Boost.FileSystem : le système de fichiers de Boost

par Matthieu Brucher (http://matthieu-brucher.developpez.com/) (Blog)

Date de publication : 10/07/2006

Dernière mise à jour : 21/08/2006

Gérer un système de fichiers en C++, ce n'est à la base pas possible de manière portable. Maintenant, Boost.FileSystem propose un ensemble de classes à cet effet. Le but n'est pas de concurrencer des langages de scripts, mais fournir des classes portables lorsque le C++ a déjà été choisi.

Introduction

Cette bibliothèque doit être générée et des fichiers .lib/.a ou .dll/.so seront à inclure à la compilation et à l'exécution. Toutes les classes et les fonctions vivent dans le namespace **boost::filesystem**.

I - La classe path définie par "path.hpp"

La classe **path** est une classe décrivant un dossier ou un fichier. Elle peut être vue comme une liste de chaînes de caractères, les caractères de séparation étant ceux de la plateforme. On peut afficher une instance d'une telle classe sous plusieurs formes : la forme naturelle de **Boost.FileSystem** est d'utiliser '/', mais la forme native est aussi accessible, donc avec '\' sous Windows, '/' sous Linux, ':' sous Mac et d'autres formes sur d'autres plateformes.

Sous Linux et autres systèmes POSIX, une attention particulière devra être portée sur le fait qu'il n'y a qu'un seul dossier racine - "/" sous Linux -, tandis que d'autres OS en ont plusieurs - sous Windows, on a "C:\", "D:\", ... -. Il faudra donc faire attention à l'utilisation des fonctions **root()** et **root_directory()** qui retournent un autre résultat sur ces derniers OS.

Ne programmez jamais des chemins absolus, utilisez des chemins relatifs et lorsqu'un chemin absolu est nécessaire, laissez l'OS compléter, utilisez un chemin que l'utilisateur aurait rentré ou utilisez la fonction **initial_path()** qui sera étudiée plus tard.

Enfin, faites attention, **operateur==** utilise une comparaison lexicographique, donc

```
path("abc") == path("ABC");
```

retournera toujours faux, contrairement à l'opération d'équivalence **equivalent()** - en cours d'écriture :| - qui utilisera la comparaison adaptée à l'OS.

I-A - Exposé de la classe path

```
namespace boost
{
  namespace filesystem
  {
    class path
    {
    public:
      typedef bool (*name_check)( const std::string & name );

      // compiler generates copy constructor,
      // copy assignment, and destructor

      // constructors:
      path();
      path( const std::string & src );
      path( const char * src );
      path( const std::string & src, name_check checker );
      path( const char * src, name_check checker );

      // append operations:
      path & operator /= ( const path & rhs );
      path   operator /  ( const path & rhs ) const;

      // conversion functions:
      const std::string & string() const;
      std::string native_file_string() const;
      std::string native_directory_string() const;

      // modification functions:
      path &      normalize();
```

```
        // decomposition functions:
        path        root_path() const;
        std::string root_name() const;
        std::string root_directory() const;
        path        relative_path() const;
        std::string leaf() const;
        path        branch_path() const;

        // query functions:
        bool empty() const;
        bool is_complete() const;
        bool has_root_path() const;
        bool has_root_name() const;
        bool has_root_directory() const;
        bool has_relative_path() const;
        bool has_leaf() const;
        bool has_branch_path() const;

        // iteration:
        typedef implementation-defined iterator;
        iterator begin() const;
        iterator end() const;

        // default name_check mechanism:
        static bool default_name_check_writable();
        static void default_name_check( name_check new_check );
        static name_check default_name_check();

        // relational operators:
        bool operator==( const path & that ) const;
        bool operator!=( const path & that ) const;
        bool operator<( const path & that ) const;
        bool operator<=( const path & that ) const;
        bool operator>( const path & that ) const;
        bool operator>=( const path & that ) const;
      };

      path operator / ( const char * lhs, const path & rhs );
      path operator / ( const std::string & lhs, const path & rhs );

      // name_check functions
      bool portable_posix_name( const std::string & name );
      bool windows_name( const std::string & name );
      bool portable_name( const std::string & name );
      bool portable_directory_name( const std::string & name );
      bool portable_file_name( const std::string & name );
      bool no_check( const std::string & name );
      bool native( const std::string & name );
    }
  }
```

I-B - Les types prédéfinis

Le typedef le plus important est le premier défini de la classe, **name_check**. Ce type est un type de fonction qui permettra de vérifier la validité d'un chemin. **iterator** définit quant à lui le type des itérateurs qu'on peut utiliser.

I-C - Les constructeurs et destructeurs

Aucun destructeur n'est défini, en revanche, les constructeurs sont parfois en double. En fait, ils peuvent prendre en entrée rien du tout - le constructeur par défaut -, ou un **const char *** ou un **const string&**, accompagné ou non d'une fonction de vérification de type **name_check**.

I-D - Les fonctions d'ajouts, de conversion et de normalisation

Les opérateurs d'ajouts utilisent l'opérateur "/", qui est le standard choisi par boost comme caractère de séparation, il y a donc une logique. Il existe donc **operator/** et **operator/=** qui permettent d'ajouter un chemin à une instance de classe.

Trois fonctions de conversion existent. **string()** retourne une chaîne de caractères représentant le chemin que l'instance décrit avec le caractère "/" comme séparateur. **native_file_string()** et **native_directory_string()** retourne une chaîne de caractères en utilisant la manière native à l'OS de représenter un chemin de fichier ou de dossier.

La normalisation d'une instance de classe **path** est appelée à l'aide de la fonction **normalize()** qui retire récursivement les sous-chaînes de type "foo/../".

I-E - Les fonctions de décomposition et les itérateurs

La première fonction est **root_path()** qui retourne en fait la concaténation du résultat de 2 autres fonctions, **root_name()** et **root_directory()**. La première de ces fonctions retourne le premier élément du chemin si celui-ci est une racine - par exemple "c:" sous Windows - tandis que la deuxième retourne "/" si l'instance de path contient une racine. La fonction **relative_path()** retourne la portion relative de l'instance; que la fonction **leaf()** retourne le dernier élément de l'instance tandis que **branch_path()** permet tout simplement de récupérer le chemin contenu dans l'instance sans le dernier élément, celui retourné par **leaf()**.

En ce qui concerne les itérateurs, les deux fonctions classiques, **begin()** et **end()**, sont fournies.

I-F - Les fonctions de test

Les premières fonctions importantes sont **empty()**, retournant **true** si le chemin est vide, **is_complete()** retournant **true** si le chemin est absolu. D'autres fonctions sont disponibles, **has_root_path()** retourne **true** si **has_root_name()** retourne **true** - lorsqu'il y a une racine dans l'instance de la classe - ou si **has_root_directory()** retourne **true** - si **root_directory()** n'est pas vide -. De même pour **has_relative_path()**, **has_leaf()** et **has_branch_path()** retournant **true** si l'instance a un chemin relatif, un dernier élément ou un chemin sans le dernier élément non vide.

Comme indiqué précédemment, on peut spécifier une fonction de vérification de validité du chemin. **default_name_check_writable()** retourne **true** si aucune fonction de test n'a été spécifiée quand **default_name_check()** permet de la spécifier, mais une seule fois. La fonction **default_name_check()** retourne la fonction de vérification sélectionnée.

Les opérateurs de comparaison classiques ont été définis, comme **operator==**, **operator!=**, **operator==**, **operator<**, **operator<=**, **operator>** et **operator>=**. Ces comparaisons utilisent uniquement l'ordre lexicographique.

I-G - Les fonctions externes

Deux fonctions surchargées **operator/** permettent de concaténer un **const char*** ou un **const string&** avec une instance de classe **path**. Enfin, des fonctions permettent de vérifier la conformité d'un chemin stocké dans un **string**, comme **portable_posix_name()**, **windows_name()**, **portable_name()**, **portable_directory_name()**, **portable_file_name()**, **no_check()**, **native()** qui retournent **true** lorsque le chemin est compatible POSIX, Windows, les deux, décrit un dossier sans ".", un nom de fichier avec un ".", dans tous les cas, ou lorsque le chemin est valide sous le système d'exploitation utilisé.

I-H - Exemple de retour des fonctions

p.string()	Elements	p.root_path()	p.root_name()	p.root_directory()	p.relative_path()	p.root_directory() / p.relative_path()	p.root_path() / p.relative_path()	p.branch_path()	p.leaf()
All Systems									
/	/	/	""	/	""	/	""	""	/
foo	foo	""	""	""	foo	foo	foo	""	foo
/foo	/, foo	/	""	/	foo	/foo	foo	/	foo
foo/bar	foo, /, bar	""	""	""	foo/bar	foo/bar	foo/bar	foo	bar
/foo/bar	/, foo, /, bar	/	""	/	foo/bar	/foo/bar	foo/bar	/foo	bar
.	.	""	""	""	.	.	.	""	.
..	..	""	""	""	..	..	..	""	..
../foo	.., /, foo	""	""	""	../foo	../foo	../foo	..	foo
Windows									
C:	C:	C:	C:	""	""	""	C:	""	C:
C:/	C:, /	C:/	C:	/	""	/	C:/	C:	/
C:..	C:, ..	C:	C:	""	..	C:..	C:..	C:	..
C:foo	C:, foo	C:	C:	""	foo	foo	C:foo	C:	foo
C:/foo	C:, /, foo	C:/	C:	/	foo	/foo	C:foo	C:/	foo
//shr	//shr	//shr	//shr	""	""	""	//shr	""	//shr
//shr/	//shr, /	//shr/	//shr	/	""	/	//shr	//shr	/
//shr/foo	//shr, /, foo	//shr/	//shr	/	foo	/foo	//shr/foo	//shr/	foo
prn:	prn:	prn:	prn:	""	""	""	prn:	""	prn:

II - Les opérations usuelles définies dans "operations.hpp"

Cet en-tête est sans doute le plus utile. Il définit un itérateur sur un dossier ainsi que plusieurs fonctions qui vous permettront de faire presque tout ce qu'on faisait en ligne de commande.

