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.]]