SIMPLEXpress Documentation

Release 0.1

MousePaw Media

Jun 13, 2021

CONTENTS

SIMPLEXpress is a simple-to-use alternative to regular expressions. Instead of requiring the user to memorize dozens of special characters with multiple uses and complex rules, SIMPLEXpress hard-reserves two characters and leaves the rest for literal use. Using a few simple syntactic rules, the user can easily write powerful expressions.

See SIMPLEXpress' `README.md`, `CHANGELOG.md`, `BUILDING.md`, and `LICENSE.md` for more information.

CHAPTER

ONE

CONTENTS:

1.1 Setup and Compiling

1.1.1 Building

SIMPLExpress is a static library which needs to be compiled before it can be used.

Note: Complete building instructions can be found in BUILDING.md

Environment and Dependencies

To build SIMPLExpress, you must have the following installed on your system:

- CMake 3.1+ (cmake.org)
- Clang 3.4+ OR GCC 5.3+ (MSVC is not presently supported.)
- CPGF 1.60, from either:
  - libdeps 1.0 (GitHub) [recommended], *or*
  - CPGF (GitHub).
- PawLIB, available on GitHub.

Build Configuration

Follow the instructions in the `pawlib` repository to set up PawLIB (which requires first setting up CPGF, whether by using our `libdeps` repository or by following PawLIB's alternative directions), then clone the `simplexpress` repository into an adjacent folder.

Compiling SIMPLExpress

Make sure that PawLIB is built, also ensuring CPGF is built before building PawLIB. If you're using `libdeps`, you can simply run `make cpgf` in that repository. Then in the PawLIB repository, run `make ready` to build PawLIB as a static library.

Finally, in the SIMPLExpress repository, run `make ready` to build SIMPLExpress for use in your project.

Note: If you experience any errors in compiling, we invite you to contact us. See *Finding Support*.

1.1.2 Linking to SIMPLExpress

To use SIMPLExpress in your C++ project, you will need to statically link to it, and both PawLIB and CPGF.

Important: Linking order is important in C++! You should link to SIMPLExpress first, then PawLIB, and then its dependencies (namely CPGF).

You can point your compiler and linker at the `simplexpress/include` and `simplexpress/lib` directories, followed by `pawlib/include/` and `pawlib/lib/` directories.

If you need help finding the paths for CPGF, refer to the path in the PawLIB build configuration (see *Build Configuration*). The `include/` and `lib/` directories should be in the path specified.

1.1.3 Using SIMPLExpress

All `#include` commands follow the format `#include "simplexpress/HEADER.hpp"`, where `HEADER.hpp` is the header file you wish to include.

1.1.4 Building Tester

If you want to use the SIMPLExpress tester application, you can build that as well. Once you've confirmed that SIMPLExpress itself can build without errors, run `make tester` to build the tester.

1.2 Simplex

1.2.1 What is Simplex?

Simplex is the basic class used in SIMPLEXpress to store a model and process matches against the model. Simplex is designed to be easier to read and write compared to standard regex. The priority with Simplex models is ease of use.

1.2.2 Using Simplex

Including Simplex

To include Simplex, use the following:

```
#include "simplexpress/simplex.hpp"
```

Creating a Simplex

A `Simplex` object is created by whatever means is convenient.

When the Simplex is first created, you must specify the *model* you are using to process matches. A onestring (from PawLIB) is preferred, but for models using only `char`, you can define it in the declaration, too:

```
// Using onestring
onestring model = "^d/";
Simplex simplex(model);
// Using char
Simplex char_simplex("^l/");
```

The major functions of Simplex, `match`, `snag` and `lex`, can be also used without creating a Simplex object. (See the function pages for details on use.) Creating a Simplex object is the more efficient of these two methods when you will be reusing the model, however.

1.3 Simplex Model

1.3.1 What is a Simplex model?

As a regular expression parser, SIMPLEXpress uses the model to determine what exactly you are trying to match. The model communicates to the SIMPLEXpress code the specifics of what qualifies as a match or not.