II-A - La classe directory_iterator

La classe directory_iterator

```
class directory_iterator
{
public:
  typedef path                    value_type;
  typedef std::ptrdiff_t          difference_type;
  typedef const path *            pointer;
  typedef const path &            reference;
  typedef std::input_iterator_tag iterator_category;

  directory_iterator();
  explicit directory_iterator( const path & directory_ph );

  // Autres fonctions pour un itérateur
  // ...
};
```

La classe **directory_iterator** est, comme son nom l'indique, un itérateur. Il est de catégorie **std::input_iterator_tag**, et donc définit toutes les fonctions usuelles d'un itérateur - comparaison, incrémentation, déréférencement, ... -, l'objet pointé étant de type **path**.

II-B - Les fonctions usuelles

Les fonctions usuelles d'operation.hpp

```
bool exists( const path & ph );
bool symbolic_link_exists( const path & ph );
bool is_directory( const path & ph );
bool is_empty( const path & ph );
bool equivalent( const path & ph1, const path & ph2 );

boost::intmax_t file_size( const path & ph );

std::time_t last_write_time( const path & ph );
void last_write_time( const path & ph, std::time_t new_time );

bool create_directory( const path & directory_ph );
bool remove( const path & ph );
unsigned long remove_all( const path & ph );
void rename( const path & from_path,
             const path & to_path );
void copy_file( const path & source_file,
                const path & target_file );

const path & initial_path();
path current_path();
path complete( const path & ph,
               const path & base = initial_path() );
path system_complete( const path & ph );
```

Les premières fonctions utiles permettent de connaître l'état d'une instance de **path**. Par exemple, **exists()** teste l'existence d'un fichier/dossier, **symbolic_link_exists()** si le chemin est un lien symbolique, **is_directory()** si le chemin est un dossier, **is_empty()** indique si le fichier ou le dossier est vide, enfin **is_equivalent()** indique si 2 chemins sont équivalents.

file_size() retourne la taille d'un fichier, quand **last_write_time()** retourne le temps de la dernière écriture sur le fichier. Une fonction surchargée de **last_write_time()** permet de mettre à jour ce temps.

D'autres fonctions dont utiles pour modifier l'arborescence. Par exemple, **create_directory()** crée un nouveau dossier, **remove()** détruit un chemin - et retourne **true** si le chemin existait -, **remove_all()** détruit tous les dossiers et fichiers contenu dans un chemin, et retourne le nombre d'éléments détruits. Enfin, **copy_file()** permet de copier un fichier et **rename()** de renommer un chemin.

Enfin, **initial_path()** retourne le contenu de **current_path()** lors de la première invocation en stockant ce résultat, **current_path()** retournant le chemin courant tel que l'OS le connaît, **complete()** permet de compléter un chemin relatif et **system_complete()** résout un chemin relatif comme le ferait l'OS.

III - La gestion des flux sur des fichiers avec "fstream.hpp"

Flux spéciaux avec fstream.cpp

```
template < class charT, class traits = std::char_traits<charT> >
class basic_filebuf : public std::basic_filebuf<charT,traits>
{
public:
  virtual ~basic_filebuf() {}

  std::basic_filebuf<charT,traits> * open( const path & file_ph,
    std::ios_base::openmode mode );
};

typedef basic_filebuf<char> filebuf;
typedef basic_filebuf<wchar_t> wfilebuf;

template < class charT, class traits = std::char_traits<charT> >
class basic_ifstream : public std::basic_ifstream<charT,traits>
{
public:
  basic_ifstream() {}
  explicit basic_ifstream( const path & file_ph,
    std::ios_base::openmode mode = std::ios_base::in );
  virtual ~basic_ifstream() {}
  void open( const path & file_ph,
    std::ios_base::openmode mode = std::ios_base::in );
};

typedef basic_ifstream<char> ifstream;
typedef basic_ifstream<wchar_t> wifstream;

template < class charT, class traits = std::char_traits<charT> >
class basic_ofstream : public std::basic_ofstream<charT,traits>
{
public:
  basic_ofstream() {}
  explicit basic_ofstream( const path & file_ph,
    std::ios_base::openmode mode = std::ios_base::out );
  virtual ~basic_ofstream() {}
  void open( const path & file_ph,
    std::ios_base::openmode mode = std::ios_base::out );
};

typedef basic_ofstream<char> ofstream;
typedef basic_ofstream<wchar_t> wofstream;

template < class charT, class traits = std::char_traits<charT> >
class basic_fstream : public std::basic_fstream<charT,traits>
{
public:
  basic_fstream() {}
  explicit basic_fstream( const path & file_ph,
    std::ios_base::openmode mode = std::ios_base::in|std::ios_base::out );
  virtual ~basic_fstream() {}
  void open( const path & file_ph,
    std::ios_base::openmode mode = std::ios_base::in|std::ios_base::out );
};

typedef basic_fstream<char> fstream;
typedef basic_fstream<wchar_t> wfstream;
```

