Boost.Checks

Pierre Talbot

Table of Contents

Boost.Checks

Overview

This library provides a collection of functions for validating and creating check digits.

Most are primarily for checking the accuracy of short strings of typed input (though it obviously also provides against a mis-scan by a device like a bar code or card reader). The well-known ISBN is a typical example. All single altered digits, most double altered digits, and transpositions of two digits are caught, and the input rejected as an invalid ISBN.

Important

This is not (yet) an official Boost library. It was a Google Summer of Code project (2011) whose mentor organization was Boost. It remains a library under construction, the code is quite functional, but interfaces, library structure, and names may still be changed without notice. The current version is available at

https://svn.boost.org/svn/boost/sandbox/SOC/2011/checks/libs/checks/doc/pdf/checks.pdf PDF documentation

https://svn.boost.org/svn/boost/sandbox/SOC/2011/checks/libs/checks/doc/html/index.html HTML documentation

https://svn.boost.org/svn/boost/sandbox/SOC/2011/checks/boost/checksboost Boost Sandbox checks source code

[note Comments and suggestions (even bugs!) to Pierre Talbot pierre.talbot.6114(at) herslibramont (dot).be

Document Conventions

- **Tutorials** are listed in the *Table of Contents* and include many examples that should help you get started quickly.
- **Source code** of the many *Examples* will often be your quickest start.
- **Reference section** prepared using Doxygen will provide the function and class signatures, but there is also an *index* of these.
- The main *index* will also help, especially if you know a word describing what it does, without needing to know the exact name chosen for the function.

This documentation makes use of the following naming and formatting conventions.

- C++ Code is in `fixed width font` and is syntax-highlighted.

- Other code is in `teletype fixed-width font`.
- Replaceable text that you will need to supply is in *`italics`*.
- If a name refers to a free function, it is specified like this: `free_function()`; that is, it is in code font and its name is followed by `()` to indicate that it is a free function.
- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by <> to indicate that it is a class template.
- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.
- Many code snippets assume an implicit namespace, for example, `std::` or `boost::checks`.
- If you have a feature request, or if it appears that the implementation is in error, please check the TODO section first, as well as the rationale section.

If you do not find your idea/complaint, please reach the author either through the Boost development list, or email the author(s) direct .

Admonishments

Note

In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Tip

These blocks contain information that you may find helpful while coding.

Important

These contain information that is imperative to understanding a concept. Failure to follow suggestions in these blocks will probably result in undesired behavior. Read all of these you find.

Warning

Failure to heed this will lead to incorrect, and very likely undesired, results.

Introduction

The checks are required in a numerous kind of domains such as the distribution chain (bar codes), the cards number (bank, fidelity cards, ...) and many others. These codes and numbers are often copied or scanned by humans or machines, and both make errors. We need a way to control it and this is why some people created a check digit. A check digit is aimed to control the validity of a number and catch mismatched input (we'll detail further the different errors). Another functionnality of this library is to calculate the check digit of a number. There are other functionnalities more specific to a number, for example, we can *transform* an ISBN-10 to an ISBN-13.

There are a lot of codes and numbers that use a check digit, for instance : the ISBN for the books or the IBAN for the internationnal account numbers. But many of those are specialisation of well-known algorithms such as Luhn or modulus 11 algorithm. For example : ISBN-13 is a specialisation of the EAN-13 which is a specialisation of the modulus 10 algorithm.

This library is divided into two parts : a low level part (Luhn, modulus 11, modulus 10, ...) and a higher level library (ISBN-10, EAN-13, IBAN, VISA number, ...). The higher level library will use the low level with filter on the length, first X characters, ... Theorically, the user should only use the high level library which is more specific. In some cases, the user would like to use the lower level library because some kind of exotic numbers (social number of india,...) are not provided by the library.

Type of errors

The following sections will describe some of the errors that an user or a device can make. Those are the most frequent and we are not exhaustive, however we will find out how well our algorithms work. We will calculate (in a mathematical way) the probability of failures and the factors which affect it.

Alteration

Single error

If the digits are added, an alteration of one digit will always render a different sum and therefore the check digit.

Multiple error

If more than one digit is altered, a simple sum can't ensure that the check digit will be different. In fact it depends on the compensation of the altered digits. For example : 1 + 2 + 3 = 6. If we alter 2 digits, the sum could become : 2 + 2 + 2 = 6. The result is equal because 1 + 3 = 2 + 2, the digits altered are compensated.

Transposition

A transposition on a simple sum is impossible to detect because the addition is commutative, the order is not important. A solution is to associate the position of a digit with a weight.

Note

A transposition error is only catched if the two digits transposed have a different weight and if their values with their weight or the weight of the other digit are not the same.

Length

The length is not often a problem because many codes and numbers have a fixed length. But if the user do not specify the size, an error could be uncatched if the check digit of the new sequence of digit is equal to the last digit of this sequence.

