|  |  |
| --- | --- |
|  | **Back of Card** |
|  | Caposocrates/MusicUnzipper/source |
|  | MusicUnzipper.vcxproj  This is the main project file for VC++ projects generated using an Application Wizard.  It contains information about the version of Visual C++ that generated the file, and  information about the platforms, configurations, and project features selected with the  Application Wizard.  MusicUnzipper.vcxproj.filters  This is the filters file for VC++ projects generated using an Application Wizard.  It contains information about the association between the files in your project  and the filters. This association is used in the IDE to show grouping of files with  similar extensions under a specific node (for e.g. ".cpp" files are associated with the  "Source Files" filter).  MusicUnzipper.cpp  This is the main application source file.  /////////////////////////////////////////////////////////////////////////////  Other standard files:  StdAfx.h, StdAfx.cpp  These files are used to build a precompiled header (PCH) file  named MusicUnzipper.pch and a precompiled types file named StdAfx.obj.  ///////////////////////////////////////////////////////////////////////////// |
|  | constants.hpp  #pragma once  constexpr auto \_\_\_IGNORING = L"\_\_\_IGNORING";  constexpr auto \_\_\_PARSED = L"\_\_\_PARSED";  constexpr auto \_\_\_READY = L"\_\_\_READY";  constexpr auto \_\_\_PROCESSED = L"\_\_\_PROCESSED";  constexpr auto \_\_\_ERROR = L"\_\_\_ERROR";  constexpr auto \_\_\_BEGIN = L"\_\_\_BEGIN";  constexpr auto \_\_\_END = L"\_\_\_END"; |
|  | interpret.cpp  #include "interpret.h"  #include "representation.h"  #include <boost\range\algorithm.hpp>  #include <iostream>  #include <string>  const auto locale = std::locale(); //not thread-safe  auto is\_not\_space(const wchar\_t c) -> bool  {  return c != L' ' && c != L'\_' && c != L'-';  }  auto trim(sub\_str range\_ref) -> sub\_str  {  return sub\_str(  boost::find\_if(  range\_ref  , is\_not\_space  )  , (boost::find\_if(  ad::reverse(range\_ref)  , is\_not\_space  )  ).base()  );  }  auto print(sub\_str str\_range) -> std::wostream&  {  boost::copy(str\_range, std::ostreambuf\_iterator<wchar\_t>(std::wcout));  return std::wcout;  }  auto dash\_or\_underscore(const wchar\_t c) -> bool  {  return c == L'\_' || c == L'-';  };  auto maybe\_capitalize(char\_range str) -> void  {  constexpr wchar\_t\* to\_capitalize[]{  L"ep"  , L"lp"  , L"ato"  };  for (const auto check : to\_capitalize)  {  if (alg::iequals(str, check, locale))  {  alg::to\_upper(str, locale);  return;  }  }  };  auto sanitize(sub\_str str) -> sub\_str  {  char\_range last\_str = {};  auto char\_count = 0;  bool last\_char\_was\_space = true;  auto sub\_parts = std::vector<char\_range>();  for (wchar\_t& c : str)  {  if (last\_char\_was\_space && !dash\_or\_underscore(c))  {  last\_str = {&c, &c + 1};  alg::to\_upper(last\_str, locale);  last\_char\_was\_space = false;  }  if (dash\_or\_underscore(c))  {  c = L' ';  last\_char\_was\_space = true;  last\_str = {last\_str.begin(), last\_str.begin() + char\_count};  char\_count = 0;  maybe\_capitalize(last\_str);  }  else  {  ++char\_count;  }  }  return str;  };  auto explode\_path(const fs::path& path, const std::string& intermediate\_path) -> representation  {  auto filename = path.filename().wstring();  auto path\_str = sub\_str(filename);  path\_str.drop\_back(4); //shear off ".zip"  auto parts = std::vector<sub\_str>();  {  auto contiguous\_underscores = 0;  alg::split(parts, path\_str, [&](const wchar\_t c) -> bool  {  if (c == L'\_')  {  ++contiguous\_underscores;  }  else  {  contiguous\_underscores = 0;  }  return 3 <= contiguous\_underscores;  });  }  if (parts.empty())  {  throw std::runtime\_error(std::string("bad filename!"));  }  return{path.wstring(), path.parent\_path()  .append(intermediate\_path)  .append(sanitize(trim(parts.front())))  .append(sanitize(trim(parts.back())))};  }  auto split\_string\_to\_rep(const std::wstring& str, const std::string& intermediate\_path) -> representation  {  constexpr auto divider = L" -> ";  constexpr auto div\_sz = 4; //4 chars  const auto pos = str.find(divider);  if (pos == std::wstring::npos)  {  return explode\_path(str, intermediate\_path);  }  const auto b = std::begin(str), e = std::end(str);  return{{b, b + pos},{b + pos + div\_sz, e}};  } |
