

Checks

Pierre Talbot

Copyright © 2011 Pierre Talbot

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Checks	3
Preface	3
Status	4
Document Conventions	4
Tutorial	6
Starting with Checks	6
Extending the library	9
Common check algorithms	13
Type of errors	14
Alteration	14
Transposition	14
Length	14
Shift	14
Phonetic	14
Modular sum algorithms	14
Luhn algorithm	15
Modulus 10 algorithm	15
Modulus 11 algorithm	15
Verhoeff Algorithm	15
Summary of Algorithms	16
Acknowledgements	16
References	16
Rationale	17
History	17
Version Info	17
Checks Reference	18
Header <boost/checks/amex.hpp>	18
Header <boost/checks/basic_check_algorithm.hpp>	22
Header <boost/checks/basic_checks.hpp>	25
Header <boost/checks/checks_fwd.hpp>	29
Header <boost/checks/ean.hpp>	42
Header <boost/checks/isbn.hpp>	44
Header <boost/checks/iteration_sense.hpp>	47
Header <boost/checks/limits.hpp>	50
Header <boost/checks/luhn.hpp>	52
Header <boost/checks/mastercard.hpp>	55
Header <boost/checks/modulus10.hpp>	58
Header <boost/checks/modulus11.hpp>	60
Header <boost/checks/modulus97.hpp>	63
Header <boost/checks/translation_exception.hpp>	67
Header <boost/checks/upc.hpp>	67
Header <boost/checks/verhoeff.hpp>	68
Header <boost/checks/visa.hpp>	70
Header <boost/checks/weight.hpp>	73

Header <boost/checks/weighted_sum.hpp>	75
Class Index	77
Typedef Index	78
Function Index	80
Macro Index	83
Index	84

DRAFT

Checks

Preface

Checks are required on alphanumeric identifiers in numerous domains such as the distribution chain (product bar codes), banking (bank account, credit cards, fidelity cards, ...) and many others. These codes and numbers are often typed, copied or scanned by humans or machines; and both make errors. We need a way to detect errors and this is why check digits have been designed.

A check digit aims to control the validity of a alphanumeric string and catch the as many changes as possible. ([types of error](#) discuss possible alterations and the efficiency of their detection).

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

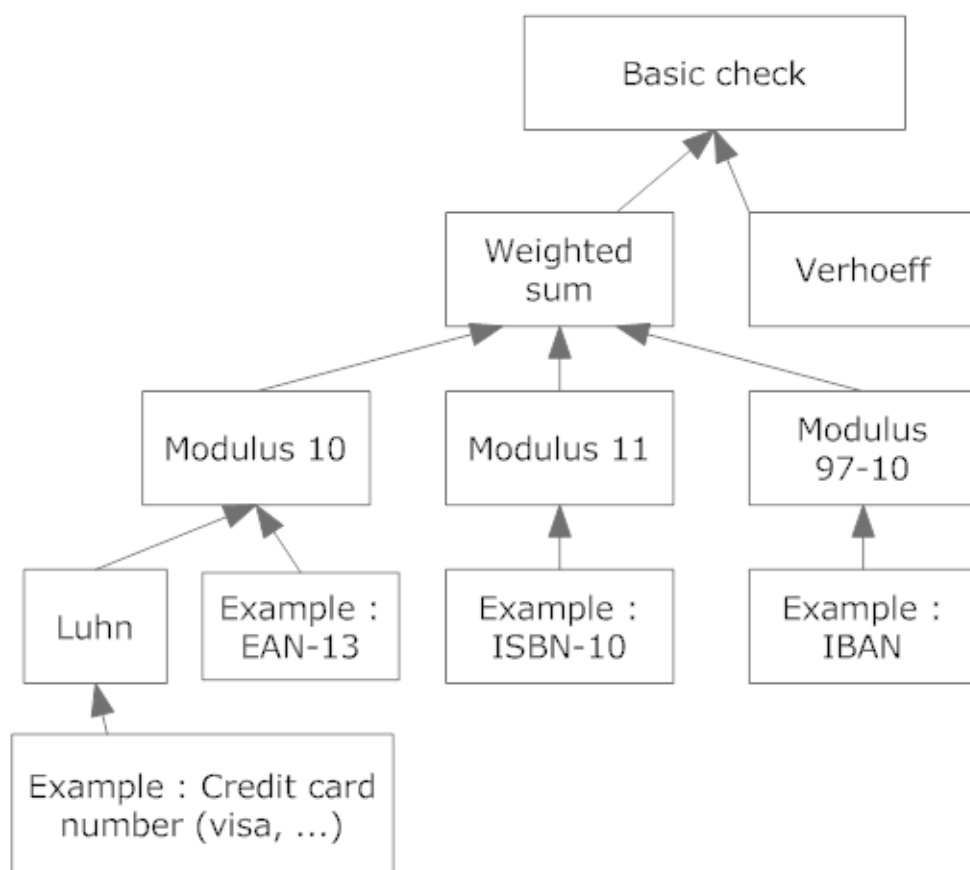
Scott McMurray has identified four fairly distinct types of check:

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.

The functions in this Boost.Checks library are primarily for the first category : catching human-entry errors (though it obviously also provides against a mis-scan or mis-transmit by a device like a bar code or card reader.)

This library supports four families of check : Modulus 10, Modulus 11, Modulus 97-10 and Verhoeff. A lot of other check systems are inherited from these families.

The following diagram shows the hierarchy used in Boost.Checks:



Numerous check algorithms are in use worldwide, and this is why this library is designed to help you to cater for less common checks, and even to create your own check systems. If you are interested, see [extending the library](#).

Status



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/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/checks/boost/> Boost Sandbox checks source code.



Note

Comments and suggestions (even bugs!) to Pierre Talbot.

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 starting point.
- **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 in color.
- 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.

Tutorial

In this section, we will quickly learn to use this library. But most important is this quote of Lao Tseu :

“Give a Man a Fish, Feed Him For a Day. Teach a Man to Fish, Feed Him For a Lifetime.”

So we'll also learn to extend this library and create your own check functions.

Starting with Checks

There are two main functions for each check system.

- to validate a sequence: `check_<number>`.
- to provides a check digit for a sequence: `compute_<number>`.

All the examples of this section are in the file [checks_examples.cpp](#).

Credit card numbers check

We will start with some credit card numbers checking.

Please first include these headers:

```
#include <boost/checks/visa.hpp>
#include <boost/checks/amex.hpp>
#include <boost/checks/mastercard.hpp>
```

Three credit card checks are implemented: [Visa credit card](#), [Mastercard credit card](#), and American Express. The following examples show us how to compute and check numbers:

```
std::string visa_credit_card_number = "4000 0807 0620 0007" ;
if( boost::checks::check_visa( visa_credit_card_number ) )
    std::cout << "The VISA credit card number : " << visa_credit_card_number << " is valid." << std::endl ;

std::string amex_credit_card_number = "3458 2531 9273 09" ;
char amex_checkdigit = boost::checks::compute_amex( amex_credit_card_number ) ;
std::cout << "The check digit of the American Express number : " << amex_credit_card_number << " is " << amex_checkdigit << "." << std::endl ;

std::string mastercard_credit_card_number = "5320 1274 8562 157" ;
mastercard_credit_card_number += boost::checks::compute_mastercard( mastercard_credit_card_number ) ;
std::cout << "This is a valid Mastercard number : " << mastercard_credit_card_number << std::endl ;
```

This provides the output:

```
The VISA credit card number : 4000 0807 0620 0007 is valid.
The check digit of the American Express number : 3458 2531 9273 09 is 4.
This is a valid Mastercard number : 5320 1274 8562 1570
```

Multi check digits

Some checks use two check digits, for example, the mod97-10 algorithm used to calculate the check digits of the [International Bank Account Number \(IBAN\)](#).

We add an extra parameter to retrieve the two check digits. The include file is:

```
#include <boost/checks/modulus97.hpp>
```

and the next example shows us how to use this function:

```
std::string mod97_10_number = "1234567890123456789" ;
std::string mod97_10_checkdigits = " " ;
boost::checks::compute_mod97_10 ( mod97_10_number , mod97_10_checkdigits.begin() ) ;
std::cout << "The number : " << mod97_10_number << " have the check digits : " << mod97_10_checkdigits << "." << std::endl ;

mod97_10_number = "85212547851652 " ;
boost::checks::compute_mod97_10 ( mod97_10_number , mod97_10_number.end() - 2 );
std::cout << "A complete mod97-10 number : " << mod97_10_number << std::endl ;
```

which provides the output:

```
The number : 1234567890123456789 have the check digits : 68.
A complete mod97-10 number : 8521254785165211
```

Catching errors

We will now see how the library reacts with simple errors. The first error is that the number of characters (size of sequence) doesn't fit the requirements. The second error shows that some number must respect pattern, here the three first digit of an ISBN-13 must be "978" or "979". An exception is thrown if any one of these errors are encountered. We will use the [International Article Number \(EAN\)](#) and [International Standard Book Number \(ISBN\)](#) headers:

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

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

Two examples of number error:

```
std::string ean13_number = "540011301748" ; // Incorrect size.
try
{
    boost::checks::check_ean13 ( ean13_number ) ;
}
catch ( std::invalid_argument e )
{
    std::cout << e.what() << std::endl ;
}

std::string isbn13_number = "977-0321227256" ; // Third digit altered.
try
{
    boost::checks::check_isbn13( isbn13_number ) ;
}
catch ( std::invalid_argument e )
{
    std::cout << e.what() << std::endl ;
}
```

The output shows us the detailed message the exception provides:

```
Too few or too much valid values in the sequence.
The third digit should be 8 or 9.
```

And with integer array

The C-arrays of integers are also supported. In the other examples, we check "number" but with an ASCII code, we can use integer value as well. The following will show us the result of the computation of two same numbers but in different format. We'll use the header:

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

And the examples:

```
std::string isbn10_number = "020163371" ; // More Effective C++: 35 New Ways to Improve Your Programs and Designs, Scott Meyers.
int isbn10_integer_number[] = {0,2,0,1,6,3,3,7,1} ;

std::cout << "ISBN10 : " << isbn10_number << ". Check digit : " << boost::checks::compute_isbn10( isbn10_number ) << std::endl ;
std::cout << "ISBN10 integer version. Check digit : " << boost::checks::compute_isbn10( isbn10_integer_number ) << std::endl ;
```

As you can see in the output, the "X" check digit is represented by its integer value (10) with the integer C-array:


```
ISBN10 : 020163371. Check digit : X
ISBN10 integer version. Check digit : 10
```

Extending the library

The re-usability of this library is an important feature. In fact, we can't code every existing check systems - there are dozens of checksums in use worldwide. This section gives examples of how to extend this library to cater for existing check systems not yet provided, and even to create your own check system.

Example with the Routing transit number

We will show how to extend this library with the [Routing transit number \(RTN\)](#). The first thing to do is to read the check digit calculation procedure. So we can notice few points:

1. It is a weighted sum and the weight sequence is: 3,7,1.
2. It is using a modulus 10.
3. The size of the RTN is 9.

We can create the [rtn.hpp](#) file.

The library supports the weighted sum and the modulus 10 algorithm, so the work will be easy. We can run through the number from right to left or left to right (sense) depending on the weight sequence. We will begin with the leftmost digit because it is more "readable" (at least for Latin language users).

We need these include files.

```
#include <boost/checks/modulus10.hpp>
#include <boost/checks/basic_checks.hpp>
```

```
#define RTN_SIZE 9
#define RTN_SIZE_WITHOUT_CHECKDIGIT 8

typedef boost::checks::weight<3,7,1> rtn_weight ;
typedef boost::checks::leftmost rtn_sense ;
```

We must put the weights and the sense together into an algorithm type:

```
typedef boost::checks::modulus10_algorithm < rtn_weight, rtn_sense, 0> rtn_check_algorithm ;
typedef boost::checks::modulus10_algorithm < rtn_weight, rtn_sense, 0> rtn_compute_algorithm ;
```

As the hard part is already done, we can build our check functions now:

```

template <typename check_range>
bool check_rtn (const check_range& check_seq)
{
    return boost::checks::check_sequence<rtn_check_algorithm, RTN_SIZE> ( check_seq ) ;
}

template <typename check_range>
typename rtn_compute_algorithm::checkdigit<check_range>::type compute_rtn (const check_range& check_seq)
{
    return boost::checks::compute_checkdigit<rtn_compute_algorithm, RTN_SIZE_WITHOUT_CHECKDIGIT> ( check_seq ) ;
}

```

And that's all!



Note

`boost::checks::compute_checkdigit` and `boost::checks::check_sequence` are both defined in [basic_checks.hpp](#)

We can code a RTN sample in the file [checks_tutorial.cpp](#):

```

std::string rtn_number = "111000025" ;
if ( check_rtn ( rtn_number ) )
    std::cout << "The Routing Transit Number: " << rtn_number << " is valid." << std::endl ;
rtn_number = "11100002";
std::cout << "The check digit of the number: " << rtn_number << " is " << compute_rtn (rtn_number) << "." << std::endl ;

```

and the output is:

```

The Routing Transit Number: 111000025 is valid.
The check digit of the number: 11100002 is 5.

```

Example with the Vehicle Identification Number (VIN)

This second example is quite more complex because the [Vehicle Identification Number \(VIN\)](#) is not a default implemented check algorithm. Like for the [Routing transit number \(RTN\)](#), we must read the documentation first, and then we can extract a few elements:

- The number contains letters that must be translated to compute or check the check digit.
- The check digit is not at the end of the number. It's at the 9th position, in the midst of the number.
- The letters Q, I, or O are not valid (presumably to avoid confusion with digits 0 and 1).
- This uses a custom modulus 11 algorithm, so the check digit range is [0..9, X]

The library already has support for modulus 11 algorithm in the header:

```
#include <boost/checks/modulus11.hpp>
```

We create the [vin.hpp](#) file. Step by step, let's now complete this file.

