ph

A c++ library, build tool, interpreter, and IDE

Introduction

Ph is a software project that aims to be the facto software for both building and testing the best C++ software. It aims to cut loose from other languages in the build process, instead integrating them directly in the source code such as this:

```
import Ph;

using namespace ph;

auto main (int i, char** s) -> int
{

    Arguments auto args = parse_args (i, s);

    Error auto err = len (args) > 0 ? true : false;

    if (err)
    {

    }

    return err;
}
```

Figure 1:

"Ph" is a lightning fast C++ library and also a build tool (such as Cmake or Make) for turning C++ source code into something useful.

It introduces c++ concepts and modules, by which the former it is built around,

thus, making it lucrative for industrial applications where the need for precision is critical, and where most of the bugs can be catched before the c++ files are even compiled, thus simply just generating syntax errors for developers. Introducing this core language feature has huge pros for industrial-strength generic components, AKA good software. This is what we want! We dont want python's "throw in whatever type u desire", and not java's somewhat pragmatic "please specify the type". We want to be able to say "hey, other coders out there using this function im about to type, just throw in a String". String is just a concept that we specify. It could be everything from "std::string" to old plain c string "char const *". We say "hey, String can be either a "std::string" or a "char const*"*. OR it could be anything that we could do the following with:

```
template <typename S>
concept String = requires (S s)
{
    s.size ();
    {s [0]} -> char;
}
```

Goals with this project ==========

- ☒ Motivation
- ☐ Just c++, even for building.

  All software are built around a set of programming languages, often one for front-end and one for back-end. The reason for this, unknown. One can only guess.

  I can not stress this enough, but writing software in one language has huge upside effects, and it's much cheaper. Your teams can speek the same language, thus making it much easier for further intrigues.
- ☐ C++ interpreter

  Similar to python interpreter.

Details

file dependency

Project dependencies

The following graph describes that basically "ph" is a set of files which will either be transformed into a documentation file or into the executable software.

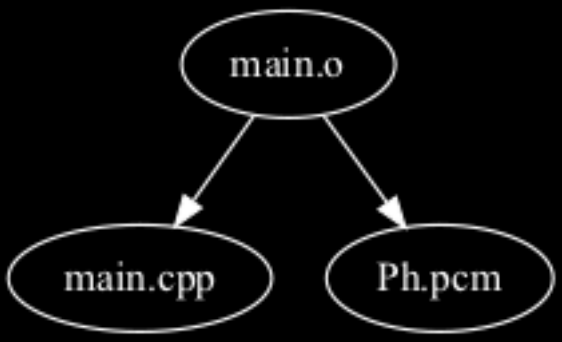

Figure 2:

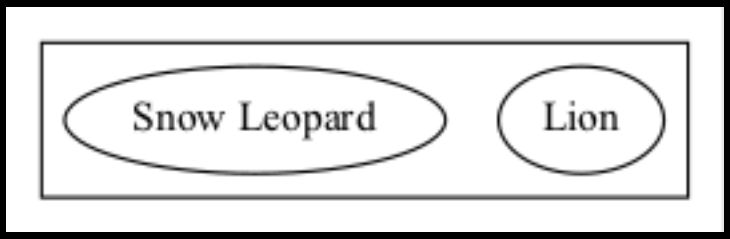

Figure 3:

