
Hippo

Release 0.2.3

Caleb Zulawski

Oct 14, 2019

CONTENTS

1	Hierarchical Information as Pretty-Printed Objects	1
1.1	Introduction	1
1.2	Printing user-defined types via reflection	2
1.3	Formatting	4
1.4	Out-of-the-box type support	9
1.5	Advanced usage	11
1.6	Can't find what you're looking for?	13
	Index	15

HIERARCHICAL INFORMATION AS PRETTY-PRINTED OBJECTS

Hippo is a header-only library for C++17 that provides facilities for pretty-printing user-defined types.

For more information on a specific feature, see the pages below:

1.1 Introduction

1.1.1 Using Hippo

Hippo is a header-only library, so install the headers however you'd like.

To begin using Hippo, include the following:

```
#include "hippo/hippo.h"
```

1.1.2 Printing a value

Printing values is performed by the `hippo::print()` or `hippo::print_to()` functions. Both functions take a value and a `hippo::configuration` and produce a pretty-printed output.

A simple example of printing a vector:

```
#include "hippo/hippo.h"
#include "hippo/std/vector.h"
#include <iostream>

int main() {
    std::vector<int> v {0, 1, 2};
    hippo::print_to(std::cout, v, hippo::configuration());
}
```

This example will print:

```
std::vector [0, 1, 2]
```

1.1.3 Interface

struct configuration

Global configuration values applied to all printers.

Public Members

`std::uint64_t indent`

The number of spaces to indent for each indentation level (defaults to 0)

`std::uint64_t width`

The number of output columns, not a hard limit but best-effort (defaults to 60)

template<typename T>

`std::vector<std::string> hippo::print(const T &t, const hippo::configuration &config)`

Print any printable value `t` with configuration `config`

template<typename T>

`std::ostream &hippo::print_to(std::ostream &os, const T &t, const hippo::configuration &config)`

Print any printable value `t` with configuration `config` to the specified `std::ostream`

1.2 Printing user-defined types via reflection

A key feature of Hippo is the ease of printing user-defined types.

1.2.1 Classes

Hippo provides utilities for printing user-defined types. Consider the following types:

```
struct Foo {  
    int a;  
    float b;  
};  
  
struct Bar {  
    std::vector<Foo> foos;  
};
```

To print these types, we reflect them using `HIPPO_CLASS_BEGIN`, `HIPPO_MEMBER`, and `HIPPO_CLASS_END`:

```
#include "hippo/hippo.h"           // reflection macros  
#include "hippo/std/vector.h"      // std::vector printer  
  
HIPPO_CLASS_BEGIN(Foo)  
    HIPPO_MEMBER(a)  
    HIPPO_MEMBER(b)  
HIPPO_CLASS_END()  
  
HIPPO_CLASS_BEGIN(Bar)  
    HIPPO_MEMBER(foos)  
HIPPO_CLASS_END()
```

The printers for `int`, `float`, and `std::vector` are all provided by Hippo. Once we've provided the printer for `Foo`, we are able to use it to print `Bar`. A printed instance of `Bar` might look something like this:

```
Bar {  
  foos: std::vector [  
    Foo { a: 1, b: 0.5 },  
    Foo { a: 2, b: -3.1 }  
  ]  
}
```

(continues on next page)

(continued from previous page)

```

    ]
}

```

1.2.2 Enums

Like classes, enums can be reflected with `HIPPO_ENUM_BEGIN`, `HIPPO_ENUM_VALUE`, and `HIPPO_ENUM_END`:

```

enum Foo {
    Bar,
    Baz
};

HIPPO_ENUM_BEGIN(Foo)
    HIPPO_ENUM_VALUE(Bar)
    HIPPO_ENUM_VALUE(Baz)
HIPPO_ENUM_END()

```

1.2.3 Base classes

Hippo can also reflect base classes with `HIPPO_BASE`:

```

struct Foo : Bar {
    /* members */
};

HIPPO_CLASS_BEGIN(Foo)
    HIPPO_BASE(Bar)
    /* members */
HIPPO_CLASS_END()

```

1.2.4 Custom member access expressions

In some cases, it's useful to use another expression to access a member. This is accomplished by using the `HIPPO_MEMBER_EXPR` macro, which allows a custom expression to be provided, operating on the input object:

```

class Foo {
    int bar;
public:
    Foo(int bar) : bar(bar) {}
    int get_bar() const { return bar; }
};

HIPPO_CLASS_BEGIN(Foo)
    HIPPO_MEMBER_EXPR(bar, object.get_bar())
HIPPO_CLASS_END()

```

1.2.5 Interface

Class reflection

HIPPO_CLASS_BEGIN (Type)

Begin the definition of a printer specialization for a class Type

HIPPO_CLASS_END ()

End the definition of a printer specialization for a class.