For the most part, SIMPLEXpress models are read literally. The only time they are not literal is if you include something into a Unit, and the only reserved characters outside of a Unit are the characters used to mark the beginning of a Unit, or escape them.

Note: Behind the scenes, each individual literal character is also considered a Unit, so when you see references to Units, consider the context to know whether they may also include individual literals.

Throughout this document, where relevant, current functionality as of v0.1 and planned functionality will be split into separate subsections.

1.3.2 Reserved characters

Outside a Unit

There are only three hard reserved characters in Simplex models, which are used outside of a Unit:

- `^`: Used to indicate the beginning of a non-literal Unit.
- `~`: Used to indicate the beginning of a snag Unit.
- `%`: Used to escape any of the three reserved characters as needed.

Inside a Unit (Soft Reserved)

Inside of a Unit, various characters are used to indicate different behavior.

- `/`: Closes Unit. After all other characters describing Unit.
- `!`: NOT. Before the matcher.
- `?`: Optional. After matcher.
- `+`: Multiple. After the matcher.
- `*`: Optional *and* Multiple. After matcher.

Planned Functionality (Soft Reserved Characters)

- `[ ]`: Set. Match any one of the Unit values within. Space delimited.
- `<>`: Literal Set: Any literal character within.
- `( )`: Group. Allows for literal characters, strings, and further Units (simplex-ception!) within a Unit. For example, `^(abc)?/` matches optional abc.
- `%`: Escape following character (to make literal). Affects exactly one character. Modifiers following that character will affect that character's Unit.
- `{ }`: Exclusion. Anything within is checked, but is not returned as part of the result. Parallels regex "lookahead".
- `$`: Line beginning or end. Logically, we can combine the two together, because nothing can follow a line end, and nothing can precede a line beginning. In multi-line mode, this would match a line break.
- `#1`, `#2-3`, (etc, any number): Exact number or range of matches.

Examples:

- `^[(abc) (123)]/` matches either `abc` or `123`, but not both.
- `^[lu d]/` matches either an uppercase letter or a digit, but not both.
- `^[(abc) d]/` matches either `abc` or a digit.
- `^[<abc> d]/` matches either `a`, `b`, `c`, or a digit.

1.3.3 Units

Current Unit Functionality (v0.1)

Units are the basic blocks of a Simplex model. Each Unit indicates one discrete item that Simplex needs to match. A literal Unit consists of a single character to be matched exactly (any single character - other than the three hard reserved characters mentioned above - if used outside of a non-literal Unit). A non-literal Unit consists of the opening Unit marker, indicators for any desired, modifiers, and the matcher. The most common 'matcher' will be one of the one or two character *Specifiers* which indicates a match such a digit, an alphanumeric character, and so on.

As of verison 0.1, these are the only functioning matchers.

Planned Unit Functionality

In later versions, matchers will also be able to include various tools for sets and groups. See *Planned Functionality (Soft Reserved Characters)* for more information.

1.3.4 Modifiers

Note: Individual literals cannot have modifiers of any kind. If you need to use modifiers on a literal, you must place it into a specifier Unit.

Modifiers affect the behavior of the Unit in various ways. The base behavior of a specifier Unit, for example, `^a/` for an alphanumeric match, will match exactly one alphanumeric character, and will fail the match if there is no alphanumeric character present in the corresponding position in the input string. Modifiers are as follows:

