exemple de code :

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

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

   // Obtient le chemin du répertoire de travail
   if (getcwd(buffer, sizeof(buffer)) != NULL) {
      printf("Répertoire de travail actuel : %s\n", buffer);
   } else {
      perror("getcwd");
   }
```

```
    return 0;
}
```

Ce code utilise `getcwd` pour obtenir le chemin du répertoire de travail actuel et l'affiche à l'aide de `printf`. Compilez et exécutez le programme comme décrit précédemment pour voir le chemin du répertoire de travail actuel.

Ls

La commande `ls` (list) affiche les fichiers et répertoires dans le répertoire actuel. Pour simuler cela en C, vous pouvez utiliser la bibliothèque `dirent.h` pour parcourir les fichiers et répertoires du répertoire. Voici un exemple de code :

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

int main() {
    DIR *dir;
    struct dirent *entry;

    // Ouvre le répertoire actuel
    dir = opendir(".");

    if (dir == NULL) {
        perror("opendir");
        return 1;
    }

    // Parcourt les fichiers et répertoires du répertoire
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }
```

```
    // Ferme le répertoire
    closedir(dir);

    return 0;
}
```

Ce code ouvre le répertoire actuel en utilisant `opendir` et parcourt ensuite chaque entrée à l'aide de `readdir`, affichant le nom de chaque fichier ou répertoire avec `printf`. Compilez et exécutez le programme comme décrit précédemment pour voir la liste des fichiers et répertoires du répertoire actuel.

Cd

La commande `cd` (change directory) est généralement utilisée pour changer le répertoire de travail. Cependant, en C, le changement de répertoire nécessite l'utilisation de la fonction `chdir` de la bibliothèque `unistd.h`. Notez que le changement de répertoire dans un programme C affecte uniquement le programme en cours d'exécution et ne modifie pas le répertoire de travail du shell parent.

Voici un exemple simple de code C pour simuler la commande `cd` :

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

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <directory>\n", argv[0]);
        return 1;
    }

    // Change le répertoire de travail
    if (chdir(argv[1]) == -1) {
```

```
        perror("chdir");
        return 1;
    }

    // Affiche le nouveau répertoire de travail
    char buffer[1024];
    if (getcwd(buffer, sizeof(buffer)) != NULL) {
        printf("Répertoire de travail actuel : %s\n", buffer);
    } else {
        perror("getcwd");
    }

    return 0;
}
```

Cela prend un argument, le chemin du répertoire vers lequel vous voulez changer, utilise `chdir` pour changer le répertoire de travail, puis utilise `getcwd` pour afficher le nouveau répertoire de travail. Compilez et exécutez le programme comme décrit précédemment. Notez que le changement de répertoire affecte uniquement le programme en cours d'exécution. Après la fin du programme, le répertoire de travail du shell parent reste inchangé.

Mv

La commande `mv` (move) est utilisée pour déplacer ou renommer des fichiers et des répertoires. En C, vous pouvez utiliser les fonctions système `rename` pour déplacer ou renommer des fichiers. Voici un exemple simple de code C pour simuler la commande `mv` :

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

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc != 3) {
```

```
    fprintf(stderr, "Usage: %s <source> <destination>\n", argv[0]);
    return 1;
  }

  // Utilise la fonction rename pour déplacer ou renommer le fichier
  if (rename(argv[1], argv[2]) == -1) {
    perror("rename");
    return 1;
  }

  printf("Fichier déplacé ou renommé avec succès de %s vers %s\n", argv[1], argv[2]);

  return 0;
}
```

Cela prend deux arguments, le fichier source à déplacer ou renommer, et la destination du fichier. Le programme utilise ensuite la fonction `rename` pour effectuer l'opération. Compilez et exécutez le programme comme décrit précédemment.

Notez que la fonction `rename` est également utilisée pour renommer des fichiers. Si le fichier de destination existe déjà, il sera remplacé par le fichier source. Si vous avez besoin de fonctionnalités plus avancées, comme la gestion de répertoires, vous devrez implémenter une logique supplémentaire.