1. The weight sequence is : 2,3,4,5,6,7,8,9,10.

2. We run through the sequence from right to left.

We create the types associated with these two observations:

```
#include <boost/checks/modulus11.hpp>
#include <boost/checks/basic_checks.hpp>

#define VIN_SIZE 17
#define VIN_SIZE_WITHOUT_CHECKDIGIT 16
#define VIN_CHECKDIGIT_POS 8

typedef boost::checks::weight<2,3,4,5,6,7,8,9,10> vin_weight ;
typedef boost::checks::rightmost vin_sense ;
```

We will now attack the harder part of the work: we need to build the adapted structure. To create our own algorithm, first we need to declare the structure with inheritance:

```
template <unsigned int number_of_virtual_value_skipped = 0>
struct vin_algorithm : boost::checks::modulus11_algorithm<vin_weight, vin_sense, number_of_virtual_value_skipped>
```

The classic modulus 11 algorithm doesn't permit the translation of letters (only the 'x' if it's the check digit). But the VIN number uses nearly the full latin alphabet (they omitted O, Q, and I to avoid confusion with numerals 1 and 0). We choose to launch the `std::invalid_argument` exception (that has the effect of stopping the algorithm) if one of these letter is encountered. The other letters must be transformed using this table:

Table 1. Letter to digit VIN conversion table

Conversion value	1	2	3	4	5	6	7	8	9
	A (1)	B (2)	C (3)	D (4)	E (5)	F (6)	G (7)	H (8)	I (N/A)
	J (10)	K (11)	L (12)	M (13)	N (14)	O (N/A)	P (16)	Q (N/A)	R (18)
		S (19)	T (20)	U (21)	V (22)	W (23)	X (24)	Y (25)	Z (26)

We need to find an algorithm that converts a letter into its conversion value, the following function does the job:

```
X = X % 10 + X/10 + ((X > 18) ? 1 : 0) .
```

Also the check digit can only be in the range $[0..9,X]$, so we choose to launch the `std::invalid_argument` exception if another letter is read. With the check digit, and following the modulus 11 algorithm, if the check digit is equal to X, the integer value is 10. But this algorithm is different and we must subtract the check digit from 11.

Let's see the code now:

```

template <typename value>
static int translate_to_valid_value(const value &current_value, const unsigned int val_id_value_counter )
{
    int valid_value = 0;
    try
    {
        valid_value = boost::lexical_cast<int>( current_value ) ;
    }
    catch( boost::bad_lexical_cast )
    {
        // Transform the value to be between 1 and 26.
        if( current_value >= 'a' && current_value <= 'z' )
            valid_value = current_value - 'a' + 1 ;
        else if( current_value >= 'A' && current_value <= 'Z' )
            valid_value = current_value - 'A' + 1 ;
        else
            throw boost::checks::translation_exception() ;

        if ( valid_value == 9 || valid_value == 15 || valid_value == 17)
            throw std::invalid_argument( "The letter I, O and Q are not allowed." );

        if ( valid_value_counter == VIN_CHECKDIGIT_POS && number_of_virtual_value_skipped == 0)
        {
            if ( valid_value != 24 )
                throw std::invalid_argument( "The check digit should be a digit or X or x." );
            else
                valid_value = 10 ;
            valid_value = 11 - valid_value ;
        }
        else
            valid_value = valid_value % 10 + valid_value / 10 + (valid_value > 18) ;
    }
    if( valid_value > 10)
        throw boost::checks::translation_exception() ;

    return valid_value ;
}

```

The operation function is partially copied from the function `operate_on_valid_value` in the file [weighted_sum.hpp](#). We need to control the fact that the check digit is in the midst of the number. If there is a check digit into the sequence, we mustn't apply a weight, and we must avoid shift of the full weight sequence for the future iteration.

```

static void operate_on_valid_value( const int current_valid_value, const unsigned int val_id_value_counter, int &checksum )
{
    if( number_of_virtual_value_skipped == 0 && valid_value_counter == VIN_CHECKDIGIT_POS )
        checksum += current_valid_value ;
    else
    {
        unsigned int weight_position = valid_value_counter - (number_of_virtual_value_skipped == 0 && valid_value_counter > VIN_CHECKDIGIT_POS) ;
        int current_weight = vin_weight::weight_associated_with_pos( weight_position ) ;
        checksum += current_valid_value * current_weight ;
    }
}

```

Finally the calculation of the check digit is different from the classic modulus 11 algorithm, so we need to re-implement it:

```
template <typename checkdigit>
static typename checkdigit compute_checkdigit( int checksum )
{
    typedef typename boost::checks::modulus11_algorithm<vin_weight, vin_sense, number_of_virtual_value_skipped> mod11 ;
    return mod11::translate_checkdigit<checkdigit>(checksum % 11) ;
}
```

We can now write the VIN type algorithm:

```
typedef vin_algorithm <0> vin_check_algorithm ;
typedef vin_algorithm <1> vin_compute_algorithm ;
```

And write the functions:

```
template <typename check_range>
bool check_vin (const check_range& check_seq)
{
    return boost::checks::check_sequence<vin_check_algorithm, VIN_SIZE> ( check_seq ) ;
}

template <typename check_range>
typename vin_compute_algorithm::checkdigit<check_range>::type compute_vin (const check_range& check_seq)
{
    return boost::checks::compute_checkdigit<vin_compute_algorithm, VIN_SIZE_WITHOUT_CHECKDIGIT> ( check_seq ) ;
}
```



Note

This algorithm doesn't support full integer array that are not pre-computed (Example: (A) 10 -> 1 ; (M) 13 -> 4). It can be an exercise for the reader.

Some basic examples are coded in the file [checks_tutorial.cpp](#) .

```
std::string vin_number = "1M8GDM9AXKP042788";
if ( check_vin ( vin_number ) )
    std::cout << "The Vehicle Identification Number: " << vin_number << " is correct." << std::endl ;

vin_number = "1M8GDM9AKP042788" ;
std::cout << "The check digit of " << vin_number << " is " << compute_vin ( vin_number ) << std::endl ;
```

that provides the following output:

```
The Vehicle Identification Number: 1M8GDM9AXKP042788 is correct.
The check digit of 1M8GDM9AKP042788 is X
```

Common check algorithms

This section will discuss the objectives of the algorithms used in Boost.Checks. A check algorithm is firstly designed to:

1. Catch the most errors as possible that a human (or machine) can make.
2. Minimize the cost of the check digit for fast computation.

3. Minimize the size (or number) of the check digit(s).

We cannot have our cake and eat it, that's why we often choose between the size or the efficiency. Critical numbers, for example, the [International Bank Account Number \(IBAN\)](#), use two check digits.

The main difference with these algorithms (and the other checksum algorithms such as CRC or cryptographic hashes) is that we don't analyse the binary content of the number or the lexical values meaning, so "123" is equivalent to 123.

Table 2. Error catching summary

	1 Alteration	2 Alterations	Twin transpositions
Luhn	18/18 (100%)		88/90 (97.78%)
Verhoeff	18/18 (100%)		90/90 (100%)

Type of errors

This section will describe some common errors that an user or a device can make.

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 caught 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 uncaught 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 some number. The number obtained is called the *check digit*; in many codes it is appended as the last digit (or letter). This simplistic algorithm detects any *alteration* of one single digit but

doesn't detect a simple *transposition* or two (or more) digits 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 worldwide use. We could describe a number and its check digit calculation with three characteristics : length, weight and the modulus. So we could design a generic function - but we won't - it wouldn't be efficient and would be unnecessarily complicated. The next parts will present three different algorithms for which we have choose to design this implementation.



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 weight pattern used is from the rightmost digit (the check digit) doubling the value of every second digit. It applies a modulus 10 on the sum, so 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 any one digit are all caught. The alterations of more than one digit are not all caught. All **transpositions** on digits with different weight are caught, but the sequence "90" or "09" is not because:

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

The two digits have the same value if they are doubled or not. Because Luhn alternates a weight of 1 and 2, any transpositions of digits with the same weight are not caught.

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 may be useful 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 decreases by one for the second position, one again for the third, etc.

Verhoeff Algorithm

Verhoeff's check equation catches all single errors, all adjacent transpositions, over 95% of twin errors, over 94% of jump transpositions and jump twin errors, and most phonetic errors.

Summary of Algorithms

Here a summary of the different algorithms studied.

Table 3. Summary of the modular sum algorithms

Algorithm	Modulus	Weight pattern	check digit range
Luhn	10	... 2 1 2 1	0..9
Modulus 10	10	custom	0..9
Modulus 11	11	... 2 1 10 ... 4 3 2 1	0..9 + 'X'
Modulus 97	97		
Verhoeff			

Acknowledgements

The author thanks

- Paul A. Bristow who was the mentor of this Google Summer of Code project for his infinite patience and his wise advice.
- Google for funding this Google Summer of Code 2011 project.
- Joseph A Gallian for his assistance with tables of values for the Verhoeff algorithm.

References

1. [Routing transit number \(RTN\)](#)
2. [Vehicle Identification Number \(VIN\)](#)
3. [Code 39](#)
4. Verhoeff, J, Error Detecting Codes, Mathematical Centre Tract 29, The Mathematical Centre, Amsterdam 1969.
5. [Verhoeff_algorithm](#)
6. H. Peter Gumm, New class of check-digit methods for arbitrary number systems, IEEE Trans. on Information Theory, 31(1985) 102-105.
7. Paul Putter, & Neal R Wagner, Communication of ther ACM, Jan 1989, Vol 32, pages 106 - 110. Jonathan Mohr <http://www.augustana.ab.ca/~mohrj/>
8. Numerical Recipes in C++, p 904 - 905 W H Press, S A Teukolsky, W T Vetterling, B P Flannery, ISBN 0 521 75033 4.
9. SNOMED Clinical Terms (First Release Technical Reference Manual)
10. First Release (January 2002), Annex B. Check-digit computation, p 30 to 34
11. http://www.snomed.org/Spanish%20Documents/SNOMEDCT_Core_Technical%20Reference%20Manual_US.pdf uses Jonathan Mohr Javascript checkdigit.htm (copy in checkdigit.txt) based on code at <http://www.augustana.ab.ca/~mohrj/algorithms/check-digit.html>
- 12 J A Gallian, Table 3 p 514, Error detection methods, ACM Surveys, 28(3) 504-517 ISSN 0360-0300 (1996)

13. J A Gallian & S Winters, American Math Monthly, 95, (1988) Modular Arithmetic in the Marketplace, p 548 - 551.

Rationale

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

Scope of the project

- Scott McMurray has identified 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.

Function parameter

- For more flexibility, this library uses the range concept. So you can use old C-array or `std::string`,...
- If there is only one check digit in the number, this check digit is returned in the same raw type than in the range sequence.
- If there is more than one check digit, an extra parameter is required. This must be an `OutputIterator`, the function returns an iterator at one pass the end of the check digit stored into this iterator.

Use of template parameters to pass size and weights

- This feature means that much of the commonality between the various check systems can be implemented in one place and reused.
- It also makes it possible to implement other check systems (of which there are very many in use worldwide) and to devise new ones without writing new code.

Performance

- Performance is not a major objective, but all the current algorithms are implemented with a $O(n)$ complexity, where n is the number of digits or characters.

History

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

Version Info

Last edit to Quickbook file checks.qbk was at 05:33:15 PM on 2011-Sep-06.



Warning

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

Checks Reference

Header <boost/checks/amex.hpp>

This file provides tools to compute and validate an American Express credit card number.

```
AMEX_SIZE  
AMEX_SIZE_WITHOUT_CHECKDIGIT
```

```
namespace boost {  
    namespace checks {  
        template<unsigned int number_of_virtual_value_skipped = 0>  
            class amex_algorithm;  
  
        typedef amex_algorithm< 0 > amex_check_algorithm; // This is the type of the Amex algorithm for validating a check digit.  
        typedef amex_algorithm< 1 > amex_compute_algorithm; // This is the type of the Amex algorithm for computing a check digit.  
        template<typename check_range> bool check_amex(const check_range &);  
        template<typename check_range>  
            boost::checks::amex_compute_algorithm::checkdigit< check_range >::type  
            compute_amex(const check_range &);  
    }  
}
```

Class template amex_algorithm

boost::checks::amex_algorithm — This class can be used to compute or validate checksum with the Luhn algorithm, but filter following the Amex pattern.

Synopsis

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

template<unsigned int number_of_virtual_value_skipped = 0 // Help functions
// to provide same
// behavior on
// sequence with
// and without
// check digits. No
// "real" value in
// the sequence
// will be skipped.
>
class amex_algorithm :
{
public:

    // public static functions
    static checkdigit compute_checkdigit(int);
    static checkdigits_iter compute_multicheckdigit(int, checkdigits_iter);
    static void filter_valid_value_with_pos(const unsigned int,
                                           const unsigned int);
    static void operate_on_valid_value(const int, const unsigned int, int &);
    static int translate_to_valid_value(const value &, const unsigned int);
    static bool validate_checksum(int);
};
```

Description

amex_algorithm public static functions

1. `static checkdigit compute_checkdigit(int checksum);`

Compute the check digit with a simple modulus 10.

Parameters: checksum is the checksum used to extract the check digit.
Returns: The modulus 10 check digit of checksum.
Throws: [boost::checks::translation_exception](#) if the check digit cannot be translated into the checkdigit type.

2. `static checkdigits_iter
compute_multicheckdigit(int checksum, checkdigits_iter checkdigits);`

Compute the check digit(s) of a sequence.

This function should be overloaded if you want your algorithm to compute more than one check digit (through it works for just one check digit too).

Parameters: checkdigits is the iterator with which the check digit(s) will be written.
 checksum is the checksum used to extract the check digit(s).
Requires: checkdigits must be a valid initialized iterator.
Returns: checkdigits.

3. `static void filter_valid_value_with_pos(const unsigned int current_valid_value,
 const unsigned int current_value_position);`

Verify that a number matches the Amex pattern.

This function use the macro AMEX_SIZE to find the real position from left to right.

Parameters:	<code>current_valid_value</code>	is the current valid value analysed.
	<code>current_value_position</code>	is the number of valid value already counted (the current value is not included). This is also the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).
Throws:	<code>std::invalid_argument</code> if the first character is not equal to 3 or the second is not equal to 4 or 7. The exception contains a descriptive message of what was expected.	

```
4. static void operate_on_valid_value(const int current_valid_value,
                                     const unsigned int valid_value_counter,
                                     int & checksum);
```

Compute the Luhn algorithm operation on the checksum.

This function become obsolete if you don't use `luhn_weight`. It is using operator "<<" to make internal multiplication.

Parameters:	checksum	is the current checksum.
	current_valid_value	is the current valid value analysed.
	valid_value_counter	is the number of valid value already counted (the current value is not included).
		This is also the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).
Postconditions:	checksum is equal to the new computed checksum.	

[illegible]

translate a value of the sequence into an integer valid value.

Parameters:	<code>current_value</code> is the current value analysed in the sequence that must be translated. <code>valid_value_counter</code> is the number of valid value(s) already counted (the current value is not included). This is also the position (beyond the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).
Returns:	the translation of the current value in the range [0..9].
Throws:	boost::checks::translation_exception is thrown if the translation of <code>current_value</code> failed. This will automatically throw if the value is not a digit ($0 \leq i < 11$).

```
6. static bool validate_checksum(int checksum);
```

Validate a checksum with a simple modulus 10.

Parameters: checksum is the checksum to validate.
Returns: true if the checksum is correct, false otherwise.

Function template check_amex

`boost::checks::check_amex` — Validate a sequence according to the `amex_check_algorithm` type.

Synopsis

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

template<typename check_range> bool check_amex(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
 Requires: `check_seq` is a valid range.
 Returns: `true` if the check digit is correct, `false` otherwise.
 Throws: `std::invalid_argument` if `check_seq` doesn't contain exactly `AMEX_SIZE` digits. if the two first digits (from the leftmost) don't match the Amex pattern.

Function template `compute_amex`

`boost::checks::compute_amex` — Calculate the check digit of a sequence according to the `amex_compute_algorithm` type.

Synopsis

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

template<typename check_range>
boost::checks::amex_compute_algorithm::checkdigit< check_range >::type
compute_amex(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
 Requires: `check_seq` is a valid range.
 Returns: The check digit. The check digit is in the range `[0..9]`.
 Throws: `std::invalid_argument` if `check_seq` doesn't contain exactly `AMEX_SIZE_WITHOUT_CHECKDIGIT` digits. if the two first digits (from the leftmost) don't match the amex pattern. if the check digit cannot be translated into the `checkdigit` type.

Macro `AMEX_SIZE`

`AMEX_SIZE` — This macro defines the size of a American Express card number (15).

Synopsis

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

AMEX_SIZE
```

Macro `AMEX_SIZE_WITHOUT_CHECKDIGIT`

`AMEX_SIZE_WITHOUT_CHECKDIGIT` — This macro defines the size of a American Express card number without its check digit (14).

Synopsis

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

AMEX_SIZE_WITHOUT_CHECKDIGIT
```

Header <boost/checks/basic_check_algorithm.hpp>

This file provides a class that should be used as an "interface" because most of the static functions should be re-implemented using inheritance.

The class implements static functions that are common to many algorithms.

```
namespace boost {  
    namespace checks {  
        template<typename iteration_sense,  
                unsigned int number_of_virtual_value_skipped = 0>  
            class basic_check_algorithm;  
    }  
}
```

Class template basic_check_algorithm

boost::checks::basic_check_algorithm — The main check algorithm class that provides every static function that can be overloaded. Most of the functions must be re-implemented to have the desired behavior.

Synopsis

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

template<typename iteration_sense,    // must meet the iteration_sense concept
        // requirements.
        unsigned int number_of_virtual_value_skipped = 0 // Helper functions
                                                    // to provide the
                                                    // same behavior on
                                                    // sequence with
                                                    // and without
                                                    // checkdigits. No
                                                    // "real" value in
                                                    // the sequence
                                                    // will be skipped.
>
class basic_check_algorithm {
public:
    // types
    typedef iteration_sense iteration_sense; // This is the sense or direction of the iteration ↴
    (begins with the right or the leftmost value).

    // member classes/structs/unions

    // Template rebinding class used to define the type of the check digit(s) of
    // check_range.
    template<typename check_range // The type of the sequence to check.
    >
    class checkdigit {
    public:
        // types
        typedef boost::range_value< check_range >::type type;
    };

    // public static functions
    template<typename checkdigit> static checkdigit compute_checkdigit(int);
    template<typename checkdigits_iter>
        static checkdigits_iter compute_multicheckdigit(int, checkdigits_iter);
    static void filter_valid_value_with_pos(const unsigned int,
                                           const unsigned int);
    static void operate_on_valid_value(const int, const unsigned int, int &);
    template<typename value>
        static int translate_to_valid_value(const value &, const unsigned int);
    static bool validate_checksum(int);
};
```

Description

basic_check_algorithm public static functions

1.

```
template<typename checkdigit>
    static checkdigit compute_checkdigit(int checksum);
```

Compute the check digit of a sequence.

This function should be overloaded if you want to compute the check digit of a sequence.

Parameters: checksum is the checksum used to extract the check digit.

Requires: The type checkdigit must provides the default initialisation feature.

Returns: default initialized value of checkdigit.

2.

```
template<typename checkdigits_iter>
static checkdigits_iter
compute_multicheckdigit(int checksum, checkdigits_iter checkdigits);
```

Compute the check digit(s) of a sequence.

This function should be overloaded if you want your algorithm to compute more than one check digit (through it works for just one check digit too).

Parameters: `checkdigits` is the iterator with which the check digit(s) will be written.

`checksum` is the checksum used to extract the check digit(s).

Requires: `checkdigits` must be a valid initialized iterator.

Returns: `checkdigits`.

3.

```
static void filter_valid_value_with_pos(const unsigned int current_valid_value,
                                       const unsigned int current_value_position);
```

Filtering of a valid value according to its position.

This function should be overloaded if you want to filter the values with their positions.

Parameters: `current_valid_value` is the current valid value analysed.

`current_value_position` is the position (above the valid values) of the current value analysed
($0 \leq \text{valid_value_counter} < n$).

Postconditions: Do nothing.

4.

```
static void operate_on_valid_value(const int current_valid_value,
                                   const unsigned int valid_value_counter,
                                   int & checksum);
```

Compute an operation on the checksum with the current valid value.

This function should be overloaded if you want to calculate the checksum of a sequence.

Parameters: `checksum` is the current checksum.

`current_valid_value` is the current valid value analysed.

`valid_value_counter` is the number of valid value(s) already counted (the current value is not included).

 This is also the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).

Postconditions: Do nothing. The checksum is unchanged.

5.

```
template<typename value>
static int translate_to_valid_value(const value & current_value,
                                   const unsigned int valid_value_counter);
```

translate a value of the sequence into an integer valid value.

Parameters: `current_value` is the current value analysed in the sequence that must be translated.

`valid_value_counter` is the number of valid value(s) already counted (the current value is not included).

 This is also the position (beyond the valid values) of the current value analysed
($0 \leq \text{valid_value_counter} < n$).

Returns: the translation of the current value in the range [0..9].

Throws: `boost::checks::translation_exception` is thrown if the translation of `current_value` failed.

 This will automatically throw if the value is not a digit ($0 \leq i < 11$).

6.

```
static bool validate_checksum(int checksum);
```


Validate the checksum.

This function should be overloaded if you want to check a sequence.

Parameters: checksum is the checksum to validate.

Returns: true always (unless overloaded to check a sequence).

Class template checkdigit

boost::checks::basic_check_algorithm::checkdigit — Template rebinding class used to define the type of the check digit(s) of check_range.

Synopsis

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

// Template rebinding class used to define the type of the check digit(s) of
// check_range.
template<typename check_range // The type of the sequence to check.
>
class checkdigit {
public:
    // types
    typedef boost::range_value< check_range >::type type;
};
```

Description

This function should be overloaded if you want to change the type of the check digit.

Header <boost/checks/basic_checks.hpp>

This file provides a set of basic functions used to compute and validate check digit(s) and checksum.

```

namespace boost {
    namespace checks {
        template<typename algorithm, typename check_range>
            bool check_sequence(const check_range &);
        template<typename algorithm, size_t size_expected, typename check_range>
            bool check_sequence(const check_range &);
        template<typename algorithm, typename check_range>
            algorithm::checkdigit< check_range >::type
            compute_checkdigit(const check_range &);
        template<typename algorithm, size_t size_expected, typename check_range>
            algorithm::checkdigit< check_range >::type
            compute_checkdigit(const check_range &);
        template<typename algorithm, typename size_contract, typename iterator>
            int compute_checksum(iterator, iterator);
        template<typename algorithm, typename size_contract, typename check_range>
            int compute_checksum(const check_range &);
        template<typename algorithm, typename check_range,
            typename checkdigit_iterator>
            checkdigit_iterator
            compute_multicheckdigit(const check_range &, checkdigit_iterator);
        template<typename algorithm, size_t size_expected, typename check_range,
            typename checkdigit_iterator>
            checkdigit_iterator
            compute_multicheckdigit(const check_range &, checkdigit_iterator);
    }
}

```

Function template check_sequence

boost::checks::check_sequence — Validate a sequence according to algorithm.

Synopsis

```

// In header: <boost/checks/basic_checks.hpp>

template<typename algorithm, typename check_range>
    bool check_sequence(const check_range & check_seq);

```

Description

Parameters: check_seq is the sequence of value to check.
 Requires: check_seq is a valid range.
 Returns: true if the checkdigit is correct, false otherwise.
 Throws: std::invalid_argument if check_seq contains no valid value.

Function template check_sequence

boost::checks::check_sequence — Validate a sequence according to algorithm.

Synopsis

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

template<typename algorithm, size_t size_expected, typename check_range>
bool check_sequence(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
`size_expected > 0` (enforced by static assert).
Returns: `true` if the checkdigit is correct, `false` otherwise.
Throws: `std::invalid_argument` if `check_seq` doesn't contain `size_expected` valid values.

Function template `compute_checkdigit`

`boost::checks::compute_checkdigit` — Calculate the check digit of a sequence according to algorithm.

Synopsis

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

template<typename algorithm, typename check_range>
algorithm::checkdigit< check_range >::type
compute_checkdigit(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: The check digit of the type of a value in `check_seq`.
Throws: `std::invalid_argument` if `check_seq` contains no valid value.

Function template `compute_checkdigit`

`boost::checks::compute_checkdigit` — Calculate the check digit of a sequence according to algorithm.

Synopsis

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

template<typename algorithm, size_t size_expected, typename check_range>
algorithm::checkdigit< check_range >::type
compute_checkdigit(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
`size_expected > 0` (enforced by static assert).

Returns: The check digit of the type of a value in `check_seq`.
Throws: `std::invalid_argument` if `check_seq` doesn't contain `size_expected` valid values.

Function template `compute_checksum`

`boost::checks::compute_checksum` — Run through a sequence and calculate the checksum with the algorithm policy class.

Synopsis

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

template<typename algorithm, typename size_contract, typename iterator>
int compute_checksum(iterator seq_begin, iterator seq_end);
```

Description

Parameters: `seq_begin` Beginning of the sequence.
`seq_end` Ending of the sequence.
Requires: `seq_begin` and `seq_end` are valid iterators.
Returns: The checksum of the sequence calculated with `algorithm`.
Throws: `size_contract::exception_size_failure` If the terms of the contract are not respected.

Function template `compute_checksum`

`boost::checks::compute_checksum` — Create iterators according to the `algorithm::iterator` policy. And call the iterator overload version of `compute_checksum`.

Synopsis

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

template<typename algorithm, typename size_contract, typename check_range>
int compute_checksum(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: The checksum of the sequence calculated with `algorithm`.
Throws: `size_contract::exception_size_failure` If the terms of the contract are not respected.

Function template `compute_multicheckdigit`

`boost::checks::compute_multicheckdigit` — Calculate the checkdigits of a sequence according to `algorithm`.

Synopsis

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

template<typename algorithm, typename check_range,
        typename checkdigit_iterator>
checkdigit_iterator
compute_multicheckdigit(const check_range & check_seq,
                       checkdigit_iterator checkdigits);
```

Description

Parameters: `check_seq` is the sequence of value to check.
`checkdigits` is the output iterator in which the check digits will be written.

Requires: `check_seq` is a valid range.
`checkdigits` is a valid initialized iterator and have enough reserved place to store the check digits.

Returns: An iterator initialized at one pass the end of `checkdigits`.

Throws: `std::invalid_argument` if `check_seq` contains no valid value.

Function template `compute_multicheckdigit`

`boost::checks::compute_multicheckdigit` — Calculate the checkdigits of a sequence according to algorithm.

Synopsis

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

template<typename algorithm, size_t size_expected, typename check_range,
        typename checkdigit_iterator>
checkdigit_iterator
compute_multicheckdigit(const check_range & check_seq,
                       checkdigit_iterator checkdigits);
```

Description

Parameters: `check_seq` is the sequence of value to check.
`checkdigits` is the output iterator in which the check digits will be written.

Requires: `check_seq` is a valid range.
`checkdigits` is a valid initialized iterator and have enough reserved place to store the check digits.
`size_expected > 0` (enforced by static assert).

Returns: An iterator initialized at one pass the end of `checkdigits`.

Throws: `std::invalid_argument` if `check_seq` doesn't contain `size_expected` valid values.