HIPPO_BASE (Type)

Register Type as a base class in a class printer specialization.

HIPPO_MEMBER (Name)

Register Name as a member in a class printer specialization.

HIPPO_MEMBER_EXPR (Name, Expression)

Register Name as a member, printed as Expression, in a class printer specialization.

Enum reflection

HIPPO_ENUM_BEGIN (Type)

Begin the definition of a printer specialization for an enum Type

HIPPO_ENUM_END ()

End the definition of a printer specialization for an enum.

HIPPO_ENUM_VALUE (Value)

Register an enum value named Value

1.3 Formatting

Some printable types support formatting. Formatting is applied with the `hippo::formatter` adapter, which itself is a printable type that applies a format to its contents.

1.3.1 Formatting numbers

The following example shows how numbers can be formatted for a user-defined type:

```
struct Foo {
    int bar;
    float baz;
};

static hippo::integer_format hex() {
    hippo::integer_format fmt;
    fmt.base = hippo::integer_format::base_type::hex;
    return fmt;
}

static hippo::float_format scientific() {
    hippo::float_format fmt;
    fmt.format = hippo::float_format::notation_type::scientific;
    return fmt;
};
```

(continues on next page)

(continued from previous page)

```

HIPPO_CLASS_BEGIN(Foo)
    HIPPO_CLASS_MEMBER_EXPR(Foo, hippo::formatter(object.bar, hex()))
    HIPPO_CLASS_MEMBER_EXPR(Foo, hippo::formatter(object.baz, scientific()))
HIPPO_CLASS_END()

```

1.3.2 Using formatting to print polymorphic types

Polymorphic types can be printed by use of `hippo::derived_type_printer`:

```

#include "hippo/hippo.h"
#include "hippo/std/memory.h"
#include <iostream>

struct Foo {
    virtual ~Foo() = default;
};

struct Bar : Foo {};
struct Baz : Foo {};

HIPPO_CLASS_BEGIN(Foo)
HIPPO_CLASS_END()

HIPPO_CLASS_BEGIN(Bar)
    HIPPO_BASE(Foo)
HIPPO_CLASS_END()

HIPPO_CLASS_BEGIN(Baz)
    HIPPO_BASE(Foo)
HIPPO_CLASS_END()

int main() {
    std::shared_ptr<Foo> foo = std::make_shared<Foo>();
    std::shared_ptr<Foo> bar = std::make_shared<Bar>();
    std::shared_ptr<Foo> baz = std::make_shared<Baz>();
    hippo::dynamic_type_format<Foo> dyn_fmt;
    dyn_fmt.printers.push_back(std::make_shared<hippo::derived_type_printer<Foo, Bar>>
↳ ());
    dyn_fmt.printers.push_back(std::make_shared<hippo::derived_type_printer<Foo, Baz>>
↳ ());
    hippo::pointer_format<Foo> fmt = std::move(dyn_fmt);
    hippo::print_to(std::cout, hippo::formatter(foo, fmt), hippo::configuration());
    hippo::print_to(std::cout, hippo::formatter(bar, fmt), hippo::configuration());
    hippo::print_to(std::cout, hippo::formatter(baz, fmt), hippo::configuration());
}

```

Once Bar and Baz are registered with the pointer format, the printer is able to use RTTI to determine which printer to use. The following is printed:

```

std::shared_ptr containing [ Foo { } ]
std::shared_ptr containing [ Bar { Base Foo { } } ]
std::shared_ptr containing [ Baz { Base Foo { } } ]

```

1.3.3 Interface

```
template<typename T>
struct formatter
```

A printable type that applies formats to other printable types.