Rmdir

La commande `rmdir` est utilisée pour supprimer des répertoires vides. En C, vous pouvez utiliser la fonction système `rmdir` pour effectuer cette opération. Voici un exemple simple de code C pour simuler la commande `rmdir` :

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

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

```
    // Vérifie le nombre d'arguments
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <directory>\n", argv[0]);
        return 1;
    }

    // Utilise la fonction rmdir pour supprimer le répertoire
    if (rmdir(argv[1]) == -1) {
        perror("rmdir");
        return 1;
    }

    printf("Répertoire vidé avec succès : %s\n", argv[1]);

    return 0;
}
```

Cela prend un seul argument, le répertoire à supprimer, et utilise la fonction `rmdir` pour effectuer l'opération. Compilez et exécutez le programme comme décrit précédemment.

Remarque : La fonction `rmdir` ne supprime que les répertoires vides. Si le répertoire contient des fichiers ou d'autres répertoires, la suppression échouera. Pour supprimer des répertoires non vides, vous devrez utiliser une approche récursive pour vider le répertoire d'abord.

Rm

La commande `rm` est utilisée pour supprimer des fichiers ou des répertoires, et de manière récursive pour supprimer des répertoires et leur contenu. En C, vous pouvez utiliser la fonction système `remove` pour supprimer un fichier, et la fonction `rmdir` pour supprimer un répertoire vide. Pour supprimer des répertoires et leur contenu, vous pouvez utiliser une approche récursive.

Voici un exemple de code C simple pour simuler la commande `rm` :

```c
```

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

// Fonction récursive pour supprimer un répertoire et son contenu
void remove_directory(const char *path);

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file or directory>\n", argv[0]);
        return 1;
    }

    // Parcourt chaque argument
    for (int i = 1; i < argc; i++) {
        // Utilise la fonction remove pour supprimer un fichier
        if (remove(argv[i]) == -1) {
            // En cas d'échec, essaye de supprimer un répertoire
            perror("remove");

            // Supprime récursivement le répertoire et son contenu
            remove_directory(argv[i]);
        } else {
            printf("Fichier supprimé avec succès : %s\n", argv[i]);
        }
    }

    return 0;
```

```
}

void remove_directory(const char *path) {
    DIR *dir;
    struct dirent *entry;

    // Ouvre le répertoire
    dir = opendir(path);

    if (dir == NULL) {
        perror("opendir");
        return;
    }

    // Parcourt les fichiers et répertoires du répertoire
    while ((entry = readdir(dir)) != NULL) {
        // Ignore les entrées "." et ".."
        if (strcmp(entry->d_name, ".") != 0 && strcmp(entry->d_name, "..") != 0) {
            // Construit le chemin complet de l'entrée
            char entry_path[1024];
            snprintf(entry_path, sizeof(entry_path), "%s/%s", path, entry->d_name);

            // Si c'est un répertoire, supprime récursivement
            if (entry->d_type == DT_DIR) {
                remove_directory(entry_path);
            } else {
                // Sinon, utilise la fonction remove pour supprimer le fichier
                if (remove(entry_path) == -1) {
                    perror("remove");
                } else {
                    printf("Fichier supprimé avec succès : %s\n", entry_path);
```

```
            }
        }
    }
}

// Ferme le répertoire
closedir(dir);