Shift

Phonetic

Modular sum algorithms

A *modular sum algorithm* computes the sum of a sequence of digits modulus a number. The number obtained is called the *check digit*, in many codes it is added as the last digit. This rubbish algorithm detect all *alteration* of one digit but doesn't detect a simple *transposition* if the check digit is not transposed. This is why even the most basic algorithms introduce the notion of *weight*. The weight is the contribution of a number to the final sum. The following algorithms presented are the base of many, many codes and numbers in the world. We could describe a number and its check digit calculation with three characteristics : length, weigth and the modulus. So we could design a generic function but we won't. It wouldn't be efficient and would be unnecessarily more complicated. The next parts will present the three different algorithms that we have choose to design.

Note

We may add other algorithms later

Luhn algorithm

Description

The Luhn algorithm is used with a lot of codes and numbers, the most well-known usage is the verification of the *credit card numbers*. It produces a check digit from a sequence with an unlimited length. The weigth pattern used is : from the rightmost digit (the check digit) double the value of every second digit. It use a modulus 10 on the sum, the range of the check digit is from 0 to 9.

Note

When a digit is doubled, we subtract 9 from the result if it exceeds 9

Errors

Alterations of one digit are all catched. The alterations of more than one digit are not all catched. All **transpositions** on digits with different weight are catched but the sequence "90" or "09" because:

```
9*2 = 18 and 18-9 = 9
9 * 2 = 9 * 1
0 * 2 = 0 * 1
```

The two digits have the same value if there are doubled or not. Seeing that Luhn alternates a weight of 1 and 2, the transpositions on digits with the same weight are not catched.

Modulus 10 algorithm

Description

This algorithm use a modulus 10 as the Luhn algorithm but use a custom weight pattern. The sum is made without subtraction if the multiplication of a digit exceeds 9. The custom weight pattern is interresting for many codes and numbers that aren't implemented in the high level library. The user can easily craft his own check function with this weight pattern.

Modulus 11 algorithm

The modulus 11 algorithm use a modulus of 11, so we have 11 possible check digits. The ten first characters are the figures from 0 to 9, the eleventh is a special character choose by the designer of the number. It is typically 'X' or 'x'. The weight of a digit is related to its position. The weight of the first character is equal to the total length of the number (with the check digit included). The weight decrease by one for the second position, one again for the third, etc.

Examples

In this part, we'll see some basic examples which show how to check or compute a number with an algorithm. After this, we'll discuss about a few specific cases.

Firstly, we need to include the file including the check and compute functions.

```
#include <boost/checks/modulus.hpp>
```

Check a number

We can check a number which is delimitated by two iterators. The most simple is using std::string.

```
std::string luhn_number = "123455" ; // Initialisation
// Verification
if( boost::checks::check_luhn( luhn_number.begin(), luhn_number.end(), 6) )
  std::cout << "The number " << luhn_number << " is a valid luhn number." << std::endl;
```

This provides this output:

```
The number 123455 is a valid luhn number.
```

Compute a check digit

We can compute a check digit which is in the range [0..9] (+ 'X' for the modulus 11 algorithm). The check digit can be of the type "char" or a custom type such as *wchar_t*.

This example will show how to compute a serie of check digit retrieved from a UTF-8 file and put the result into another UTF-8 file.

```
std::wifstream mod11_ifile("files/checks_file_unicode_input.txt", std::ios::in);
std::wofstream mod11_ofile("files/checks_file_unicode_out↵
put.txt", std::ios::out | std::ios::trunc );

std::wstring tmp_mod11_number ;

while( std::getline(mod11_ifile, tmp_mod11_number) )
{
  wchar_t check_digit = boost::checks::compute_mod11<wchar_t>(tmp_mod11_number.be↵
gin(), tmp_mod11_number.end() );
  mod11_ofile << tmp_mod11_number << " check digit : " << check_digit << std::endl ;
}
mod11_ofile.close();
mod11_ifile.close();
```

The contents of the input and output files are showed on the following table. We tested the program with 3 differents numbers, the digits of this number are separated with the unicode character '†'.

Table 1. Contents of the input and output files

Input file	Output file
1†2†3†4†5†1	1†2†3†4†5†1 check digit : X
2†3†4†5†6†7	2†3†4†5†6†7 check digit : 6
3†4†5†6†7†8	3†4†5†6†7†8 check digit : 1

Use the weight pattern

We can use a custom weight pattern with the modulus 10 algorithm. In the following example, we'll see how to compute and check a Routing transit number (RTN).

```
// Design the weight pattern for the Routing transit number (RTN)
boost::array<unsigned int, 3> rtn_weight_pattern = {3,7,1};

std::string rtn_number = "12345678" ;

rtn_number += boost::checks::compute_mod10(rtn_number.begin(), rtn_number.end(), rtn_weight_pat↵
tern, 8) ;

if ( boost::checks::check_mod10( rtn_number.begin(), rtn_number.end(), rtn_weight_pattern, 9) )
  std::cout << "The routing transit number (RTN) : " << rtn_number << " is valid." << std::endl ;
```

Special cases

1. Numbers one beside the others

The first special case is to test a collection of number which are in the same container without special separation.

```
// Create a string with numbers
std::string luhn_numbers = "";
for(int i=9; i >= 0; --i)
{
std::ostringstream num_gen;
num_gen << 1023456 << i ;
std::string num = num_gen.str();
char checkdigit = boost::checks::compute_luhn(num.begin(), num.end(), num.size());
luhn_numbers = luhn_numbers + num + checkdigit ;
}

// Check all the numbers from this string
std::string::iterator luhn_iter = luhn_numbers.begin();
while(luhn_iter != luhn_numbers.end())
  if(boost::checks::check_luhn(luhn_iter, luhn_numbers.end(), 9))
    std::cout << "The number is valid." << std::endl;
  else
    std::cout << "The number is invalid." << std::endl;
```

This provides this output:

```
The number is valid.
The number is valid.
The number is valid.
The number is valid.
The number is valid.
The number is valid.
The number is valid.
The number is valid.
The number is valid.
The number is valid.
```

Summary

Here a summary of the different algorithms studied.