- Negative: the `!` character preceding the matcher will negate the specifier (or other matcher). The single character requirement (if not otherwise modified) remains the same; it instead becomes one single character of anything *but* the matcher. Using our alphanumeric example, if you wanted a single character that was *anything non-alphanumeric* (symbols, for example), your model would look like `^!a/`. Note this must come before the matcher, placing it afterward will not work.
- Optional: the `?` character after the matcher indicates that the Unit is optional. Thus, either 0 or 1 matching characters will satisfy the requirement. So, to match whether there is an alphanumeric character in that position or not, you would use the model `^a?/`.
- Multiple: the + character after the matcher indicates the Unit can be multiple. This reads as "one or more"; it has to have at least *one* matching character to pass, but will match and consume as many characters from the input as fit the criteria until it encounters a character that does *not* match. To match one or more alphanumeric characters in a row, your model would look like `^a+/`.
- Multiple *and* Optional: the * character after the matcher indicates that the Unit is both optional and multiple. Thus, 0 or more matching characters will cause the match to pass. As with Multiple, it will continue to match and consume characters from the input as long as it encounters characters that match. The model for matching any number (including none) of alphanumeric characters in a row looks like `^a*/`.
- Snag: the ~ character in place of the Unit opener `^` indicates a snag Unit. Snag Units behave a little differently from the the other modifiers and can be used with any of them. Snag Units are used with the snag function to return matched characters; if used in a match function, behavior is identical to a regular Unit. This difference is useful when you know that specific parts of your matches are not meaningful except that they are there, but need to capture other parts of the matches to process. Using the alphanumeric example, we can demonstrate a slightly more complicated model. For example, you need to capture one or more alphanumeric characters before a comma, but you know the comma is going to be there regardless and don't need to store it; you could write a model like `~a+/,` (in this case the comma is a literal Unit), and then when using the snag function your

return array would contain anything before the comma. Example: `12345,` would return an array containing the onestring `12345`; `s,` would return an array containing `s`, and so on.

Note: If you have an optional Unit between two other potentially overlapping Units, the model will still fail if the optional is not met. For example, with the model `~l+/~d*/~l+/`, an input of `stevefred` would fail, whereas an input of `steve12345fred` would pass. The reasoning for this is that in the case of `stevefred` there are not, in fact, two separate "one or more latin letters" chunks, only one, which we have no way to arbitrarily split.

1.3.5 Specifiers

Current Specifier Functionality (v0.1)

Specifiers consist of a single letter that indicates which type of character is being matched. Currently operational specifiers are:

- `a`: alphanumeric
- `d`: digit
- `l`: latin Letter
- `n`: newline (*n*)
- `o`: math operator
- `p`: punctuation
- `r`: Carriage return (*r*)
- `s`: Literal space
- `t`: tab
- `w`: whitespace
- `.`: Any character.

Planned Specifier Functionality

- Multi character specifiers:
  - Inclusion of `u` or `l` after a specifier that includes letters to indicate upper or lower case.
  - `c`: classification (Reserved for later expanded character classes, such as `c_hangal` for Hangal characters) (2.0-3.0)
  - `u#`: unicode (accepts `u78` or `u57-78`) (2.0)
- `e`: extended Latin (2.0)
- `g`: greek (2.0)
- `i`: iPA (2.0)

1.4 Match

1.4.1 Simplex Match

The `match()` member function is used to determine whether input matches the specified *model* or not. When used with a generated Simplex object, it takes either a onestring or a string of `char` as a parameter (the input to check), and the function returns true if the input is an exact match to the model, false if not. It will only return true if the *entire* input matches the *entire* model. If you want to check for a match at the beginning of input, use *lex* instead.

Simplex is a single pass parser; if you want to include matches later than the beginning of input, build the model accordingly (ie, with a Unit like `^.*/` - which is any character, one or more, optionally - at the beginning of the units you're trying to match or lex).

```
// Using the model example on the main Simplex page:
onestring model = "^d/";

onestring match_input1 = "3";
bool match1 = simplex.match(match_input1);
// match1 is true
onestring match_input2 = "abc";
bool match2 = simplex.match(match_input2);
// match2 is false
onestring match_input3 = "3a";
bool match3 = simplex.match(match_input3);
// match3 is false
```

Note: Match treats *snag* groups (~) as regular units.