// Supprime le répertoire lui-même une fois qu'il est vide
if (rmdir(path) == -1) {
    perror("rmdir");
} else {
    printf("Répertoire vidé avec succès : %s\n", path);
}
}
```

Ce code prend en charge la suppression de fichiers et de répertoires, et supprime récursivement les répertoires et leur contenu. Compilez et exécutez le programme comme décrit précédemment. Notez que la suppression récursive de répertoires nécessite une approche soigneuse pour éviter la suppression accidentelle de fichiers importants. Utilisez cette fonction avec précaution.

Touch

La commande `touch` est utilisée pour créer des fichiers vides ou mettre à jour les horodatages de modification et d'accès d'un fichier existant. En C, vous pouvez utiliser la fonction `open` avec le flag `O_CREAT` pour créer un fichier et `utime` pour mettre à jour les horodatages. Voici un exemple de code C pour simuler la commande `touch` :

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <utime.h>
#include <sys/stat.h>
```

```c
#include <unistd.h>

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file1> [file2 ...]\n", argv[0]);
        return 1;
    }

    // Parcourt chaque argument
    for (int i = 1; i < argc; i++) {
        // Utilise la fonction open avec le flag O_CREAT pour créer le fichier s'il n'existe pas
        int fd = open(argv[i], O_CREAT | O_WRONLY, 0644);

        if (fd == -1) {
            perror("open");
            return 1;
        }

        // Ferme le fichier
        close(fd);

        // Met à jour les horodatages du fichier
        struct utimbuf utime_buf;
        utime_buf.actime = utime_buf.modtime = time(NULL);

        if (utime(argv[i], &utime_buf) == -1) {
            perror("utime");
            return 1;
        }
```

```
        printf("Fichier créé ou horodatages mis à jour avec succès : %s\n", argv[i]);
    }

    return 0;
}
```

Ce code crée des fichiers vides pour chaque argument fourni et met à jour les horodatages pour refléter le temps actuel. Compilez et exécutez le programme comme décrit précédemment.

Notez que la fonction `utime` est utilisée pour mettre à jour les horodatages. Certains systèmes d'exploitation peuvent utiliser `utimes` ou `utimensat` à la place, en fonction de la disponibilité des fonctions sur la plateforme. Vous pouvez adapter le code en conséquence si nécessaire.

Cat

La commande `cat` est utilisée pour concaténer et afficher le contenu des fichiers. En C, vous pouvez utiliser la fonction `open` pour ouvrir les fichiers et `read` pour lire leur contenu. Voici un exemple de code C simple pour simuler la commande `cat` :

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

#define BUFFER_SIZE 4096

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file1> [file2 ...]\n", argv[0]);
        return 1;
    }
```

```
// Parcourt chaque argument
for (int i = 1; i < argc; i++) {
    // Utilise la fonction open pour ouvrir le fichier
    int fd = open(argv[i], O_RDONLY);

    if (fd == -1) {
        perror("open");
        return 1;
    }

    // Lit et affiche le contenu du fichier
    char buffer[BUFFER_SIZE];
    ssize_t bytesRead;

    while ((bytesRead = read(fd, buffer, sizeof(buffer))) > 0) {
        if (write(STDOUT_FILENO, buffer, bytesRead) == -1) {
            perror("write");
            close(fd);
            return 1;
        }
    }

    // Vérifie les erreurs de lecture
    if (bytesRead == -1) {
        perror("read");
        close(fd);
        return 1;
    }

    // Ferme le fichier
    close(fd);
```

```
        // Ajoute une ligne vide entre les fichiers (si plusieurs fichiers sont fournis)
        if (i < argc - 1) {
            if (write(STDOUT_FILENO, "\n", 1) == -1) {
                perror("write");
                return 1;
            }
        }
    }