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 check_range> bool check_ean13(const check_range &);
        template<typename check_range> bool check_ean8(const check_range &);
        template<typename check_range> bool check_isbn10(const check_range &);
        template<typename check_range> bool check_isbn13(const check_range &);
        template<size_t size_expected, typename check_range>
            bool check_luhn(const check_range &);
        template<typename check_range> bool check_luhn(const check_range &);
        template<typename check_range> bool check_mastercard(const check_range &);
        template<size_t size_expected, typename check_range>
            bool check_mod97_10(const check_range &);
        template<typename check_range> bool check_mod97_10(const check_range &);
        template<size_t size_expected, typename check_range>
            bool check_modulus11(const check_range &);
        template<typename check_range> bool check_modulus11(const check_range &);
        template<typename check_range> bool check_upca(const check_range &);
        template<size_t size_expected, typename check_range>
            bool check_verhoeff(const check_range &);
        template<typename check_range> bool check_verhoeff(const check_range &);
        template<typename check_range> bool check_visa(const check_range &);
        template<typename check_range>
            boost::checks::ean_compute_algorithm::checkdigit< check_range >::type
            compute_ean13(const check_range &);
        template<typename check_range>
            boost::checks::ean_compute_algorithm::checkdigit< check_range >::type
            compute_ean8(const check_range &);
        template<typename check_range>
            boost::checks::mod11_compute_algorithm::checkdigit< check_range >::type
            compute_isbn10(const check_range &);
        template<typename check_range>
            boost::checks::isbn13_compute_algorithm::checkdigit< check_range >::type
            compute_isbn13(const check_range &);
        template<size_t size_expected, typename check_range>
            boost::checks::luhn_compute_algorithm::checkdigit< check_range >::type
            compute_luhn(const check_range &);
        template<typename check_range>
            boost::checks::luhn_compute_algorithm::checkdigit< check_range >::type
            compute_luhn(const check_range &);
        template<typename check_range>
            boost::checks::mastercard_compute_algorithm::checkdigit< check_range >::type
            compute_mastercard(const check_range &);
        template<size_t size_expected, typename check_range,
            typename checkdigits_iter>
            checkdigits_iter compute_mod97_10(const check_range &, checkdigits_iter);
        template<typename check_range, typename checkdigits_iter>
            checkdigits_iter compute_mod97_10(const check_range &, checkdigits_iter);
        template<size_t size_expected, typename check_range>
            boost::checks::mod11_compute_algorithm::checkdigit< check_range >::type
            compute_modulus11(const check_range &);
        template<typename check_range>
            boost::checks::mod11_compute_algorithm::checkdigit< check_range >::type
            compute_modulus11(const check_range &);
        template<typename check_range>
            boost::checks::upc_compute_algorithm::checkdigit< check_range >::type
            compute_upca(const check_range &);
        template<size_t size_expected, typename check_range>
            boost::checks::verhoeff_compute_algorithm::checkdigit< check_range >::type
            compute_verhoeff(const check_range &);
        template<typename check_range>
            boost::checks::verhoeff_compute_algorithm::checkdigit< check_range >::type
    }
}

```

```
compute_verhoeff(const check_range &);  
template<typename check_range>  
boost::checks::visa_compute_algorithm::checkdigit< check_range >::type  
compute_visa(const check_range &);  
}  
}
```

Function template check_ean13

boost::checks::check_ean13 — Validate a sequence according to the ean_check_algorithm type.

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>  
  
template<typename check_range> bool check_ean13(const check_range & check_seq);
```

Description

Parameters: check_seq is the sequence of value to check.
Requires: check_seq is a valid range.
Returns: true if the check digit is correct, false otherwise.
Throws: std::invalid_argument if check_seq doesn't contain exactly EAN13_SIZE digits.

Function template check_ean8

boost::checks::check_ean8 — Validate a sequence according to the ean_check_algorithm type.

Synopsis

```
// In header: <boost/checks/checks_fwd.hpp>  
  
template<typename check_range> bool check_ean8(const check_range & check_seq);
```

Description

Parameters: check_seq is the sequence of value to check.
Requires: check_seq is a valid range.
Returns: true if the check digit is correct, false otherwise.
Throws: std::invalid_argument if check_seq doesn't contain exactly EAN8_SIZE digits.

Function template check_isbn10

boost::checks::check_isbn10 — Validate a sequence according to the mod11_check_algorithm type.

Synopsis

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

template<typename check_range>
    bool check_isbn10(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: `true` if the check digit is correct, `false` otherwise.
Throws: `std::invalid_argument` if `check_seq` doesn't contain exactly ISBN10_SIZE digits.

Function template `check_isbn13`

`boost::checks::check_isbn13` — Validate a sequence according to the `isbn13_check_algorithm` type.

Synopsis

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

template<typename check_range>
    bool check_isbn13(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: `true` if the check digit is correct, `false` otherwise.
Throws: `std::invalid_argument` if `check_seq` doesn't contain exactly EAN13_SIZE digits.

Function template `check_luhn`

`boost::checks::check_luhn` — Validate a sequence according to the `luhn_check_algorithm` type.

Synopsis

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

template<size_t size_expected, typename check_range>
    bool check_luhn(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
`size_expected > 0` (enforced by static assert).
Returns: `true` if the check digit is correct, `false` otherwise.
Throws: `std::invalid_argument` if `check_seq` doesn't contain `size_expected` valid values.

Function template check_luhn

boost::checks::check_luhn — Validate a sequence according to the luhn_check_algorithm type.

Synopsis

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

template<typename check_range> bool check_luhn(const check_range & check_seq);
```

Description

Parameters: check_seq is the sequence of value to check.
Requires: check_seq is a valid range.
Returns: true if the check digit is correct, false otherwise.
Throws: std::invalid_argument if check_seq contains no valid value.

Function template check_mastercard

boost::checks::check_mastercard — Validate a sequence according to the mastercard_check_algorithm type.

Synopsis

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

template<typename check_range>
bool check_mastercard(const check_range & check_seq);
```

Description

Parameters: check_seq is the sequence of value to check.
Requires: check_seq is a valid range.
Returns: True if the check digit is correct, false otherwise.
Throws: std::invalid_argument if check_seq doesn't contain exactly MASTERCARD_SIZE digits. if the two first digits (from the leftmost) don't match the Mastercard pattern.

Function template check_mod97_10

boost::checks::check_mod97_10 — Validate a sequence according to the mod97_10_check_algorithm type.

Synopsis

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

template<size_t size_expected, typename check_range>
bool check_mod97_10(const check_range & check_seq);
```

Description

Parameters: check_seq is the sequence of value to check.
Requires: check_seq is a valid range.

size_expected > 0 (enforced by static assert).
Returns: True if the two check digits are correct, false otherwise.
Throws: std::invalid_argument if check_seq doesn't contain size_expected valid values.

Function template check_mod97_10

boost::checks::check_mod97_10 — Validate a sequence according to the mod97_10_check_algorithm type.

Synopsis

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

template<typename check_range>
bool check_mod97_10(const check_range & check_seq);
```

Description

Parameters: check_seq is the sequence of value to check.
Requires: check_seq is a valid range.
Returns: true if the two check digits are correct, false otherwise.
Throws: std::invalid_argument if check_seq contains no valid value.

Function template check_modulus11

boost::checks::check_modulus11 — Validate a sequence according to the mod11_check_algorithm type.

Synopsis

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

template<size_t size_expected, typename check_range>
bool check_modulus11(const check_range & check_seq);
```

Description

Parameters: check_seq is the sequence of value to check.
Requires: check_seq is a valid range.
size_expected > 0 (enforced by static assert).
Returns: true if the check digit is correct, false otherwise.
Throws: std::invalid_argument if check_seq doesn't contain size_expected valid values.

Function template check_modulus11

boost::checks::check_modulus11 — Validate a sequence according to the mod11_check_algorithm type.

Synopsis

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

template<typename check_range>
    bool check_modulus11(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: `true` if the check digit is correct, `false` otherwise.
Throws: `std::invalid_argument` if `check_seq` contains no valid value.

Function template `check_upca`

`boost::checks::check_upca` — Validate a sequence according to the `upc_check_algorithm` type.

Synopsis

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

template<typename check_range> bool check_upca(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: `true` if the check digit is correct, `false` otherwise.
Throws: `std::invalid_argument` if `check_seq` doesn't contain exactly `UPCA_SIZE` digits.

Function template `check_verhoeff`

`boost::checks::check_verhoeff` — Validate a sequence according to the `verhoeff_check_algorithm` type.

Synopsis

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

template<size_t size_expected, typename check_range>
    bool check_verhoeff(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
`size_expected > 0` (enforced by static assert).
Returns: `true` if the check digit is correct, `false` otherwise.
Throws: `std::invalid_argument` if `check_seq` doesn't contain `size_expected` valid values.

Function template check_verhoeff

boost::checks::check_verhoeff — Validate a sequence according to the verhoeff_check_algorithm type.

Synopsis

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

template<typename check_range>
    bool check_verhoeff(const check_range & check_seq);
```

Description

Parameters: check_seq is the sequence of value to check.
Requires: check_seq is a valid range.
Returns: true if the check digit is correct, false otherwise.
Throws: std::invalid_argument if check_seq contains no valid value.

Function template check_visa

boost::checks::check_visa — Validate a sequence according to the visa_check_algorithm type.

Synopsis

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

template<typename check_range> bool check_visa(const check_range & check_seq);
```

Description

Parameters: check_seq is the sequence of value to check.
Requires: check_seq is a valid range.
Returns: true if the check digit is correct, false otherwise.
Throws: std::invalid_argument if check_seq doesn't contain exactly VISA_SIZE digits. if the first digit (from the leftmost) doesn't match the Visa pattern.

Function template compute_ean13

boost::checks::compute_ean13 — Calculate the check digit of a sequence according to the ean_compute_algorithm type.

Synopsis

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

template<typename check_range>
    boost::checks::ean_compute_algorithm::checkdigit< check_range >::type
    compute_ean13(const check_range & check_seq);
```

Description

Parameters: check_seq is the sequence of value to check.

Requires: `check_seq` is a valid range.
Returns: The check digit. The check digit is in the range [0..9].
Throws: `std::invalid_argument` if `check_seq` doesn't contain exactly `EAN13_SIZE_WITHOUT_CHECKDIGIT` digits. if the check digit cannot be translated into the checkdigit type.

Function template `compute_ean8`

`boost::checks::compute_ean8` — Calculate the check digit of a sequence according to the `ean_compute_algorithm` type.

Synopsis

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

template<typename check_range>
    boost::checks::ean_compute_algorithm::checkdigit< check_range >::type
    compute_ean8(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: The check digit. The check digit is in the range [0..9].
Throws: `std::invalid_argument` if `check_seq` doesn't contain exactly `EAN8_SIZE_WITHOUT_CHECKDIGIT` digits. if the check digit cannot be translated into the checkdigit type.

Function template `compute_isbn10`

`boost::checks::compute_isbn10` — Calculate the check digit of a sequence according to the `mod11_compute_algorithm` type.

Synopsis

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

template<typename check_range>
    boost::checks::mod11_compute_algorithm::checkdigit< check_range >::type
    compute_isbn10(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: The check digit. The check digit is in the range [0..9,X].
Throws: `std::invalid_argument` if `check_seq` doesn't contain exactly `ISBN10_SIZE_WITHOUT_CHECKDIGIT` digits. if the check digit cannot be translated into the checkdigit type.

Function template `compute_isbn13`

`boost::checks::compute_isbn13` — Calculate the check digit of a sequence according to the `isbn13_compute_algorithm` type.

Synopsis

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

template<typename check_range>
    boost::checks::isbn13_compute_algorithm::checkdigit< check_range >::type
    compute_isbn13(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: The check digit. The check digit is in the range [0..9].
Throws: `std::invalid_argument` if `check_seq` doesn't contain exactly `EAN13_SIZE_WITHOUT_CHECKDIGIT` digits. if the check digit cannot be translated into the `checkdigit` type.

Function template `compute_luhn`

`boost::checks::compute_luhn` — Calculate the check digit of a sequence according to the `luhn_compute_algorithm` type.

Synopsis

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

template<size_t size_expected, typename check_range>
    boost::checks::luhn_compute_algorithm::checkdigit< check_range >::type
    compute_luhn(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
`size_expected > 0` (enforced by static assert).
Returns: The check digit. The check digit is in the range [0..9].
Throws: `std::invalid_argument` if `check_seq` doesn't contain `size_expected` valid values. if the check digit cannot be translated into the `checkdigit` type.

Function template `compute_luhn`

`boost::checks::compute_luhn` — Calculate the check digit of a sequence according to the `luhn_compute_algorithm` type.

Synopsis

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

template<typename check_range>
    boost::checks::luhn_compute_algorithm::checkdigit< check_range >::type
    compute_luhn(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
 Requires: `check_seq` is a valid range.
 Returns: The check digit. The check digit is in the range [0..9].
 Throws: `std::invalid_argument` if `check_seq` contains no valid value. if the check digit cannot be translated into the `checkdigit` type.

Function template `compute_mastercard`

`boost::checks::compute_mastercard` — Calculate the check digit of a sequence according to the `mastercard_compute_algorithm` type.

Synopsis

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

template<typename check_range>
    boost::checks::mastercard_compute_algorithm::checkdigit< check_range >::type
    compute_mastercard(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
 Requires: `check_seq` is a valid range.
 Returns: The check digit. The check digit is in the range [0..9].
 Throws: `std::invalid_argument` if `check_seq` doesn't contain exactly `MASTERCARD_SIZE_WITHOUT_CHECKDIGIT` digits. if the two first digits (from the leftmost) do not match the Mastercard pattern. if the check digit cannot be translated into the `checkdigit` type.

Function template `compute_mod97_10`

`boost::checks::compute_mod97_10` — Calculate the check digits of a sequence according to the `mod97_10_compute_algorithm` type.

Synopsis

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

template<size_t size_expected, typename check_range,
        typename checkdigits_iter>
    checkdigits_iter
    compute_mod97_10(const check_range & check_seq,
                     checkdigits_iter mod97_checkdigits);
```

Description

Parameters: `check_seq` is the sequence of value to check.
`mod97_checkdigits` is the `OutputIterator` in which the two check digits will be stored.
 Requires: `check_seq` is a valid range.
`size_expected > 0` (enforced by static assert).
`mod97_checkdigits` should have enough reserved place to store the two check digits.
 Returns: The check digits are stored into `mod97_checkdigits`. The range of these is [0..9][0..9].

Throws: `std::invalid_argument` if `check_seq` doesn't contain `size_expected` valid values. if the check digits cannot be translated into the `checkdigits_iter` type.

Function template `compute_mod97_10`

`boost::checks::compute_mod97_10` — Calculate the check digits of a sequence according to the `mod97_10_compute_algorithm` type.

Synopsis

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

template<typename check_range, typename checkdigits_iter>
    checkdigits_iter
    compute_mod97_10(const check_range & check_seq,
                     checkdigits_iter mod97_checkdigits);
```

Description

Parameters: `check_seq` is the sequence of value to check.
`mod97_checkdigits` is the `OutputIterator` in which the two check digits will be stored.

Requires: `check_seq` is a valid range.
`mod97_checkdigits` should have enough reserved place to store the two check digits.

Returns: The check digits are stored into `mod97_checkdigits`. The range of these is `[0..9][0..9]`.

Throws: `std::invalid_argument` if `check_seq` contains no valid value. if the check digits cannot be translated into the `checkdigits_iter` type.

Function template `compute_modulus11`

`boost::checks::compute_modulus11` — Calculate the check digit of a sequence according to the `mod11_compute_algorithm` type.

Synopsis

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

template<size_t size_expected, typename check_range>
    boost::checks::mod11_compute_algorithm::checkdigit< check_range >::type
    compute_modulus11(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.

Requires: `check_seq` is a valid range.
`size_expected > 0` (enforced by static assert).

Returns: The check digit. The check digit is in the range `[0..9,X]`.

Throws: `std::invalid_argument` if `check_seq` doesn't contain `size_expected` valid values. if the check digit cannot be translated into the `checkdigit` type.

Function template `compute_modulus11`

`boost::checks::compute_modulus11` — Calculate the check digit of a sequence according to the `mod11_compute_algorithm` type.

Synopsis

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

template<typename check_range>
    boost::checks::mod11_compute_algorithm::checkdigit< check_range >::type
    compute_modulus11(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: The check digit. The check digit is in the range [0..9,X].
Throws: `std::invalid_argument` if `check_seq` contains no valid value. if the check digit cannot be translated into the `checkdigit` type.

Function template `compute_upca`

`boost::checks::compute_upca` — Calculate the check digit of a sequence according to the `upc_compute_algorithm` type.

Synopsis

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

template<typename check_range>
    boost::checks::upc_compute_algorithm::checkdigit< check_range >::type
    compute_upca(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: The check digit. The check digit is in the range [0..9].
Throws: `std::invalid_argument` if `check_seq` doesn't contain exactly `UPCA_SIZE_WITHOUT_CHECKDIGIT` digits. if the check digit cannot be translated into the `checkdigit` type.

Function template `compute_verhoeff`

`boost::checks::compute_verhoeff` — Calculate the check digit of a sequence according to the `verhoeff_compute_algorithm` type.

Synopsis

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

template<size_t size_expected, typename check_range>
    boost::checks::verhoeff_compute_algorithm::checkdigit< check_range >::type
    compute_verhoeff(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.

Requires: `check_seq` is a valid range.
`size_expected > 0` (enforced by static assert).
Returns: The check digit. The check digit is in the range [0..9].
Throws: `std::invalid_argument` if `check_seq` doesn't contain `size_expected` valid values. if the check digit cannot be translated into the `checkdigit` type.

Function template `compute_verhoeff`

`boost::checks::compute_verhoeff` — Calculate the check digit of a sequence according to the `verhoeff_compute_algorithm` type.

Synopsis

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

template<typename check_range>
    boost::checks::verhoeff_compute_algorithm::checkdigit< check_range >::type
    compute_verhoeff(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: The check digit. The check digit is in the range [0..9].
Throws: `std::invalid_argument` if `check_seq` contains no valid value. if the check digit cannot be translated into the `checkdigit` type.

Function template `compute_visa`

`boost::checks::compute_visa` — Calculate the check digit of a sequence according to the `visa_compute_algorithm` type.

Synopsis

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

template<typename check_range>
    boost::checks::visa_compute_algorithm::checkdigit< check_range >::type
    compute_visa(const check_range & check_seq);
```

Description

Parameters: `check_seq` is the sequence of value to check.
Requires: `check_seq` is a valid range.
Returns: The check digit. The check digit is in the range [0..9].
Throws: `std::invalid_argument` if `check_seq` doesn't contain exactly `VISA_SIZE_WITHOUT_CHECKDIGIT` digits. if the first digit (from the leftmost) doesn't match the Visa pattern. if the check digit cannot be translated into the `checkdigit` type.

Header `<boost/checks/ean.hpp>`

This file provides tools to compute and validate an European Article Numbering (EAN) of size 8 or 13.

```
EAN13_SIZE
EAN13_SIZE_WITHOUT_CHECKDIGIT
EAN8_SIZE
EAN8_SIZE_WITHOUT_CHECKDIGIT
```

```
namespace boost {
  namespace checks {
    typedef boost::checks::modulus10_algorithm< ean_weight, ean_sense, 0 > ean_check_algorithm; ↵
    // This is the type of the EAN algorithm for validating a check digit.
    typedef boost::checks::modulus10_algorithm< ean_weight, ean_sense, 1 > ean_compute_algorithm; ↵
    // This is the type of the EAN algorithm for computing a check digit.
    typedef boost::checks::rightmost ean_sense; // This is the running sense or direction to ↵
    check an EAN.
    typedef boost::checks::weight< 1, 3 > ean_weight; // This is the weight used by EAN system.
  }
}
```

Macro EAN13_SIZE

EAN13_SIZE — This macro defines the size of an EAN-13 (13).

Synopsis

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

EAN13_SIZE
```

Macro EAN13_SIZE_WITHOUT_CHECKDIGIT

EAN13_SIZE_WITHOUT_CHECKDIGIT — This macro defines the size of an EAN-13 without its check digit (12).

Synopsis

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

EAN13_SIZE_WITHOUT_CHECKDIGIT
```

Macro EAN8_SIZE

EAN8_SIZE — This macro defines the size of an EAN-8 (8).

Synopsis

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

EAN8_SIZE
```

Macro EAN8_SIZE_WITHOUT_CHECKDIGIT

EAN8_SIZE_WITHOUT_CHECKDIGIT — This macro defines the size of a EAN-8 without its check digit (7).

Synopsis

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

EAN8_SIZE_WITHOUT_CHECKDIGIT
```

Header <boost/checks/isbn.hpp>

This file provides tools to compute and validate an International Standard Book Number (ISBN) of size 10 or 13.

The ISBN-13 is derived from the EAN number, so EAN macro or type are used.

```
ISBN10_SIZE
ISBN10_SIZE_WITHOUT_CHECKDIGIT
```

```
namespace boost {
  namespace checks {
    template<unsigned int number_of_virtual_value_skipped = 0>
      class isbn13_algorithm;

    typedef boost::checks::isbn13_algorithm< 0 > isbn13_check_algorithm; // This is the type ↴
of the ISBN-13 algorithm for validating a check digit.
    typedef boost::checks::isbn13_algorithm< 1 > isbn13_compute_algorithm; // This is the type ↴
of the ISBN-13 algorithm for computing a check digit.
  }
}
```

Class template isbn13_algorithm

boost::checks::isbn13_algorithm — This class can be used to compute or validate checksum with a basic modulus 10 but using a custom filter for the ISBN-13 prefix.

Synopsis

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

template<unsigned int number_of_virtual_value_skipped = 0 // Help functions
// to provide same
// behavior on
// sequence with
// and without
// check digits. No
// "real" value in
// the sequence
// will be skipped.
>
class isbn13_algorithm : public boost::checks::modulus10_algorithm< boost::checks::ean_weight,
boost::checks::ean_sense, number_of_virtual_value_skipped >
{
public:

    // public static functions
    static checkdigit compute_checkdigit(int);
    static checkdigits_iter compute_multicheckdigit(int, checkdigits_iter);
    static void filter_valid_value_with_pos(const unsigned int,
                                           const unsigned int);
    static void operate_on_valid_value(const int, const unsigned int, int &);
    static int translate_to_valid_value(const value &, const unsigned int);
    static bool validate_checksum(int);
};
```

Description

isbn13_algorithm public static functions

1. `static checkdigit compute_checkdigit(int checksum);`

Compute the check digit with a simple modulus 10.

Parameters: checksum is the checksum used to extract the check digit.
Returns: The modulus 10 check digit of checksum.
Throws: [boost::checks::translation_exception](#) if the check digit cannot be translated into the checkdigit type.

2. `static checkdigits_iter
compute_multicheckdigit(int checksum, checkdigits_iter checkdigits);`

Compute the check digit(s) of a sequence.

This function should be overloaded if you want your algorithm to compute more than one check digit (through it works for just one check digit too).

Parameters: checkdigits is the iterator with which the check digit(s) will be written.
 checksum is the checksum used to extract the check digit(s).
Requires: checkdigits must be a valid initialized iterator.
Returns: checkdigits.

3. `static void filter_valid_value_with_pos(const unsigned int current_valid_value,
 const unsigned int current_value_position);`

Verify that a number matches the ISBN-13 pattern.

This function use the macro EAN13_SIZE to find the real position from left to right.

Parameters:	<code>current_valid_value</code>	is the current valid value analysed.
	<code>current_value_position</code>	is the number of valid value already counted (the current value is not included).

This is also the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).

Throws: `std::invalid_argument` if the three first character are not equal to 978 or 979. The exception contains a descriptive message of what was expected.

```
4. static void operate_on_valid_value(const int current_valid_value,
                                     const unsigned int valid_value_counter,
                                     int & checksum);
```

Compute an operation on the checksum with the current valid value.

Parameters:	checksum	is the current checksum.
	current_valid_value	is the current valid value analysed.
	valid_value_counter	is the number of valid values already counted (the current value is not included).

This is also the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).

Postconditions: The current weight multiplied by the current value is added to the checksum.

```
5. static int translate_to_valid_value(const value & current_value,  
                                     const unsigned int valid_value_counter);
```

translate a value of the sequence into an integer valid value.

Parameters:	<code>current_value</code>	is the current value analysed in the sequence that must be translated.
	<code>valid_value_counter</code>	is the number of valid value(s) already counted (the current value is not included).

This is also the position (beyond the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).

Returns: the translation of the current value in the range [0..9].

Throws: `boost::checks::translation_exception` is thrown if the translation of `current_value` failed.

This will automatically throw if the value is not a digit ($0 \leq i < 11$).

```
6. static bool validate_checksum(int checksum);
```

Validate a checksum with a simple modulus 10.

Parameters: checksum is the checksum to validate.

Returns: true if the checksum is correct, false otherwise.

Macro ISBN10 SIZE

ISBN10 SIZE — This macro defines the size of an ISBN-10.

Synopsis

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

ISBN10 SIZE

Macro ISBN10_SIZE_WITHOUT_CHECKDIGIT

ISBN10_SIZE_WITHOUT_CHECKDIGIT — This macro defines the size of an ISBN-10 without its check digit.

Synopsis

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

ISBN10_SIZE_WITHOUT_CHECKDIGIT
```

Header <boost/checks/iteration_sense.hpp>

Provides two sense or direction of iteration to run through the sequence, either from right to left or left to right.

```
namespace boost {
    namespace checks {
        class leftmost;
        class rightmost;
    }
}
```

Class leftmost

boost::checks::leftmost — Policy class that provides methods to run through a sequence from left to right.

Synopsis

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

class leftmost {
public:
    // member classes/structs/unions

    // Template rebinding class used to define the type of a const iterator for
    // seq_range.
    template<typename seq_range // The type of the sequence to check.
            >
    class iterator {
    public:
        // types
        typedef boost::range_const_iterator< seq_range >::type type;
    };

    // public static functions
    template<typename seq_range>
    static iterator< seq_range >::type begin(seq_range &);
    template<typename seq_range>
    static iterator< seq_range >::type end(seq_range &);
};
```

Description

leftmost public static functions

1.

```
template<typename seq_range>
    static iterator< seq_range >::type begin(seq_range & sequence);
```

Get the beginning of the sequence.

Returns: An iterator represents the beginning of the sequence.

2.

```
template<typename seq_range>
    static iterator< seq_range >::type end(seq_range & sequence);
```

Get the ending of the sequence.

Returns: An iterator represents one past the end of the sequence.

Class template iterator

boost::checks::leftmost::iterator — Template rebinding class used to define the type of a const iterator for seq_range.

Synopsis

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

// Template rebinding class used to define the type of a const iterator for
// seq_range.
template<typename seq_range // The type of the sequence to check.
        >
class iterator {
public:
    // types
    typedef boost::range_const_iterator< seq_range >::type type;
};
```

Description

Class rightmost

boost::checks::rightmost — Policy class that provides methods to run through a sequence from right to left.

Synopsis

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

class rightmost {
public:
    // member classes/structs/unions

    // Template rebinding class used to define the type of a const reverse
    // iterator for seq_range.
    template<typename seq_range // The type of the sequence to check.
        >
    class iterator {
    public:
        // types
        typedef boost::range_const_reverse_iterator< seq_range >::type type;
    };

    // public static functions
    template<typename seq_range>
        static iterator< seq_range >::type begin(seq_range &);
    template<typename seq_range>
        static iterator< seq_range >::type end(seq_range &);
};
```

Description

rightmost public static functions

1.

```
template<typename seq_range>
    static iterator< seq_range >::type begin(seq_range & sequence);
```

Get the beginning of the sequence.

Returns: A reverse iterator representing the beginning of the sequence.

