IOSqueak Documentation

Release 2.0

MousePaw Media

Nov 09, 2022

CONTENTS

IOSqueak provides tools for creating beautifully formatted output, without having to memorize arcane codes. It also allows you to categorize and prioritize your messages, and route them wherever they're needed.

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

CHAPTER

ONE

CONTENTS

1.1 Channel

1.1.1 What is Channel?

Channel is designed as a wrapper and, depending on usage, a replacement for `std::cout` and `printf()`. Its sports a number of unique and useful features.

- Multiple asynchronous outputs.
- Message priorities (verbosity).
- Message categories.
- Built-in output formatting.
- Advanced memory tools.

1.1.2 Setting up Channel

Including Channel

To include Channel, use the following:

```
#include "iosqueak/channel.hpp"
```

Channel Instance

For convenience, a single static global instance of Channel, `ioc`, exists in that header. It is suitable for most purposes, though a custom channel instance may be declared. All inputs and outputs that the developer wishes to interface with one another via this class must share the same instance.

Documentation Assumptions

For purposes of expediency, the default global static instance `ioc` will be used in this documentation.

1.1.3 Concepts

Channel uses two unique concepts, **Verbosity** and **Category**, to determine where and how a message is routed.

Category

The benefit to having categories on messages is that you can route different kinds of messages to different outputs. For example, you might send all errors and warnings to a debug terminal, and reserve "normal" messages for game notifications.

Category	Enum	Use
Normal	`IOCat::normal`	Regular use messages, especially those you want the user to see.
Warning	`IOCat::warning`	Warnings about potential problems.
Error	`IOCat::error`	Error messages.
Debug	`IOCat::debug`	Messages that might help you track down problems.
Testing	`IOCat::testing`	Messages related solely to testing.
All	`IOCat::all`	All of the above.

One of the advantages of this system is that you can actually leave messages in the code, and just control when and how they are processed and broadcast. This means you can actually ship with debugging statements still alive in the code, allowing you to diagnose problems on any machine.

You can control which of these categories messages are broadcast from using the echo settings (*Internal Broadcast Settings (Echo)*) and signals (*Category Signals (signal_c_...)*).

Verbosity

Some messages we need to see every time, and others only in special circumstances. This is what verbosity is for.

Verbosity	Enum	Use
Quiet	`IOVrb::quiet`	Only essential messages and errors. For normal end-use. Shipping default.
Normal	`IOVrb::normal`	Common messages and errors. For common and normal end-user testing.
Chatty	`IOVrb::chatty`	Most messages and errors. For detailed testing and debugging.
TMI	`IOVrb::tmi`	Absolutely everything. For intense testing, detailed debugging, and driving the developers crazy.

One example of verbosity in action would be in debugging messages. A notification about a rare and potentially problematic function being called might be `IOVrb::normal`, while the output of a loop iterator would probably be `IOVrb::tmi`.

You can control which of these categories messages are broadcast from using the echo settings (*Internal Broadcast Settings (Echo)*) and signals (*Verbosity Signals (signal_v_...)*).

1.1.4 Output

General

All output is done using the stream insertion (<<) operator, in the same manner as with `std::cout`. Before a message is broadcast, a stream control flags such as `IOCtrl::endl` must be passed.

`IOCtrl::endl` serves as an "end of transmission" [EoT] flag, clears any formatting set during the stream, and inserts a final newline character before flushing the stream. Thus, `\n` is not needed if the output should be displayed on a single line. This functionality also allows a single transmission to be split up over multiple lines, if necessary. Other stream control enumerations have different behaviors. (See *Stream Control*)

```
ioc << "This is the first part. ";
//Some more code here.
ioc << "This is the second part." << IOCtrl::endl;
```

Strings

Channel natively supports string literals, cstring (char arrays), `std::string`, and `onestring`.