    return 0;
}
```

Ce code lit le contenu de chaque fichier spécifié en argument et l'affiche sur la sortie standard (`stdout`). Compilez et exécutez le programme comme décrit précédemment.

Notez que la gestion d'erreurs est effectuée pour détecter les éventuelles erreurs lors de l'ouverture, la lecture ou l'écriture des fichiers.

Head

Tail

More

Less

File

Cp

La commande `cp` est utilisée pour copier des fichiers et des répertoires. En C, pour copier un fichier, vous pouvez utiliser la fonction `open` pour ouvrir le fichier source et la fonction `creat` ou `open` avec le flag `O_WRONLY | O_CREAT | O_TRUNC` pour créer le fichier de destination. Ensuite, vous pouvez utiliser la fonction `read` pour lire le contenu du fichier source et `write` pour écrire dans le fichier de destination.

Voici un exemple simple de code C pour simuler la commande `cp` :

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

#define BUFFER_SIZE 4096

void copy_file(const char *source, const char *destination);

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source> <destination>\n", argv[0]);
        return 1;
    }

    // Copie le fichier source vers le fichier de destination
    copy_file(argv[1], argv[2]);

    return 0;
}

void copy_file(const char *source, const char *destination) {
    // Ouvre le fichier source en lecture
    int source_fd = open(source, O_RDONLY);

    if (source_fd == -1) {
        perror("open");
        exit(1);
    }
```

```
// Ouvre ou crée le fichier de destination en écriture
int dest_fd = open(destination, O_WRONLY | O_CREAT | O_TRUNC, 0644);

if (dest_fd == -1) {
    perror("open");
    close(source_fd);
    exit(1);
}

// Copie le contenu du fichier source vers le fichier de destination
char buffer[BUFFER_SIZE];
ssize_t bytesRead;

while ((bytesRead = read(source_fd, buffer, sizeof(buffer))) > 0) {
    if (write(dest_fd, buffer, bytesRead) == -1) {
        perror("write");
        close(source_fd);
        close(dest_fd);
        exit(1);
    }
}

// Vérifie les erreurs de lecture
if (bytesRead == -1) {
    perror("read");
    close(source_fd);
    close(dest_fd);
    exit(1);
}
```

```
    // Ferme les fichiers
    close(source_fd);
    close(dest_fd);

    printf("Copie réussie de %s vers %s\n", source, destination);
}
```

Ce code copie le contenu du fichier source vers le fichier de destination. Compilez et exécutez le programme comme décrit précédemment. Notez que si le fichier de destination existe déjà, son contenu sera écrasé.

Cmp

La commande `cmp` est utilisée pour comparer deux fichiers byte par byte et afficher la première paire de différences rencontrée. En C, vous pouvez utiliser la fonction `open` pour ouvrir les fichiers, `read` pour lire leurs contenus et comparer les données pour détecter des différences.

Voici un exemple simple de code C pour simuler la commande `cmp` :

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

#define BUFFER_SIZE 4096

int compare_files(const char *file1, const char *file2);

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <file1> <file2>\n", argv[0]);
```

```
        return 1;
    }

    // Compare les fichiers
    int result = compare_files(argv[1], argv[2]);

    if (result == 0) {
        printf("Les fichiers %s et %s sont identiques.\n", argv[1], argv[2]);
    } else if (result == 1) {
        printf("Les fichiers %s et %s diffèrent à la position %ld.\n", argv[1], argv[2], (long)result);
    } else {
        perror("Erreur lors de la comparaison des fichiers");
        return 1;
    }

    return 0;
}

int compare_files(const char *file1, const char *file2) {
    // Ouvre le premier fichier en lecture
    int fd1 = open(file1, O_RDONLY);

    if (fd1 == -1) {
        perror("open");
        return -1;
    }

    // Ouvre le deuxième fichier en lecture
    int fd2 = open(file2, O_RDONLY);

    if (fd2 == -1) {
```

```
    perror("open");
    close(fd1);
    return -1;
}

// Compare le contenu des fichiers
char buffer1[BUFFER_SIZE];
char buffer2[BUFFER_SIZE];
ssize_t bytesRead1, bytesRead2;
off_t position = 0;

do {
    bytesRead1 = read(fd1, buffer1, sizeof(buffer1));
    bytesRead2 = read(fd2, buffer2, sizeof(buffer2));

    if (bytesRead1 != bytesRead2 || memcmp(buffer1, buffer2, bytesRead1) != 0) {
        // Les fichiers diffèrent à la position actuelle
        close(fd1);
        close(fd2);
        return position;
    }

    position += bytesRead1;

} while (bytesRead1 > 0);

// Les fichiers sont identiques
close(fd1);
close(fd2);

return 0;
```

}

```

Ce code ouvre les deux fichiers spécifiés en argument, lit leurs contenus par blocs, et compare ces blocs pour détecter des différences. Si une différence est trouvée, le programme affiche la position où la première différence a été détectée. Si les fichiers sont identiques, il retourne 0. Si une erreur se produit, il retourne -1. Compilez et exécutez le programme comme décrit précédemment.