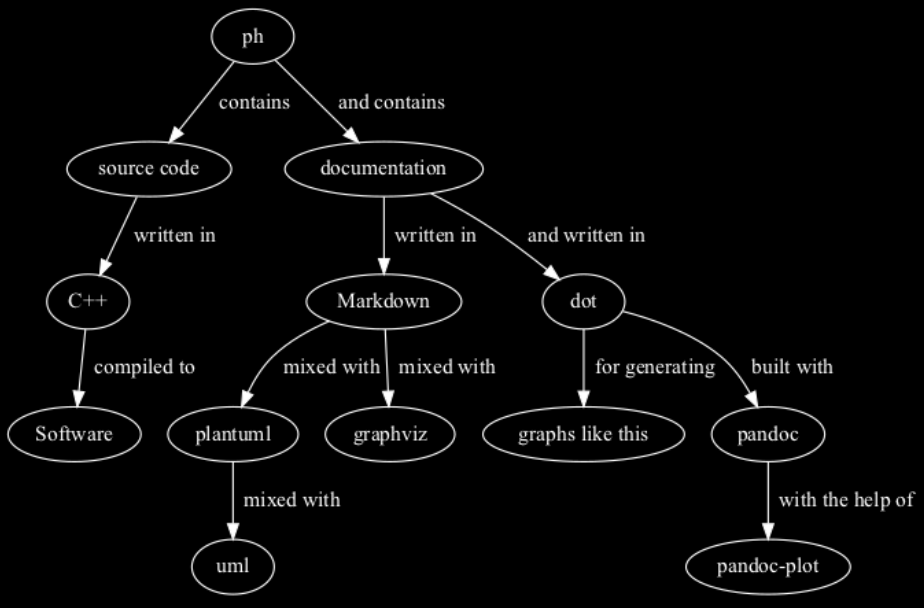

Figure 4:

Source code written in

- C++

and built with

- Make

About

Documentation

The documentation for Ph is dependent on the the following languages:

- Markdown
- dot
- UML
- English

and built with

- pandoc
- pandoc-plot
- pandoc-plantuml-filter

Contribute

At the moment, the project is very dependent on Make (for compiling C++ files into different things and then finally everything into an executable).

Also pandoc (for documentations).

Reasons for this? It's still very young.

Please help me develop this project! At the moment there are just one developer.

Licensing

Can be either open or proprietary.

Concepts architecture

Iterator-relations

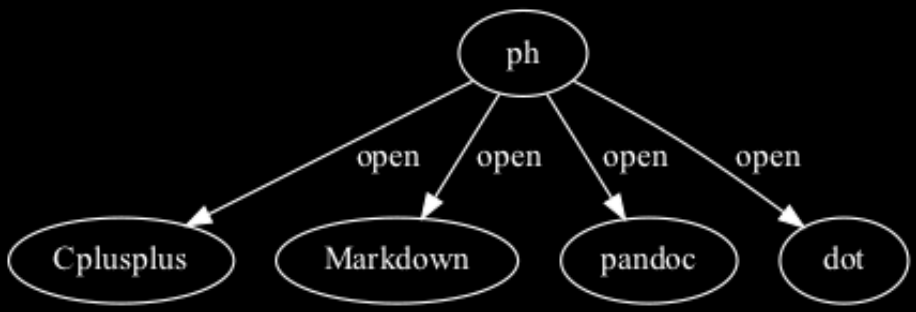

Figure 5:

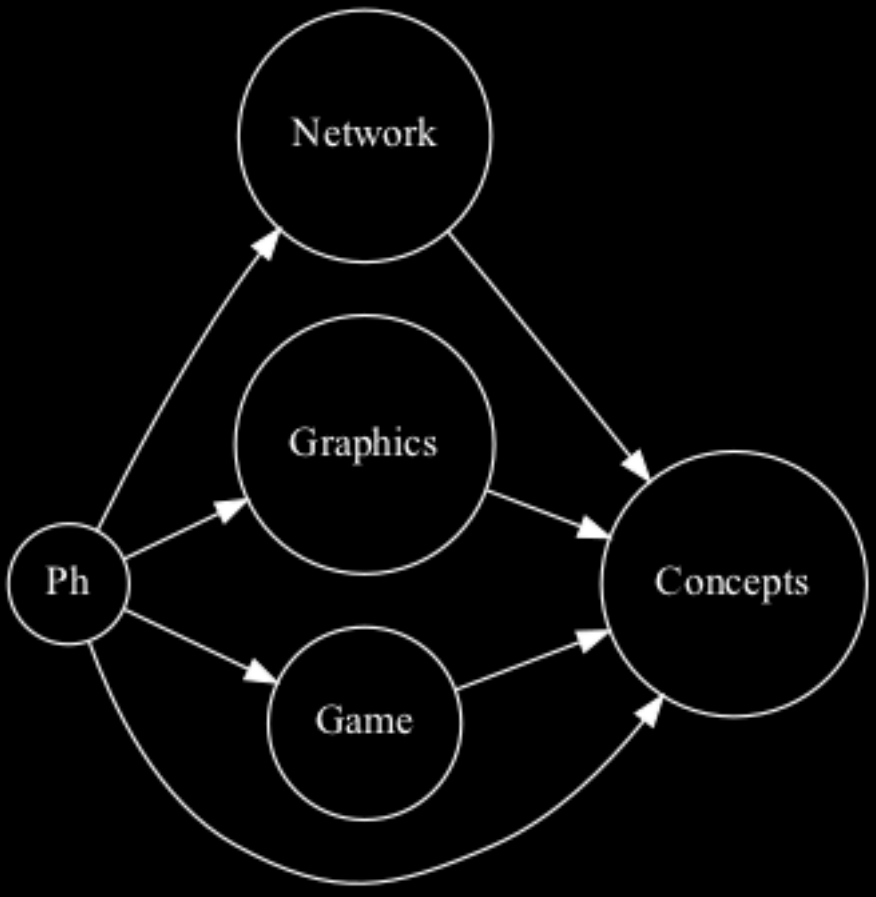

Figure 6:

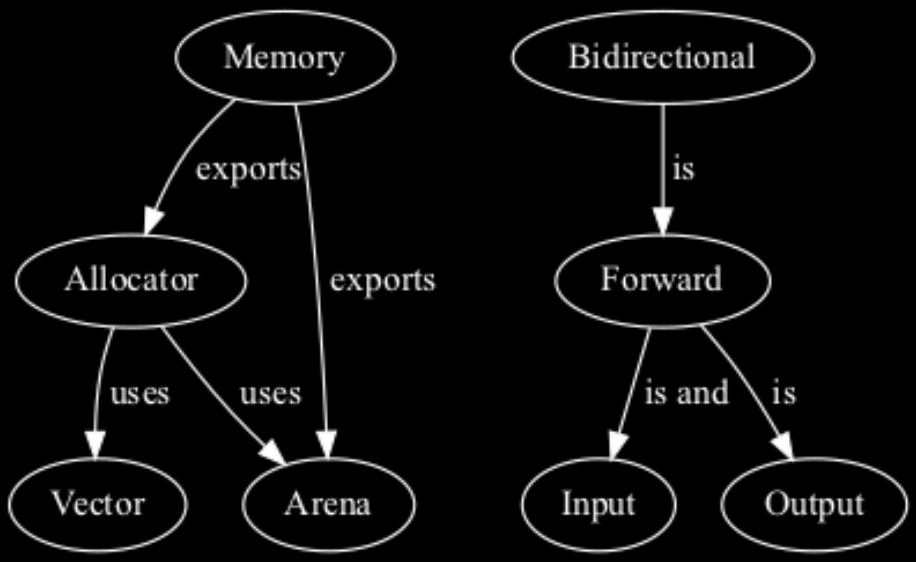

Figure 7:

Ph.Language

"A simple, yet a powerful programming language aimed against code repetition"

Ph.Language is a new programming language developed by Philip Wenkel. It is a simple, yet efficient, programming language written in the high performance language C++. It makes a great antidote for text repetition, whether it is for coding or really anything else. It also makes it much easier to create template files or folder structures for your projects, which drastically improves your production time and prevents you from making simple errors. I bet that, if you are like most people, creating a new project can really take time and effort. Probably you have some prepared base project structure which you pretty much copy-paste to the new one and just rename everything to fit your current project name. Enough of words, lets look at an example of how to use Ph.Language to simplify code repetition in c++. Then, we will look at how to use Ph.Language as a tool when writing a simple document. Last but not least, we will see how it can also be used with files and folders.

```
export module Ph.Concepts;

export import Ph.Concepts.Tuple;
export import Ph.Concepts.Bool;
export import Ph.Concepts.Done;
export import Ph.Concepts.Size;
```

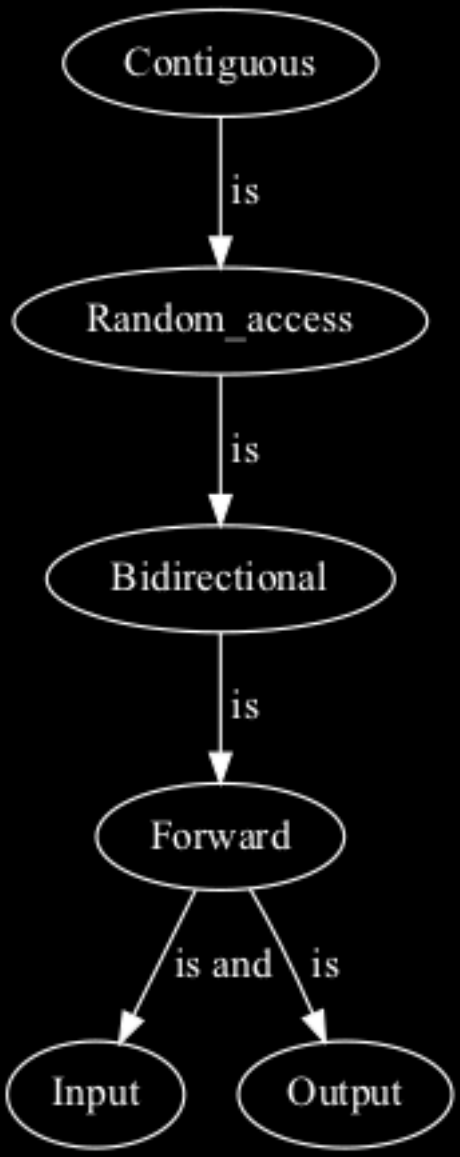
Contiguous
is
Random_access
is
Bidirectional
is
Forward
is and
is
Input
Output