These are passed in using the << operator, as with anything being output via Channel. The message will not be broadcast until an EoT (end-of-transmission) flag is passed.

```
ioc << "Hello, world!" << IOCtrl::endl;
//OUTPUT: "Hello, world!"

char* cstr = "I am a Cstring.\0";
ioc << cstr << IOCtrl::endl;
//OUTPUT: "I am a Cstring."

std::string stdstr = "I am a standard string.";
ioc << stdstr << IOCtrl::endl;
//OUTPUT: "I am a standard string."
```

Formatting

Cross-platform output formatting is built in to Channel. This means that formatting can be set using the IOFormat flags, and it will display correctly on each output and environment.

```
ioc << IOFormatTextAttr::bold << IOFormatTextFG::red << "This is bold, red text. "
    << IOFormatTextAttr::underline << IOFormatTextFG::blue << IOFormatTextBG::yellow <<
↪"This is bold, underline, blue text with a yellow background. "
    << IOFormatTextAttr::none << IOFormatTextFG::none << IOFormatTextBG::none << "This␣
↪is normal text."
    << IOCtrl::endl;
//The output is exactly what you'd expect.
```

Important: Currently, only ANSI is used. Formatting-removed and an easy-to-parse formatting flag system for custom outputs will be added soon.

Alternative, you can use the `IOFormat` object to store multiple flags. (See *Formatting Objects*)

Variable Input

Channel supports all basic C/C++ data types.

- Boolean (`bool`)
- Char (`char`)
- Integer (`int`) and its various forms.
- Float (`float`)
- Double (`double`)

Boolean

Output for boolean is pretty basic and boring.

```
bool foo = true;
ioc << foo << IOCtrl::endl;
//OUTPUT: "TRUE"
```

The output style can be adjusted, however, using the `IOFormalBoolStyle::` flags.

```
bool foo = true;
ioc << IOFormalBoolStyle::lower << foo << IOCtrl::endl;
//OUTPUT: "true"
ioc << IOFormalBoolStyle::upper << foo << IOCtrl::endl;
//OUTPUT: "True"
ioc << IOFormalBoolStyle::caps << foo << IOCtrl::endl;
//OUTPUT: "TRUE"
ioc << IOFormalBoolStyle::numeral << foo << IOCtrl::endl;
//OUTPUT: "1"
```

Char

Since char can represent both an integer and a character, Channel lets you display it as either. By default, Channel displays the char as a literal character. Using the `IOFormatCharValue::as_int` flag forces it to print as an integer.

```
char foo = 'A';
ioc << "Character " << foo << " has ASCII value "
    << IOFormatCharValue::as_int << foo << IOCtrl::endl;
//OUTPUT: Character A has ASCII value 65
```

When output as an integer, char can be used with all of the enumerations for int (see that section).

Integer

An `int` can be represented in any base (radix) from binary (base 2) to base 35 using the `IOFormatBase::` flags.

```
int foo = 12345;
ioc << "Binary: " << IOFormatBase::bin << foo << IOCtrl::endl;
ioc << "Octal: " << IOFormatBase::oct << foo << IOCtrl::endl;
ioc << "Decimal: " << IOFormatBase::dec << foo << IOCtrl::endl;
ioc << "Dozenal: " << IOFormatBase::doz << foo << IOCtrl::endl;
ioc << "Hexadecimal: " << IOFormatBase::hex << foo << IOCtrl::endl;
ioc << "Base 31: " << IOFormatBase::b31 << foo << IOCtrl::endl;

/*OUTPUT:
Binary: 11000000111001
Octal: 30071
Decimal: 12345
Dozenal: 7189
Hexadecimal: 3039
Base 31: cq7
*/
```

In bases larger than decimal (10), the letter numerals can be output as lowercase or uppercase (default) using the `IOFormatNumCase::` flags.

```
int foo = 187254;
ioc << "Hexadecimal Lower: " << IOFormatBase::hex << foo << IOCtrl::endl;
ioc << "Hexadecimal Upper: " << IOFormatNumCase::upper
    << IOFormatBase::hex << foo << IOCtrl::endl;

/*OUTPUT:
Hexadecimal Lower: 2db76
Hexadecimal Upper: 2DB76
*/
```

Float and Double

Float and Double can only be output in base 10 directly. (Hexadecimal output is only possible through a pointer memory dump. See that section.) However, the decimal places (the number of digits after the decimal point) and use of scientific notation can be modified. By default, decimal places is 14, and use of scientific notation is automatic for very large and small numbers.