## Gerp

La commande `grep` est utilisée pour rechercher des motifs (patterns) dans un fichier ou un flux de texte. En C, vous pouvez utiliser des fonctions telles que `fopen`, `fgets`, et `strstr` pour effectuer des opérations similaires.

Voici un exemple simple de code C pour simuler la commande `grep` :

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

#define MAX_LINE_LENGTH 1024

void grep(const char *pattern, const char *filename);

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <pattern> <file>\n", argv[0]);
        return 1;
    }

    // Recherche le motif dans le fichier
    grep(argv[1], argv[2]);
```
```

```
    return 0;
}

void grep(const char *pattern, const char *filename) {
    // Ouvre le fichier en lecture
    FILE *file = fopen(filename, "r");

    if (file == NULL) {
        perror("fopen");
        exit(1);
    }

    char line[MAX_LINE_LENGTH];

    // Lit chaque ligne du fichier
    while (fgets(line, sizeof(line), file) != NULL) {
        // Recherche le motif dans la ligne
        if (strstr(line, pattern) != NULL) {
            // Affiche la ligne si le motif est trouvé
            printf("%s", line);
        }
    }

    // Ferme le fichier
    fclose(file);
}
```

Ce code ouvre le fichier spécifié en argument, lit chaque ligne, recherche le motif donné dans chaque ligne, et affiche les lignes contenant le motif. Compilez et exécutez le programme comme décrit précédemment.

Notez que cet exemple ne couvre pas toutes les fonctionnalités de la commande `grep` comme les options de ligne de commande étendues ou la prise en charge des expressions régulières. Pour des fonctionnalités plus avancées, vous pourriez envisager d'utiliser des bibliothèques telles que PCRE (Perl Compatible Regular Expressions) ou bien d'implémenter une logique plus complexe pour gérer ces cas.

Sort

La commande `sort` est utilisée pour trier les lignes d'un fichier ou les lignes provenant de l'entrée standard. En C, vous pouvez utiliser des fonctions telles que `fopen`, `fgets`, et `qsort` pour effectuer des opérations similaires.

Voici un exemple simple de code C pour simuler la commande `sort` :

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

#define MAX_LINE_LENGTH 1024
#define MAX_LINES 1000

// Fonction de comparaison pour qsort
int compare_lines(const void *a, const void *b);

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        return 1;
    }

    // Ouvre le fichier en lecture
    FILE *file = fopen(argv[1], "r");
```

```
if (file == NULL) {
    perror("fopen");
    exit(1);
}

char **lines = malloc(MAX_LINES * sizeof(char *));
char line[MAX_LINE_LENGTH];
int num_lines = 0;

// Lit chaque ligne du fichier
while (fgets(line, sizeof(line), file) != NULL) {
    // Supprime le caractère de nouvelle ligne à la fin de la ligne
    line[strcspn(line, "\n")] = '\0';

    // Alloue de la mémoire pour la ligne et la copie
    lines[num_lines] = strdup(line);

    if (lines[num_lines] == NULL) {
        perror("strdup");
        exit(1);
    }

    num_lines++;

    // Vérifie si le nombre maximal de lignes est atteint
    if (num_lines >= MAX_LINES) {
        fprintf(stderr, "Nombre maximal de lignes atteint. Augmentez MAX_LINES si nécessaire.\n");
        exit(1);
    }
}
```

```
    // Ferme le fichier
    fclose(file);

    // Trie les lignes
    qsort(lines, num_lines, sizeof(char *), compare_lines);

    // Affiche les lignes triées
    for (int i = 0; i < num_lines; i++) {
        printf("%s\n", lines[i]);
        free(lines[i]); // Libère la mémoire allouée pour chaque ligne
    }