1.4.2 Static Match

The `match()` function also can operate statically using two parameters instead of one. The first parameter is still the input that you would like to check, and the second parameter is then the *model* you want to check against. This works either with two assigned onestrings, or in the case of simpler models and inputs, a string of `char`:

```
// Onestrings:
onestring static_model = "^l/";
onestring static_input1 = "a";
bool static1 = Simplex::match(static_input1, static_model);
// static1 is true

// Chars
bool static2 = Simplex::match("g", static_model);
// static2 is true
bool static3 = Simplex::match(static_input1, "^d/");
// static3 is false
bool static4 = Simplex::match("7", "^d/");
// static4 is true
```

Either match function will only return true if the entire input matches the entire model.

1.5 Snag

1.5.1 Simplex Snag

The `snag()` function is used to capture the parts of an input matching snag Units in the model. If the input is not a match for the model, it will return an empty array. Otherwise, it will return a FlexArray of onestrings, and each snag Unit generates its own onestring in the return, whether it is a single character or multi character Unit.

The member function uses the *model* specified when defining the Simplex object, and takes a single parameter, which is either a onestring or a string of `char`, to be processed against the model.

For example, to get *just* the numbers out of a time, you could set up a model like this:

```
onestring model = "~d+/:~d+/ ~[(AM)(PM)]?/";
// This model will snag one or more digits, followed by matching a literal :
// then snag one or more digits again, then optionally snag either AM or PM.
// NOTE: the syntax for the final unit, [(AM)(PM)], is not currently
// implemented in v0.1
Simplex times(model);
FlexArray<onestring> match1 = times.snag("18:05")
// contains ("18", "05", "")
FlexArray<onestring> match2 = times.snag("04:15 PM")
// contains ("04", "15", "PM")
FlexArray<onestring> match3 = times.snag("11 03 pm")
// empty, because it doesn't match the model (no ':')
```

1.5.2 Static Snag

The `snag()` function also can operate statically using two parameters instead of one. The first parameter is still the input that you would like to check, and the second parameter is then the *model* you want to check against. This works either with two assigned onestrings, or in the case of simpler models and inputs, a string of `char`:

```
// Onestrings:
onestring static_model = "~l/";
onestring static_input1 = "a";

FlexArray<onestring> static1 = Simplex::snag(static_input1, static_model);
// static1.at(0) is "a"

// Chars
FlexArray<onestring> static2 = Simplex::match("g", static_model);
// static2.at(0) is "g"
FlexArray<onestring> static3 = Simplex::match(static_input1, "~d/");
// static3 is an empty array
FlexArray<onestring> static4 = Simplex::match("7", "~d/");
// static4.at(0) is "7"
```

Either snag function will only return snag strings if the entire input matches the entire model (in other words, if `match()` returns true). Otherwise the array will be empty.

1.6 Lex

1.6.1 Simplex Lex

The `lex()` member function is used to determine whether the beginning of input matches the specified *model* or not, and also processes any *snag* units included. When used with a generated Simplex object, it takes either a onestring or a string of `char` as a parameter (the input to check), and the function returns a SimplexResult object with the details of the match. The boolean `match` in this object will be true if the beginning of input matches the model, false if not. (It will return true whether there are characters remaining after the matched characters or not. If you want the match to succeed *only* if the entire input matches the entire model, use *match* instead.) If match is true, the FlexArray<onestring> `snag_array` will be populated with the results of any snag units, and `match_length` will contain the total length of matched characters.

Simplex is a single pass parser; if you want to include matches in the middle of an input, build the model accordingly (ie, with a Unit like `^.*/` - which is any character, one or more, optionally - at the beginning of the units you're trying to match or lex).

```
// Using the model example on the main Simplex page:
onestring model = "^d/";
Simplex simplex(model);

onestring lex_input1 = "3";
SimplexResult lex1 = simplex.lex(lex_input1);
// lex1.match is true, and lex1.matched_length = 1
onestring lex_input2 = "abc";
SimplexResult lex2 = simplex.lex(search_input2);
// lex2.match is false, and lex2.matched_length = 0
onestring lex_input3 = "3a";
SimplexResult lex3 = simplex.lex(lex_input3);
// lex3.match is true, and lex3.matched_length = 1
```

Note: Lex contains a true result on the final input, while match would not. It also provides the length of 1 so you can know that only the first character matched the model you provided.

