scnlib

Release 0.4.0-dev

Elias Kosunen

Nov 13, 2020

CONTENTS:

CHAPTER

ONE

GUIDE

1.1 Installation and building

NOTE: the following instructions assume a Unix-like system with a command line. If your development environment is different (e.g. Visual Studio), these steps cannot be followed verbatim. Some IDEs, like Visual Studio, have CMake integration built into them. In some other environments, you may have to use the CMake GUI or your system's command line. To build a CMake project without `make`, use `cmake --build ..`

`scnlib` uses CMake for building. If you're already using CMake for your project, integration is easy with `find_package`.

```
$ mkdir build
$ cd build
$ cmake ..
$ make -j
$ make install
```

```
# Find scnlib package
find_package(scn CONFIG REQUIRED)

# Target which you'd like to use scnlib
# scn::scn-header-only to use the header-only version
add_executable(my_program ...)
target_link_libraries(my_program scn::scn)
```

1.1.1 Tests and benchmarks

To build and run the tests and benchmarks for scnlib, clone the repository, and build it with CMake.

Building and running tests:

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
$ make -j
$ ctest -j4
```

Building and running the runtime performance benchmarks:

```
# create build directory like above
$ cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INTERPROCEDURAL_OPTIMIZATION=ON -DSCN_
↪NATIVE_ARCH=ON ..
```

(continues on next page)

(continued from previous page)

```
$ make -j

# disable CPU frequency scaling
$ sudo cpupower frequency-set --governor performance

# run the benchmark of your choosing
$ ./benchmark/runtime/integer/bench-int

# re-enable CPU frequency scaling
$ sudo cpupower frequency-set --governor powersave
```

1.1.2 Without CMake

Headers for the library can be found from the `include/` directory, and source files from the `src/` directory.

Building and linking the library:

```
$ mkdir build
$ cd build
$ c++ -c -I../include/ ../src/*.cpp
$ ar rcs libscn.a *.o
```

`libscn.a` can then be linked, as usual.

```
# in your project
$ c++ ... -Lpath-to-scn/build -lscn
```

When building as header-only, `src/` has to be in the include path, and `SCN_HEADER_ONLY` must be defined to `1`: In this case, a separate build stage obviously isn't required

```
# in your project
$ c++ ... -Ipath-to-scn/include -Ipath-to-scn/src -DSCN_HEADER_ONLY=1
```

1.2 Basic usage

`scn::scan` can be used to parse various values from a source range.

A range is an object that has a beginning and an end. Examples of ranges are string literals, `std::string` and `std::vector<char>`. Objects of these types, and more, can be passed to `scn::scan`. To learn more about the requirements on these ranges, see the API documentation on source ranges.

After the source range, `scn::scan` is passed a format string. This is similar in nature to `scanf`, and has virtually the same syntax as `std::format` and {fmt}. In the format string, arguments are marked with curly braces `{}`. Each `{}` means that a single value is to be scanned from the source range. Because `scnlib` uses variadic templates, type information is not required in the format string, like it is with `scanf` (like `%d`).

After the format string, references to arguments to be parsed are given.

```
// Scanning an int
int i;
scn::scan("123", "{}", i):
// i == 123
```

(continues on next page)

(continued from previous page)

```
// Scanning a double
double d;
scn::scan("3.14", "{}", d);
// d == 3.14

// Scanning multiple values
int a, b;
scn::scan("0 1 2", "{} {}", a, b);
// a == 0
// b == 1
// Note, that " 2" was not scanned,
// because only two integers were requested

// Scanning a string means scanning a "word" --
//   that is, until the next whitespace character
// this is the same behavior as with iostreams
std::string str;
scn::scan("hello world", "{}", str);
// str == "hello"
```

Compare the above example to the same implemented with `std::istringstream`:

```
int i;
std::istringstream{"123"} >> i;

double d;
std::istringstream{"3.14"} >> d;

int a, b;
std::istringstream{"0 1 2"} >> a >> b;

std::string str;
std::istringstream{"hello world"} >> str;
```

Or with `sscanf`:

```
int i;
std::sscanf("123", "%d", &i);

double d;
std::sscanf("3.14", "%lf", &d);

int a, b;
std::sscanf("0 1 2", "%d %d", &a, &b);

// Not really possible with scanf!
// Using a fixed size buffer
char buf[6] = {0};
std::sscanf("hello world", "%5s", buf);
// buf == "hello"
```

1.3 Error handling and return values

scnlib does not use exceptions. The library compiles with `-fno-exceptions -fno-rtti` and is perfectly usable without them.

Instead, it uses return values to signal errors. This return value is truthy if the operation succeeded. Using the `.error()` member function more information about the error can be gathered.

If an error occurs, the arguments that were not scanned are not written to. Beware of using possibly uninitialized variables.

```
int i;
// "foo" is not an integer
auto result = scn::scan("foo", "{}", i);
// fails, i still uninitialized
if (!result) {
    std::cout << result.error().msg() << '\n';
}
```

Unlike with `scanf`, partial successes are not supported. Either the entire scanning operation succeeds, or a failure is returned.

```
int a, b;
// "foo" is still not an integer
auto result = scn::scan("123 foo", "{} {}", a, b);
// fails -- result == false
// a is written to, a == 123
// b is still uninitialized
```

Oftentimes, the entire source range is not scanned, and the remainder of the range may be useful later. The leftover range can be accessed with the member function `.range()`.

```
int i;
auto result = scn::scan("123 456", "{}", i);
// result == true
// i == 123
// result.range() == " 456"

result = scn::scan(result.range(), "{}", i);
// result == true
// i == 456
// result.range() == ""
```

The return type of `.range()` is a valid range, but it is of an library-internal, user-unnameable type. Its type is not the same as the source range. If possible for the given source range type, the `.reconstruct()` member function can be used to create a range of the original source range type. Note, that `.reconstruct()` is not usable with string literals.

```
std::string source{"foo bar"};
std::string str;
auto result = scn::scan(source, "{}", str);
// result == true
// str == "foo"
// result.reconstruct() == " bar"
```

The result type has some additional useful member functions. These include:

- `.empty()`: returns `true` if the leftover range is empty, meaning that there are definitely no values to scan from the source range any more.
- `.string()`, `.string_view()`, and `.span()`: like `.reconstruct()`, except they work for every contiguous range, and return a value of the type specified in the function name

See the API documentation for more details.

To enable multiple useful patterns, the library provides a function `scn::make_result`. This function will return the result object for a given source range, that can be later reassigned to. For example:

```
auto result = scn::make_result("foo");
int i;
if (result = scn::scan(result.range(), "{}", i)) {
    // success
    // i is usable
} else {
    // failure
    // result contains more info, i not usable
}
```

Or:

```
auto result = scn::make_result("123 456");
int i;
while (result = scn::scan(result.range(), "{}", i)) {
    // success
    // i is usable:
    // iteration #1: i == 123
    // iteration #2: i == 456
}
// failure
// can either be an invalid value or EOF
// in this case, it's EOF
// i not modified, still i == 456
```

1.4 Files and `stdin`

To easily read from `stdin`, use `scn::input` or `scn::prompt`. They work similarly `scn::scan`.

```
// Reads an integer from stdin
int i;
auto result = scn::input("{}", i);

// Same, but with an accompanying message sent to stdout
int i;
auto result = scn::prompt("Write an integer: ", "{}", i);
```

To use `scn::scan` with `stdin`, use `scn::cstdin()`. It returns a `scn::file&`, which is a range mapping to a `FILE*`.

```
int i;
auto result = scn::scan(scn::cstdin(), "{}", i);
```

`scn::input` and `scn::prompt` sync with `<cstdio>` automatically, so if you wish to mix-and-match `scn::input` and `scanf`, it's possible without any further action. `scn::scan` and `scn::cstdin()` don't do this, but you must explicitly call `scn::cstdin().sync()` when synchronization is needed.

```
int i, j;
scn::input("{}", i);
std::scanf("%d", &j);

int i, j;
scn::scan(scn::cstdin(), "{}", i);
scn::cstdin().sync(); // needed here, because we wish to use <cstdio>
std::scanf("%d", &j);
```

You can also scan from other file handles than `stdin`. You can either use `scn::file` or `scn::owning_file`, depending on if you want to handle the lifetime of the `FILE*` yourself, or let the library handle it, respectively.

```
auto f = std::fopen("file.txt", "r");
scn::file file{f};
f.close();
// file now unusable

scn::owning_file file{"file.txt", "r"};
```

Both `scn::file` and `scn::owning_file` are valid source ranges, and can be passed to `scn::scan` and other scanning functions. `scn::owning_file` is a child class of `scn::file`, so `scn::owning_file& -> scn::file&` is a valid conversion.

There's also `scn::mapped_file` for easier management of memory mapped files, see the API documentation for more.

1.5 Other scanning functions

1.5.1 `scn::scan_default`

Oftentimes, specific parsing configuration through the format string is not required. In this case, `scn::scan_default` can be used. Using it has some performance benefits, as a format string doesn't need to be parsed.

Using `scn::scan_default` with N args has the same semantics as using `scn::scan` with a format string with N space-separated `"{}"` s.

```
int a, b;
auto result = scn::scan_default("123 456", a, b);
// result == true
// a == 123
// b == 456

// Equivalent to:
int a, b;
auto result = scn::scan("123 456", "{} {}", a, b);
```

1.5.2 scn::scan_value

If you only wish to scan a single value with default options, you can avoid using output arguments by using `scn::scan_value`. The return value of `scn::scan_value<T>` contains a `.value()` member function that returns a `T` if the operation succeeded.

```
auto result = scn::scan_value<int>("123");
// result == true
// result.value() == 123
```

As is evident by the presence of an extra member function, the return type of `scan_value` is not the same as the one of `scan`. The return type of `scan` inherits from `scn::wrapped_error`, but the return type of `scan_value` inherits from `scn::expected<T>`. To use `make_result` with `make_value`, this needs to be taken into account:

```
auto result = scn::make_result<scn::expected<int>>(...);
result = scn::scan_value<int>(result.range());
```

The return types of `scan` and `scan_value` are not compatible, and cannot be assigned to each other.

1.5.3 Localization: scn::scan_localized

By default, `scnlib` isn't affected by changes to the global C or C++ locale. All functions will behave as if the global locale was set to `"C"`.

A `std::locale` can be passed to `scn::scan_localized` to scan with a locale. This is mostly used with numbers, especially floats, giving locale-specific decimal separators.

Because of the way `std::locale` is, parsing with a locale is significantly slower than without one. This is because the library effectively has to use iostreams for parsing.

```
// Reads a localized float
double d;
auto result = scn::scan_localized(std::locale{"fi_FI.UTF-8"}, "2,73", "{}", d);
// result == true
// d == 2.73
```

Because `scan_localized` uses iostreams under the hood, the results will not be identical to `scn::scan`, even if `std::locale::classic()` was passed.

1.5.4 scn::getline

`scn::getline` works similarly to `std::getline`. It takes a range to read from, a string to read into, and optionally a delimeter character defaulting to `'\n'`.

```
std::string line;
auto result = scn::getline("first\nsecond\nthird", line);
// result == true
// line == "first"
// result.range() == "second\nthird" (note that the delim '\n' is skipped)

// setting '\n' explicitly
result = scn::getline(result.range(), line, '\n');
// result == true
// line == "second"
// result.range() == "third"
```

(continues on next page)

(continued from previous page)

```
// delim doesn't have to be '\n' or even whitespace
result = scn::getline(result.range(), line, 'r');
// result == true
// line == "thi"
// result.range() == "d"
```

If the string to read into passed to `scn::getline` is a `scn::string_view`, and the source range is contiguous, the `string_view` is modified to point into the source range. This increases performance (no copying or memory allocations) at the expense of lifetime safety.

```
std::string source = "foo\nbar";
scn::string_view line;
auto result = scn::getline(source, line);
// result == true
// line == "foo"
// result.range() == "bar"
// line.data() == source.data() (point to the same address -- `line` points to␣
↪`source`)
```

1.5.5 `scn::ignore_until` and `scn::ignore_until_n`

`scn::ignore_until_n` is like `std::istream::ignore`. It takes an integer N and a character C, and reads the source range until either N characters were read or character C was found from the source range.

`scn::ignore_until` works in the same way, except the only condition for stopping to read is finding the end character. This is effectively equivalent to passing `std::numeric_limits<std::ptrdiff_t>::max()` as N to `scn::ignore_until_n`.

1.5.6 `scn::scan_list` and temporaries

To easily scan multiple values of the same type, `scn::scan_list` can be used. It takes a source range and a container to write the scanned values to. Its return type is similar to that of `scn::scan`.

```
std::vector<int> list;
auto result = scn::scan_list("123 456 789", list);
// result == true
// list == [123, 456, 789]
```

`scn::scan_list` can also be passed a third argument marking a delimeter character:

```
std::vector<int> list;
auto result = scn::scan_list("123, 456, 789", list, ',');
// result == true
// list == [123, 456, 789]
```

`scn::scan_list` will read until an invalid value or delimeter is found or the source range is exhausted. `scn::scan_list_until` can be used to control this behavior. As its third argument, it takes a character, until which it will read the source range, similar to `getline`. The delimeter character argument is still the last argument and optional.

```
std::vector<int> list;
auto result = scn::scan_list_until("123 456 789\n123", list, '\n');
```

(continues on next page)

(continued from previous page)

```
// result == true
// list == [123, 456, 789]
// result.range() == "123"
```

If you've already allocated memory for the list, `scan_list` and `scan_list_until` can be passed a `scn::span`. Because the container must be passed to `scan_list` as an lvalue reference, the `span` must be constructed separately, which can be tedious. The library provides some helpers for this.

```
// Doing everything explicitly
std::vector<int> list(64, 0);
auto span = scn::make_span(list);
auto result = scn::scan_list("123 456 789", span);
// result == true
// list == span == [123, 456, 789]

// Using scn::temp
// Takes an rvalue and makes it usable as an argument to scanning functions requiring␣
↪an lvalue reference
// Useful with spans and other views
std::vector<int> list(64, 0);
auto result = scn::scan_list("123 456 789", scn::temp(scn::make_span(list)));
// result == true
// list == span == [123, 456, 789]

// Using scn::make_span_list_wrapper
// Takes a container and returns a span into it, already wrapped with scn::temp
// Effectively equivalent to the example above
std::vector<int> list(64, 0);
auto result = scn::scan_list("123 456 789", scn::make_span_list_wrapper(list));
// result == true
// list == span == [123, 456, 789]
```

`scn::temp` can be also utilized elsewhere

```
std::vector<char> buffer(64, 0);
// Reads up to 64 chars into the buffer
auto result = scn::scan_default(source, scn::temp(scn::make_span(buffer)));
```

1.6 User types

To scan a value of a program-defined type, specialize `scn::scanner`

```
struct int_and_double {
    int i;
    double d;
};

template <typename CharT>
struct scn::scanner<CharT, int_and_double> : scn::empty_parser<CharT> {
    template <typename Context>
    error scan(int_and_double& val, Context& ctx)
    {
        auto r = scn::scan(ctx.range(), "[{}, {}]", val.i, val.d);
        ctx.range() = std::move(r.range());
```

(continues on next page)

(continued from previous page)

```
        return r.error();
    }
};

// ...

int_and_double val;
auto result = scn::scan("[123, 3.14]", "{}", val);
// result == true
// val.i == 123
// val.d == 3.14
```

The above example inherits from `scn::empty_parser`. This implements the format string functionality for this type. `scn::empty_parser` is a good default choice, as it only accepts empty format strings. You could also inherit from other scanner types (like `scn::scanner<CharT, int>`), or implement `parse()` by hand (see `reader.h` in the library source code).

Alternatively, you could also include the header `<scn/istream.h>`. This enables scanning of types with a `std::istream` compatible `operator>>`. Using this functionality is discouraged, as using iostreams to scan these values presents some difficulties with error recovery, and will lead to worse performance. Specializing `scn::scanner` should be preferred.

1.7 Tuple-based scanning API

By including `<scn/tuple_return.h>`, you'll get access to an alternative API, which returns the scanned values in a tuple instead of output parameters. See Rationale for why this is not the default API.

These functions are slightly slower compared to their output-parameter equivalents, both in runtime and compile time.

```
#include <scn/tuple_return.h>

// Way more usable with C++17 structured bindings
// Can also be used without them
auto [result, i] = scn::scan_tuple<int>("123", "{}");
// result == true
// i == 123
```

1.8 Miscellaneous

1.8.1 `string` vs `string_view` vs `span<char>`

Three types that at first glance might appear quite similar, have significant differences what comes to how they're scanned by the library.

`std::string` works very similarly to how it works with `<iostream>`. It scans a "word": a sequence of letters separated by spaces. More precisely, it reads the source range into the string, until a whitespace character is found or the range reaches its end.

`span<char>` works like `istream.read`: it copies bytes from the range into the buffer it's pointing to. `string_view` works like `std::string`, except it doesn't copy, but changes its data pointer to point into the source stream. Scanning a `string_view` works only with contiguous ranges, and may lead to lifetime issues, but it will give you better performace (avoids copying and allocation).

```
scn::string_view source{"hello world"};

std::string str;
scn::scan(source, "{}", str);
// str == "hello"

scn::string_view sv;
scn::scan(source, "{}", sv);
// sv == "hello"
// sv.data() == source.data() -- sv points to source
// Make sure that `source` outlives `sv`

std::vector<char> buffer(5, '\0'); // 5 bytes, all zero
scn::span<char> s = scn::make_span(buffer);
scn::scan(source, "{}", s);
// s == buffer == "hello"
// Reads 5 bytes, doesn't care about whitespace
// No lifetime problems, the data is copied into the span/the buffer it points to
```

1.8.2 Wide ranges

Source ranges have an associated character type, either `char` or `wchar_t`. This character type is determined by the type of deferencing an iterator into the range, which is either `CharT` or `scn::expected<CharT>`. For most use cases, this type is `char`. In this case, the range is said to be narrow. If the character type is `wchar_t`, the range is said to be wide.

The return types of scanning narrow and wide ranges are incompatible and cannot be mixed.

`char`, `std::string`, `scn::string_view`, and `scn::span<char>` cannot be scanned from a wide range. `wchar_t`, `std::wstring`, `scn::wstring_view`, and `scn::span<wchar_t>` cannot be scanned from a narrow range.

Wide ranges are useful if your source data is wide (often the case on Windows). Narrow ranges should be preferred if possible, however.

The encoding of wide ranges is assumed to be whatever is set in the global C locale. The encoding must be ASCII-compatible.

CHAPTER

TWO

API DOCUMENTATION

2.1 Scanning functions

Main part of the public API.

Generally, the functions in this group take a range, a format string, and a list of arguments. The arguments are parsed from the range based on the information given in the format string.

If the function takes a format string and a range, they must share character types. Also, the format string must be convertible to `basic_string_view<CharT>`, where `CharT` is that aforementioned character type.

template<typename **Range**, typename **Format**, typename ...**Args**>
auto `scn::scan`(*Range* &&*r*, **const** *Format* &*f*, *Args*&... *a*) -> detail::scan_result_for_range<*Range*>

The most fundamental part of the scanning API.

Reads from the range in `r` according to the format string `f`.

```
int i;
scn::scan("123", "{}", i);
// i == 123
```

template<typename **Range**, typename ...**Args**>
auto `scn::scan_default`(*Range* &&*r*, *Args*&... *a*) -> detail::scan_result_for_range<*Range*>

Equivalent to *scan*, but with a format string with the appropriate amount of space-separated `"{}"`s for the number of arguments.

Because this function doesn't have to parse the format string, performance is improved.

Adapted from the example for *scan*

```
int i;
scn::scan_default("123", i);
// i == 123
```

See *scan*

template<typename **Locale**, typename **Range**, typename **Format**, typename ...**Args**>
auto `scn::scan_localized`(**const** *Locale* &*loc*, *Range* &&*r*, **const** *Format* &*f*, *Args*&... *a*) -> detail::scan_result_for_range<*Range*>

Read from the range in `r` using the locale in `loc`.

`loc` must be a `std::locale`. The parameter is a template to avoid inclusion of `<locale>`.

Use of this function is discouraged, due to the overhead involved with locales. Note, that the other functions are completely locale-agnostic, and aren't affected by changes to the global C locale.

```
double d;
scn::scan_localized(std::locale{"fi_FI"}, "3,14", "{}", d);
// d == 3.14
```

See *scan*

template<typename **T**, typename **Range**>
auto `scn::scan_value`(*Range* &&*r*) -> detail::generic_scan_result_for_range<*expected*<*T*>, *Range*>

Scans a single value with the default options, returning it instead of using an output parameter.

The parsed value is in `ret.value()`, if `ret == true`. The return type of this function is otherwise similar to other scanning functions.

```
auto ret = scn::scan_value<int>("42");
if (ret) {
  // ret.value() == 42
}
```

template<typename **Format**, typename ...**Args**, typename **CharT** = ranges::range_value_t<*Format*>>
auto `scn::input`(**const** *Format* &*f*, *Args*&... *a*) -> detail::scan_result_for_range<*basic_file*<*CharT*>&>

Otherwise equivalent to *scan*, expect reads from `stdin`.

Character type is determined by the format string. Syncs with `<cstdio>`.

template<typename **CharT**, typename **Format**, typename ...**Args**>
auto `scn::prompt`(**const** *CharT* **p*, **const** *Format* &*f*, *Args*&... *a*) -> decltype(*input*(*f*, *a*...))

Equivalent to *input*, except writes what's in `p` to `stdout`.

```
int i{};
scn::prompt("What's your favorite number? ", "{}", i);
// Equivalent to:
//   std::fputs("What's your favorite number? ", stdout);
//   scn::input("{}", i);
```

template<typename **Range**, typename **String**, typename **CharT**>
auto `scn::getline`(*Range* &&*r*, *String* &*str*, *CharT* *until*) -> detail::scan_result_for_range<*Range*>

Read the range in `r` into `str` until `until` is found.

`until` will be skipped in parsing: it will not be pushed into `str`, and the returned range will go past it.

`r` and `str` must share character types, which must be `CharT`.

If `str` is convertible to a *`basic_string_view`*:

- And if `r` is a `contiguous_range`:
  - `str` is set to point inside `r` with the appropriate length
- if not, returns an error

Otherwise, clears `str` by calling `str.clear()`, and then reads the range into `str` as if by repeatedly calling `str.push_back`. `str.reserve` is also required to be present.

```
auto source = "hello\nworld"
std::string line;
auto result = scn::getline(source, line, '\n');
// line == "hello"
// result.range() == "world"
```

(continues on next page)

(continued from previous page)

```
// Using the other overload
result = scn::getline(result.range(), line);
// line == "world"
// result.empty() == true
```

template<typename **Range**, typename **String**, typename **CharT** = **typename** detail::extract_char_type<ranges::iterator_t<detail::r
auto `scn::getline`(*Range* &&*r*, *String* &*str*) -> detail::scan_result_for_range<*Range*>

Equivalent to *getline* with the last parameter set to `'\n'` with the appropriate character type.

In other words, reads `r` into `str` until `'\n'` is found.

The character type is determined by `r`.

template<typename **Range**, typename **CharT**>
auto `scn::ignore_until`(*Range* &&*r*, *CharT* *until*) -> detail::scan_result_for_range<*Range*>

Advances the beginning of `r` until `until` is found.

The character type of `r` must be `CharT`.

template<typename **Range**, typename **CharT**>
auto `scn::ignore_until_n`(*Range* &&*r*, ranges::range_difference_t<*Range*> *n*, *CharT* *until*) -> detail::scan_result_for_range<*Range*>

Advances the beginning of `r` until `until` is found, or the beginning has been advanced `n` times.

The character type of `r` must be `CharT`.

template<typename **Range**, typename **Container**, typename **CharT** = **typename** detail::extract_char_type<ranges::iterator_t<*Ran*
auto `scn::scan_list`(*Range* &&*r*, *Container* &*c*, *CharT* *separator* = detail::zero_value<*CharT*>::value) -> detail::scan_result_for_range<*Range*>

Reads values repeatedly from `r` and writes them into `c`.

The values read are of type `Container::value_type`, and they are written into `c` using `c.push_back`.

The values must be separated by separator character `separator`, followed by whitespace. If `separator == 0`, no separator character is expected.

The range is read, until:

- `c.max_size()` is reached, or
- range `EOF` was reached, or
- unexpected separator character was found between values.

In all these cases, an error will not be returned, and the beginning of the returned range will point to the first character after the scanned list.

To scan into `span`, use *span_list_wrapper*. *make_span_list_wrapper*

```
std::vector<int> vec{};
auto result = scn::scan_list("123 456", vec);
// vec == [123, 456]
// result.empty() == true

result = scn::scan_list("123, 456", vec, ',');
// vec == [123, 456]
// result.empty() == true
```

template<typename **Range**, typename **Container**, typename **CharT** = **typename** detail::extract_char_type<ranges::iterator_t<*Ran*

auto `scn::scan_list_until`(*Range* &&*r*, *Container* &*c*, *CharT* *until*, *CharT* *separator* = detail::zero_value<*CharT*>::value) -> detail::scan_result_for_range<*Range*>

Otherwise equivalent to *scan_list*, except with an additional case of stopping scanning: if `until` is found where a separator was expected.

```
std::vector<int> vec{};
auto result = scn::scan_list_until("123 456\n789", vec, '\n');
// vec == [123, 456]
// result.range() == "789"
```

See *scan_list*

2.2 Source range

Various kinds of ranges can be passed to scanning functions.

Fundamentally, a range is something that has a beginning and an end. Examples of ranges are a string literal, a C-style array, and a `std::vector`. All of these can be passed to `std::begin` and `std::end`, which then return an iterator to the range. This notion of ranges was standardized in C++20 with the Ranges TS. This library provides barebone support of this functionality.

2.2.1 Source range requirements

Ranges passed to scanning functions must be:

- bidirectional
- default and move constructible

Using C++20 concepts:

```
template <typename Range>
concept scannable_range =
    std::ranges::bidirectional_range<Range> &&
    std::default_constructible<Range> &&
    std::move_constructible<Range>;
```

A bidirectional range is a range, the iterator type of which is bidirectional: http://eel.is/c++draft/iterator.concepts#iterator.concept.bidir. Bidirectionality means, that the iterator can be moved both forwards: `++it` and backwards `--it`.

Note, that both random-access and contiguous ranges are refinements of bidirectional ranges, and can be passed to the library. In fact, the library implements various optimizations for contiguous ranges.

2.2.2 Recommended range requirements

In addition, to limit unnecessary copies and possible dynamic memory allocations, the ranges should be passed as an lvalue, and/or be a `view`: http://eel.is/c++draft/range.view. A `view` is a `range` that is cheap to copy: think `string_view` or `span`.

Passing a non-view as an rvalue will work, but it may cause worse performance, especially with larger source ranges.

```
// okay: view
scn::scan(std::string_view{...}, ...);

// okay: lvalue
std::string source = ...
scn::scan(source, ...);

// worse performance: non-view + rvalue
scn::scan(std::string{...}, ...);
```

In order for the `.reconstruct()` member function to compile in the result object, the range must be a `pair-reconstructible-range` as defined by https://wg21.link/p1664r1, i.e. be constructible from an iterator and a sentinel.

If the source range is contiguous, and/or its `value_type` is its character type, various fast-path optimizations are enabled inside the library implementation. Also, a `string_view` can only be scanned from such a range.

2.2.3 Character type

The range has an associated character type. This character type can be either `char` or `wchar_t`. The character type is determined by the result of `operator*` of the range iterator. If dereferencing the iterator returns

- `char` or `wchar_t`: the character type is `char` or `wchar_t`, respectively
- `expected<char>` or `expected<wchar_t>`: the character type is `char` or `wchar_t`, respectively

2.2.4 Note on string literals

Please note, that only string literals are ranges (`const char(&)[N]`), not pointers to a constant character (`const char*`). This is because:

- It's impossible to differentiate if a `const char*` is a null-terminated string, a pointer to a single `char`, or a pointer to an array of `char`. For safety reasons, `const char*` is thus not an allowed source range type.
- It's how ranges in the standard are defined: a `const char*` cannot be passed to `std::ranges::begin` or `std::ranges::end` (it doesn't have a clear beginning or an end, for the reason explained above), so it's not even a range to begin with.

Therefore, this code is allowed, as it uses a string literal (`const char(&)[N]`) as the source range type:

```
int i;
scn::scan_default("123", i);
```

But this code isn't, as the source range type used is not a range, but a pointer to constant character (`const char*`):

```
const char* source = "123";
int i;
scn::scan_default(source, i); // compiler error
```

This issue can be avoided by using a `string_view`:

```
const char* source = "123";
int i;
scn::scan_default(scn::string_view{source}, i);
// std::string_view would also work
```

2.3 Return type

The return type of the scanning functions is based on the type of the given range. It contains an object of that range type, representing what was left over of the range after scanning. The type is designed in such a way as to minimize copying and dynamic memory allocations. The type also contains an error value.

struct `scn::`**wrapped_error**

Base class for the result type returned by most scanning functions (except for *scan_value*).

scan_result_base inherits either from this class or *expected*.

Public Functions

::scn::*error* **error**() **const**

Get underlying error.

operator bool() **const**

Did the operation succeed true means success.

template<typename **WrappedRange**, typename **Base**>
class `scn::detail::`**scan_result_base** : **public** scn::detail::scan_result_base_wrapper<*Base*>

Type returned by scanning functions.

Contains an error (inherits from it: for *error*, that's *wrapped_error*; with *scan_value*, inherits from *expected*), and the leftover range after scanning.

The leftover range may reference the range given to the scanning function. Please take the necessary measures to make sure that the original range outlives the leftover range. Alternatively, if possible for your specific range type, call the *reconstruct()* member function to get a new, independent range.

Subclassed by scn::detail::intermediary_scan_result< WrappedRange, Base >

Public Functions

iterator **begin**() **const noexcept**

Beginning of the leftover range.

sentinel **end**() **const noexcept**(noexcept(**declval**<wrapped_range_type>().end()))

End of the leftover range.

bool **empty**() **const noexcept**(noexcept(**declval**<wrapped_range_type>().end()))

Whether the leftover range is empty.

ranges::subrange<iterator, sentinel> **subrange**() **const**

A subrange pointing to the leftover range.

wrapped_range_type &**range**() &

Leftover range.

If the leftover range is used to scan a new value, this member function should be used.

const wrapped_range_type &**range**() **const** &

Leftover range.

If the leftover range is used to scan a new value, this member function should be used.

wrapped_range_type **range**() &&

Leftover range.

If the leftover range is used to scan a new value, this member function should be used.

template<typename **R** = wrapped_range_type, typename = **typename** std::enable_if<*R*::is_contiguous>::type>
basic_string_view<char_type> **string_view**() **const**

These member functions enable more conventient use of the leftover range for non-scnlib use cases.

The range must be contiguous.

The lifetime semantics are as one would expect: *string_view* and *scan* reference the leftover range, *string* allocates a new string, independent of the leftover range.

template<typename **R** = wrapped_range_type, typename = **typename** std::enable_if<*R*::is_contiguous>::type>
span<char_type> **span**() **const**

These member functions enable more conventient use of the leftover range for non-scnlib use cases.

The range must be contiguous.

The lifetime semantics are as one would expect: *string_view* and *scan* reference the leftover range, *string* allocates a new string, independent of the leftover range.

template<typename **R** = wrapped_range_type, typename = **typename** std::enable_if<*R*::is_contiguous>::type>
std::basic_string<char_type> **string**() **const**

These member functions enable more conventient use of the leftover range for non-scnlib use cases.

The range must be contiguous.

The lifetime semantics are as one would expect: *string_view* and *scan* reference the leftover range, *string* allocates a new string, independent of the leftover range.

template<typename **R** = **typename** *WrappedRange*::range_type>
R **reconstruct**() **const**

Reconstructs a range of the original type, independent of the leftover range, beginning from *begin* and ending in *end*.

Compiles only if range is reconstructible.

Note, that the values scanned are only touched iff the scanning succeeded, i.e. `operator bool()` returns `true`. This means, that reading from a default-constructed value of a built-in type on error will cause UB:

```
int i;
auto ret = scn::scan("foo", "{}", i);
// ret == false
// i is still default-constructed -- reading from it is UB
```

2.3.1 Error types

class `scn::error`

Error class.

Used as a return value for functions without a success value.

Public Types

enum code

Error code.

Values:

enumerator good

No error.

enumerator end_of_range

EOF.

enumerator invalid_format_string

Format string was invalid.

enumerator invalid_scanned_value

Scanned value was invalid for given type.

e.g. a period '.' when scanning for an int

enumerator invalid_operation

Stream does not support the performed operation.

enumerator value_out_of_range

Scanned value was out of range for the desired type.

(e.g. `>2^32` for an `uint32_t`)

enumerator invalid_argument

Invalid argument given to operation.

enumerator exceptions_required

This operation is only possible with exceptions enabled.

enumerator source_error

The source range emitted an error.

enumerator unrecoverable_source_error

The source range emitted an error that cannot be recovered from.

The stream is now unusable.

enumerator unrecoverable_internal_error

enumerator max_error

Public Functions

constexpr operator bool() **const noexcept**
Evaluated to true if there was no error.

constexpr enum *code* **code**() **const noexcept**
Get error code.

constexpr bool **is_recoverable**() **const noexcept**
Returns `true` if, after this error, the state of the given input range is consistent, and thus, the range can be used for new scanning operations.

struct success_tag_t

template<typename **T**, typename **Error** = ::scn::*error*, typename **Enable** = void>
class expected
expected-like type.

For situations where there can be a value in case of success or an error code.

2.4 Convenience scan types

These types can be passed to scanning functions (`scn::scan` and alike) as arguments, providing useful functionality.

template<typename **T**>
struct temporary
Allows reading an rvalue.

Stores an rvalue and returns an lvalue reference to it via `operator()`. Create one with *temp*.

template<typename **T**, **typename** std::enable_if<!std::is_lvalue_reference<*T*>::value>::type* = nullptr>
temporary<*T*> `scn::`**temp**(*T &&val*)
Factory function for *temporary*.

Canonical use case is with span:

```
std::vector<char> buffer(32, '\0');
auto result = scn::scan("123", "{}", scn::temp(scn::make_span(buffer)));
// buffer == "123"
```

template<typename **T**>
discard_type<*T*> &`scn::`**discard**()
Scans an instance of `T`, but doesn't store it anywhere.

Uses `scn::temp` internally, so the user doesn't have to bother.

```
int i{};
// 123 is discarded, 456 is read into `i`
auto result = scn::scan("123 456", "{} {}", scn::discard<T>(), i);
// result == true
// i == 456
```

template<typename **T**>
struct span_list_wrapper
Adapts a `span` into a type that can be read into using *scan_list*.

This way, potentially unnecessary dynamic memory allocations can be avoided. To use as a parameter to *scan_list*, use *make_span_list_wrapper*.

```
std::vector<int> buffer(8, 0);
scn::span<int> s = scn::make_span(buffer);

auto wrapper = scn::span_list_wrapper<int>(s);
scn::scan_list("123 456", wrapper);
// s[0] == buffer[0] == 123
// s[1] == buffer[1] == 456
```

See *scan_list*

See *make_span_list_wrapper*

template<typename **T**>
auto `scn::make_span_list_wrapper`(*T &s*) -> *temporary*<detail::span_list_wrapper_for<*T*>>

Adapts a contiguous buffer into a type containing a `span` that can be read into using *scan_list*.

Example adapted from span_list_wrapper:

```
std::vector<int> buffer(8, 0);
scn::scan_list("123 456", scn::make_span_list_wrapper(buffer));
// s[0] == buffer[0] == 123
// s[1] == buffer[1] == 456
```

See *scan_list*

See *span_list_wrapper*

2.5 Format string

Every value to be scanned from the source range is marked with a pair of curly braces `"{}"` in the format string. Inside these braces, additional options can be specified. The syntax is not dissimilar from the one found in fmtlib.

The information inside the braces consist of two parts: the index and the scanning options, separated by a colon `':'`.

The index part can either be empty, or be an integer. If the index is specified for one of the arguments, it must be set for all of them. The index tells the library which argument the braces correspond to.

```
int i;
std::string str;
scn::scan(range, "{1} {0}", i, str);
// Reads from the range in the order of:
//   string, whitespace, integer
// That's because the first format string braces have index '1', pointing to
// the second passed argument (indices start from 0), which is a string
```

After the index comes a colon and the scanning options. The colon only has to be there if any scanning options are specified.

For `span` s, there are no supported scanning options.

2.5.1 Integral types

There are localization specifiers:

- `n`: Use thousands separator from the given locale
- `l`: Accept characters specified as digits by the given locale. Implies `n`
- (default): Use `,` as thousands separator and `[0-9]` as digits

And base specifiers:

- `d`: Decimal (base-10)
- `x`: Hexadecimal (base-16)
- `o`: Octal (base-8)
- `b..` Custom base; `b` followed by one or two digits (e.g. `b2` for binary). Base must be between 2 and 36, inclusive
- (default): Detect base. `0x`/`0X` prefix for hexadecimal, `0` prefix for octal, decimal by default
- `i`: Detect base. Argument must be signed
- `u`: Detect base. Argument must be unsigned

And other options:

- `'`: Accept thousands separator characters, as specified by the given locale (only with `custom`-scanning method)
- (default): Thousands separator characters aren't accepted

These specifiers can be given in any order, with up to one from each category.

2.5.2 Floating-point types

First, there's a localization specifier:

- `n`: Use decimal and thousands separator from the given locale
- (default): Use `.` as decimal point and `,` as thousands separator

After that, an optional `a`, `A`, `e`, `E`, `f`, `F`, `g` or `G` can be given, which has no effect.

2.5.3 `bool`

First, there are a number of specifiers that can be given, in any order:

- `a`: Accept only `true` or `false`
- `n`: Accept only `0` or `1`
- `l`: Implies `a`. Expect boolean text values as specified as such by the given locale
- (default): Accept `0`, `1`, `true`, and `false`, equivalent to `an`

After that, an optional `b` can be given, which has no effect.

2.5.4 Strings (std::string, string_view)

Only supported option is s, which has no effect

2.5.5 Characters (char, wchar_t)

Only supported option is c, which has no effect

2.5.6 Whitespace

Any amount of whitespace in the format string tells the library to skip until the next non-whitespace character is found from the range. Not finding any whitespace from the range is not an error.

2.5.7 Literal characters

To scan literal characters and immediately discard them, just write the characters in the format string. scanf-like []-wildcard is not supported. To read literal { or }, write {{ or }}, respectively.

```
std::string bar;
scn::scan("foobar", "foo{}", bar);
// bar == "bar"
```

2.6 Semantics of scanning a value

In the beginning, with every scn::scan (or similar) call, the library wraps the given range in a scn::detail::range_wrapper. This wrapper provides an uniform interface and lifetime semantics over all possible ranges. The arguments to scan are wrapped in a scn::arg_store. The appropriate context and parse context types are then constructed based on these values, the format string, and the requested locale.

These are passed to scn::vscan, which then calls scn::visit. There, the library calls begin() on the range, getting an iterator. This iterator is advanced until a non-whitespace character is found.

After that, the format string is scanned character-by-character, until an unescaped '{' is found, after which the part after the '{' is parsed, until a ':' or '}' is found. If the parser finds an argument id, the argument with that id is fetched from the argument list, otherwise the next argument is used.

The parse() member function of the appropriate scn::scanner specialization is called, which parses the parsing options-part of the format string argument, setting the member variables of the scn::scanner specialization to their appropriate values.

After that, the scan() member function is called. It reads the range, starting from the aforementioned iterator, into a buffer until the next whitespace character is found (except for char/wchar_t: just a single character is read; and for span: span.size() characters are read). That buffer is then parsed with the appropriate algorithm (plain copy for string s, the method determined by the options object for ints and floats).

If some of the characters in the buffer were not used, these characters are put back to the range, meaning that operator-- is called on the iterator.

Because how the range is read until a whitespace character, and how the unused part of the buffer is simply put back to the range, some interesting situations may arise. Please note, that the following behavior is consistent with both scanf and <iostream>.

```
char c;
std::string str;

// No whitespace character after first {}, no range whitespace is skipped
scn::scan("abc", "{}{}", c, str);
// c == 'a'
// str == "bc"

// Not finding whitespace to skip from the range when whitespace is found in
// the format string isn't an error
scn::scan("abc", "{} {}", c, str);
// c == 'a'
// str == "bc"

// Because there are no non-whitespace characters between 'a' and the next
// whitespace character ' ', ``str`` is empty
scn::scan("a bc", "{}{}", c, str);
// c == 'a'
// str == ""

// Nothing surprising
scn::scan("a bc", "{} {}", c, str);
// c == 'a'
// str == "bc"
```

Using `scn::scan_default` is equivalent to using `"{}"` in the format string as many times as there are arguments, separated by whitespace.

```
scn::scan_default(range, a, b);
// Equivalent to:
// scn::scan(range, "{} {}", a, b);
```

2.7 Files

template<typename **CharT**>
class `scn::basic_file`

Range mapping to a C FILE*.

Not copyable or reconstructible.

Subclassed by *scn::basic_owning_file< CharT >*

Public Functions

basic_file() = default

Construct an empty file.

Reading not possible: *valid()* is `false`

basic_file(FILE *f)

Construct from a FILE*.

Must be a valid handle that can be read from.

FILE ***handle**() **const**

Get the FILE* for this range.

Only use this handle for reading *sync()* has been called and no reading operations have taken place after that.

See *sync*

FILE ***set_handle**(FILE **f*)
Reset the file handle.

Calls *sync()*, if necessary, before resetting.

Return The old handle

bool **valid**() **const**
Whether the file has been opened.

void **sync**()
Synchronizes this file with the underlying FILE*.

Invalidates all non-end iterators. File must be open.

Necessary for mixing-and-matching scnlib and <cstdio>:

```
scn::scan(file, ...);
file.sync();
std::fscanf(file.handle(), ...);
```

Necessary for synchronizing result objects:

```
auto result = scn::scan(file, ...);
// only result.range() can now be used for scanning
result = scn::scan(result.range(), ...);
// .sync() allows the original file to also be used
file.sync();
result = scn::scan(file, ...);
```

class iterator

template<typename **CharT**>
class scn::**basic_owning_file** : **public** scn::*basic_file*<*CharT*>
A child class for *basic_file*, handling fopen, fclose, and lifetimes with RAII.

Public Functions

basic_owning_file() = default
Open an empty file.

basic_owning_file(**const** char **f*, **const** char **mode*)
Open a file, with fopen arguments.

basic_owning_file(FILE **f*)
Steal ownership of a FILE*.

bool **open**(**const** char **f*, **const** char **mode*)
fopen

bool **open**(FILE **f*)
Steal ownership.

void **close**()
Close file.

SCN_NODISCARD bool is_open () const
Is the file open.

template<typename **CharT**>
class scn::**basic_mapped_file** : **public** scn::detail::byte_mapped_file
Memory-mapped file range.

Manages the lifetime of the mapping itself.

Public Functions

basic_mapped_file() = default
Constructs an empty mapping.

basic_mapped_file(**const** char *f)
Constructs a mapping to a filename.

span<**const** *CharT*> **buffer**() **const**
Mapping data.

using scn::**file** = *basic_file*<char>

using scn::**wfile** = *basic_file*<wchar_t>

using scn::**owning_file** = *basic_owning_file*<char>

using scn::**owning_wfile** = *basic_owning_file*<wchar_t>

using scn::**mapped_file** = *basic_mapped_file*<char>

using scn::**mapped_wfile** = *basic_mapped_file*<wchar_t>

template<typename **CharT**>
basic_file<*CharT*> &scn::**stdin_range**()
Get a reference to the global stdin range.

file &scn::**cstdin**()
Get a reference to the global `char`-oriented stdin range.

wfile &scn::**wcstdin**()
Get a reference to the global `wchar_t`-oriented stdin range.

2.8 Lower level parsing and scanning operations

template<typename **Context**, typename **ParseCtx**>
error scn::**vscan**(*Context* &*ctx*, *ParseCtx* &*pctx*, basic_args<**typename** *Context*::char_type> *args*)
In the spirit of {fmt}/`std::format` and `vformat`, `vscan` behaves similarly to *scan*, except instead of taking a variadic argument pack, it takes an object of type `basic_args`, which type-erases the arguments to scan.

This, in effect, will decrease generated code size and compile times dramatically.

`parse_integer` and `parse_float` will provide super-fast parsing from a string, at the expense of some safety and usability guarantees. Using these functions can easily lead to unexpected behavior or UB if not used correctly and proper precautions are not taken.

template<typename **T**, typename **CharT**>
expected<**const** *CharT**> scn::**parse_integer**(*basic_string_view*<*CharT*> *str*, *T* &*val*, int *base* = 10)
Parses an integer into `val` in base `base` from `str`.

Returns a pointer past the last character read, or an error.

Parameters

- `str`: source, can't be empty, cannot have:
  - preceding whitespace
  - preceding `"0x"` or `"0"` (base is determined by the `base` parameter)
  - `'+'` sign (`'-'` is fine)
- `val`: parsed integer, must be default-constructed
- `base`: between [2,36]

template<typename **T**, typename **CharT**>
expected<**const** *CharT**> `scn::parse_float` (*basic_string_view*<*CharT*> *str*, *T &val*)

Parses float into `val` from `str`.

Returns a pointer past the last character read, or an error.

Parameters

- `str`: source, can't be empty
- `val`: parsed float, must be default-constructed

The following functions abstract away the source range in easier to understand parsing operations.

template<typename **WrappedRange**, **typename** std::enable_if<*WrappedRange*::is_contiguous>::type* = nullptr>
expected<*span*<**const typename** detail::extract_char_type<**typename** *WrappedRange*::iterator>::type>> `scn::read_zero_co`

Reads up to `n` characters from `r`, and returns a `span` into the range.

If `r.begin() == r.end()`, returns EOF. If the range does not satisfy `contiguous_range`, returns an empty `span`.

Let `count` be `min(r.size(), n)`. Returns a span pointing to `r.data()` with the length `count`. Advances the range by `count` characters.

template<typename **WrappedRange**, **typename** std::enable_if<*WrappedRange*::is_contiguous>::type* = nullptr>
expected<*span*<**const typename** detail::extract_char_type<**typename** *WrappedRange*::iterator>::type>> `scn::read_all_zer`

Reads every character from `r`, and returns a `span` into the range.

If `r.begin() == r.end()`, returns EOF. If the range does not satisfy `contiguous_range`, returns an empty `span`.

template<typename **WrappedRange**, typename **OutputIterator**, **typename** std::enable_if<*WrappedRange*::is_contiguous>::ty
error `scn::read_into` (*WrappedRange &r*, *OutputIterator &it*, ranges::range_difference_t<*WrappedRange*>
n)

Reads `n` characters from `r` into `it`.

If `r.begin() == r.end()` in the beginning or before advancing `n` characters, returns EOF. If `r` can't be advanced by `n` characters, the range is advanced by an indeterminate amout. If successful, the range is advanced by `n` characters.

template<typename **WrappedRange**, typename **Predicate**, **typename** std::enable_if<*WrappedRange*::is_contiguous>::type* = n

expected<*span*<**const typename** detail::extract_char_type<**typename** *WrappedRange*::iterator>::type>> `scn::read_until_s`

Reads characters from `r` until a space is found (as determined by `is_space`), and returns a `span` into the range.

If `r.begin() == r.end()`, returns EOF. If the range does not satisfy `contiguous_range`, returns an empty `span`.

Parameters

- `is_space`: Predicate taking a character and returning a `bool`. `true` means, that the given character is a space.
- `keep_final_space`: Whether the final found space character is included in the returned span, and is advanced past.

template<typename **WrappedRange**, typename **OutputIterator**, typename **Predicate**, **typename** std::enable_if<*WrappedR*
error `scn::read_until_space`(*WrappedRange* *&r*, *OutputIterator* *&out*, *Predicate* *is_space*, bool *keep_final_space*)

Reads characters from `r` until a space is found (as determined by `is_space`) and writes them into `out`.

If `r.begin() == r.end()`, returns EOF.

Parameters

- `is_space`: Predicate taking a character and returning a `bool`. `true` means, that the given character is a space.
- `keep_final_space`: Whether the final found space character is written into `out` and is advanced past.

template<typename **WrappedRange**, typename **OutputIterator**, typename **Sentinel**, typename **Predicate**, **typename** st
error `scn::read_until_space_ranged`(*WrappedRange* *&r*, *OutputIterator* *&out*, *Sentinel* *end*, *Predicate* *is_space*, bool *keep_final_space*)

Reads characters from `r` until a space is found (as determined by `is_space`), or `out` reaches `end`, and writes them into `out`.

If `r.begin() == r.end()`, returns EOF.

Parameters

- `is_space`: Predicate taking a character and returning a `bool`. `true` means, that the given character is a space.
- `keep_final_space`: Whether the final found space character is written into `out` and is advanced past.

template<typename **WrappedRange**, **typename** std::enable_if<*WrappedRange*::is_contiguous>::type* = nullptr>
error `scn::putback_n`(*WrappedRange* *&r*, ranges::range_difference_t<*WrappedRange*> *n*)

Puts back `n` characters into `r` as if by repeatedly calling `r.advance(-1)` .

template<typename **Context**, **typename** std::enable_if<!*Context*::range_type::is_contiguous>::type* = nullptr>

error `scn::skip_range_whitespace(Context &ctx) noexcept`

Reads from the range in `ctx` as if by repeatedly calling `read_char()`, until a non-space character is found (as determined by `ctx.locale()`), or EOF is reached.

That non-space character is then put back into the range.

2.9 Utility types

template<typename **CharT**>
class basic_string_view

A view over a (sub)string.

Used even when std::string_view is available to avoid compatibility issues.

using `scn::string_view` = *basic_string_view*<char>

using `scn::wstring_view` = *basic_string_view*<wchar_t>

template<typename **T**>
class span

A view over a contiguous range.

Stripped-down version of `std::span`.

template<typename **T**>
class optional

A very lackluster optional implementation.

Useful when scanning non-default-constructible types, especially with <tuple_return.h>:

```
// implement scn::scanner for optional<mytype>
optional<mytype> val;
scn::scan(source, "{}", val);

// with tuple_return:
auto [result, val] = scn::scan_tuple<optional<mytype>>(source, "{}");
```

CHAPTER

THREE

CMAKE USAGE

Using `scnlib` with CMake is pretty easy. Just import it in the way of your liking (`find_package`, `add_subdirectory` etc) and add `scn::scn` (or `scn::scn-header-only`) to your `target_link_libraries`.

3.1 CMake configuration options

These default to `OFF`, unless `scnlib` is built as a standalone project.

- `SCN_TESTS`: Build tests
- `SCN_EXAMPLES`: Build examples
- `SCN_BENCHMARKS`: Build benchmarks
- `SCN_DOCS`: Build docs
- `SCN_INSTALL` Generate `install` target
- `SCN_PEDANTIC`: Enable stricter warning levels

These default to `OFF`, but can be turned on if you want to:

- `SCN_USE_NATIVE_ARCH`: Add `-march=native` to build flags (gcc or clang only). Useful for increasing performance, but makes your binary non-portable.
- `SCN_USE_ASAN`, `SCN_USE_UBSAN`, `SCN_USE_MSAN`: Enable sanitizers, clang only

These default to `ON`:

- `SCN_USE_EXCEPTIONS`, `SCN_USE_RTTI`: self-explanatory

These default to `OFF`, and should only be turned on if necessary:

- `SCN_WERROR`: Stops compilation on compiler warnings
- `SCN_USE_32BIT`: Compile as 32-bit (gcc or clang only)
- `SCN_COVERAGE`: Generate code coverage report
- `SCN_BLOAT`: Generate bloat test target
- `SCN_BUILD_FUZZING`: Build fuzzer
- `SCN_BUILD_LOCALIZED_TEST`: Build localization tests, requires en_US.UTF-8 and fi_FI.UTF-8 locales
- `SCN_BUILD_BLOAT`: Build code bloat benchmarks
- `SCN_BUILD_BUILDTIME`: Build build time benchmarks

CHAPTER

FOUR

RATIONALE

4.1 Why take arguments by reference?

Relevant GitHub issue: https://github.com/eliaskosunen/scnlib/issues/2

Another frequent complaint is how the library requires default-constructing your arguments, and then passing them by reference. A proposed alternative is returning the arguments as a tuple, and then unpacking them at call site.

This is covered pretty well by the above GitHub issue, but to summarize:

- `std::tuple` has measurable overhead (~5% slowdown)
- it still would require your arguments to be default-constructible

To elaborate on the second bullet point, consider this example:

```
auto [result, i, str] =
    scn::scan_tuple<int, non_default_constructible_string>(
        range, scn::default_tag);
```

Now, consider what would happen if an error occurs during scanning the integer. The function would need to return, but what to do with the string? It *must* be default-constructed (`std::tuple` doesn't allow unconstructed members).

Would it be more convenient, especially with C++17 structured bindings? One could argue that, and that's why an alternative API, returning a `tuple`, is available, in the header `<scn/tuple_return.h>`. The rationale of putting it in a separate header is to avoid pulling in the entirety of very heavy standard headers `<tuple>` and `<functional>`.

TODO: flesh this section out

4.2 What's with all the `vscan`, `basic_args` and `arg_store` stuff?

This approach is borrowed (*cough* stolen *cough*) from fmtlib, for the same reason it's in there as well. Consider this peace of code:

```
int i;
std::string str;

scn::scan(range, scn::default_tag, i, str);
scn::scan(range, scn::default_tag, str, i);
```

If the arguments were not type-erased, almost all of the internals would have to be instantiated for every given combination of argument types.

CHAPTER

FIVE

INTRODUCTION

`scnlib` is a modern C++ library for scanning values. Think of it as more C++-y `scanf`, or the inverse of fmtlib.

The repository lives on the scnlib GitHub.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

S