Decimal places can be modified using the `IOFormatDecimalPlaces(#)` flag. Scientific notation can be turned on with `IOFormatSciNotation::on`, and off using `IOFormatSciNotation::none`. It can also be reset to automatic with `IOFormatSciNotation::automatic`.

```
float foo = 12345.12345678912345;
ioc << "Decimal places 5, no sci: " << IOFormatDecimalPlaces(5) << foo << IOCtrl::endl;
ioc << "Decimal places 10, sci: " << IOFormatDecimalPlaces(10)
    << IOFormatSciNotation::on << foo << IOCtrl::endl;

/*OUTPUT:
Decimal places 5, no sci: 12345.12304
```

(continues on next page)

(continued from previous page)

```
Decimal places 10, sci: 1.2345123046e+4
*/
```

Both types work the same.

Pointer Output

One of the most powerful features of Channel is its handling of pointers. In addition to printing the value at known pointer types, it can print the address or raw memory for ANY pointer, even for custom objects.

Pointer Value

By default, Channel will attempt to print the value at the pointers. This can also be forced using `IOFormatPtr::value`.

```
int foo = 12345;
int* foo_ptr = &foo;
ioc << "Value of foo: " << IOFormatPtr::value << foo_ptr << IOCtrl::endl;

char* bar = "My name is Bob, and I am a coder.\0";
ioc << "Value of bar: " << bar << IOCtrl::endl;

/*OUTPUT:
Value of foo: 12345
Value of bar: My name is Bob, and I am a coder.
*/
```

Pointer Address

Channel can print out the address of the pointer in hexadecimal using `IOFormatPtr::address`. It displays with lowercase letter numerals by default, though these can be displayed in uppercase using `IOFormatNumCase::upper`. It is capable of doing this with any pointer, even for custom objects.

```
int foo = 12345;
int* foo_ptr = &foo;
ioc << "Address of foo: " << IOFormatPtr::address << foo_ptr << IOCtrl::endl;

char* bar = "My name is Bob, and I am a coder.\0";
ioc << "Address of bar: " << IOFormatPtr::address << IOFormatNumCase::upper
    << bar << IOCtrl::endl;

/*OUTPUT:
Address of foo: 0x7ffc33518308
Address of bar: 0x405AF0
*/
```

Pointer Memory Dump

Channel is capable of dumping the raw memory at any pointer using `IOFormatPtr::memory`. The function is safe for pointers to most objects and atomic types, as the memory dump will automatically determine the size and will never overrun the size of the variable. With char pointers (cstring), the only danger is when the cstring is not null terminated.

Spacing can be added between bytes (`IOFormatMemSep::byte`) and bytewords (`IOFormatMemSep::word`), or both (`IOFormatMemSep::all`). By default, the memory dumps with no spacing (`IOFormatMemSep::none`).

```
int foo = 12345;
int* foo_ptr = &foo;
ioc << "Memory dump of foo: " << IOFormatPtr::memory << IOFormatMemSep::byte
    << foo_ptr << IOCtrl::endl;

char* bar = "My name is Bob, and I am a coder.\0";
ioc << "Memory dump of bar: " << IOFormatPtr::memory << IOFormatMemSep::all
    << bar << IOCtrl::endl;

/*OUTPUT:
Memory dump of foo: 39 30 00 00
Memory dump of bar: 4d 79 20 6e 61 6d 65 20 | 69 73 20 42 6f 62 2c 20 | 61 6e 64 20 49␣
↪20 61 6d | 20 61 20 63 6f 64 65 72 | 2e 00
*/
```

The following dumps the raw memory for a custom object.

```
//Let's define a struct as our custom object, and make an instance of it.
struct CustomStruct
{
    int foo = 12345;
    double bar = 123.987654321;
    char faz[15] = "Hello, world!\0";
    void increment(){foo++;bar++;}
};
CustomStruct blah;

ioc << IOFormatPtr::memory << IOFormatMemSep::all << &blah << IOCtrl::endl;
/*OUTPUT:
39 30 00 00 00 00 00 00 | ad 1c 78 ba 35 ff 5e 40 | 48 65 6c 6c 6f 2c 20 77 | 6f 72 6c␣
↪64 21 00 00 00
*/
```

You can also read memory from a void pointer, though you must specify the number of bytes to read using `IOMemReadSize()`.

> **Warning:** This feature must be used with caution, as reading too many bytes can trigger segfaults or any number of memory errors. Use the sizeof operator in the read_bytes() argument to prevent these types of problems. (See code).

Bitset

Channel is able to intelligently output the contents of any bitset. It temporarily forces use of the `IOFormatPtr::memory` flag to ensure proper output.

One may use any of the `IOFormatMemSep::` flags to control the style of output. By default, `IOFormatMemSep::none` is used.

```
bitset<32> foo = bitset<32>(12345678);
ioc << IOFormatMemSep::all << foo << IOCtrl::endl;
/* OUTPUT:
4e 61 bc 00
*/
```

Formatting Objects

If you find yourself regularly using particular formatting flags (`IOFormat...::`), you can store them in an IOFormat object for reuse. Flags are passed into the `IOFormat` object with the stream insertion operator (<<), and then the `IOFormat` object itself can be passed to the Channel.

```
IOFormat fmt;
fmt << IOFormatTextAttr::bold << IOFormatTextFG::red << IOFormatTextBG::black;

ioc << fmt << "This is bold, red text on a black background." << IOCtrl::endl;

ioc << fmt << IOFormatBG::blue << "This is bold, red text on a blue background."
    << IOCtrl::endl;
```

As you can see, anything passed to the Channel *after* the `IOFormat` object overrides prior options.

IOFormat supports all the flags beginning with `IOFormat...`.

Stream Control

There are multiple enums for controlling Channel's output.

For example, one might want to display progress on the same line, and then move to a new line for a final message. This can be accomplished via…

```
ioc << "Let's Watch Progress!" << IOCtrl::endl;
ioc << fg_blue << ta_bold;
for(int i=0; i<100; i++)
{
    //Some long drawn out code here.
    ioc << i << "%" << IOCtrl::sendc;
}
ioc << io_endl;
ioc << "Wasn't that fun?" << io_endl;

/* FINAL OUTPUT:
Let's Watch Progress!
100%
Wasn't that fun?
*/
```

The complete list of stream controls is as follows. Some notes…

- EoM indicates "End of Message", meaning Channel will broadcast the message at this point.
- n is a newline.
- r is simply a carriage return (move to start of current line).
- Clear means all formatting flags are reset to their defaults.
- Flush forces stdout to refresh. This is generally necessary when overwriting a line or moving to a new line after overwriting a previous one.

Command	EoM	Clear	r	n	Flush
`IOCtrl::clear`		X			
`IOCtrl::flush`					X
`IOCtrl::end`	X	X			
`IOCtrl::endc`	X	X	X		X
`IOCtrl::endl`	X	X		X	X
`IOCtrl::send`	X				
`IOCtrl::sendc`	X		X		X
`IOCtrl::sendl`	X			X	X
`IOCtrl::r`			X		
`IOCtrl::n`				X	

Cursor Movement

Channel can move the cursor back and forth on ANSI-enabled terminals using the *IOCursor::left* and *IOCursor::right* flags.

```
std::string buffer;
ioc << "Hello, world!"
            << IOCursor::left
            << IOCursor::left
            << IOCtrl::end;
std::getline(std::cin, buffer);

/* Will now wait for user input, while displaying "Hello, world!"
 * with the cursor highlighting the 'd' character.
 */
```

Important: Currently, only ANSI is used. Windows support, formatting-removed, and an easy-to-parse formatting flag system for custom outputs will be added soon.

Internal Broadcast Settings (Echo)

Channel can internally output to either `printf()` or `std::cout` (or neither). By default, it uses printf(). However, as stated, this can be changed.

Channel's internal output also broadcasts all messages by default. This can also be changed.

These settings are modified by passing a `IOEchoMode::` flag to the `configure_echo()` member function.

```
//Set to use `std::cout`
ioc.configure_echo(IOEchoMode::cout);

//Set to use `printf` and show only error messages (any verbosity)
ioc.configure_echo(IOEchoMode::printf, IOVrb::tmi, IOCat::error);

//Set to use `cout` and show only "quiet" verbosity messages.
ioc.configure_echo(IOEchoMode::cout, IOVrb::quiet);

//Turn off internal output.
ioc.configure_echo(IOEchoMode::none);
```

External Broadcast with Signals

One of the primary features of Channel is that it can be connected to multiple outputs using signals. Examples of this might be if you want to output to a log file, or display messages in a console in your interface.

Main Signal (`signal_all`)

The main signal is `signal_all`, which requires a callback function of the form `void callback(std::string, IOVrb, IOCat)`, as seen in the following example.

```
//This is our callback function.
void print(std::string msg, IOVrb vrb, IOCat cat)
{
    //Handle the message however we want.
    std::cout << msg;
}

//We connect the callback function to `signal_all` so we get all messages.
ioc.signal_all.add(&print);
```

Category Signals (`signal_c_...`)

Almost all categories have a signal: `signal_c_normal`, `signal_c_warning`, `signal_c_error`, `signal_c_testing`, and `signal_c_debug`.

Note: `IOCat::all` is used internally, and does not have a signal. Use `signal_all` instead.

The callbacks for category signals require the form `void callback(std::string, IOVrb)`. Below is an example.

```
//This is our callback function.
void print_error(std::string msg, IOVrb vrb)
{

//Handle the message however we want.
std::cout << msg;

}

//We connect the callback function to signal_c_error to get only error messages.
ioc.signal_c_error.add(&print_error);
```

Verbosity Signals (`signal_v_...`)

Each verbosity has a signal: `signal_v_quiet`, `signal_v_normal`, `signal_v_chatty`, and `signal_v_tmi`. A signal is broadcast when any message of that verbosity or lower is transmitted.

The callbacks for verbosity signals require the form `void callback(std::string, IOCat)`. Below is an example inside the context of a class.

```
class TestClass
{
    public:
        TestClass(){}
        void output(std::string msg, IOCat cat)
        {
            //Handle the message however we want.
            std::cout << msg;
        }
        ~TestClass(){}
};

TestClass testObject;
ioc.signal_v_normal.add(&testObject, TestClass::output)
```

1.1.5 Flag Lists

Category (`IOCat::`)

Flag	Use
`IOCat::none`	No category; **NEVER broadcasted**. Does not have a correlating signal.
`IOCat::normal`	The default value - anything that doesn't fit elsewhere.
`IOCat::warning`	Warnings, but not necessarily errors.
`IOCat::error`	Error messages.
`IOCat::debug`	Debug messages, such as variable outputs.
`IOCat::testing`	Messages in tests. (Goldilocks automatically suppresses these during benchmarking.)
`IOCat::all`	All message categories. Does not have a correlating signal.

Cursor Control (`IOCursor::`)

Flag	Use
`IOCursor::left`	Moves the cursor left one position.
`IOCursor::right`	Moves the cursor right one position.

Echo Mode (`IOEchoMode::`)

Note: These cannot be passed directly to Channel.

Flag	Use
`IOEchoMode::none`	No internal output.
`IOEchoMode::printf`	Internal output uses `printf()`.
`IOEchoMode::cout`	Internal output uses `std::cout`.

Base/Radix Format (`IOFormatBase::`)

Flag	Base
`IOFormatBase::bin`	2
`IOFormatBase::b2`	2
`IOFormatBase::ter`	3
`IOFormatBase::b3`	3
`IOFormatBase::quat`	4
`IOFormatBase::b4`	4
`IOFormatBase::quin`	5
`IOFormatBase::b5`	5
`IOFormatBase::sen`	6
`IOFormatBase::b6`	6
`IOFormatBase::sep`	7
`IOFormatBase::b7`	7
`IOFormatBase::oct`	8
`IOFormatBase::b8`	8
`IOFormatBase::b9`	9
`IOFormatBase::dec`	10
`IOFormatBase::b10`	10
`IOFormatBase::und`	11
`IOFormatBase::b11`	11
`IOFormatBase::duo`	12
`IOFormatBase::doz`	12
`IOFormatBase::b12`	12
`IOFormatBase::tri`	13
`IOFormatBase::b13`	13
`IOFormatBase::tetra`	14
`IOFormatBase::b14`	14
`IOFormatBase::pent`	15

continues on next page

Table 1 – continued from previous page

Flag	Base
`IOFormatBase::b15`	15
`IOFormatBase::hex`	16
`IOFormatBase::b16`	16
`IOFormatBase::b17`	17
`IOFormatBase::b18`	18
`IOFormatBase::b19`	19
`IOFormatBase::vig`	20
`IOFormatBase::b20`	20
`IOFormatBase::b21`	21
`IOFormatBase::b22`	22
`IOFormatBase::b23`	23
`IOFormatBase::b24`	24
`IOFormatBase::b25`	25
`IOFormatBase::b26`	26
`IOFormatBase::b27`	27
`IOFormatBase::b28`	28
`IOFormatBase::b29`	29
`IOFormatBase::b30`	30
`IOFormatBase::b31`	31
`IOFormatBase::b32`	32
`IOFormatBase::b33`	33
`IOFormatBase::b34`	34
`IOFormatBase::b35`	35
`IOFormatBase::b36`	36

Boolean Format (`IOFormalBoolStyle::`)

Char Value (`IOFormatCharValue::`)

Enum	Action
`IOFormatCharValue::as_char`	Output chars as ASCII characters.
`IOFormatCharValue::as_int`	Output chars as integers.

Memory Separators (`IOFormatMemSep::`)

Enum	Action
`IOFormatMemSep::no`	Output memory dump as one long string.
`IOFormatMemSep::byte`	Output memory dump with spaces between bytes.
`IOFormatMemSep::word`	Output memory dump with bars between words (8 bytes).
`IOFormatMemSep::all`	Output memory dump with spaces between bytes and bars between words.

Numeral Case (`IOFormatNumCase::`)

Enum	Action
`IOFormatNumCase::lower`	Print all letter digits as lowercase.
`IOFormatNumCase::upper`	Print all letter digits as uppercase.

Pointer Format (`IOFormatPtr::`)

Enum	Action
`IOFormatPtr::value`	Print the value at the address.
`IOFormatPtr::address`	Print the actual memory address.
`IOFormatPtr::memory`	Dump the hexadecimal representation of the memory at the address.

Scientific Notation Format (`IOFormatSciNotation::`)

Enum	Action
`IOFormatSciNotation::none`	No scientific notation.
`IOFormatSciNotation::auto`	Automatically select the best option.
`IOFormatSciNotation::on`	Force use of scientific notation.

> **Warning:** `IOFormatSciNotation::none` has been known to cause truncation in very large and very small values, regardless of decimal places.

Decimal places(`IOFormatDecimalPlaces()`)

`IOFormatDecimalPlaces(n)` where `n` is the decimal places, as an integer representing the number of decimal places.

Text Attributes(`IOFormatTextAttr::`)

Enum	Action
`IOFormatTextAttr::none`	Turn off all attributes.
`IOFormatTextAttr::bold`	**Bold text**.
`IOFormatTextAttr::underline`	Underlined text.
`IOFormatTextAttr::invert`	Invert foreground and background colors.

Text Background Color(`IOFormatTextBG::`)

Enum	Action
`IOFormatTextBG::none`	Default text background.
`IOFormatTextBG::black`	Black text background.
`IOFormatTextBG::red`	Red text background.
`IOFormatTextBG::green`	Green text background.
`IOFormatTextBG::yellow`	Yellow text background.
`IOFormatTextBG::blue`	Blue text background.
`IOFormatTextBG::magenta`	Meganta text background.
`IOFormatTextBG::cyan`	Cyan text background.
`IOFormatTextBG::white`	White text background.

Text Foreground Color(`IOFormatTextFG::`)

Enum	Action
`IOFormatTextFG::none`	Default text foreground.
`IOFormatTextFG::black`	Black text foreground.
`IOFormatTextFG::red`	Red text foreground.
`IOFormatTextFG::green`	Green text foreground.
`IOFormatTextFG::yellow`	Yellow text foreground.
`IOFormatTextFG::blue`	Blue text foreground.
`IOFormatTextFG::magenta`	Meganta text foreground.
`IOFormatTextFG::cyan`	Cyan text foreground.
`IOFormatTextFG::white`	White text foreground.

Memory Dump Read Size (`IOMemReadSize()`)

`IOMemReadSize(n)` where `n` is the number of bytes to read and print, starting at the memory address. **Only used with void pointers.**

Warning: Misuse triggers undefined behavior, including SEGFAULT. Use with caution.

Verbosity (`IOVrb::`)

Enum	Use
`IOVrb::quiet`	Only essential messages and errors. For normal end-use. Shipping default.
`IOVrb::normal`	Common messages and errors. For common and normal end-user testing.
`IOVrb::chatty`	Most messages and errors. For detailed testing and debugging.
`IOVrb::tmi`	Absolutely everything. For intense testing, detailed debugging, and driving the developers crazy.

1.2 IOFormat

1.3 Stringy: String Utilities

These functions allow working with (and between) C-string, std::string, and other types. They're used by the rest of IOSqueak, but may be useful to others as well.

1.3.1 Including Stringy

To use the Stringy functions in your code, use the following:

```
#include "iosqueak/stringy.hpp"
```

1.3.2 Integer to std::string [itos()]

We can convert any integer data type, signed or unsigned, to a std::string using `itos()`.

`itos()` converts the integer to a std::string. It accepts three arguments, two of which are required:

- the integer to convert,
- the base you're working in, represented as an integer (default=10),
- whether to represent digits greater than 9 as uppercase (default=false)

```
// The integer to convert.
int foo = -16753;

/* Convert the float to a std::string. We're passing all the arguments,
 * even though only the first two are required, for the sake of example.
 */
std::string foo_s = stringy::itos(foo, 10, false);

// Print out the std::string.
ioc << foo_s << IOCtrl::endl;

// OUTPUT: -16753
```

Important: Enumerations are not implicitly cast to ints with this function. Therefore, you must `static_cast<int>()` any enumeration variables before passing them to this function.

1.3.3 Integer to C-String [`itoa()` & `intlen()`]

We can convert any integer data type, signed or unsigned, to a C-string using `itoa()` and `intlen()`.

`intlen()` returns the character count necessary to represent the integer as a string. It accepts three arguments, two of which are required:

- the integer being measured,
- the base you're working in, represented as an integer, and
- whether to include space for the sign (default=true).

`itoa()` converts the integer to a C-string. It accepts five arguments, two of which are required:

- the C-string to write to,
- the integer to convert,
- the base you're working in, represented as an integer (default=10),
- the number of characters in the integer (default=0, meaning it will be internally calculated), and
- whether to represent digits greater than 9 as uppercase (default=false)

Combining these functions allows us to flexibly convert any integer to a C-string, without having to know anything in advance.

```
// The integer to convert.
int foo = -16753;

/* We use the intlen function to determine the size of our C-string
 * Note that we are adding one to leave room for our null terminator. */
char foo_a[stringy::intlen(foo, 10, true) + 1];

/* Convert the integer to a C-string. We're passing all the arguments,
 * even though only the first two are required, for the sake of example.
 * 0 for the fourth argument (size) causes the function to internally
 * calculate the size of the integer again, which is another call to
 * intlen(). You might save some execution time by specifying this instead.
 */
stringy::itoa(foo_a, foo, 10, 0, false);

// Print out the C-string.
ioc << foo_a << IOCtrl::endl;

// OUTPUT: -16753
```

Note: It is generally going to be more practical to use `itos()` instead.

Important: Enumerations are not implicitly cast to ints with this function. Therefore, you must `static_cast<int>()` any enumeration variables before passing them to this function.

1.3.4 Float to String [ftos()]

We can convert any floating-point number data type (float, double, or long double) to a std::string using *ftos()*.

We need to specify the number of decimal places - in our case, the number of digits after the decimal point - to work with. Because of the nature of floating point numbers, the conversion is *not* perfect, as we'll see shortly.

`ftos()` converts the number into a C-string. It accepts three arguments, one of which are required:

- the number to convert,
- the number of decimal places (default=14), and
- whether to use scientific notation - 0=none, 1=automatic, and 2=force scientific notation (default=1).

```
// The integer to convert.
float foo = -65.78325;

/* Convert the float to a std::string. */
std::string foo_s = stringy::ftos(foo, 5, 1);

// Print out the std::string.
ioc << foo_s << IOCtrl::endl;

// OUTPUT: -65.78324
```

As you can see, the output is off by 0.00001. Again, this is because of how floating point numbers work, and the number of decimal places we specified. If we were to raise the decimal places to the default 14, our output would actually have been "-65.78324891505623".

1.3.5 Float to C-String [ftoa() & floatlen()]

We can convert any floating-point data type (float, double, or long double) to a C-string using `ftoa()` and `floatlen()`.

In both functions, we need to specify the number of decimal places - in our case, the number of digits after the decimal point - to work with. Because of the nature of floating point numbers, the conversion is *not* perfect, as we'll see shortly.

`floatlen()` returns the character count necessary to represent the floating-point number as a string. It accepts three arguments, only one of which is required:

- the number to count the characters in,
- the number of decimal places (default=14), and
- whether to count the symbols (default=true)

`ftoa()` converts the number into a C-string. It accepts four arguments, two of which are required:

- the C-string to write to,
- the number to convert,
- the number of decimal places (default=14), and
- whether to use scientific notation - 0=none, 1=automatic, and 2=force scientific notation (default=1).

```
// The integer to convert.
float foo = -65.78325;

/* Convert the float to a std::string. */
```

(continues on next page)

(continued from previous page)

```
std::string foo_s = stringy::ftos(foo, 5, 1);

// Print out the std::string.
ioc << foo_s << IOCtrl::endl;

// OUTPUT: -65.78324
```

As you can see, the output is off by 0.00001. Again, this is because of how floating point numbers work, and the number of decimal places we specified. If we were to raise the decimal places to the default 14, our output would actually have been "-65.78324891505623".

Note: It is generally going to be more practical to use `ftos()` instead.

1.3.6 Split String By Tokens [`split_string`]

This will split a `std::string` by a given token and store it in a `std::vector`. The token will be stripped out in the process.

Later versions of this will support Onestring and FlexArray.

```
std::string splitMe = "What if we:Want to split:A string:By colons?";
std::vector<std::string> result;

stringy::stdsplit(splitMe, ":", result);
// result now contains "What if we", "Want to split", "A string", "By colons?"
```

1.3.7 Reverse C-String [`reverse_c_string()`]

This will reverse a given C-string in place, overriding the string.

```
char foo[14] = "Hello, world!";
stringy::reverse_c_string(foo);
ioc << foo << IOCtrl::endl;
```

CHAPTER

TWO

INDICES AND TABLES

Note: The index is still a work in progress. If you'd like to help with this, please see our Contribution Guide.

- genindex
- search

INDEX