Table 2. Summary of the modular sum algorithms

Algorithm	Modulus	Weight pattern	check digit range
Luhn	10	...21	0..9
Modulus 10	10	custom	0..9
Modulus 11	11	n...4321	0..9 + 'X'

ISBN checking

The functions defined at are for validating and computing check digits of International Standard Book Number (ISBN) strings.

Error detecting

You probably want a section on how good things are at detecting changes in the string. This has been well studied for the Verhoeff/Gumm system.

Some of your tests might be devoted to confirming that this is really true?

All alterations of a single digit will be detected.

Most alterations of two digits will be detected.

All two digit transpositions will be detected.

Synopsis

```
// This function checks if an `isbn10` is a valid ISBN.
bool is_isbn10(const std::string& isbn);

// This function computes and returns the check digit for a given 9 digit ISBN in `isbn`.
char isbn_check_digit(const std::string& isbn);
```

Both functions assume that `isbn` is a 10-digit ISBN containing only ANSI digits '0' to '9', or ANSI letters 'X', 'Y', 'x', 'y'.

Universal Product Code (UPC) checking

UPC is a sub-set of International Article Number (EAN) - see Universal Product Code (UPC).

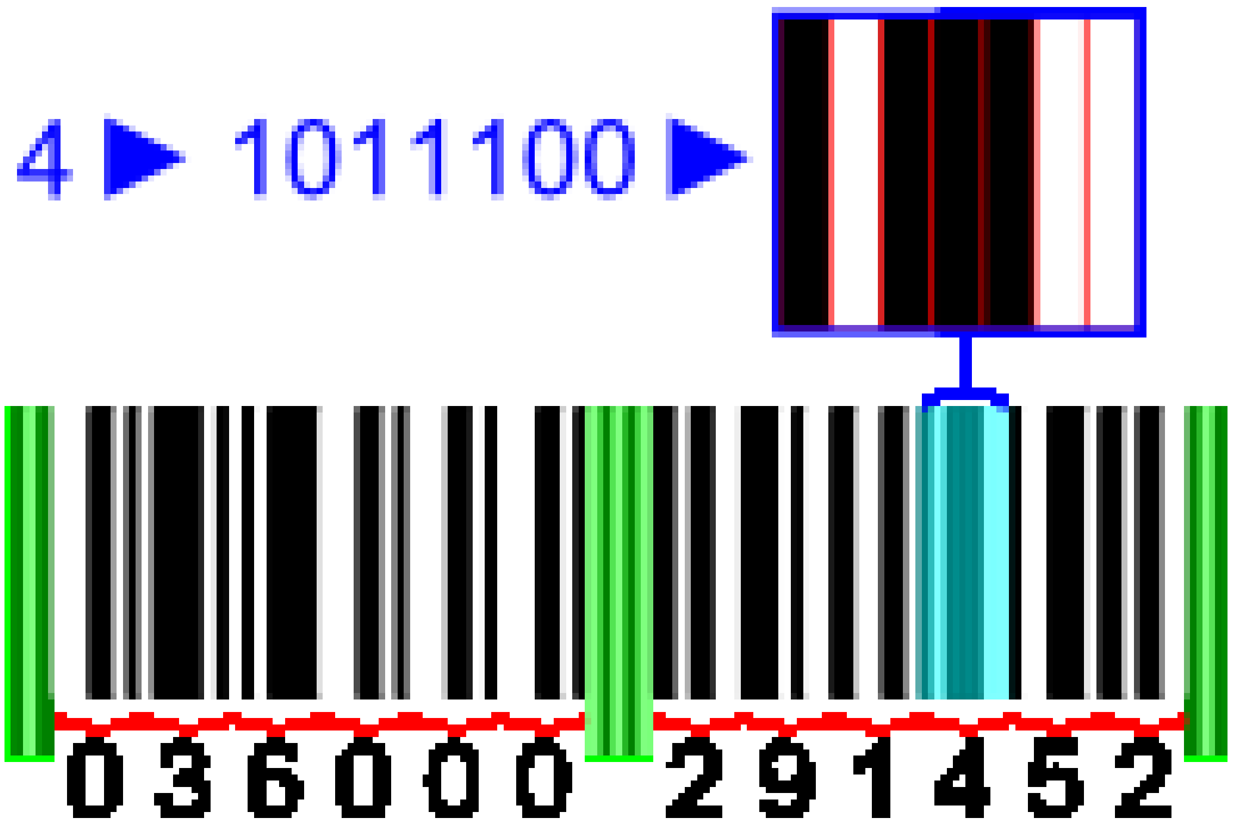
The original UPC is still in use and has 12 decimal digits, for example, a UPC for a box of tissues)

"03600029145X" where X is the check digit, in this case having a value of 2.

On products, it is usually printed as a barcode, but the decimal digits are visible too.

UPC EANUCC 12 barcode

This is a local copy of the barcode:



International Article Number (EAN) checking

EAN is a super-set of the original US 12-digit Universal Product Code (UPC) UPC13 digit (12 + check digit) barcoding standard.

See ????

Tutorial Examples

Here some general tutorial stuff.

followed by specific examples??

ISBN example

Here is a really trivial example of making an ISBN check and also computing the check digit.

First we need to include the appropriate file including the check and compute functions.

Then assuming that the ISBN is in a `std::string`, we can check a complete ISBN, and also compute the check digit from one lacking the check digit.

This provides this output:

Hints and Tips

- This manual is also available as a single PDF file which may be easily emailed and printed.
- This is another tip?

Acknowledgements

- Thanks to Paul A. Bristow who is the mentor of this project for her infinite patience and his wise advices.
- UPC EANUCC 12 barcode is copied from Wikipedia under the Creative Commons license.

FAQs

- Why are checks needed?
- How many alterations to the strings are detected (or undetected)?

References

1. International Standard Book Number (ISBN)
2. International Standard Serial Number (ISSN)

Rationale

This section records the rationale and compromises for some design decisions.

Function parameter

- Functions that take a single `std::string` are convenient for users with simple requirements. This may be simplest what a string is typed in.
- Functions that use std:: iterators `begin` and `end` allow efficient use of a decimal digits string within other text, for example when a record is retrieve from a database.

Scope of the project

- Scott McMurray has identifed four fairly distinct types of check:
  1. ISBN/ISSN/UPC/EAN/VISA/etc, for catching human-entry errors.
  2. hash functions as in hash tables, which only care about distribution.
  3. checksums like CRC32, for catching data transmission errors.
  4. and cryptographic hash functions, the only ones useful against malicious adversaries.

This project is directed first at the first class. Others might be the subject of future additions or other libraries.

- Performance is not a major objective, as most input is tiny, and the number of items often likely to be quite small.
- Convenience and flexibility for the user is the highest priority.

History

1. Project started by Pierre Talbot June 2011 as a Google Summer of Code Project.
2. First release in Boost Sandbox for public comment ????

TODO

This section lists items that are acknowledged as work still TODO.

1. Produce 1st version for comment.
2. Produce version for pre-review.

Version Info

Last edit to Quickbook file D:\boost-sandbox\SOC\2011\checks\libs\checks\doc\checks.qbk was at 05:02:47 PM on 2011-Jul-17.

Tip

This version information should appear on the pdf version (but is redundant on html where the last revised date is on the bottom line of the home page).

Warning

Home page "Last revised" is GMT, not local time. Last edit date is local time.

Caution

It does not give the last edit date of other included .qbk files, so may mislead!

Checks Reference

Header <boost/checks/adler.hpp>

Header <boost/checks/amex.hpp>

Header <boost/checks/checks_fwd.hpp>

Boost.Checks forward declaration of function signatures.

This file can be used to copy a function signature, but is mainly provided for testing purposes.

```
namespace boost {
  namespace checks {
    template<typename luhn_iter>
      bool check_luhn(luhn_iter &, const luhn_iter &, std::size_t = 0);
    template<typename mod10_iter, typename weight_t>
      bool check_mod10(mod10_iter &, const mod10_iter &, const weight_t &,
                       std::size_t = 0);
    template<typename mod11_iter>
      bool check_mod11(mod11_iter &, const mod11_iter &, std::size_t = 0);
    template<typename mod97_iter>
      bool check_mod97(mod97_iter &, const mod97_iter &, std::size_t = 0);
    template<typename luhn_iter>
      char compute_luhn(luhn_iter &, const luhn_iter &, std::size_t = 0);
    template<typename luhn_checkdigit, typename luhn_iter>
      luhn_checkdigit
      compute_luhn(luhn_iter &, const luhn_iter &, std::size_t = 0);
    template<typename mod10_iter, typename weight_t>
      char compute_mod10(mod10_iter &, const mod10_iter &, const weight_t &,
                         std::size_t = 0);
    template<typename mod10_checkdigit, typename mod10_iter,
             typename weight_t>
      mod10_checkdigit
      compute_mod10(mod10_iter &, const mod10_iter &, const weight_t &,
                    std::size_t = 0);
    template<typename mod11_iter>
      char compute_mod11(mod11_iter &, const mod11_iter &, std::size_t = 0);
    template<typename mod11_checkdigit, typename mod11_iter>
      mod11_checkdigit
      compute_mod11(mod11_iter &, const mod11_iter &, std::size_t = 0);
    template<typename mod97_iter>
      char compute_mod97(mod97_iter &, const mod97_iter &, std::size_t = 0);
  }
}
```