|  | interpret.h  #pragma once  #include "types.hpp"  #include <iosfwd>  struct representation;  auto explode\_path(const fs::path& path, const std::string& extended\_path) -> representation;  auto split\_string\_to\_rep(const std::wstring& str, const std::string& extended\_path) -> representation; |
|  | representation.cpp  include "representation.h"  #include <boost\algorithm\string\predicate.hpp> //for ilexicographical\_compare  #include <iostream>  #include <string>  const auto locale = std::locale(); //not thread-safe  auto rep\_less::operator()(const representation& lhs, const representation& rhs) const -> bool  {  return alg::ilexicographical\_compare(lhs.old\_path.wstring(), rhs.old\_path.wstring(), locale);  }  auto rep\_less::operator()(const std::wstring& lhs, const representation& rhs) const -> bool  {  return alg::ilexicographical\_compare(lhs, rhs.old\_path.wstring(), locale);  }  auto rep\_less::operator()(const representation& lhs, const std::wstring& rhs) const -> bool  {  return alg::ilexicographical\_compare(lhs.old\_path.wstring(), rhs, locale);  }  auto operator<<(std::wostream& os, const representation& r) -> std::wostream&  {  return os << r.old\_path.wstring() << L" -> " << r.new\_path.wstring();  } |
|  | representation.h  #pragma once  #include "types.hpp"  #include <iosfwd>  #include <set>  struct representation  {  fs::path old\_path;  fs::path new\_path;  };  struct rep\_less  {  using is\_transparent = std::true\_type;  auto operator()(const representation& lhs, const representation& rhs) const -> bool;  auto operator()(const std::wstring& lhs, const representation& rhs) const -> bool;  auto operator()(const representation& lhs, const std::wstring& rhs) const -> bool;  };  using rep\_set = std::set<representation, rep\_less>;  auto operator<<(std::wostream& os, const representation& r) -> std::wostream&; |
|  | targetver.h  #pragma once  // Including SDKDDKVer.h defines the highest available Windows platform.  // If you wish to build your application for a previous Windows platform, include WinSDKVer.h and  // set the \_WIN32\_WINNT macro to the platform you wish to support before including SDKDDKVer.h.  #include <SDKDDKVer.h> |
|  | types.h  #pragma once  #include <boost\algorithm\string.hpp>  #include <boost\filesystem.hpp>  #include <boost\range\adaptors.hpp>  #include <boost\range\sub\_range.hpp>  namespace fs = boost::filesystem;  namespace alg = boost::algorithm;  namespace ad = boost::adaptors;  using sub\_str = boost::sub\_range<std::wstring>;  using char\_range = boost::iterator\_range<wchar\_t\*>; |
|  | stdafx.cpp  // stdafx.cpp : source file that includes just the standard includes  // MusicUnzipper.pch will be the pre-compiled header  // stdafx.obj will contain the pre-compiled type information  #include "stdafx.h"  // TODO: reference any additional headers you need in STDAFX.H  // and not in this file |
|  | stdafx.h  // stdafx.h : include file for standard system include files,  // or project specific include files that are used frequently, but  // are changed infrequently  //  #pragma once  #include "targetver.h"  #include <stdio.h>  #include <tchar.h>  // TODO: reference additional headers your program requires here |
|  | intermediate.h  #pragma once  #include "representation.h"  #include "types.hpp"  #include <set>  struct intermediate\_data  {  std::set<fs::path> ignored\_files;  rep\_set parsed\_filenames;  rep\_set ready\_files;  rep\_set processed\_files;  rep\_set errored\_files;  };  auto read(const fs::path& directory\_path, const std::string& extended\_path, const fs::path& intermediate\_path) -> intermediate\_data;  auto write(  const fs::path& intermediate\_path  , const intermediate\_data& data  ) -> void; |