    // Libère la mémoire allouée pour le tableau de pointeurs
    free(lines);

    return 0;
}

// Fonction de comparaison pour qsort
int compare_lines(const void *a, const void *b) {
    return strcmp(*(const char **)a, *(const char **)b);
}
```

Ce code lit les lignes du fichier spécifié en argument, les stocke dans un tableau de pointeurs, trie les lignes à l'aide de `qsort`, puis les affiche. Compilez et exécutez le programme comme décrit précédemment.

Notez que cet exemple suppose que le nombre maximal de lignes à trier est défini par `MAX_LINES`. Vous pouvez ajuster cette valeur en fonction de vos besoins.

Tar

Tar

La commande `tar` est utilisée pour créer des archives tar et pour extraire ou afficher leur contenu. En C, vous pouvez utiliser la bibliothèque `libtar` pour effectuer des opérations similaires. Notez que cette bibliothèque n'est pas standard et doit être installée séparément.

Voici un exemple simple de code C utilisant `libtar` pour créer une archive tar avec des fichiers :

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

int create_tar(const char *tar_filename, const char **files, int num_files);

int main() {
    const char *tar_filename = "example.tar";
    const char *files[] = {"file1.txt", "file2.txt", "file3.txt"};
    int num_files = sizeof(files) / sizeof(files[0]);

    if (create_tar(tar_filename, files, num_files) != 0) {
        fprintf(stderr, "Erreur lors de la création de l'archive tar.\n");
        return 1;
    }

    printf("Archive tar créée avec succès : %s\n", tar_filename);

    return 0;
}

int create_tar(const char *tar_filename, const char **files, int num_files) {
    Tar *tar;
```

```
    int ret;

    // Initialise la structure Tar
    tar = tar_open(tar_filename, NULL, O_WRONLY | O_CREAT, 0644, TAR_GNU);

    if (tar == NULL) {
        perror("tar_open");
        return -1;
    }

    // Ajoute chaque fichier à l'archive
    for (int i = 0; i < num_files; i++) {
        ret = tar_append_tree(tar, files[i], files[i]);

        if (ret != 0) {
            fprintf(stderr, "Erreur lors de l'ajout du fichier %s à l'archive.\n", files[i]);
            tar_close(tar);
            return -1;
        }
    }

    // Ferme l'archive
    tar_close(tar);

    return 0;
}
```

Dans cet exemple, nous utilisons la bibliothèque `libtar` pour créer une archive tar (`example.tar`) contenant les fichiers spécifiés dans le tableau `files`. Assurez-vous d'avoir `libtar` installée sur votre système avant de compiler et d'exécuter ce code.

Compilez le code avec la commande :

```bash
gcc -o create_tar create_tar.c -ltar
```

Ensuite, exécutez le programme généré :

```bash
./create_tar
```

Cela devrait créer un fichier `example.tar` contenant les fichiers spécifiés. Vous pouvez également explorer la bibliothèque `libtar` pour des fonctionnalités plus avancées, comme l'extraction d'une archive ou la liste de son contenu.

Ln

La commande `ln` est utilisée pour créer des liens entre des fichiers. En C, vous pouvez utiliser la fonction `link` pour créer un lien symbolique entre deux fichiers.

Voici un exemple simple de code C pour simuler la création d'un lien symbolique, similaire à la commande `ln` :

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

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source_file> <link_name>\n", argv[0]);
        return 1;
```

```
    }

    // Crée un lien symbolique
    if (symlink(argv[1], argv[2]) == -1) {
        perror("symlink");
        return 1;
    }

    printf("Lien symbolique créé avec succès: %s -> %s\n", argv[2], argv[1]);

    return 0;
}
```

Ce code crée un lien symbolique du fichier spécifié en premier argument (`source_file`) vers le nom de lien spécifié en deuxième argument (`link_name`). Compilez et exécutez le programme comme décrit précédemment.

Notez que ce code crée un lien symbolique. Si vous souhaitez créer un lien dur, vous pouvez utiliser la fonction `link` au lieu de `symlink`. Cependant, la création de liens durs n'est possible que sur le même système de fichiers.

Ln –s

La commande `ln -s` est utilisée pour créer des liens symboliques dans un système de fichiers Unix/Linux. Un lien symbolique est un pointeur vers un fichier ou un répertoire existant, et il peut pointer vers des emplacements situés sur différents systèmes de fichiers.

En C, vous pouvez utiliser la fonction `symlink` pour créer un lien symbolique entre deux fichiers ou répertoires. Voici un exemple de code C qui simule la création d'un lien symbolique en utilisant des arguments de ligne de commande similaires à `ln -s` :

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