Function template check_luhn

boost::checks::check_luhn

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>


template<typename luhn_iter>
  bool check_luhn(luhn_iter & begin, const luhn_iter & end,
                  std::size_t nbr_digits = 0);
```

Description

Validate the check digit of the number provided with the Luhn algorithm.

Parameters:	`begin`	The beginning of the sequence to check.
	`end`	One off the limit of the sequence to check.
	`nbr_digits`	The number of digits on which the Luhn algorithm will operate. If the size is < 1, the luhn algorithm will validate the check digit with all the digit encountered.
Requires:	The parameters begin and end must be two valid initialized iterators. They represent a sequence of character encoded in big-endian mode in a format compatible with the 7 bits ASCII.	
Postconditions:	begin is equal to the position of the check digit plus one if the expression provided is correct, otherwise is equal to end.	
Returns:	true is returned if the expression given have a valid check digit and have nbr_digits (or more than 0 digit if nbr_digits is equal to 0).	

Function template check_mod10

boost::checks::check_mod10

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>


template<typename mod10_iter, typename weight_t>
  bool check_mod10(mod10_iter & begin, const mod10_iter & end,
                   const weight_t & weight, std::size_t nbr_digits = 0);
```

Description

Validate the check digit of the number provided with the modulus 10 algorithm.

Parameters:	`begin`	The beginning of the sequence to check.
	`end`	One off the limit of the sequence to check.
	`nbr_digits`	The number of digits on which the modulus 10 algorithm will operate. If the size is < 1, the modulus 10 algorithm will validate the check digit with all the digit encountered.
	`weight`	The weight pattern of the sequence starting on the left of the expression. The size of weight should be greater than 0.
Requires:	begin and end are valid initialized iterators. They represent a sequence of character encoded in big-endian mode in a format compatible with the 7 bits ASCII.	
Postconditions:	begin is equal to the position of the check digit plus one if the expression provided is correct, otherwise is equal to end.	
Returns:	true is returned if the expression given have a valid check digit and have nbr_digits (or more than 0 digit if nbr_digits is equal to 0) and if the size of weight is greater than 0. Returns false otherwise.	

Function template check_mod11

boost::checks::check_mod11

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>


template<typename mod11_iter>
  bool check_mod11(mod11_iter & begin, const mod11_iter & end,
                   std::size_t nbr_digits = 0);
```

Description

Calculate the check digit of the number provided with the modulus 11 algorithm.

Parameters:	`begin`	The beginning of the sequence to check.
	`end`	One off the limit of the sequence to check.
	`nbr_digits`	The number of digits on which the MOD11 algorithm will operate. If the size is < 1, the MOD11 algorithm will calculate the check digit with all the digit encountered. If the size isn't precised and if the character 'X' or 'x' is encountered, the end of the number will be considered ; if the size is precised, the character 'X' or 'x' will be ignored unless it is the last character.
Requires:	begin and end are valid initialized iterators. They represent a sequence of character encoded in big-endian mode in a format compatible with the 7 bits ASCII.	
Postconditions:	begin is equal to the position of the last digit encountered plus one if the expression provided is correct, otherwise is equal to end.	
Returns:	0 is returned if the expression given have not nbr_digits (or no digit if nbr_digits is equal to 0). Otherwise the ASCII character of the check digit is returned.	

Function template check_mod97

boost::checks::check_mod97

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>


template<typename mod97_iter>
  bool check_mod97(mod97_iter & begin, const mod97_iter & end,
                   std::size_t nbr_digits = 0);
```

Description

Validate the check digit of the number provided with the modulus 97 algorithm.

Parameters:	`begin`	The beginning of the sequence to check.
	`end`	One off the limit of the sequence to check.
	`nbr_digits`	The number of digits on which the modulus 97 algorithm will operate. If the size is < 1, the modulus 97 algorithm will validate the check digit with all the digit encountered.
Requires:	begin and end are valid initialized iterators. They represent a sequence of character encoded in big-endian mode in a format compatible with the 7 bits ASCII.	
Postconditions:	begin is equal to the position of the check digits plus one if the expression provided is correct, otherwise is equal to end.	
Returns:	true is returned if the expression given have a valid check digit and have nbr_digits (or more than 0 digit if nbr_digits is equal to 0).	

Function template compute_luhn

boost::checks::compute_luhn

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>


template<typename luhn_iter>
  char compute_luhn(luhn_iter & begin, const luhn_iter & end,
                    std::size_t nbr_digits = 0);
```

Description

Calculate the check digit of the number provided with the Luhn algorithm.

Parameters:	`begin`	The beginning of the sequence to check.
	`end`	One off the limit of the sequence to check.
	`nbr_digits`	The number of digits on which the Luhn algorithm will operate. If the size is < 1, the luhn algorithm will calculate the check digit with all the digit encountered.
Requires:	begin and end are valid initialized iterators. They represent a sequence of character encoded in big-endian mode in a format compatible with the 7 bits ASCII.	
Postconditions:	begin is equal to the position of the last digit encountered plus one if the expression provided is correct, otherwise is equal to end.	
Returns:	0 is returned if the expression given have not nbr_digits (or no digit if nbr_digits is equal to 0). Otherwise the ASCII character of the check digit is returned.	

Function template compute_luhn

boost::checks::compute_luhn

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>


template<typename luhn_checkdigit, typename luhn_iter>
  luhn_checkdigit
  compute_luhn(luhn_iter & begin, const luhn_iter & end,
               std::size_t nbr_digits = 0);
```

Description

Calculate the check digit of the number provided with the Luhn algorithm.

