C++1y - File System

Le Technical Specification — File System propose d'ajouter des fonctionnalités de gestion des fichiers et répertoires en C++. Jusqu'à maintenant, il fallait passer par des bibliothèques externes, comme par exemple Boost.Filesystem ou Qt, ou utiliser les bibliothèques fournies par le système d'exploitation (POSIX sur Linux, Win32 sur Windows). Ce TS devrait être validé cette année par le comité, l'implémentation dans les compilateurs devrait suivre très rapidement.

Les classes et énumérations

Le draft N3803 est relativement simple à lire, puisqu'il n'y a pas de modification du langage, mais simplement des ajouts sous forme de bibliothèque. Ceux qui ont déjà utilisé une bibliothèque de gestion de fichiers retrouveront des éléments familiers. Du coup, je ne vais pas reprendre en détail le document (ce qui reviendrait en gros à le traduire), mais je vais donner des exemples de code pour répondre à des problématiques spécifiques. Pour ceux intéressés par les détails, je vais juste décrire l'organisation du draft.

Le document commence à définir un ensemble de notions généralistes sur les systèmes de fichiers : ce qu'est un fichier, un répertoire, un nom de fichier ou de répertoire, un lien, ainsi que les notions de chemin absolu, relatif ou canonique.

Ensuite, le document décrit les différents types :

- la classe `path`, qui représente un chemin vers un fichier ou répertoire ;
- la classe `filesystem_error` pour la gestion des erreurs ;
- l'énumération `file_type` pour lister les types (fichier, répertoire, lien, etc.) ;
- l'énumération `copy_options` pour lister les options possibles lors

d'une copie (écraser les fichiers existants, parcourir les sous-répertoires, suivre les liens symboliques, etc.) ;
- l'énumération `perms` pour lister les permissions d'un fichier (propriétaire du fichier, groupe d'utilisateurs, etc.) ;
- la classe `file_status` pour gérer le type de fichier et les permissions d'accès ;
- la classe `directory_entry` représente une entrée dans un répertoire, avec un chemin et les statuts du fichier ;
- la classe `directory_iterator` est un itérateur permettant de parcourir les éléments d'un répertoire ;
- la classe `recursive_directory_iterator` est un itérateur permettant de parcourir les éléments d'un répertoire et de ses sous-répertoires.

Pour terminer, le document définit l'ensemble des fonctions libres pour copier, renommer, déplacer ou supprimer des fichiers et répertoires, manipuler les chemins, tester le type et les accès des fichiers, créer des liens.

L'ensemble des fonctionnalités est accessible via l'en-tête `<filesystem>`.

La gestion des erreurs

Pour gérer les erreurs en C, il est classique d'utiliser un code d'erreur en retour de fonction. L'un des problèmes de cette approche est qu'il devient vite fastidieux d'ajouter du code pour tester toutes les erreurs et beaucoup de développeurs ne vérifient plus toutes les erreurs.

Le C++ ajoute une nouvelle approche pour la gestion des erreurs : les exceptions. Cette différence majeure par rapport au C fait qu'il est très dangereux de porter directement du code C en C++. Un code aussi simple que le suivant ne sera pas valide en C++ (risque de fuite mémoire).

```
Object p1 = new Object;
Object P2 = new Object;
```

```
if (!p1 || !p2) {
    delete p1;
    delete p2;
}
```

Il est donc conseillé en C++ d'utiliser des classes assurant le RAII : les pointeurs intelligents (`shared_ptr`, `weak_ptr`, `unique_ptr`), les conteneurs standards (`vector`, `list`, `map`, etc.) et plus généralement les outils de la STL (`string`, `stream`, etc). Je vous conseille la lecture de l'article Retour de fonctions ou exceptions ?

En complément, le C++11 introduit un nouveau mot-clé `noexcept` qui permet d'indiquer qu'une fonction ne retournera pas d'exception (c'est-à-dire ne lançant pas d'exception et n'appelant que des fonctions `noexcept`).

Les fonctions de *File System* existent en deux versions, l'une utilisant la gestion d'erreur par exception et l'autre par code d'erreur. Pour garder les signatures des fonctions similaires, le code d'erreur est passé comme argument (référence non constante) et non en retour de fonction.

Ainsi, pour tester l'existence d'un fichier, la fonction `exists` existe selon les deux signatures suivantes :

```
bool exists(const path& p);
bool exists(const path& p, error_code& ec) noexcept;
```

La première version ne prend pas de code d'erreur en argument, la seconde oui (et est donc déclarée en `noexcept`). Pour les utiliser :

```
// avec les exceptions
try {
    auto file_exists(R"(C:\Mon fichier.txt)");
} catch(...) {
    cerr << R"(Le fichier n'existe pas !)" << endl;
}

// avec code d'erreur
error_code ec;
auto file_exists(R"(C:\Mon fichier.txt)", ec);
```

```
if (ec) {
    cerr << R"(Le fichier n'existe pas !)" << endl;
}
```

(Je ne suis pas sûr du code pour tester le code d'erreur, le type `error_code` n'est pas défini).