Public Types

```
template<>
using value_type = std::remove_const_t<T>
    The type to format.

template<>
using printer_type = hippo::printer<value_type>
    The printer for T

template<>
using format_type = typename printer_type::format_type
    The format configuration for T
```

Public Functions

```
formatter (const value_type &value, const format_type &format)
    Construct a formatter that prints value with the format described by format. The constructed
    formatter does not own value or format, so both must remain in scope for the lifetime of the
    formatter.
```

```
template<typename T>
struct formatter<T*>
    Specialization of formatter for pointer types.
```

Public Types

```
template<>
using value_type = std::remove_const_t<std::decay_t<T>>
    The type to format.

template<>
using printer_type = hippo::printer<value_type*>
    The printer for T

template<>
using format_type = typename printer_type::format_type
    The format configuration for T
```

Public Functions

```
formatter (const value_type *value, const format_type &format)
    Construct a formatter that prints value with the format described by format. The constructed
    formatter does not own value or format, so both must remain in scope for the lifetime of the
    formatter.
```

```
struct no_format
    Format for non-formattable types.
```

Number format configurations

struct integer_format
Format for integer values.

Public Types

enum base_type
Integer base description.

Values:

oct
Octal.

dec
Decimal.

hex
Hexadecimal.

Public Members

base_type **base**
Numeric base.

struct float_format
Format for floating-point values.

Public Types

enum notation_type
Notation format description.

Values:

standard
Format with `std::defaultfloat`

fixed
Format with `std::fixed`

scientific
Format with `std::scientific`

Public Members

notation_type **notation**
Notation format, defaults to `standard`

`std::optional<std::size_t>` **precision**
Precision for `std::setprecision`

Pointer configurations

using `hippo::pointer_format` = `std::variant<standard_pointer_format<T>, address_format, dynamic_type_format<T>>`
Format for printing a pointer.

`template<typename T>`

struct `standard_pointer_format`

Format option for non-polymorphic pointers. A non-null pointer is dereferenced and printed.

Public Types

`template<>`

using `format_type` = `typename hippo::printer::format_type`

Format type of T

Public Members

`format_type` **format**

The format used for printing.

struct `address_format`

Format option for printing pointers as addresses (rather than printing the dereferenced pointer)

`template<typename T>`

struct `dynamic_type_format`

Format option for printing polymorphic types. A non-null pointer is checked against the registered types, dereferenced, and printed.

Public Types

`template<>`

using `base_format_type` = `typename hippo::printer<T>::format_type`

Format type of the base class.

Public Members

`std::vector<std::shared_ptr<base_type_printer<T>>>` **printers**

Printers for derived types, in preference order.

Printers are called one by one and returns the first successful output.

`base_format_type` **base_format**

If none of the derived printers are successful, the base class is printed with this format.

`template<typename Base>`

struct `base_type_printer`

Abstract base for printers of polymorphic pointers.

Subclassed by `hippo::derived_type_printer< Base, Derived >`

Public Functions

```
virtual std::optional<hippo::object> print (const Base *b, std::uint64_t current_indent, const
hippo::configuration &config) = 0
```

Prints b if possible, otherwise the return value is empty.

```
template<typename Base, typename Derived>
struct derived_type_printer : public hippo::base_type_printer<Base>
Printer for a polymorphic type from a base class pointer.
```

Public Types

```
template<>
using printer_type = hippo::printer<Derived>
Printer specialization for Derived

template<>
using format_type = typename printer_type::format_type
Format type of Derived
```

Public Functions

```
derived_type_printer()
Construct a printer using the default format.

derived_type_printer(const format_type &format)
Construct a printer using the specified format format

std::optional<hippo::object> print (const Base *b, std::uint64_t current_indent, const
hippo::configuration &config)
Prints b if it is a Derived, otherwise returns nothing.
```

1.4 Out-of-the-box type support

As discussed in *Printing user-defined types via reflection*, struct, class, enum, and enum class are all supported via macros.

In addition to user defined types, most types provided by the language are automatically supported. All builtin types are supported, as well as many from the standard library.

1.4.1 Supported standard library types

Strings

In addition to `const char *`, Hippo supports `std::string` via "hippo/std/string.h".

Containers

Support for all containers is available:

- `std::array` via "hippo/std/array.h"

- `std::vector` via `"hippo/std/vector.h"`
- `std::list` via `"hippo/std/list.h"`
- `std::forward_list` via `"hippo/std/forward_list.h"`
- `std::deque` via `"hippo/std/deque.h"`
- `std::set` and `std::multiset` via `"hippo/std/set.h"`
- `std::unordered_set` and `std::unordered_multiset` via `"hippo/std/unordered_set.h"`
- `std::map` and `std::multimap` via `"hippo/std/map.h"`
- `std::unordered_map` and `std::unordered_multimap` via `"hippo/std/unordered_map.h"`

All containers can be formatted with the format configuration of the inner type(s). Map types can be formatted with:

```
using hippo::map_format = std::pair<typename hippo::printer<Key>::format_type, typename hippo::printer<Value>::format_type>;  
    Format for map types.
```

Tuples

Both `std::pair` and `std::tuple` are supported, by `"hippo/std/utility.h"` and `"hippo/std/tuple.h"`, respectively.