Parameters:	`begin`	The beginning of the sequence to check.
	`end`	One off the limit of the sequence to check.
	`nbr_digits`	The number of digits on which the Luhn algorithm will operate. If the size is < 1, the luhn algorithm will calculate the check digit with all the digit encountered.
Requires:	begin and end are valid initialized iterators. They represent a sequence of character encoded in big-endian mode in a format compatible with the 7 bits ASCII.	
Postconditions:	begin is equal to the position of the last digit encountered plus one if the expression provided is correct, otherwise is equal to end.	
Returns:	0 is returned if the expression given have not nbr_digits (or no digit if nbr_digits is equal to 0). Otherwise the ASCII character of the check digit is returned.	

Function template compute_mod10

boost::checks::compute_mod10

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>


template<typename mod10_iter, typename weight_t>
  char compute_mod10(mod10_iter & begin, const mod10_iter & end,
                     const weight_t & weight, std::size_t nbr_digits = 0);
```

Description

Compute the check digit of the number provided with the modulus 10 algorithm.

Parameters:	`begin`	The beginning of the sequence to check.
	`end`	One off the limit of the sequence to check.
	`nbr_digits`	The number of digits on which the modulus 10 algorithm will operate. If the size is < 1, the modulus 10 algorithm will calculate the check digit with all the digit encountered.
	`weight`	The weight pattern of the sequence starting on the left of the expression. The size of weight should be greater than 0.
Requires:	begin and end are valid initialized iterators. They represent a sequence of character encoded in big-endian mode in a format compatible with the 7 bits ASCII.	
Postconditions:	begin is equal to the position of the last digit encountered plus one if the expression provided is correct, otherwise is equal to end.	
Returns:	0 is returned if the expression given have not nbr_digits (or no digit if nbr_digits is equal to 0) or weight have a size of 0. Otherwise the ASCII character of the check digit is returned.	

Function template compute_mod10

boost::checks::compute_mod10

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>


template<typename mod10_checkdigit, typename mod10_iter, typename weight_t>
  mod10_checkdigit
  compute_mod10(mod10_iter & begin, const mod10_iter & end,
                const weight_t & weight, std::size_t nbr_digits = 0);
```

Description

Compute the check digit of the number provided with the modulus 10 algorithm.

Parameters:	`begin`	The beginning of the sequence to check.
	`end`	One off the limit of the sequence to check.
	`nbr_digits`	The number of digits on which the modulus 10 algorithm will operate. If the size is < 1, the modulus 10 algorithm will calculate the check digit with all the digit encountered.
	`weight`	The weight pattern of the sequence starting on the left of the expression. The size of weight should be greater than 0.
Requires:	begin and end are valid initialized iterators. They represent a sequence of character encoded in big-endian mode in a format compatible with the 7 bits ASCII.	
Postconditions:	begin is equal to the position of the last digit encountered plus one if the expression provided is correct, otherwise is equal to end.	
Returns:	0 is returned if the expression given have not nbr_digits (or no digit if nbr_digits is equal to 0) or weight have a size of 0. Otherwise the ASCII character of the check digit is returned.	

Function template compute_mod11

boost::checks::compute_mod11

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>


template<typename mod11_iter>
  char compute_mod11(mod11_iter & begin, const mod11_iter & end,
                     std::size_t nbr_digits = 0);
```

Description

Validate the check digit of the number provided with the modulus 11 algorithm.

Parameters:	`begin`	The beginning of the sequence to check.
	`end`	One off the limit of the sequence to check.
	`nbr_digits`	The number of digits on which the Luhn algorithm will operate. If the size is < 1, the luhn algorithm will validate the check digit with all the digit encountered.
Requires:	begin and end are valid initialized iterators. They represent a sequence of character encoded in big-endian mode in a format compatible with the 7 bits ASCII.	
Postconditions:	begin is equal to the position of the check digit plus one if the expression provided is correct, otherwise is equal to end.	
Returns:	true is returned if the expression given have a valid check digit and have nbr_digits (or more than 0 digit if nbr_digits is equal to 0).	

Function template compute_mod11

boost::checks::compute_mod11

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>


template<typename mod11_checkdigit, typename mod11_iter>
  mod11_checkdigit
  compute_mod11(mod11_iter & begin, const mod11_iter & end,
                std::size_t nbr_digits = 0);
```

Description

Validate the check digit of the number provided with the modulus 11 algorithm.

Parameters:	`begin`	The beginning of the sequence to check.
	`end`	One off the limit of the sequence to check.
	`nbr_digits`	The number of digits on which the Luhn algorithm will operate. If the size is < 1, the luhn algorithm will validate the check digit with all the digit encountered.
Requires:	begin and end are valid initialized iterators. They represent a sequence of character encoded in big-endian mode in a format compatible with the 7 bits ASCII.	
Postconditions:	begin is equal to the position of the check digit plus one if the expression provided is correct, otherwise is equal to end.	
Returns:	true is returned if the expression given have a valid check digit and have nbr_digits (or more than 0 digit if nbr_digits is equal to 0).	