Il est préférable d'utiliser la version avec exception par défaut.

Les fonctions libres

Beaucoup de fonctions prennent en paramètre un objet de type `path` ou de type `file_status`. Comme il existe un constructeur (non `explicit`) de `path` prenant en paramètre une chaîne, il est possible d'appeler les fonctions directement avec une chaîne.

```
auto my_filename = R"(C:\mon fichier.exe)";
auto my_path = path(my_filename );
auto my_status = status(my_path);
auto my_another_status = status(my_filename);
```

Connaître le répertoire courant

```
auto current = current_path();
```

Chemin relatif, absolu, canonique

Un chemin n'est pas représenté par une écriture unique, plusieurs chaînes de caractères peuvent correspondre à un même chemin (et donc au même fichier ou même répertoire) :

- le chemin relatif est le chemin défini à partir d'un emplacement (le plus souvent, l'application) ;
- le chemin absolu est défini à partir de la racine, avec ou sans définir le chemin de base ;
- le chemin canonique est le chemin absolu sans lien symbolique

ou élément `.` et `..`

Les fonctions `absolute` et `canonical` permettent d'obtenir les chemins absolu et canonique :

```
auto path_absolute = absolute(old_path);
auto path_absolute = absolute("my_file.cpp");
auto path_absolute = absolute("my_file.cpp", "/home/me");

auto path_canonical = canonical(old_path);
auto path_canonical = canonical("my_file.cpp");
auto path_canonical = canonical("my_file.cpp", "/home/me");
```

La fonction `system_complete` permet de retourner un chemin absolu à partir du répertoire courant ou de root.

```
auto p = system_complete(my_file);
    // est équivalent à :
auto p = absolute(my_file, current_path());
```

Statuts des fichiers

La fonction `statuts` permet de connaître le type et les permissions d'un fichier.

```
auto file_statuts = statuts(my_file);
auto perms = file_statuts.permissions();
if (file_statuts.type() = file_type.regular)
    cout << "Est un fichier régulier" << endl;
```

Ces fonctions sont similaires aux fonctions suivantes, permettant de tester directement le type d'un fichier.

```
auto b1 = exists(my_filename);
auto b2 = equivalent(my_filename, "mon_fichier.exe");

auto b3 = is_directory(my_filename); // est un répertoire ?
auto b4 = is_regular_file(my_filename); // est un fichier ?
auto b5 = is_symlink(my_filename); // est un lien symbolique
```

```
?
auto b6 = is_other(my_filename); // est autre chose qu'un
fichier, un répertoire et un lien ?

auto b7 = is_block_file(my_filename); // type block ?
auto b8 = is_character_file(my_filename); // type caractère
?
auto b9 = is_socket(m_filename); // type socket ?
auto b10 = is_fifo(my_filename); // type fifo ?
```

La fonction `statuts` peut émettre une exception `filesystem_error` si le type détecté est `file_type::none`. Attention, les types `file_type::not_found` et `file_type::unknown` ne sont pas considérés comme des erreurs et ne lancent pas d'exception (cela est considéré comme étant normal dans le fonctionnement des systèmes de fichiers).

Pour terminer, la fonction `status_known` permet de tester si un statut est connu (c'est-à-dire qu'il n'est pas de type `file_type::none`).

```
if (status_known(file_statuts))
    cout << "Statut connu" << endl;
```

La fonction `symlink_status` est similaire à statuts, mais retourne `file_type::symlink` si le chemin correspond à un lien symbolique.

```
auto file_statuts = symlink_status(my_file);
```

La date de la dernière modification

```
auto t = last_write_time(my_filename);
last_write_time(my_filename, chrono::system_clock::now());
```

Taille des fichiers

```
auto size = file_size(my_filename);
resize_file(my_filename, new_size);
```

```
auto b = is_empty(my_filename);
```

Il est également possible d'obtenir des informations sur le système de fichiers, avec la fonction `space`. Celle-ci prend en paramètre un chemin vers n'importe quel fichier du système de fichiers.

```
auto s = space(my_file);
cout << s.capacity << endl; // capacité
cout << s.free << endl; // espace libre
cout << s.available << endl; // espace
    // disponible, en fonction des autorisations d'accès
```

Gérer les permissions d'accès

Les permissions sont définies dans l'énumération `perms`, pour le propriétaire, le groupe et les autres :