They can be formatted with:

```
using hippo::pair_format = std::pair<typename hippo::printer<First>::format_type, typename hippo::printer<Second>::format_type>;  
    Format for std::pair  
using hippo::tuple_format = std::tuple<typename hippo::printer<T>::format_type...>;  
    Format for std::tuple
```

Smart pointers

In addition to plain pointers, `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr` are supported via `"hippo/std/memory.h"`. These types are all formattable by `hippo::pointer_format`.

Sum types

`std::optional` is supported via `"hippo/std/optional.h"` and is formattable with the inner type's format configuration. `std::variant` is supported via `"hippo/std/variant.h"` and is formattable with:

```
using hippo::variant_format = std::variant<typename hippo::printer<T>::format_type...>;
```

Chrono

`std::chrono::duration` is supported via `"hippo/std/chrono.h"` and is formattable with the inner type's format configuration.

Complex

`std::complex` is supported via `"hippo/std/complex.h"` and is formattable with the inner type's format configuration.

Atomic

`std::atomic` is supported via `"hippo/std/atomic.h"` and is formattable with the inner type's format configuration.

Bitset

`std::bitset` is supported via `"hippo/std/bitset.h"` and is not formattable.

1.5 Advanced usage

Sometimes it is necessary to access the internal workings to create a more complicated printer.

1.5.1 Representing of lines of text

Before we get into printing types, we must understand how outputs are represented. In Hippo, a line of text is represented by `hippo::line`, which tracks the indentation level of the line separately from the contents.

struct line

Describes a printed line of text.

Public Functions

`line` (`std::uint64_t indent`)

Construct an empty line with the given indentation level.

`line` (`std::uint64_t indent`, `std::string string`)

Construct a line with the given indentation level and string.

Public Members

`std::uint64_t indent`

The indentation level of the line.

`std::string string`

The contents of the line.

When an object is printed, the generated lines of text are then collected into a `hippo::object`. Any lines that are shorter than `hippo::configuration::width` are condensed into a single line if possible. Multiple `hippo::object` may be condensed as well, but only if all of the objects are a single line.

`using hippo::object = std::variant<hippo::line, std::list<hippo::line>>`

Describes the printed output of any object, either as a single or multiple lines.

`inline hippo::object hippo::condense(const std::list<hippo::line> &lines, const hippo::configuration &config)`

Condense a collection of lines into a single object. Multiple lines will be condensed into one if the indented result is less than the configured output width.

```
inline hippo::object hippo::condense(const std::list<hippo::object> &objects, const
                                     hippo::configuration &config)
```

Condense a collection of objects into a single object. If any of the input objects are multiline, the output is not condensed, otherwise the lines will be condensed if the indented result is less than the configured output width.

1.5.2 Defining a printer

Printers are added for a type by specializing the `hippo::printer` struct. This class is declared as follows:

```
template<typename T, typename U = T>
```

```
struct printer
```

The core pretty-printer type. T is the type to be printed. U is provided for optionally making SFINAE possible.

Specializations must fulfill the following interface:

```
template<> hippo::printer<Foo> {
    using format_type = /* any default-constructible and copy-constructible type */
    static ::hippo::object print(const Foo &f,
                                std::uint64_t current_indent,
                                const ::hippo::configuration &config,
                                const format_type &format = format_type());
}
```

1.5.3 Convenient utilities

The following operations are so common when creating printers that Hippo provides them.

Manipulating lines

```
struct prepend_visitor
```

Visitor over objects that prepends a string to the first line.

Public Functions

```
void operator() (hippo::line &line)
```

Prepend to a single line.

```
void operator() (std::list<hippo::line> &lines)
```

Prepend to the beginning of many lines.

Public Members

```
std::string prefix
```

The string to prepend.

```
struct append_visitor
```

Visitor over objects that appends a string to the last line.

Public Functions

void **operator**() (hippo::*line* &line)
Append to a single line.

void **operator**() (std::list<hippo::*line*> &lines)
Append to the end of many lines.

Public Members

std::string **suffix**
The string to append.