2.

```
template<typename seq_range>
    static iterator< seq_range >::type end(seq_range & sequence);
```

Get the ending of the sequence.

Returns: A reverse iterator represents one past the end of the sequence.

Class template iterator

boost::checks::rightmost::iterator — Template rebinding class used to define the type of a const reverse iterator for seq_range.

Synopsis

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

// Template rebinding class used to define the type of a const reverse
// iterator for seq_range.
template<typename seq_range // The type of the sequence to check.
>
class iterator {
public:
    // types
    typedef boost::range_const_reverse_iterator< seq_range >::type type;
};
```

Description

Header **<boost/checks/limits.hpp>**

Provides two types of size contract to manage the expected size of the check sequence.

```
namespace boost {
    namespace checks {
        template<typename exception_size_failure = std::invalid_argument>
            class no_null_size_contract;
        template<size_t expected_size,
                typename exception_size_failure = std::invalid_argument>
            class strict_size_contract;
    }
}
```

Class template **no_null_size_contract**

`boost::checks::no_null_size_contract` — This is a contract class used to verify that a sequence does not have a size of zero.

Synopsis

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

template<typename exception_size_failure = std::invalid_argument // If the
                                                                // size is
                                                                // null, a
                                                                // exception
                                                                // _size_fai
                                                                // lure
                                                                // exception
                                                                // will be
                                                                // thrown.
                                                                // Default
                                                                // exception
                                                                // class is
                                                                // std::inva
                                                                // lid_argum
                                                                // ent.
>
class no_null_size_contract {
public:

    // public static functions
    static bool reach_one_past_the_end(const size_t);
    static void respect_size_contract(const size_t);
};
```

Description

no_null_size_contract public static functions

1. `static bool reach_one_past_the_end(const size_t);`
2. `static void respect_size_contract(const size_t valid_value_counter);`

Warns if the expected interval of value [0..n) is exceeded.

Returns: false.

Enforce the size contract.

Parameters: valid_value_counter Number of valid values in the sequence.

Throws: exception_size_failure if the terms of the contract are not respected. (valid_value_counter == 0).

Class template strict_size_contract

boost::checks::strict_size_contract — This is a contract class used to verify that a sequence has the expected size.

Synopsis

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

template<size_t expected_size,    // The expected size of the sequence.
        // (Expected_size > 0, enforced with static
        // assert).
        typename exception_size_failure = std::invalid_argument // If the
                                                                    // size is
                                                                    // not
                                                                    // respected
                                                                    // an
                                                                    // exception
                                                                    // _size_fai
                                                                    // lure
                                                                    // exception
                                                                    // will be
                                                                    // thrown.
                                                                    // Default
                                                                    // exception
                                                                    // class is
                                                                    // std::inva
                                                                    // lid_argum
                                                                    // ent.
>
class strict_size_contract {
public:

    // public static functions
    static bool reach_one_past_the_end(const size_t);
    static void respect_size_contract(const size_t);
};
```

Description

strict_size_contract public static functions

1. `static bool reach_one_past_the_end(const size_t valid_value_counter);`

Tells if the expected interval of value [0..n) is outstripped.

Parameters: valid_value_counter Number of valid values in the sequence already counted.
Returns: true if valid_value_counter is one past the end of the expected size, else false.

2. `static void respect_size_contract(const size_t valid_value_counter);`

Enforce the size contract.

Parameters: valid_value_counter Number of valid values in the sequence.
Throws: exception_size_failure If the terms of the contract are not respected. (valid_value_counter != expected_size).

Header <boost/checks/luhn.hpp>

This file provides tools to compute and validate sequence with the Luhn algorithm.

```

namespace boost {
    namespace checks {
        template<unsigned int number_of_virtual_value_skipped = 0>
            class luhn_algorithm;

        typedef luhn_algorithm< 0 > luhn_check_algorithm; // This is the type of the Luhn algorithm for validating a check digit.
        typedef luhn_algorithm< 1 > luhn_compute_algorithm; // This is the type of the Luhn algorithm for computing a check digit.
        typedef boost::checks::rightmost luhn_sense; // This is the running sense to check an Luhn number.
        typedef boost::checks::weight< 1, 2 > luhn_weight; // This is the weight used by the Luhn algorithm.
    }
}

```

Class template luhn_algorithm

boost::checks::luhn_algorithm — This class can be used to compute or validate checksum with the Luhn algorithm.

Synopsis

```

// In header: <boost/checks/luhn.hpp>

template<unsigned int number_of_virtual_value_skipped = 0 // Help functions
                                                    // to provide same
                                                    // behavior on
                                                    // sequence with
                                                    // and without
                                                    // check digits. No
                                                    // "real" value in
                                                    // the sequence
                                                    // will be skipped.
>
class luhn_algorithm : public boost::checks::modulus10_algorithm< luhn_weight, luhn_sense, number_of_virtual_value_skipped >
{
public:

    // public static functions
    static checkdigit compute_checkdigit(int);
    static checkdigits_iter compute_multicheckdigit(int, checkdigits_iter);
    static void filter_valid_value_with_pos(const unsigned int,
                                           const unsigned int);
    static void operate_on_valid_value(const int, const unsigned int, int &);
    static int translate_to_valid_value(const value &, const unsigned int);
    static bool validate_checksum(int);
};

```

Description

luhn_algorithm public static functions

1. `static checkdigit compute_checkdigit(int checksum);`

Compute the check digit with a simple modulus 10.

Parameters: checksum is the checksum used to extract the check digit.

Returns: The modulus 10 check digit of checksum.
 Throws: [boost::checks::translation_exception](#) if the check digit cannot be translated into the checkdigit type.

2.

```
static checkdigits_iter
compute_multicheckdigit(int checksum, checkdigits_iter checkdigits);
```

Compute the check digit(s) of a sequence.

This function should be overloaded if you want your algorithm to compute more than one check digit (through it works for just one check digit too).

Parameters: `checkdigits` is the iterator with which the check digit(s) will be written.
`checksum` is the checksum used to extract the check digit(s).
 Requires: `checkdigits` must be a valid initialized iterator.
 Returns: `checkdigits`.

3.

```
static void filter_valid_value_with_pos(const unsigned int current_valid_value,
                                       const unsigned int current_value_position);
```

Filtering of a valid value according to its position.

This function should be overloaded if you want to filter the values with their positions.

Parameters: `current_valid_value` is the current valid value analysed.
`current_value_position` is the position (above the valid values) of the current value analysed (0 <= `valid_value_counter` < n).
 Postconditions: Do nothing.

4.

```
static void operate_on_valid_value(const int current_valid_value,
                                  const unsigned int valid_value_counter,
                                  int & checksum);
```

Compute the Luhn algorithm operation on the checksum.

This function become obsolete if you don't use `luhn_weight`. It is using operator "<<" to make internal multiplication.

Parameters: `checksum` is the current checksum.
`current_valid_value` is the current valid value analysed.
`valid_value_counter` is the number of valid value already counted (the current value is not included).
This is also the position (above the valid values) of the current value analysed (0 <= `valid_value_counter` < n).
 Postconditions: `checksum` is equal to the new computed checksum.

5.

```
static int translate_to_valid_value(const value & current_value,
                                   const unsigned int valid_value_counter);
```

translate a value of the sequence into an integer valid value.

Parameters: `current_value` is the current value analysed in the sequence that must be translated.
`valid_value_counter` is the number of valid value(s) already counted (the current value is not included).
This is also the position (beyond the valid values) of the current value analysed (0 <= `valid_value_counter` < n).
 Returns: the translation of the current value in the range [0..9].
 Throws: [boost::checks::translation_exception](#) is thrown if the translation of `current_value` failed.
This will automatically throw if the value is not a digit (0 <= i < 11).

6.

```
static bool validate_checksum(int checksum);
```

Validate a checksum with a simple modulus 10.

Parameters: `checksum` is the checksum to validate.

Returns: `true` if the checksum is correct, `false` otherwise.

Header <boost/checks/mastercard.hpp>

This file provides tools to compute and validate a Mastercard credit card number.

```
MASTERCARD_SIZE  
MASTERCARD_SIZE_WITHOUT_CHECKDIGIT
```

```
namespace boost {  
    namespace checks {  
        template<unsigned int number_of_virtual_value_skipped = 0>  
            class mastercard_algorithm;  
  
        typedef mastercard_algorithm< 0 > mastercard_check_algorithm; // This is the type of the ↵  
Mastercard algorithm for validating a check digit.  
        typedef mastercard_algorithm< 1 > mastercard_compute_algorithm; // This is the type of the ↵  
Mastercard algorithm for computing a check digit.  
    }  
}
```

Class template `mastercard_algorithm`

`boost::checks::mastercard_algorithm` — This class can be used to compute or validate checksum with the Luhn algorithm, but filter following the Mastercard pattern.

Synopsis

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

template<unsigned int number_of_virtual_value_skipped = 0 // Help functions
// to provide same
// behavior on
// sequence with
// and without
// check digits. No
// "real" value in
// the sequence
// will be skipped.
>
class mastercard_algorithm :
    public boost::checks::luhn_algorithm< number_of_virtual_value_skipped >
{
public:

    // public static functions
    static checkdigit compute_checkdigit(int);
    static checkdigits_iter compute_multicheckdigit(int, checkdigits_iter);
    static void filter_valid_value_with_pos(const unsigned int,
                                           const unsigned int);
    static void operate_on_valid_value(const int, const unsigned int, int &);
    static int translate_to_valid_value(const value &, const unsigned int);
    static bool validate_checksum(int);
};
```

Description

mastercard_algorithm public static functions

1. `static checkdigit compute_checkdigit(int checksum);`

Compute the check digit with a simple modulus 10.

Parameters: checksum is the checksum used to extract the check digit.
Returns: The modulus 10 check digit of checksum.
Throws: [boost::checks::translation_exception](#) if the check digit cannot be translated into the checkdigit type.

2. `static checkdigits_iter
compute_multicheckdigit(int checksum, checkdigits_iter checkdigits);`

Compute the check digit(s) of a sequence.

This function should be overloaded if you want your algorithm to compute more than one check digit (through it works for just one check digit too).

Parameters: checkdigits is the iterator with which the check digit(s) will be written.
 checksum is the checksum used to extract the check digit(s).
Requires: checkdigits must be a valid initialized iterator.
Returns: checkdigits.

3. `static void filter_valid_value_with_pos(const unsigned int current_valid_value,
 const unsigned int current_value_position);`

Verify that a number matches the Mastercard pattern.

This function use the macro `MASTERCARD_SIZE` to find the real position from left to right.

Parameters:	<code>current_valid_value</code>	is the current valid value analysed.
	<code>current_value_position</code>	is the number of valid values already counted (the current value is not included). This is also the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).
Throws:	<code>std::invalid_argument</code> if the first character is not equal to 5 or the second is not between 1 and 5. The exception contains a descriptive message of what was expected.	

```
4. static void operate_on_valid_value(const int current_valid_value,
                                     const unsigned int valid_value_counter,
                                     int & checksum);
```

Compute the Luhn algorithm operation on the checksum.

This function become obsolete if you don't use `luhn_weight`. It is using operator "<<" to make internal multiplication.

Parameters:	checksum	is the current checksum.
	current_valid_value	is the current valid value analysed.
	valid_value_counter	is the number of valid value already counted (the current value is not included).
		This is also the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).
Postconditions:	checksum is equal to the new computed checksum.	

[illegible]

translate a value of the sequence into an integer valid value.

Parameters:	<p><code>current_value</code> is the current value analysed in the sequence that must be translated.</p> <p><code>valid_value_counter</code> is the number of valid value(s) already counted (the current value is not included).</p> <p>This is also the position (beyond the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).</p>
Returns:	the translation of the current value in the range [0..9].
Throws:	<p>boost::checks::translation_exception is thrown if the translation of <code>current_value</code> failed.</p> <p>This will automatically throw if the value is not a digit ($0 \leq i < 11$).</p>

```
6. static bool validate_checksum(int checksum);
```

Validate a checksum with a simple modulus 10.

Parameters: checksum is the checksum to validate.
Returns: true if the checksum is correct, false otherwise.

Macro MASTERCARD_SIZE

MASTERCARD_SIZE — This macro defines the size of a Mastercard number.

Synopsis

```
// In header: <boost/checks/mastercard.hpp>
MASTERCARD_SIZE
```

Macro `MASTERCARD_SIZE_WITHOUT_CHECKDIGIT`

`MASTERCARD_SIZE_WITHOUT_CHECKDIGIT` — This macro defines the size of a Mastercard number without its check digit.

Synopsis

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

MASTERCARD_SIZE_WITHOUT_CHECKDIGIT
```

Header `<boost/checks/modulus10.hpp>`

This file provides tools to compute and validate classic modulus 10 checksum.

```
namespace boost {
    namespace checks {
        template<typename mod10_weight, typename iteration_sense,
                unsigned int number_of_virtual_value_skipped = 0>
            class modulus10_algorithm;
    }
}
```

Class template `modulus10_algorithm`

`boost::checks::modulus10_algorithm` — This class can be used to compute or validate checksum with a basic modulus 10.

Synopsis

```
// In header: <boost/checks/modulus10.hpp>

template<typename mod10_weight,    // must meet the weight concept
        // requirements.
        typename iteration_sense,  // must meet the iteration_sense concept
        // requirements.
        unsigned int number_of_virtual_value_skipped = 0 // Help functions
                                                    // to provide same
                                                    // behavior on
                                                    // sequence with
                                                    // and without
                                                    // check digits. No
                                                    // "real" value in
                                                    // the sequence
                                                    // will be skipped.
>
class modulus10_algorithm : public boost::checks::weighted_sum_algorithm< mod10_weight, iteration_sense, number_of_virtual_value_skipped >
{
public:

    // public static functions
    template<typename checkdigit> static checkdigit compute_checkdigit(int);
    static checkdigits_iter compute_multicheckdigit(int, checkdigits_iter);
    static void filter_valid_value_with_pos(const unsigned int,
                                           const unsigned int);
    static void operate_on_valid_value(const int, const unsigned int, int &);
    static int translate_to_valid_value(const value &, const unsigned int);
    static bool validate_checksum(int);
};
```

Description

modulus10_algorithm public static functions

1.