Figure 8:

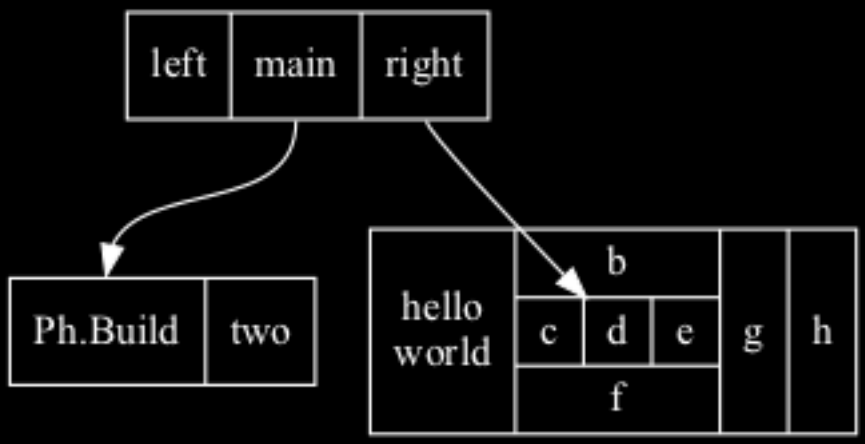

Figure 9:

```
export import Ph.Concepts.Bit;
export import Ph.Concepts.Byte;
export import Ph.Concepts.Core;
export import Ph.Concepts.Char;
export import Ph.Concepts.String;
export import Ph.Concepts.Strings;
export import Ph.Concepts.Pointer;
export import Ph.Concepts.Number;
export import Ph.Concepts.Error;
export import Ph.Concepts.Void;

export import Ph.Concepts.Iterator;
export import Ph.Concepts.Array;

export import Ph.Concepts.Class;
export import Ph.Concepts.Enum;
export import Ph.Concepts.Function;
export import Ph.Concepts.Any_of;
export import Ph.Concepts.File;
export import Ph.Concepts.Vector;
export import Ph.Concepts.Common;
export import Ph.Concepts.Constant;
export import Ph.Concepts.Range;
export import Ph.Concepts.Element;a
export import Ph.Concepts.Types;

export module Ph.Concepts;
```

```
@ (library) =
    Tuple
    Bool
    Types

$ (library : add library)
{
    export import Ph.Concepts.${library}
}
```

or

```
export module Ph.Concepts;
@ (add library) -> {export import Ph.Concepts.${0}}
@ (library) =
    Tuple
    Bool
    Types
```

```
template <int>
struct Foo;

template <>
struct Foo <0>
{
    inline static constexpr int i = 0;
};

template <>
struct Foo <1>
{
    inline static constexpr int i = 1;
};

template <>
struct Foo <2>
{
    inline static constexpr int i = 2;
};
```

As you can see, every template specialization of Foo is pretty much the same, except for two places, where only a number changes. Lets use Ph.Language to help us out with this boring and cumbersome code repetition.

```
template <int>
struct Foo;

@(type){inline static constexpr int}
```

```
$(0 i 3)
{
    template <>
    struct Foo <${i}>
    {
        ${type} i = ${i};
    };
}
```

That's it!

Lets write a cv for our new job application!

```
@(first name){Philip}
@(last name){Wenkel}
@(name){${first name} ${last name}}
@(company name){Google}

My name is ${name} and I am interested in $(job){coding} at your company ${company name}.
On my spare time, i love ${job}! #{elaborate on this one...}

Yours sincerely, ${name}
```

This will result in the following output:

```
My name is Philip Wenkel and I am interested in coding at your company Google.
On my spare time, i love coding!

Yours sincerely, Philip Wenkel
```

Usage

```
Ph.Language input_file.txt output_file.txt
```