Dans ce fichier sont définis les mêmes classes que celles existants dans **std::**, sauf qu'à la place de prendre un **const char*** ou un **string** en entrée, elles prennent une variable de type **path**. Logique, non ?

IV - Les exceptions dans "exception.hpp"

Les exceptions de Boost.FileSystem

```
enum error_code
{
  no_error = 0,
  system_error,     // system generated error; if possible, is translated
                    // to one of the more specific errors below.
  other_error,      // library generated error
  security_error,   // includes access rights, permissions failures
  read_only_error,
  io_error,
  path_error,
  not_found_error,
  not_directory_error,
  busy_error,       // implies trying again might succeed
  already_exists_error,
  not_empty_error,
  is_directory_error,
  out_of_space_error,
  out_of_memory_error,
  out_of_resource_error
};

class filesystem_error : public std::exception
{
public:

  filesystem_error(
    const std::string & who,
    const std::string & message );

  filesystem_error(
    const std::string & who,
    const path & path1,
    const std::string & message,
    error_code ec = other_error );

  filesystem_error(
    const std::string & who,
    const path & path1,
    sys_err sys_err_code );

  filesystem_error(
    const std::string & who,
    const path & path1,
    const path & path2,
    sys_err sys_err_code );

  ~filesystem_error() throw();

  virtual const char * what() const throw();

  sys_err native_error() const;
  error_code error() const;
  const std::string & who() const;
  const path & path1() const;
  const path & path2() const;
};
```

Qui dit C++ dit gestion des exceptions, et Boost.FileSystem n'échappe pas à la règle. Une classe dérivée de **std::runtime_error** est définie, **filesystem_error**, qui est renvoyée lors des erreurs rencontrées dans les autres fonctions décrites dans ce tutoriel. Outre le traditionnel **what()**, un indicateur de code d'erreur est disponible par la fonction **error()**.

V - Des fonctions d'aide dans "convenience.hpp"

Les fonctions pratiques

```
bool create_directories( const path & ph );
std::string extension( const path & ph );
std::string basename( const path & ph );
path change_extension( const path & ph, const std::string & new_extension );
```

Grâce à **create_directories()**, il est possible de créer toute l'arborescence d'un dossier d'un coup. **extension()** permet de retourner l'extension d'un fichier si un '.' est trouvé dans la chaîne retournée par **leaf()**. De même, **basename()** permet de retourner le nom sans l'extension. Enfin, **change_extension()** permet de changer l'extension d'un fichier.

VI - Petit exemple

Le petit bout de code suivant aura pour simple objectif de montrer comment copier des fichiers portant une certaine extension vers les mêmes fichiers avec une autre extension, le tout dans l'arborescence courante.

```
#include <iostream>
#include <boost/filesystem/path.hpp>
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/convenience.hpp>

void iterThroughDirectories(const boost::filesystem::path& path);

int main( int argc, char *argv[])
{
  iterThroughDirectories(boost::filesystem::initial_path());
}

void iterThroughDirectories(const boost::filesystem::path& path)
{
  std::cout << "Entering : " << path.string() << std::endl;
  for(boost::filesystem::directory_iterator it(path); it !=
 boost::filesystem::directory_iterator(); ++it)
  {
   if(boost::filesystem::is_directory(*it))
   {
    iterThroughDirectories(*it);
   }
    else if(boost::filesystem::extension(*it) == ".exe")
    {
      boost::filesystem::path newfile = boost::filesystem::change_extension(*it, ".bak");
      std::cout << "Copying " << (*it).string() << " to " << newfile.string() << std::endl;
      boost::filesystem::copy_file(*it, newfile);
    }
  }
}
```

Conclusion

Le plus gros du travail est fourni par la classe **path**, et c'est logique. Cette classe, portable, servira à tous ceux qui utilisent déjà Boost et qui ne veulent pas ajouter une nouvelle bibliothèque dans leur projet. Certes, Boost.FileSystem n'est pas exhaustif, mais les fonctionnalités importantes sont présentes, et c'est ce qui est important dans une bibliothèque portable.