```
int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <target> <link_name>\n", argv[0]);
        return 1;
    }

    // Crée un lien symbolique
    if (symlink(argv[1], argv[2]) == -1) {
        perror("symlink");
        return 1;
    }

    printf("Lien symbolique créé avec succès: %s -> %s\n", argv[2], argv[1]);

    return 0;
}
```

Dans cet exemple, `argv[1]` est le fichier ou le répertoire cible, et `argv[2]` est le nom du lien symbolique. Compilez et exécutez le programme comme décrit précédemment.

Lors de l'exécution du programme avec `./votre_programme fichier_cible lien_symbolique`, cela devrait créer un lien symbolique nommé `lien_symbolique` pointant vers le fichier ou le répertoire cible `fichier_cible`.

Chmod

La commande `chmod` est utilisée pour modifier les permissions d'accès aux fichiers dans les systèmes Unix/Linux. En C, vous pouvez utiliser la fonction `chmod` de la bibliothèque standard `<sys/stat.h>` pour accomplir des opérations similaires.

Voici un exemple de code C qui simule la fonctionnalité de la commande `chmod` en utilisant des arguments de ligne de commande :

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

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <mode> <file>\n", argv[0]);
        return 1;
    }

    // Convertit la chaîne de mode en un nombre octal
    int mode = strtol(argv[1], NULL, 8);

    // Modifie les permissions du fichier
    if (chmod(argv[2], mode) == -1) {
        perror("chmod");
        return 1;
    }

    printf("Les permissions du fichier %s ont été modifiées avec succès.\n", argv[2]);

    return 0;
}
```

Dans cet exemple, `argv[1]` est la chaîne de mode octal (par exemple, "755") que vous souhaitez appliquer au fichier spécifié en `argv[2]`. Compilez et exécutez le programme comme décrit précédemment.

Lors de l'exécution du programme avec `./votre_programme 755 fichier`, cela devrait modifier les permissions du fichier `fichier` selon le mode spécifié (dans cet exemple, 755).

Umask

La commande `umask` est utilisée pour définir ou afficher le masque de création de fichiers par défaut sur les systèmes Unix/Linux. En C, le masque de création de fichiers par défaut peut être manipulé en utilisant la fonction `umask` de la bibliothèque `<sys/stat.h>`.

Voici un exemple simple de code C qui simule la fonctionnalité de la commande `umask` :

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

int main(int argc, char *argv[]) {
    // Affiche le masque de création de fichiers actuel
    mode_t current_mask = umask(0);
    printf("Le masque de création de fichiers actuel est : %o\n", current_mask);

    // Remet le masque de création de fichiers à sa valeur précédente
    umask(current_mask);

    return 0;
}
```

Dans cet exemple, le programme utilise `umask(0)` pour afficher le masque de création de fichiers actuel. La fonction `umask` renvoie également la valeur précédente du masque, ce qui peut être utilisé pour remettre le masque à sa valeur d'origine si nécessaire.

Compilez et exécutez le programme comme décrit précédemment.

Notez que l'utilisation de `umask` pour afficher le masque de création de fichiers actuel n'est pas une pratique standard, mais c'est un moyen de simuler le comportement de la commande `umask` dans un programme C.

Chown

La commande `chown` est utilisée pour changer le propriétaire et/ou le groupe d'un fichier ou d'un répertoire dans les systèmes Unix/Linux. En C, vous pouvez utiliser la fonction `chown` de la bibliothèque `<unistd.h>` pour accomplir des opérations similaires.

Voici un exemple de code C qui simule la fonctionnalité de la commande `chown` en utilisant des arguments de ligne de commande :

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

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc != 4) {
        fprintf(stderr, "Usage: %s <owner> <group> <file>\n", argv[0]);
        return 1;
    }

    // Convertit les arguments de propriétaire et groupe en identifiants numériques
    uid_t owner = atoi(argv[1]);
    gid_t group = atoi(argv[2]);

    // Modifie le propriétaire et le groupe du fichier
    if (chown(argv[3], owner, group) == -1) {
        perror("chown");
        return 1;
    }
```

```
    printf("Le propriétaire et le groupe du fichier %s ont été modifiés avec succès.\n", argv[3]);

    return 0;
}
```