Function template compute_mod97

boost::checks::compute_mod97

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>


template<typename mod97_iter>
  char compute_mod97(mod97_iter & begin, const mod97_iter & end,
                     std::size_t nbr_digits = 0);
```

Description

!! The modulus 97 algorithm wait for a sequence of numbers only ! and will not do anything else that sum the digits and calculate the modulus of this sum. If you need to check an IBAN use the algorithms in iban.hpp Compute the check digit of the number provided with the modulus 97 algorithm.

Parameters:	`begin`	The beginning of the sequence to check.
	`end`	One off the limit of the sequence to check.
	`nbr_digits`	The number of digits on which the modulus 97 algorithm will operate. If the size is < 1, the modulus 97 algorithm will calculate the check digit with all the digit encountered.
Requires:	begin and end are valid initialized iterators. They represent a sequence of character encoded in big-endian mode in a format compatible with the 7 bits ASCII.	
Postconditions:	begin is equal to the position of the last digit encountered plus one if the expression provided is correct, otherwise is equal to end.	
Returns:	0 is returned if the expression given have not nbr_digits (or no digit if nbr_digits is equal to 0). Otherwise the ASCII characters of the check digits are returned.	

Header <boost/checks/crc.hpp>

Header <boost/checks/ean.hpp>

```
namespace boost {
  namespace checks {
    template<typename In> char ean13_check_digit(In, In);
    template<typename In> char ean8_check_digit(In, In);
    template<typename In, typename Prefix>
      bool Is_ean13(In, In, Prefix = null, Prefix = null);
    template<typename In> bool Is_ean8(In, In);
  }
}
```

Function template ean13_check_digit

boost::checks::ean13_check_digit

Synopsis

```
// In header: <boost/checks/ean.hpp>


template<typename In> char ean13_check_digit(In ean_begin, In ean_end);
```

Description

Compute the check digit of the European Article Numbering of size 13 (EAN-13) provided.

Function template ean8_check_digit

boost::checks::ean8_check_digit

Synopsis

```
// In header: <boost/checks/ean.hpp>


template<typename In> char ean8_check_digit(In ean_begin, In ean_end);
```

Description

Compute the check digit of the European Article Numbering of size 8 (EAN-8) provided.

Function template Is_ean13

boost::checks::Is_ean13

Synopsis

```
// In header: <boost/checks/ean.hpp>


template<typename In, typename Prefix>
  bool Is_ean13(In ean_begin, In ean_end, Prefix ean_prefix_begin = null,
                Prefix ean_prefix_end = null);
```

Description

Check the validity of the European Article Numbering of size 13 (EAN-13) provided.

Parameters:	`ean_begin`	Represents the beginning of the EAN sequence to check.
	`ean_end`	Represents one off the end of the EAN sequence to check.
	`ean_prefix_begin`	Represents the beginning of the prefixes sequence to allow. Default : null, all the prefixes are allowed.
	`ean_prefix_end`	Represents the ending of the prefixes sequence to allow. Default : null, all the prefixes are allowed.
Requires:	ean_begin and ean_end are valid initialized iterators. If ean_prefix_begin and ean_prefix_end are passed as arguments, they must be valid initialized iterators.	
Postconditions:	ean_begin, ean_end, ean_prefix_begin, and ean_prefix_end are unchanged.	
Returns:	True if the EAN delimited by ean_begin and ean_end is a valid EAN of size 13 with a prefix	

Function template Is_ean8

boost::checks::Is_ean8

Synopsis

```
// In header: <boost/checks/ean.hpp>


template<typename In> bool Is_ean8(In ean_begin, In ean_end);
```

Description

Check the validity of the European Article Numbering of size 8 (EAN-8) provided.

Header <boost/checks/EANcheck.cpp>

```

bool EAN8check(std::string s);
char EAN8compute(std::string s);
bool EANcheck(std::string s);
char EANcompute(std::string s);
```

Header <boost/checks/EANcheck.hpp>

```

bool EANcheck(std::string s);
char EANcompute(std::string s);
```

Header <boost/checks/fletcher.hpp>

Header <boost/checks/iban.hpp>

Header <boost/checks/IBMCheck.hpp>

```

bool IBMcheck(string s);
char IBMcompute(string s);
```

Header <boost/checks/isan.hpp>

Header <boost/checks/isbn.hpp>

```
namespace boost {
  namespace checks {
    template<typename In> bool Is_isbn10(In, In);
    template<typename In> bool Is_isbn13(In, In);
    template<typename In> char isbn10_check_digit(In, In);
    template<typename In> char isbn13_check_digit(In, In);
  }
}
```

Function template Is_isbn10

boost::checks::Is_isbn10

Synopsis

```
// In header: <boost/checks/isbn.hpp>


template<typename In> bool Is_isbn10(In isbn_begin, In isbn_end);
```

Description

Check the validity of the International Standard Book Number (ISBN) of size 10.

Parameters:	`isbn_begin`	Represents the beginning of the ISBN sequence to check.
	`isbn_end`	Represents one off the end of the ISBN sequence to check.
Requires:	isbn_begin and isbn_end are valid initialized iterators.The length of the sequence should be at least of size 10 and the sequence should contains only dash(es) and digits.	
Postconditions:	isbn_begin and isbn_end are inchanged.	
Returns:	true if the sequence is a valid ISBN of size 10, otherwise false.	

Function template Is_isbn13

boost::checks::Is_isbn13

Synopsis

```
// In header: <boost/checks/isbn.hpp>


template<typename In> bool Is_isbn13(In isbn_begin, In isbn_end);
```

Description

Check the validity of the International Standard Book Number (ISBN) of size 13. (It is a ISBN encapsulated into a EAN).

Function template isbn10_check_digit

boost::checks::isbn10_check_digit

Synopsis

```
// In header: <boost/checks/isbn.hpp>


template<typename In> char isbn10_check_digit(In isbn_begin, In isbn_end);
```

Description

Compute the check digit of the International Standard Book Number (ISBN) of size 10.

Parameters:	`isbn_begin`	Represents the beginning of the ISBN sequence to check.
	`isbn_end`	Represents one off the end of the ISBN sequence to check.
Requires:	isbn_begin and isbn_end are valid initialized iterators. The length of the sequence should be of size 9 and the sequence should contains only digits and dashes.	
Postconditions:	isbn_begin and isbn_end are inchanged.	
Returns:	The check digit of the ISBN of size 9 provided, which can be between '0' and '9' or 'X'. Otherwise -1 is returned if the ISBN of size 9 provided is not correct.	

Function template isbn13_check_digit

boost::checks::isbn13_check_digit

Synopsis

```
// In header: <boost/checks/isbn.hpp>


template<typename In> char isbn13_check_digit(In isbn_begin, In isbn_end);
```

Description

Compute the check digit of the International Standard Book Number (ISBN) of size 13. (It is a ISBN encapulsed into a EAN).

Header <boost/checks/ISBN_PAB.hpp>

Obselete versio of ISBN check and compute.

```

bool ISBNcheck(string s);
bool ISBNcheck(std::string s);
char ISBNcompute(std::string s);
char ISBNcompute(string s);
```

Header <boost/checks/isbn_Vasconcelos.hpp>

```
namespace boost {
  bool is_isbn(const std::string &);
  char isbn_check_digit(const std::string &);
}
```

Function is_isbn

boost::is_isbn

Synopsis

```
// In header: <boost/checks/isbn_Vasconcelos.hpp>


bool is_isbn(const std::string & isbn);
```

Description

This function checks if a `isbn' is a valid ISBN

Function isbn_check_digit

boost::isbn_check_digit

Synopsis

```
// In header: <boost/checks/isbn_Vasconcelos.hpp>


char isbn_check_digit(const std::string & isbn);
```

Description

This function computes the check digit for a given ISBN in `isbn'

Header <boost/checks/ISSN_PAB.hpp>

```

bool ISSNcheck(string s);
char ISSNcompute(string s);
```

Header <boost/checks/luhn.hpp>

Header <boost/checks/mastercard.hpp>

Header <boost/checks/modulus.hpp>

Header <boost/checks/upc.hpp>

```
namespace boost {
  namespace checks {
    template<typename In> bool Is_upca(In, In);
    template<typename In> char upca_check_digit(In, In);
  }
}
```

Function template Is_upca

boost::checks::Is_upca

Synopsis

```
// In header: <boost/checks/upc.hpp>


template<typename In> bool Is_upca(In upc_begin, In upc_end);
```

Description

Check the validity of the Universal Product Code category A (UPC-A) provided.

Parameters:	`upc_begin` Represents the beginning of the UPC sequence to check.
Requires:	upc_begin and upc_end are valid initialized iterators. The length of the sequence should be of size 12 and the sequence should contains only digits.
Postconditions:	upc_begin and upc_end are unchanged.
Returns:	true if the UPC sequence is valid which it means that the check digit is equals to the last character. Otherwise false.

Function template upca_check_digit

boost::checks::upca_check_digit

Synopsis

```
// In header: <boost/checks/upc.hpp>


template<typename In> char upca_check_digit(In upc_begin, In upc_end);
```

Description

Compute the check digit of the Universal Product Code category A (UPC-A) provided /tparam Iterator which represents the beginning or the ending of a sequence of character. /param [in] upc_begin Represents the beginning of the UPC sequence to check. /param [in] upc_end Represents one off the end of the UPC sequence to check. /pre upc_begin and upc_end are valid initialized iterators. The length of the sequence should be of size 11 and the sequence should contains only digits. /post upc_begin and upc_end are unchanged. /result 0 if the check digit couldn't be calculated (Exemple : wrong size, ...). Otherwise, the check digit character between '0' and '9'.

Header <boost/checks/UPCcheck.cpp>

```

bool UPCcheck(std::string s);
char UPCcompute(std::string s);
```

Header <boost/checks/UPCcheck.hpp>

```

bool UPCcheck(std::string s);
char UPCcompute(std::string s);
```

Header <boost/checks/verhoeff.hpp>

Header <boost/checks/visa.hpp>

Header <boost/checks/VISACheck.hpp>

```

bool VISAcheck(string s);
char VISAcompute(string s);
```

Class Index

Typedef Index

Function Index

Macro Index

Index