```
template<typename checkdigit>
static checkdigit compute_checkdigit(int checksum);
```

Compute the check digit with a simple modulus 10.

Parameters: checksum is the checksum used to extract the check digit.

Returns: The modulus 10 check digit of checksum.

Throws: [boost::checks::translation_exception](#) if the check digit cannot be translated into the checkdigit type.

2.

```
static checkdigits_iter
compute_multicheckdigit(int checksum, checkdigits_iter checkdigits);
```

Compute the check digit(s) of a sequence.

This function should be overloaded if you want your algorithm to compute more than one check digit (through it works for just one check digit too).

Parameters: checkdigits is the iterator with which the check digit(s) will be written.

checksum is the checksum used to extract the check digit(s).

Requires: checkdigits must be a valid initialized iterator.

Returns: checkdigits.

[illegible]

Filtering of a valid value according to its position.

This function should be overloaded if you want to filter the values with their positions.

Parameters:	<code>current_valid_value</code>	is the current valid value analysed.
	<code>current_value_position</code>	is the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).

Postconditions: Do nothing.

```
4. static void operate_on_valid_value(const int current_valid_value,
                                     const unsigned int valid_value_counter,
                                     int & checksum);
```

Compute an operation on the checksum with the current valid value.

Parameters:	checksum	is the current checksum.
	current_valid_value	is the current valid value analysed.
	valid_value_counter	is the number of valid values already counted (the current value is not included).
		This is also the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).

Postconditions: The current weight multiplied by the current value is added to the checksum.

[illegible]

translate a value of the sequence into an integer valid value.

Parameters:	<code>current_value</code>	is the current value analysed in the sequence that must be translated.
	<code>valid_value_counter</code>	is the number of valid value(s) already counted (the current value is not included).
		This is also the position (beyond the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).

Returns: the translation of the current value in the range [0..9].

Throws: [boost::checks::translation_exception](#) is thrown if the translation of `current_value` failed. This will automatically throw if the value is not a digit ($0 \leq i < 11$).

```
6. static bool validate_checksum(int checksum);
```

Validate a checksum with a simple modulus 10.

Parameters: checksum is the checksum to validate.
Returns: true if the checksum is correct, false otherwise.

Header `<boost/checks/modulus11.hpp>`

This file provides tools to compute and validate classic modulus 11 checksum.

```

namespace boost {
    namespace checks {
        template<typename mod11_weight, typename iteration_sense,
                unsigned int number_of_virtual_value_skipped = 0>
        class modulus11_algorithm;

        typedef modulus11_algorithm< mod11_weight, mod11_sense, 0 > mod11_check_algorithm; // This is the type of the most common modulus 11 algorithm for validating a check digit.
        typedef modulus11_algorithm< mod11_weight, mod11_sense, 1 > mod11_compute_algorithm; // This is the type of the most common modulus 11 algorithm for computing a check digit.
        typedef boost::checks::rightmost mod11_sense; // The most common iteration sense or direction used with a modulus 11 algorithm (right to left).
        typedef boost::checks::weight< 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 > mod11_weight; // The most common weight pattern used with a modulus 11 algorithm.
    }
}

```

Class template modulus11_algorithm

boost::checks::modulus11_algorithm — This class can be used to compute or validate checksum with a basic modulus 11.

Synopsis

```

// In header: <boost/checks/modulus11.hpp>

template<typename mod11_weight, // must meet the weight concept
        typename iteration_sense, // must meet the iteration_sense concept
        unsigned int number_of_virtual_value_skipped = 0 // Help functions
                                                    // to provide same
                                                    // behavior on
                                                    // sequence with
                                                    // and without
                                                    // check digits. No
                                                    // "real" value in
                                                    // the sequence
                                                    // will be skipped.
>
class modulus11_algorithm : public boost::checks::weighted_sum_algorithm< mod11_weight, iteration_sense, number_of_virtual_value_skipped >
{
public:

    // public static functions
    template<typename checkdigit> static checkdigit compute_checkdigit(int);
    static checkdigits_iter compute_multicheckdigit(int, checkdigits_iter);
    static void filter_valid_value_with_pos(const unsigned int,
                                           const unsigned int);
    static void operate_on_valid_value(const int, const unsigned int, int &);
    template<typename value>
        static int translate_to_valid_value(const value &, const unsigned int);
    static bool validate_checksum(int);

    // protected static functions
    template<typename checkdigit> static checkdigit translate_checkdigit(int);
};

```


Parameters: `current_value` is the current value analysed in the sequence that must be translated.
`valid_value_counter` is the number of valid value already counted (the current value is not included). This is also the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).

Returns: the translation of the current value in the range [0..10].

Throws: `boost::checks::translation_exception` is thrown if the translation of `current_value` failed. The translation will fail if the current value is not a digit ($0 \leq i < 10$), unless it is the rightmost digit, when the value 10 or the 'x' or 'X' character is allowed.

6.

```
static bool validate_checksum(int checksum);
```

Validate a checksum with a simple modulus 11.

Parameters: `checksum` is the checksum to validate.
Returns: true if the checksum is correct, false otherwise.

`modulus11_algorithm` protected static functions

1.

```
template<typename checkdigit>
static checkdigit translate_checkdigit(int _checkdigit);
```

Header `<boost/checks/modulus97.hpp>`

This file provides tools to compute and validate the classic modulus 97 checksum. It provides functions for convenience with the mod97-10 algorithm (ISO/IEC 7064:2003).

```
MOD97_weight_maker(z, n, unused)
NEXT(z, n, unused)
```

```
namespace boost {
    namespace checks {
        template<unsigned int weight_value> class make_mod97_weight;

        template<> struct make_mod97_weight<68>;

        template<typename mod97_weight, typename iteration_sense,
                unsigned int number_of_virtual_value_skipped = 0>
            class modulus97_algorithm;

        typedef make_mod97_weight< 1 > initial_mod97_weight; // This is the initial weight for the ↵
mod97-10 weights series.
        typedef modulus97_algorithm< mod97_10_weight, mod97_10_sense, 0 > mod97_10_check_algorithm; ↵
// This is the type of the modulus 97-10 algorithm for validating a check digit.
        typedef modulus97_algorithm< mod97_10_weight, mod97_10_sense, 2 > mod97_10_compute_algorithm; ↵
// This is the type of the modulus 97-10 algorithm for computing a check digit.
        typedef boost::checks::rightmost mod97_10_sense; // The iteration sense or direction of ↵
the sequence. From right to left.
        typedef boost::checks::weight< BOOST_PP_ENUM(96, MOD97_weight_maker,~) > mod97_10_weight; ↵
// This is weight of the mod97-10 algorithm.
    }
}
```

Class template make_mod97_weight

boost::checks::make_mod97_weight — This class is used to pre-compute the weight of the mod97-10 algorithm ($a = 1$; $a = a * 10 \% 97$;).

Synopsis

```
// In header: <boost/checks/modulus97.hpp>

template<unsigned int weight_value // is the weight value stored by make_mod97_weight.
>
class make_mod97_weight {
public:
    // types
    typedef make_mod97_weight< weight_value *10%97 > next;

    // public data members
    static const unsigned int value;
};
```

Description

This class is the terminal specialisation of [make_mod97_weight](#), so the recursion can finish.

The last value is 68, so we specialize [make_mod97_weight](#) to terminate the template recursion.

Struct make_mod97_weight<68>

boost::checks::make_mod97_weight<68>

Synopsis

```
// In header: <boost/checks/modulus97.hpp>

struct make_mod97_weight<68> {
    // types
    typedef make_mod97_weight type;

    // public data members
    static const unsigned int value;
};
```

Class template modulus97_algorithm

boost::checks::modulus97_algorithm — This class can be used to compute or validate a checksum with a basic modulus 97.

Synopsis

```
// In header: <boost/checks/modulus97.hpp>

template<typename mod97_weight,    // must meet the weight concept
        // requirements.
        typename iteration_sense,  // must meet the iteration_sense concept
        // requirements.
        unsigned int number_of_virtual_value_skipped = 0 // Help functions
                                                    // to provide same
                                                    // behavior on
                                                    // sequence with
                                                    // and without
                                                    // check digits. No
                                                    // "real" value in
                                                    // the sequence
                                                    // will be skipped.
>
class modulus97_algorithm : public boost::checks::weighted_sum_algorithm< mod97_weight, iteration_sense, number_of_virtual_value_skipped >
{
public:

    // public static functions
    static checkdigit compute_checkdigit(int);
    template<typename checkdigits_iter>
        static checkdigits_iter compute_multicheckdigit(int, checkdigits_iter);
    static void filter_valid_value_with_pos(const unsigned int,
                                           const unsigned int);
    static void operate_on_valid_value(const int, const unsigned int, int &);
    static int translate_to_valid_value(const value &, const unsigned int);
    static bool validate_checksum(int);
};
```

Description

The mod97-10 algorithm (ISO/IEC 7064:2003 Information technology -- Security techniques -- Check character systems) uses two check digits.

This algorithm use two check digits.

modulus97_algorithm public static functions

1.

```
static checkdigit compute_checkdigit(int checksum);
```

Compute the check digit of a sequence.

This function should be overloaded if you want to compute the check digit of a sequence.

Parameters: checksum is the checksum used to extract the check digit.

Requires: The type checkdigit must provides the default initialisation feature.

Returns: default initialized value of checkdigit.

2.

```
template<typename checkdigits_iter>
static checkdigits_iter
compute_multicheckdigit(int checksum, checkdigits_iter checkdigits);
```

Compute the two check digits with a simple modulus 97.

Parameters: checkdigits is the output iterator in which the two check digits will be written.

Requires: `checksum` is the checksum used to extract the check digit.
 Postconditions: `checkdigits` should have enough reserved place to store the two check digits.
 Returns: An iterator initialized at one pass to the end of the two check digits.
 Throws: [boost::checks::translation_exception](#) if the check digits cannot be translated into the `check_digits_iter` type.

3.

```
static void filter_valid_value_with_pos(const unsigned int current_valid_value,
                                       const unsigned int current_value_position);
```

Filtering of a valid value according to its position.

This function should be overloaded if you want to filter the values with their positions.

Parameters: `current_valid_value` is the current valid value analysed.
`current_value_position` is the position (above the valid values) of the current value analysed
 (0 <= `valid_value_counter` < n).

Postconditions: Do nothing.

4.

```
static void operate_on_valid_value(const int current_valid_value,
                                   const unsigned int valid_value_counter,
                                   int & checksum);
```

Compute an operation on the checksum with the current valid value.

Parameters: `checksum` is the current checksum.
`current_valid_value` is the current valid value analysed.
`valid_value_counter` is the number of valid values already counted (the current value is not included).
 This is also the position (above the valid values) of the current value analysed (0 <= `valid_value_counter` < n).

Postconditions: The current weight multiplied by the current value is added to the checksum.

5.

```
static int translate_to_valid_value(const value & current_value,
                                    const unsigned int valid_value_counter);
```

translate a value of the sequence into an integer valid value.

Parameters: `current_value` is the current value analysed in the sequence that must be translated.
`valid_value_counter` is the number of valid value(s) already counted (the current value is not included).
 This is also the position (beyond the valid values) of the current value analysed (0 <= `valid_value_counter` < n).

Returns: the translation of the current value in the range [0..9].

Throws: [boost::checks::translation_exception](#) is thrown if the translation of `current_value` failed.
 This will automatically throw if the value is not a digit (0 <= i < 11).

6.

```
static bool validate_checksum(int checksum);
```

Validate a checksum with a simple modulus 97.

Parameters: `checksum` is the checksum to validate.
 Returns: true if the checksum is correct, false otherwise.

Macro MOD97_weight_maker

MOD97_weight_maker — This macro is used to access to n-th value of `initial_mod97_weight`. (By using `make_mod97_weight`).

Synopsis

```
// In header: <boost/checks/modulus97.hpp>

MOD97_weight_maker(z, n, unused)
```

Macro NEXT

NEXT — This macro is used to access the next type.

Synopsis

```
// In header: <boost/checks/modulus97.hpp>

NEXT(z, n, unused)
```

Header <boost/checks/translation_exception.hpp>

This file provides an exception class used when the translation of a value failed.

```
namespace boost {
    namespace checks {
        class translation_exception;
    }
}
```

Class translation_exception

boost::checks::translation_exception — This class provides support for translation failure. For example, sequence value into integer, or integer into check digit type.

Synopsis

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

class translation_exception {
};
```

Header <boost/checks/upc.hpp>

This file provides tools to compute and validate an Universal Product Code.

```
UPCA_SIZE
UPCA_SIZE_WITHOUT_CHECKDIGIT
```

```

namespace boost {
    namespace checks {
        typedef boost::checks::modulus10_algorithm< upc_weight, upc_sense, 0 > upc_check_algorithm;
        // This is the type of the UPC algorithm for validating a check digit.
        typedef boost::checks::modulus10_algorithm< upc_weight, upc_sense, 1 > upc_compute_algorithm;
        // This is the type of the UPC algorithm for computing a check digit.
        typedef boost::checks::rightmost upc_sense; // This is the running sense or direction to
        check an UPC.
        typedef boost::checks::weight< 1, 3 > upc_weight; // This is the weight used by UPC system.
    }
}

```

Macro UPCA_SIZE

UPCA_SIZE — This macro defines the size of an UPC-A.

Synopsis

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

```
UPCA_SIZE
```

Macro UPCA_SIZE_WITHOUT_CHECKDIGIT

UPCA_SIZE_WITHOUT_CHECKDIGIT — This macro defines the size of an UPC-A without its check digit.

Synopsis

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

```
UPCA_SIZE_WITHOUT_CHECKDIGIT
```

Header <boost/checks/verhoeff.hpp>

This file provides tools to compute a Verhoeff checksum.