Lex will also handle snag units:

```
// A model for one or more digits
onestring model_snag = "~d+/";
Simplex simplex_snag(model_snag);

onestring lex_input4 = "123";
SimplexResult lex4 = simplex_snag(lex_input4);
// lex4.match is true, lex4.matched_length = 3,
// and lex4.snag_array[0] = "123"
onestring lex_input5 = "123steve";
SimplexResult lex5 = simplex_snag(lex_input5);
// lex5.match is true, lex5.matched_length = 3 because only the first three
// characters match, and lex5.snag_array[0] = "123" again.
```

1.6.2 Static Lex

The `lex()` function also can operate statically using two parameters instead of one. The first parameter is still the input that you would like to check, and the second parameter is then the *model* you want to check against. This works either with two assigned onestrings, or in the case of simpler models and inputs, a string of `char`:

```
// Onestrings:
onestring static_model = "^l/";
onestring static_input1 = "a";
SimplexResult static1 = Simplex::lex(static_input1, static_model);
// static1.match is true, static1.matched_length = 1

// Chars
SimplexResult static2 = Simplex::match("g", static_model);
// static2.match is true
SimplexResult static3 = Simplex::match(static_input1, "^d/");
// static3.match is false
SimplexResult static4 = Simplex::match("7", "^d/");
// static4.match is true, static4.matched_length = 1
```

Lex will return true if the characters at the beginning of the input match the model provided, either in static or member form.

1.7 Simplex Parser

If you are using `match()`, `snag()`, or `lex()`,the Simplex class takes care of everything on this page for you. However, if you want to call *Simplex Parser* directly for a SimplexResult object *without lexing*, you will first need to use `parse_model` to generate the model array.

1.7.1 Parse Model

The `Unit::parse_model` function is a helper function in the Unit class. It generates the Unit models from an input string, and is used for this purpose in the Simplex constructor. It takes a onestring containing the entered model as a parameter, and returns a `FlexArray<Unit>` object containing the parsed Unit objects.

Example:

```
// Create the model and model array objects, then parse
onestring model = "~d+/^l/";
FlexArray<Unit> model_array = Unit::parse_model(model);
```

In this example, the array contains two Units, the first of which will snag one or more digit characters, the second of which will match exactly one latin character.

1.7.2 Simplex Parser Function

The `simplex_parser()` function is internally used by the `match()`, `snag()`, and `lex()` functions to produce their results, but can be accessed directly if needed. This is where Simplex parses the input to determine whether the results match, and snags any matching snag units to return to their respective functions.

This function is not optimized or designed to be used externally, so there are no helpful overloads currently implemented. If you need to use it, though, it takes three parameters: an input string as a `onestring` to check, a `FlexArray<Unit>` containing the parsed model of units, and an optional `bool` if you want to lex rather than match the model exclusively, which defaults to false if not provided. (If you need to generate the FlexArray separately, see *Parse Model*.) It returns a `SimplexResult` object that consists of a boolean `match` (used by `match`), a FlexArray of onestrings `snag_array` (used by `snag`), which will be empty if the match fails, and a unsigned integer `match_length` primarily for use cases of `lex()`.

Example:

```
// After using the model generation in the above example...
onestring input = "123a";
SimplexResult result = simplex_parser(input, model_array);
onestring input2 = "1234";
SimplexResult result2 = simplex_parser(input, model_array);
```

The `result` object now contains the boolean `match`, which is true, the FlexArray `snag_array`, which contains one onestring `123`, and the uint_fast16_t `match_length` which is 4. However, in `result2`, `match` is false, the snag_array is empty, and the match_length is 0, because the input did not contain a match for the `^l/` Unit.