Dans cet exemple, `argv[1]` est le nouvel identifiant du propriétaire, `argv[2]` est le nouvel identifiant du groupe, et `argv[3]` est le fichier ou répertoire dont le propriétaire et le groupe doivent être modifiés. Compilez et exécutez le programme comme décrit précédemment.

Lors de l'exécution du programme avec `./votre_programme nouvel_owner nouvel_group fichier`, cela devrait changer le propriétaire et le groupe du fichier spécifié en `fichier` en utilisant les identifiants spécifiés.

Chgrp

La commande `chgrp` est utilisée pour changer le groupe d'un fichier ou d'un répertoire dans les systèmes Unix/Linux. En C, vous pouvez utiliser la fonction `chown` de la bibliothèque `<unistd.h>` pour accomplir des opérations similaires, mais en spécifiant seulement le nouvel identifiant de groupe.

Voici un exemple de code C qui simule la fonctionnalité de la commande `chgrp` en utilisant des arguments de ligne de commande :

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

int main(int argc, char *argv[]) {
    // Vérifie le nombre d'arguments
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <new_group> <file>\n", argv[0]);
        return 1;
    }
```

```
    // Convertit l'argument de nouveau groupe en identifiant numérique de groupe
    gid_t new_group = atoi(argv[1]);

    // Modifie le groupe du fichier
    if (chown(argv[2], -1, new_group) == -1) {
        perror("chown");
        return 1;
    }

    printf("Le groupe du fichier %s a été modifié avec succès.\n", argv[2]);

    return 0;
}
```

Dans cet exemple, `argv[1]` est le nouvel identifiant de groupe et `argv[2]` est le fichier ou le répertoire dont le groupe doit être modifié. La fonction `chown` est utilisée avec un identifiant de propriétaire de `-1` pour indiquer que le propriétaire du fichier ne doit pas être modifié, seul le groupe doit être changé. Compilez et exécutez le programme comme décrit précédemment.

Lors de l'exécution du programme avec `./votre_programme nouvel_group fichier`, cela devrait changer le groupe du fichier spécifié en `fichier` en utilisant le nouvel identifiant de groupe spécifié.

Test

La commande `test` est utilisée dans les scripts shell pour évaluer des expressions conditionnelles. En C, vous n'avez pas une commande directe équivalente, mais vous pouvez utiliser des structures de contrôle de flux (comme les déclarations `if`) pour effectuer des évaluations conditionnelles.

Voici un exemple simple de code C qui simule une évaluation conditionnelle similaire à la commande `test` :

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

```
int main() {
    // Exemple : vérifie si deux nombres sont égaux
    int number1 = 5;
    int number2 = 5;

    // Utilise une structure de contrôle if pour évaluer la condition
    if (number1 == number2) {
        printf("Les nombres sont égaux.\n");
    } else {
        printf("Les nombres ne sont pas égaux.\n");
    }

    // Exemple : vérifie si une chaîne de caractères est vide
    char str[] = "Hello, World!";

    // Utilise une structure de contrôle if pour évaluer la condition
    if (str[0] == '\0') {
        printf("La chaîne de caractères est vide.\n");
    } else {
        printf("La chaîne de caractères n'est pas vide.\n");
    }

    return 0;
}
```

Ce code utilise des déclarations `if` pour évaluer différentes conditions. Vous pouvez ajuster les conditions en fonction de vos besoins. Notez que le type `bool` est utilisé pour représenter les valeurs de vérité (true/false).

Si vous avez une condition spécifique à laquelle vous pensez pour la commande `test`, n'hésitez pas à fournir des détails, et je pourrai vous aider à adapter le code en conséquence.