- `none` (ne fait rien) ;
- Pour le propriétaire :
  - `owner_read` (lecture) ;
  - `owner_write` (écriture) ;
  - `owner_exec` (exécutable) ;
  - `owner_all` (tout) ;
- Pour le groupe :
  - `group_read` (lecture) ;
  - `group_write` (écriture) ;
  - `groupe_exec` (exécutable) ;
  - `group_all` (tout) ;
- Pour les autres :
  - `others_read` (lecture) ;
  - `others_write` (écriture) ;
  - `others_exec` (exécutable) ;
  - `others_all` (tout) ;
- `all` (toutes les permissions, pour tout le monde) ;
- Pour les identifiants :

- `set_uid` (user id) ;
- `set_gid` (group id) ;
- `sticky_bit` (système dépendant).

Deux valeurs supplémentaires permettent de sélectionner le mode de modification des permissions :

- `add_perms` : ajoute les permissions ;
- `remove_perms` : supprime les permissions ;
- aucun des deux : remplace les permissions.

Il existe également d'autres valeurs :

- `resolve_symlinks`
- `mask`
- `unknown`

```
permissions(my_file, perms(all));
    // donne les permissions à tout le monde

permissions(my_file, perms(others_all | remove_perms));
    // supprime toutes les permissions des autres

permissions(my_file, perms(other_read | add_perms));
    // ajoute l'autorisation en lecture pour les autres
```

Pour connaître les permissions :

```
auto perms = status.permissions()
```

Manipulation des fichiers et répertoires

Copie

Les options de copie sont définies dans l'énumération `copy_options` :

- pour `copy_file` :
  - `none` (ne rien faire) ;
  - `skip_existing` (ignorer les existants) ;
  - `overwrite_existing` (écraser les existants) ;
  - `update_existing` (mettre à jour les existants).
- pour les sous-répertoires :
  - `none` (ne rien faire) ;
  - `recursive` (parcourir les sous-répertoires).
- lien symbolique :
  - `none` (ne rien faire) ;
  - `copy_symlinks` (copier les liens symboliques) ;
  - `skip_symlinks` (passer les liens symboliques).
- type de copie :
  - `none` (ne rien faire) ;
  - `directories_only` (copier seulement les répertoires) ;
  - `create_symlinks` (créer des liens symboliques) ;
  - `create_hard_links` (créer des liens en dur).

La fonction `copy` permet de copier un fichier ou répertoire vers une destination :

```
copy("src_file.txt", "dest_file.txt");
copy("src_dir", "dest_dir");
copy("src_dir", "dest_dir", copy_options::recursive);
```

En complément, les fonctions suivantes permettent de copier uniquement les fichiers ou des liens symboliques :

```
copy_file("src", "dest");
copy_file("src", "dest", copy_options::recursive);

copy_symlink("src", "dest");
```

Suppression

```
remove("/home/me/my_file"); // supprimer un fichier
remove_all("/home/me/my_file"); // supprime les fichiers
d'un répertoire
```

Renommer

```
rename("/home/me/old_name", "/home/me/new_name");
```

Création des répertoires ou des liens symboliques

```
create_directory("/home/me/my_dir");
create_directories("/home/me/my_dir");
create_directory_symlink("/home/me/my_dir");
create_hard_link("/home/me/link");
create_symlink("/home/me/link");
```

La fonction `hard_link_count` permet de connaître le nombre de hard link d'un chemin :

```
cout << hard_link_count("/home/me/file");
```

La fonction `read_symlink` permet de récupérer le chemin pointé par un lien symbolique.

```
auto p = read_symlink(my_file);
```

Fichier et répertoire temporaire

Pour faciliter la création de fichiers temporaires, la fonction `unique_path` permet de créer un chemin qui n'existe pas, ce qui permet de créer un fichier sans supprimer ou modifier un fichier existant. Pour cela, la fonction prend une chaîne de caractères comme modèle et génère aléatoirement un nom de fichier. Les caractères `%` de la chaîne sont remplacés par une valeur hexadécimale. Par défaut, le modèle est `"---"`.

```
cout << unique_path() << endl;
    // peut générer par exemple : 25e9-a1f3-6b4b-671c
    // 2^64 valeurs possibles

cout << unique_path("test-%%%%%%%%%%.txt") << endl;
    // peut générer par exemple : test-0db7f2bf57a.txt
    // 2^44 valeurs possibles
```

La fonction `temp_directory_path` permet d'obtenir un répertoire existant pour enregistrer des fichiers temporaires. Le répertoire obtenu dépend de l'implémentation et peut correspondre par exemple à `"/tmp"` ou au retour de la fonction `GetTempPath`.

```
auto temp_path = temp_directory_path();
```

C++, C++1y