Note: Because of the way the logic works, `match_length` may be unreliable if you are //not// lexing and have not flagged lexing as true.

1.8 Tester Console (from PawLIB)

SIMPLExpress uses the included PawLIB tester application, Goldilocks, which allows you to run tests and benchmarks using GoldilocksShell.

SIMPLExpress currently includes various functionality tests. Please see Goldilocks in PawLIB documentation for more information on Goldilocks tests.

See *Using SIMPLExpress* for instructions on how to build the SIMPLExpress Tester.

Once it's built, you can run the tester from within the SIMPLExpress repository via `./tester`.

For SIMPLExpress test and suite ID naming conventions, see tests.

1.8.1 Interactive Mode

We can start Interactive Mode by running the tester application without arguments, via `./tester`. Type commands at the >> prompt.

All commands are detailed under Goldilocks Interactive.

1.8.2 Command-Line Mode

We can run tests and suites by passing arguments to our `./tester` application. This is especially useful if you want to run tests in a automated manner, such as with a continuous integration system.

You can get help via `./tester --help`.

Multiple commands may be run in a single line. They will be executed in order.

All commands are detailed under Goldilocks Command Line Interface.

1.9 SIMPLExpress Tests

For instructions on using the SIMPLExpress Tester, see console.

1.9.1 Test Namespaces

Because we use Goldilocks for multiple projects at MousePaw Media, we follow certain conventions for test and suite IDs.

The Live-In Testing Standard defines the first part of the ID. For example,

- `X-` refers to the SIMPLExpress project.
- `s` is a suite, while `t` is a test.
- `B` is a "behavior" test, and `S` is a "stress" test, etc.

The first digit indicate the major sector of SIMPLExpress the suite and its tests are related to. The second digit is the specific sector, usually a single class.

Currently, we have just a few test sectors delineated in SIMPLExpress.

ID	Sector
00	Simplex class
01	UnitParser
10	Integration Tests

Any subsequent digits indicate the test number. A number may be shared between behavior and stress tests; both use the same implementation, but vary in their variables (such as iterations).

1.10 Finding Support

If you have any trouble with MousePaw Media projects, we invite you to contact us!

1.10.1 Supported Topics

We officially offer support for the following:

- Compiling and linking to PawLIB.
- Using PawLIB.
- Using the PawLIB Tester.
- Building and linking to MousePaw Media's `libdeps` repository with supported compilers.

We only support use of PawLIB using the supported compilers and environment (see *Environment and Dependencies*).

Note: Due to the complexities of running GCC and Clang on Microsoft Windows, we do not necessary offer support for that operating system. If you are certain that you are running a supported environment on Microsoft correctly, you ARE still welcome to contact us.

Questions about use of CPGF should be directed to that project instead (cpgf.org).

Community support is available for all C++-related questions via the ##c++-friendly chatroom on Freenode IRC.

1.10.2 Contacting Support

- Email: `support@mousepawmedia.com`
- Freenode IRC: #mousepawgames
- Phabricator Ponder on DevNet (available 7am-10pm PST / 1400-0500 UTC).

1.10.3 Bug Reports and Feature Requests

If you encounter a bug in PawLIB, or would like to see a feature added, we encourage you to file a report on DevNet Phabricator Maniphest.

> **Warning:** We do **not** monitor pull requests or issues on GitHub!

To file a bug report or feature request:

1. Go to DevNet During hours (7am-10pm PST / 1400-0500 UTC), click *Connect Now*.
2. Click *Phabricator* from the main menu.
3. Sign in using your GitHub account. If this is your first time…
    - Authorize the *DevNet [MousePaw Media]* OAuth App.
    - Thoroughly read and agree to the *Community Rules*. We kept those concise, to make them easier to read and understand.
4. On the Phabricator menu at left, select *Maniphest*.
5. In the upper-right corner, select *Create Task* and choose either *Bug Report* or *Feature Request*.
6. See the link at the top of that form for instructions on how to craft a useful bug report or feature request.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search