|  | intermediate.cpp  #include "intermediate.h"  #include "interpret.h"  #include "constants.hpp"  #include <boost\range\algorithm.hpp>  #include <fstream>  #include <iterator>  const auto locale = std::locale(); //not thread-safe  enum class parse\_state  {  IGNORED  , PARSED  , READY  , NEW  , PROCESSED  , ERROR  };  auto is\_zip\_file(const fs::path& de) -> bool  {  return alg::iequals(de.extension().wstring(), L".zip", locale);  }  auto read(const fs::path& directory\_path, const std::string& extended\_path, const fs::path& intermediate\_path) -> intermediate\_data  {  auto ignored\_files = std::set<fs::path>();  auto parsed\_filenames = rep\_set();  auto ready\_files = rep\_set();  auto processed\_files = rep\_set();  auto errored\_files = rep\_set();  auto explode = [&extended\_path](fs::path path) -> representation  {  return explode\_path(path, extended\_path);  };  {  auto infile = std::wifstream(intermediate\_path.wstring(), std::ios::binary);  auto current\_line = std::wstring();  auto state = parse\_state::IGNORED;  std::for\_each(std::istreambuf\_iterator<wchar\_t>(infile), {}, [&](const wchar\_t c)  {  if (c == L'\n' || c == L'\r')  {  if (current\_line == \_\_\_IGNORING)  {  state = parse\_state::PARSED;  }  else if (current\_line == \_\_\_PARSED)  {  state = parse\_state::READY;  }  else if (current\_line == \_\_\_READY)  {  state = parse\_state::PROCESSED;  }  else if (current\_line == \_\_\_PROCESSED)  {  state = parse\_state::ERROR;  }  else if (current\_line == \_\_\_ERROR)  {  state = parse\_state::NEW;  }  else if (!current\_line.empty())  {  switch (state)  {  case parse\_state::IGNORED:  {  ignored\_files.insert(current\_line);  break;  }  case parse\_state::PARSED:  {  parsed\_filenames.insert(split\_string\_to\_rep(current\_line, extended\_path));  break;  }  case parse\_state::READY:  {  ready\_files.insert(split\_string\_to\_rep(current\_line, extended\_path));  break;  }  case parse\_state::PROCESSED:  {  processed\_files.insert(split\_string\_to\_rep(current\_line, extended\_path));  break;  }  case parse\_state::ERROR:  {  errored\_files.insert(split\_string\_to\_rep(current\_line, extended\_path));  break;  }  case parse\_state::NEW:  {  const auto temp = split\_string\_to\_rep(current\_line, extended\_path);  if (ignored\_files.find(current\_line) == ignored\_files.end()  && ready\_files.find(temp) == ready\_files.end()  && processed\_files.find(temp) == processed\_files.end()  && errored\_files.find(temp) == errored\_files.end())  {  parsed\_filenames.insert(temp);  }  break;  }  default:  {  assert(false); break;  }  }  }  current\_line.clear();  }  else  {  current\_line.push\_back(c);  }  }  );  }  auto is\_new = [&](const fs::path& file) -> bool  {  return ignored\_files.find(file) == ignored\_files.end()  && parsed\_filenames.find(file.wstring()) == parsed\_filenames.end()  && ready\_files.find(file.wstring()) == ready\_files.end()  && processed\_files.find(file.wstring()) == processed\_files.end()  && errored\_files.find(file.wstring()) == errored\_files.end();  };  auto get\_new\_zip\_files = [&directory\_path, is\_new]() -> auto  {  return fs::directory\_iterator(directory\_path)  | ad::filtered([&](const fs::directory\_entry& de) -> bool  {  const auto nonrelative\_path = fs::system\_complete(de);  return fs::is\_regular\_file(nonrelative\_path) && is\_zip\_file(nonrelative\_path) && is\_new(nonrelative\_path);  }  )  | ad::transformed([](const fs::directory\_entry& de) -> fs::path  {  return fs::system\_complete(de);  }  );  };  for (representation r : get\_new\_zip\_files() | ad::transformed(explode))  {  parsed\_filenames.emplace(std::move(r));  }  return{  std::move(ignored\_files)  , std::move(parsed\_filenames)  , std::move(ready\_files)  , std::move(processed\_files)  , std::move(errored\_files)  };  }  auto write(  const fs::path& intermediate\_path  , const intermediate\_data& data  ) -> void  {  auto outfile = std::wofstream(intermediate\_path.wstring());  boost::copy(  data.ignored\_files | ad::transformed([](const fs::path& p) -> const std::wstring&{return p.wstring();})  , std::ostream\_iterator<std::wstring, wchar\_t>(outfile, L"\n")  );  outfile << \_\_\_IGNORING << L"\n";  boost::copy(  data.parsed\_filenames  , std::ostream\_iterator<representation, wchar\_t>(outfile, L"\n")  );  outfile << \_\_\_PARSED << L"\n";  boost::copy(  data.ready\_files  , std::ostream\_iterator<representation, wchar\_t>(outfile, L"\n")  );  outfile << \_\_\_READY << L"\n";  boost::copy(  data.processed\_files  , std::ostream\_iterator<representation, wchar\_t>(outfile, L"\n")  );  outfile << \_\_\_PROCESSED << L"\n";  boost::copy(  data.errored\_files  , std::ostream\_iterator<representation, wchar\_t>(outfile, L"\n")  );  outfile << \_\_\_ERROR << L"\n";  } |
|  | unzip.h  #pragma once  struct representation;  struct intermediate\_data;  auto unzip\_all(intermediate\_data&) -> void; |
|  | unzip.cpp  #include "unzip.h"  #include "intermediate.h"  #include "representation.h"  #pragma warning( push, 1 )  #include <libzippp.h>  #pragma warning( pop )  #include <boost\range\algorithm.hpp>  #include <iostream>  #include <memory>  namespace lzip = libzippp;  using std::begin;  using std::end;  auto unzip(representation) -> bool;  auto unzip\_all(intermediate\_data& intermediate) -> void  {  for (const representation& r : intermediate.ready\_files)  {  if(!unzip(r))  {  intermediate.errored\_files.insert(r);  }  }  boost::set\_difference(  intermediate.ready\_files  , intermediate.errored\_files  , std::inserter(intermediate.processed\_files, end(intermediate.processed\_files))  , rep\_less()  );  intermediate.ready\_files.clear();  }  auto unzip(representation r) -> bool  {  lzip::ZipArchive zf(r.old\_path.string());  if (!zf.open(lzip::ZipArchive::READ\_ONLY, true))  {  return false;  }  fs::create\_directories(r.new\_path);  for (lzip::ZipEntry& entry : zf.getEntries())  {  auto binary\_data = std::unique\_ptr<char[]>(reinterpret\_cast<char\*>(entry.readAsBinary()));  auto filepath = r.new\_path;  filepath.append(L"\\").append(entry.getName());  std::ofstream output\_file(filepath.wstring(), std::ios::binary);  std::copy(  binary\_data.get()  , binary\_data.get() + entry.getSize()  , std::ostreambuf\_iterator<char>(output\_file)  );  }  return true;  } |
|  | MusicUnzipper.cpp  // MusicUnzipper.cpp : Defines the entry point for the console application.  //  #include "intermediate.h"  #include "unzip.h"  #include "constants.hpp"  #include "types.hpp"  #include "stdafx.h"  #include <boost\range\algorithm.hpp>  #include <iostream>  #include <iterator>  #include <string>  auto main(const int argc, const char\* const argv[]) -> int  {  auto command = 'd';  auto extended\_path = std::string("Music");  auto path = fs::path();  if (argc <= 1)  {  path = ".";  }  else if (1 < argc)  {  path = argv[1];  if (2 < argc)  {  command = argv[2][0];  }  if (3 < argc)  {  extended\_path = argv[3];  }  }  try  {  if (fs::exists(path) && fs::is\_directory(path))  {  const auto \_\_\_INTERMEDIATE = path / L"\_\_\_INTERMEDIATE";  auto inter\_data = read(path, extended\_path, \_\_\_INTERMEDIATE.wstring());  switch (command)  {  case 'd': case 'D':  {  std::wcout << L"Evaluating default command; parsing directory at " << path << std::endl;  //the parse has already happened at this point--we always parse  break;  }  case 'e': case 'E':  {  std::wcout << L"Evaluating execute command" << std::endl;  unzip\_all(inter\_data);  break;  }  default:  {  throw(std::runtime\_error("Invalid command!"));  }  }  write(\_\_\_INTERMEDIATE, inter\_data);  }  else  {  auto pstr = path.string();  throw std::runtime\_error(  (!fs::exists(path)  ? pstr.append(" does not exist!")  : (  !fs::is\_directory(path)  ? pstr.append(" is not a directory!")  : pstr.append(" could not be parsed")  )  )  );  }  }  catch (std::exception e)  {  std::wcout << e.what() << L"\n";  return 1;  }  return 0;  } |
|  | [[I am leaving out two documents that are not cpp code – MusicUnzipper.vcxproj and MusicUnzipper.vcxproj.filter.]] |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| **Header** |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| **Header** |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |