Converting between types and strings

Introducing `boost::coerce`

Jeroen Habraken

The problem, foo and bar

```
int
to_int(std::string const & str) {
    return foo(str);
}
```

and

```
std::string
to_string(int i) {
    return bar(i);
}
```

Table of contents

Your current options

- The old, C style functions `atoi`, `strtol` and `snprintf`
- The new, C++11 functions `stoi` and `to_string`
- The timeless, `boost::lexical_cast`

boost::coerce

- Introduction
- Synopsis
- Backends
- Default backend, `boost::spirit`
- Tags

Table of contents

- Your current options
  - The old, C style functions `atoi`, `strtol` and `snprintf`
  - The new, C++11 functions `stoi` and `to_string`
  - The timeless, `boost::lexical_cast`
- boost::coerce
  - Introduction
  - Synopsis
  - Backends
  - Default backend, `boost::spirit`
  - Tags

atoi

```
int
to_int(std::string const & str) {
    return atoi(str.c_str());
}
```

`atoi`

- Trivial to use
- No error checking, whatsoever
- Deprecated in favour of `strtol`

atoi

- Trivial to use
- No error checking, whatsoever
- Deprecated in favour of strtol

atoi

- Trivial to use
- No error checking, whatsoever
- Deprecated in favour of `strtol`

strtol

```
int
to_int(std::string const & str) {
    char const * c_str = str.c_str();

    if (std::isspace(*c_str))
        throw std::invalid_argument("to_int");

    char * end;

    errno = 0;
    int i = std::strtol(c_str, &end, 10);

    if (errno != 0 || *end != 0 || c_str == end)
        throw std::invalid_argument("to_int");

    return i;
}
```

`strtol`

- Significantly harder to use correctly
- Specific function per type
- Little extensibility

strtol

- Significantly harder to use correctly
- Specific function per type
- Little extensibility

strtol

- Significantly harder to use correctly
- Specific function per type
- Little extensibility

snprintf

```cpp
std::string
to_string(int i) {
    char buffer[BUFFER_SIZE];

    int size = snprintf(buffer, BUFFER_SIZE, "%d", i);

    if (size < 0)
        throw std::invalid_argument("to_string");
    else if (size >= BUFFER_SIZE)
        throw std::length_error("to_string");

    return buffer;
}
```

snprintf

- Buffer size
- Specific modifier per type
- Little extensibility

snprintf

- Buffer size
- Specific modifier per type
- Little extensibility

snprintf

- Buffer size
- Specific modifier per type
- Little extensibility

Table of contents

Your current options

- The old, C style functions `atoi`, `strtol` and `snprintf`
- The new, C++11 functions `stoi` and `to_string`
- The timeless, `boost::lexical_cast`

boost::coerce

- Introduction
- Synopsis
- Backends
- Default backend, `boost::spirit`
- Tags

stoi and friends

In section 21.5 we find numeric conversions:

```
int
stoi(string const & str, size_t * idx = 0, int base = 10);
```

and similarly stol, stoul, stoll and stoull for integer types.

For floating point there there are:

```
float
stof(string const & str, size_t * idx = 0);
```

and similarly stod and stold.

`stoi` and friends specification

How are these specified?

Effects: the first two functions call `strtol(str.c_str(), ptr, base)`.

stoi and friends implementation

How are these implemented?

```
inline int
stoi(string const & __str, size_t * __idx = 0, int __base = 10)
{
    return __gnu_cxx::__stoa<long, int>(&std::strtol,
        "stoi", __str.c_str(), __idx, __base);
}
```

to_string

```
std::string
to_string(int val);
```

and similarly for all integer and floating point types based on `snprintf`.

stoi and to_string

- Easier to use than their C counterparts
- Similar downsides

stoi and to_string

- Easier to use than their C counterparts
- Similar downsides

Table of contents

Your current options

- The old, C style functions `atoi`, `strtol` and `snprintf`
- The new, C++11 functions `stoi` and `to_string`
- The timeless, `boost::lexical_cast`

boost::coerce

- Introduction
- Synopsis
- Backends
- Default backend, `boost::spirit`
- Tags

boost::lexical_cast

```cpp
int
to_int(std::string const & str) {
    return boost::lexical_cast<int>(str);
}
```

and

```cpp
std::string
to_string(int i) {
    return boost::lexical_cast<std::string>(i);
}
```

boost::lexical_cast implementation

```cpp
int
to_int(std::string const & str) {
    std::stringstream interpreter;

    if (!(interpreter << str))
        throw std::invalid_argument("to_int");

    int i;
    if (!(interpreter >> i))
        throw std::invalid_argument("to_int");

    return i;
}
```

- Slow
- No extensibility
- No no-throw interface

- Slow
- No extensibility
- No no-throw interface

- Slow
- No extensibility
- No no-throw interface

Table of contents

Your current options

- The old, C style functions `atoi`, `strtol` and `snprintf`
- The new, C++11 functions `stoi` and `to_string`
- The timeless, `boost::lexical_cast`

boost::coerce

- Introduction
- Synopsis
- Backends
- Default backend, `boost::spirit`
- Tags

History

A bit of history, `SpiritCast`.

Requirements

- Generic
- Easy to use
- Fast
- Error checking
- Takes locale into account
- Extensible
- no-throw interface, default value

boost::coerce

```cpp
int
to_int(std::string const & str) {
    return boost::coerce::as_default<int>(str, 23);
}
```

and

```cpp
std::string
to_string(int i) {
    return boost::coerce::as<std::string>(i);
}
```

What it's not

```cpp
short
to_int(int i) {
    return boost::coerce::as<short>(i);
}
```

Use `boost::numeric_cast`.

Table of contents

- Your current options
  - The old, C style functions `atoi`, `strtol` and `snprintf`
  - The new, C++11 functions `stoi` and `to_string`
  - The timeless, `boost::lexical_cast`
- boost::coerce
  - Introduction
  - Synopsis
  - Backends
  - Default backend, `boost::spirit`
  - Tags

boost::coerce synopsis

A throwing interface, throwing boost::coerce::bad_cast.

```
namespace coerce { namespace traits {

    template <typename Target, typename Source>
    Target
    as(Source const &);

    template <typename Target, typename Source, typename Tag>
    Target
    as(Source const &, Tag const &);

} }
```

For example, boost::coerce::as<std::string>(23) has a Source type int and a Target type std::string.

boost::coerce synopsis

A non-throwing interface.

```
namespace coerce { namespace traits {

    template <typename Target, typename Source>
    Target
    as_default(Source const &,
               Target const & default_value = Target());

    template <typename Target, typename Source, typename Tag>
    Target
    as_default(Source const &, Tag const &,
               Target const & default_value = Target());

} }
```

The default constructed default works nicely with `boost::optional`.

Table of contents

- Your current options
  - The old, C style functions atoi, strtol and snprintf
  - The new, C++11 functions stoi and to_string
  - The timeless, boost::lexical_cast
- boost::coerce
  - Introduction
  - Synopsis
  - Backends
  - Default backend, boost::spirit
  - Tags

boost::coerce synopsis

These interfaces all wrap the following trait.

```
namespace coerce { namespace traits {

    template <
        typename Target
      , typename Source
      , typename Tag = tag::none
      , typename Enable = void
    >
    struct as;

} }
```

```
namespace coerce { namespace traits {

    template <>
    struct as<int, std::string>
        : backend { };

} }
```

boost::coerce synopsis

```
struct backend {
    template <typename Target, typename Source, typename Tag>
    static inline bool
    call(Target & target, Source const & source, Tag const &) {
        // Your implementation
    }
};
```

Table of contents

Your current options

- The old, C style functions `atoi`, `strtol` and `snprintf`
- The new, C++11 functions `stoi` and `to_string`
- The timeless, `boost::lexical_cast`

boost::coerce

- Introduction
- Synopsis
- Backends
- Default backend, `boost::spirit`
- Tags

Strings

Many different types of strings, `char[N]`, `wchar_t *`, `std::string` and `boost::iterator_range` to name a few.

Further split up into source strings and target strings.

Source strings

- `T *`
- `T[N]`
- `std::basic_string<T, Traits, Allocator>`
- `boost::iterator_range<T>`

with `T` matching `traits::is_char<T>`.

For each of these `traits::string_traits` implements a `begin(type const & value)` and `end(type const & value)` returning constant input iterators.

Target strings

- `std::basic_string<T, Traits, Allocator>`
- `std::vector<T, Allocator>`

with `T` matching `traits::is_char<T>`.

For each of these `traits::string_traits` implements a `back_inserter(type & value)` returning a back insert iterator.

Spirit

Conversions are hard, `boost::spirit` to the rescue.

To convert a string (source string) to a type `boost::spirit::qi` is used and to convert a type to a string (target string) `boost::spirit::karma` is used.

Table of contents

Your current options

- The old, C style functions `atoi`, `strtol` and `snprintf`
- The new, C++11 functions `stoi` and `to_string`
- The timeless, `boost::lexical_cast`

boost::coerce

- Introduction
- Synopsis
- Backends
- Default backend, `boost::spirit`
- Tags

Tags

```cpp
struct tag {
    template <typename Iterator, typename Target, typename Source>
    struct parser {
        parser(tag const &) {
            // A boost::spirit::qi parser
        }
    };

    template <typename Iterator, typename Target, typename Source>
    struct generator {
        generator(tag const &) {
            // A boost::spirit::karma generator
        }
    };
};
```

Tags

A default tag of tag::none building upon boost::spirit::qi::auto_ and boost::spirit::karma::auto_.

```cpp
struct none {
    template <typename Iterator, typename Target, typename Source>
    struct parser
        : spirit::traits::create_parser<Target>::type {
        parser(tag::none const &) { }
    };

    template <typename Iterator, typename Target, typename Source>
    struct generator
        : spirit::traits::create_generator<Source>::type {
        generator(tag::none const &) { }
    };
}
```

boost::coerce

```cpp
unsigned
to_int_hex(std::string const & str) {
    return boost::coerce::as<unsigned>(str,
        boost::coerce::tag::hex());
}
```

and

```cpp
std::string
to_string_hex(int i) {
    return boost::coerce::as<std::string>(i,
        boost::coerce::tag::hex());
}
```

Questions?

You can find the source code at `http://svn.boost.org/svn/boost/sandbox/coerce/` and contact me at vexocide@gmail.com.