Formatting values

```
template<typename T>
std::enable_if_t<std::is_floating_point_v<T>, std::string> hippo::apply_format (T value, const
float_format &fmt)
```

Apply format *fmt* to floating-point value

```
template<typename T>
std::enable_if_t<std::is_integral_v<T>, std::string> hippo::apply_format (T value, const
integer_format &fmt)
```

Apply format *fmt* to integer value

```
template<typename T>
hippo::object hippo::apply_format (const T *value, std::uint64_t current_indent, const
hippo::configuration &config, const pointer_format<T> &fmt)
Apply format fmt to pointer value using the current_indent indentation level and configuration
config.
```

1.6 Can't find what you're looking for?

- [genindex](#)
- [search](#)

H

hippo::address_format (C++ class), 8
 hippo::append_visitor (C++ class), 12
 hippo::append_visitor::operator() (C++ function), 13
 hippo::append_visitor::suffix (C++ member), 13
 hippo::apply_format (C++ function), 13
 hippo::base_type_printer (C++ class), 8
 hippo::base_type_printer::print (C++ function), 9
 hippo::condense (C++ function), 11
 hippo::configuration (C++ class), 1
 hippo::configuration::indent (C++ member), 2
 hippo::configuration::width (C++ member), 2
 hippo::derived_type_printer (C++ class), 9
 hippo::derived_type_printer::derived_type_printer (C++ function), 9
 hippo::derived_type_printer::print (C++ function), 9
 hippo::derived_type_printer<Base, Derived>::format_type (C++ type), 9
 hippo::derived_type_printer<Base, Derived>::printer_type (C++ type), 9
 hippo::dynamic_type_format (C++ class), 8
 hippo::dynamic_type_format::base_format (C++ member), 8
 hippo::dynamic_type_format::printers (C++ member), 8
 hippo::dynamic_type_format<T>::base_format_type (C++ type), 8
 hippo::float_format (C++ class), 7
 hippo::float_format::fixed (C++ enumerator), 7
 hippo::float_format::notation (C++ member), 7
 hippo::float_format::notation_type (C++ enum), 7
 hippo::float_format::precision (C++ member), 7
 hippo::float_format::scientific (C++ enumerator), 7
 hippo::float_format::standard (C++ enumerator), 7
 hippo::formatter (C++ class), 6
 hippo::formatter::formatter (C++ function), 6
 hippo::formatter<T *> (C++ class), 6
 hippo::formatter<T *>::format_type (C++ type), 6
 hippo::formatter<T *>::printer_type (C++ type), 6
 hippo::formatter<T *>::value_type (C++ type), 6
 hippo::formatter<T>::format_type (C++ type), 6
 hippo::formatter<T>::printer_type (C++ type), 6
 hippo::formatter<T>::value_type (C++ type), 6
 hippo::integer_format (C++ class), 7
 hippo::integer_format::base (C++ member), 7
 hippo::integer_format::base_type (C++ enum), 7
 hippo::integer_format::dec (C++ enumerator), 7
 hippo::integer_format::hex (C++ enumerator), 7
 hippo::integer_format::oct (C++ enumerator), 7
 hippo::line (C++ class), 11
 hippo::line::indent (C++ member), 11
 hippo::line::line (C++ function), 11
 hippo::line::string (C++ member), 11
 hippo::map_format (C++ type), 10
 hippo::no_format (C++ class), 6
 hippo::object (C++ type), 11
 hippo::pair_format (C++ type), 10
 hippo::pointer_format (C++ type), 8
 hippo::prepend_visitor (C++ class), 12

`hippo::prepend_visitor::operator()` (C++
function), 12

`hippo::prepend_visitor::prefix` (C++ mem-
ber), 12

`hippo::print` (C++ function), 2

`hippo::print_to` (C++ function), 2

`hippo::printer` (C++ class), 12

`hippo::standard_pointer_format` (C++
class), 8

`hippo::standard_pointer_format::format`
(C++ member), 8

`hippo::standard_pointer_format<T>::format_type`
(C++ type), 8

`hippo::tuple_format` (C++ type), 10

`hippo::variant_format` (C++ type), 10

`HIPPO_BASE` (C macro), 4

`HIPPO_CLASS_BEGIN` (C macro), 4

`HIPPO_CLASS_END` (C macro), 4

`HIPPO_ENUM_BEGIN` (C macro), 4

`HIPPO_ENUM_END` (C macro), 4

`HIPPO_ENUM_VALUE` (C macro), 4

`HIPPO_MEMBER` (C macro), 4

`HIPPO_MEMBER_EXPR` (C macro), 4