See http://en.wikipedia.org/wiki/Verhoeff_algorithm

```

namespace boost {
    namespace checks {
        template<unsigned int number_of_virtual_value_skipped = 0>
        class verhoeff_algorithm;

        typedef verhoeff_algorithm< 0 > verhoeff_check_algorithm; // This is the type of the Ver-
        hoeff algorithm for validating a check digit.
        typedef verhoeff_algorithm< 1 > verhoeff_compute_algorithm; // This is the type of the Ver-
        hoeff algorithm for computing a check digit.
        typedef boost::checks::rightmost verhoeff_iteration_sense; // This is the sense of the Ver-
        hoeff sequence iteration.
    }
}

```

Class template `verhoeff_algorithm`

`boost::checks::verhoeff_algorithm` — This class can be used to compute or validate checksum with the Verhoeff algorithm.

Synopsis

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

template<unsigned int number_of_virtual_value_skipped = 0 // Help functions
// to provide same
// behavior on
// sequence with
// and without
// check digits. No
// "real" value in
// the sequence
// will be skipped.
>
class verhoeff_algorithm : public boost::checks::basic_check_algorithm< verhoeff_iteration_sense,
number_of_virtual_value_skipped >
{
public:

    // public static functions
    template<typename checkdigit> static checkdigit compute_checkdigit(int);
    static checkdigits_iter compute_multicheckdigit(int, checkdigits_iter);
    static void filter_valid_value_with_pos(const unsigned int,
                                           const unsigned int);
    static void operate_on_valid_value(const int, const unsigned int, int &);
    static int translate_to_valid_value(const value &, const unsigned int);
    static bool validate_checksum(int);
};
```

Description

`verhoeff_algorithm` public static functions

1.

```
template<typename checkdigit>
static checkdigit compute_checkdigit(int checksum);
```

Compute the check digit with the Verhoeff inverse table.

Parameters: `checksum` is the checksum used to extract the check digit.
Returns: The Verhoeff check digit of checksum.
Throws: [boost::checks::translation_exception](#) if the check digit cannot be translated into the checkdigit type.

2.

```
static checkdigits_iter
compute_multicheckdigit(int checksum, checkdigits_iter checkdigits);
```

Compute the check digit(s) of a sequence.

This function should be overloaded if you want your algorithm to compute more than one check digit (through it works for just one check digit too).

Parameters: `checkdigits` is the iterator with which the check digit(s) will be written.
 `checksum` is the checksum used to extract the check digit(s).
Requires: `checkdigits` must be a valid initialized iterator.
Returns: `checkdigits`.

[illegible]

Filtering of a valid value according to its position.

This function should be overloaded if you want to filter the values with their positions.

Parameters:	<code>current_valid_value</code>	is the current valid value analysed.
	<code>current_value_position</code>	is the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).

Postconditions: Do nothing.

```
4. static void operate_on_valid_value(const int current_valid_value,
                                     const unsigned int valid_value_counter,
                                     int & checksum);
```

Compute the Verhoeff scheme on the checksum with the current valid value.

This function use the classic table d and p of the Verhoeff algorithm.

Parameters:	checksum	is the current checksum.
	current_valid_value	is the current valid value analysed.
	valid_value_counter	is the number of valid value already counted (the current value is not included).
		This is also the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).

Postconditions: checksum is equal to the new computed checksum.

[illegible]

translate a value of the sequence into an integer valid value.

Parameters:	<code>current_value</code>	is the current value analysed in the sequence that must be translated.
	<code>valid_value_counter</code>	is the number of valid value(s) already counted (the current value is not included). This is also the position (beyond the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).

Returns: the translation of the current value in the range [0..9].

Throws: [boost::checks::translation_exception](#) is thrown if the translation of `current_value` failed.

This will automatically throw if the value is not a digit ($0 \leq i < 11$).

```
6. static bool validate_checksum(int checksum);
```

Validate the Verhoeff checksum.

Parameters: checksum is the checksum to validate.

Returns: true if the checksum is correct, false otherwise.

Header **<boost/checks/visa.hpp>**

This file provides tools to compute and validate a Visa credit card number.

```
VISA_SIZE
VISA_SIZE WITHOUT CHECKDIGIT
```

```

namespace boost {
    namespace checks {
        template<unsigned int number_of_virtual_value_skipped = 0>
            class visa_algorithm;

        typedef visa_algorithm< 0 > visa_check_algorithm; // This is the type of the Visa algorithm for validating a check digit.
        typedef visa_algorithm< 1 > visa_compute_algorithm; // This is the type of the Visa algorithm for computing a check digit.
    }
}

```

Class template visa_algorithm

boost::checks::visa_algorithm — This class can be used to compute or validate checksum with the Luhn algorithm, but filter following the Visa pattern.

Synopsis

```

// In header: <boost/checks/visa.hpp>

template<unsigned int number_of_virtual_value_skipped = 0 // Helper functions
// to provide same
// behavior on a
// sequence with
// and without
// check digits. No
// "real" value in
// the sequence
// will be skipped.
>
class visa_algorithm :
    public boost::checks::luhn_algorithm< number_of_virtual_value_skipped >
{
public:

    // public static functions
    static checkdigit compute_checkdigit(int);
    static checkdigits_iter compute_multicheckdigit(int, checkdigits_iter);
    static void filter_valid_value_with_pos(const unsigned int,
                                           const unsigned int);
    static void operate_on_valid_value(const int, const unsigned int, int &);
    static int translate_to_valid_value(const value &, const unsigned int);
    static bool validate_checksum(int);
};

```

Description

visa_algorithm public static functions

1. `static checkdigit compute_checkdigit(int checksum);`

Compute the check digit with a simple modulus 10.

Parameters: checksum is the checksum used to extract the check digit.

Returns: The modulus 10 check digit of checksum.

Throws: [boost::checks::translation_exception](#) if the check digit cannot be translated into the checkdigit type.

2.

```
static checkdigits_iter
compute_multicheckdigit(int checksum, checkdigits_iter checkdigits);
```

Compute the check digit(s) of a sequence.

This function should be overloaded if you want your algorithm to compute more than one check digit (through it works for just one check digit too).

Parameters: `checkdigits` is the iterator with which the check digit(s) will be written.
`checksum` is the checksum used to extract the check digit(s).
 Requires: `checkdigits` must be a valid initialized iterator.
 Returns: `checkdigits`.

3.

```
static void filter_valid_value_with_pos(const unsigned int current_valid_value,
                                       const unsigned int current_value_position);
```

Verify that a number matches the Visa pattern.

This function use the macro `VISA_SIZE` to find the real position from left to right.

Parameters: `current_valid_value` is the current valid value analysed.
`current_value_position` is the number of valid value already counted (the current value is not included).
 This is also the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).
 Throws: `std::invalid_argument` if the first character is not equal to 4. The exception contains a descriptive message of what was expected.

4.

```
static void operate_on_valid_value(const int current_valid_value,
                                  const unsigned int valid_value_counter,
                                  int & checksum);
```

Compute the Luhn algorithm operation on the checksum.

This function become obsolete if you don't use `luhn_weight`. It is using operator "<<" to make internal multiplication.

Parameters: `checksum` is the current checksum.
`current_valid_value` is the current valid value analysed.
`valid_value_counter` is the number of valid value already counted (the current value is not included).
 This is also the position (above the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).
 Postconditions: `checksum` is equal to the new computed checksum.

5.

```
static int translate_to_valid_value(const value & current_value,
                                   const unsigned int valid_value_counter);
```

translate a value of the sequence into an integer valid value.

Parameters: `current_value` is the current value analysed in the sequence that must be translated.
`valid_value_counter` is the number of valid value(s) already counted (the current value is not included).
 This is also the position (beyond the valid values) of the current value analysed ($0 \leq \text{valid_value_counter} < n$).
 Returns: the translation of the current value in the range [0..9].
 Throws: `boost::checks::translation_exception` is thrown if the translation of `current_value` failed.
 This will automatically throw if the value is not a digit ($0 \leq i < 11$).

6.

```
static bool validate_checksum(int checksum);
```


Validate a checksum with a simple modulus 10.

Parameters: `checksum` is the checksum to validate.
Returns: `true` if the checksum is correct, `false` otherwise.

Macro `VISA_SIZE`

`VISA_SIZE` — This macro defines the size of a Visa number (16).

Synopsis

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

VISA_SIZE
```

Macro `VISA_SIZE_WITHOUT_CHECKDIGIT`

`VISA_SIZE_WITHOUT_CHECKDIGIT` — This macro defines the size of a Visa number without its check digit (15).

Synopsis

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

VISA_SIZE_WITHOUT_CHECKDIGIT
```

Header `<boost/checks/weight.hpp>`

Provides a template overridden struct to encapsulate a compile-time weight sequence.

```
_WEIGHT_factory(z, weight_size, unused)
BOOST_CHECK_LIMIT_WEIGHTS
```

```
namespace boost {
    namespace checks {
        template<BOOST_PP_ENUM_BINARY_PARAMS(BOOST_CHECK_LIMIT_WEIGHTS, int weight_value, =0 BOOST_PP_IN_
TERCEPT) >
            class weight;
    }
}
```

Class template `weight`

`boost::checks::weight` — The weight metafunction encapsulate 0 to `BOOST_CHECK_LIMIT_WEIGHTS` weights.

Synopsis

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

template<BOOST_PP_ENUM_BINARY_PARAMS(BOOST_CHECK_LIMIT_WEIGHTS, int weight_value,=0 BOOST_PP_IN_
TERCEPT) >
class weight {
public:

    // public static functions
    static int weight_associated_with_pos(const unsigned int);
};
```

Description

There are BOOST_CHECK_LIMIT_WEIGHTS partial specialisations of this class.

weight public static functions

1. `static int weight_associated_with_pos(const unsigned int value_pos);`

Get the weight at the current value position.

Parameters: value_pos is the position of the current value. (0 <= value_pos < n).
Returns: The weight value at the position value_pos.

Macro _WEIGHT_factory

_WEIGHT_factory

Synopsis

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

_WEIGHT_factory(z, weight_size, unused)
```

Macro BOOST_CHECK_LIMIT_WEIGHTS

BOOST_CHECK_LIMIT_WEIGHTS — The BOOST_CHECK_LIMIT_WEIGHTS macro defines the maximum number of weights accepted by the library.

Synopsis

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

BOOST_CHECK_LIMIT_WEIGHTS
```

Description

This macro expands to 100. For compile-time saving, you can decrease it if the algorithm(s) used have a lower weight size sequence. A contrario, you can increase it till 236 (see Boost.Preprocessor for more details about this limit.)

Header <boost/checks/weighted_sum.hpp>

This file provides tools to compute weighted sum.

```
namespace boost {
  namespace checks {
    template<typename weight, typename iteration_sense,
             unsigned int number_of_virtual_value_skipped = 0>
    class weighted_sum_algorithm;
  }
}
```

Class template weighted_sum_algorithm

boost::checks::weighted_sum_algorithm — This class permits to add to the current checksum the weight multiplied by the current value.

Synopsis

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

template<typename weight,    // must meet the weight concept requirements.
         typename iteration_sense, // must meet the iteration_sense concept
                                     // requirements.
         unsigned int number_of_virtual_value_skipped = 0 // Helper function
                                                         // to provide same
                                                         // behavior on
                                                         // sequence with
                                                         // and without
                                                         // checkdigits. No
                                                         // "real" value in
                                                         // the sequence
                                                         // will be skipped.
>
class weighted_sum_algorithm :
  public boost::checks::basic_check_algorithm< iteration_sense >
{
public:

  // public static functions
  static checkdigit compute_checkdigit(int);
  static checkdigits_iter compute_multicheckdigit(int, checkdigits_iter);
  static void filter_valid_value_with_pos(const unsigned int,
                                          const unsigned int);
  static void operate_on_valid_value(const int, const unsigned int, int &);
  static int translate_to_valid_value(const value &, const unsigned int);
  static bool validate_checksum(int);
};
```

Description

weighted_sum_algorithm public static functions

1. `static checkdigit compute_checkdigit(int checksum);`

Compute the check digit of a sequence.

This function should be overloaded if you want to compute the check digit of a sequence.

Parameters: `checksum` is the checksum used to extract the check digit.
 Requires: The type `checkdigit` must provide the default initialisation feature.
 Returns: default initialized value of `checkdigit`.

2.

```
static checkdigits_iter
compute_multicheckdigit(int checksum, checkdigits_iter checkdigits);
```

Compute the check digit(s) of a sequence.

This function should be overloaded if you want your algorithm to compute more than one check digit (through it works for just one check digit too).

Parameters: `checkdigits` is the iterator with which the check digit(s) will be written.
`checksum` is the checksum used to extract the check digit(s).
 Requires: `checkdigits` must be a valid initialized iterator.
 Returns: `checkdigits`.

3.

```
static void filter_valid_value_with_pos(const unsigned int current_valid_value,
                                       const unsigned int current_value_position);
```

Filtering of a valid value according to its position.

This function should be overloaded if you want to filter the values with their positions.

Parameters: `current_valid_value` is the current valid value analysed.
`current_value_position` is the position (above the valid values) of the current value analysed
 (0 <= `valid_value_counter` < n).

Postconditions: Do nothing.

4.

```
static void operate_on_valid_value(const int current_valid_value,
                                   const unsigned int valid_value_counter,
                                   int & checksum);
```

Compute an operation on the checksum with the current valid value.

Parameters: `checksum` is the current checksum.
`current_valid_value` is the current valid value analysed.
`valid_value_counter` is the number of valid values already counted (the current value is not included).
 This is also the position (above the valid values) of the current value analysed (0 <= `valid_value_counter` < n).

Postconditions: The current weight multiplied by the current value is added to the checksum.

5.

```
static int translate_to_valid_value(const value & current_value,
                                   const unsigned int valid_value_counter);
```

translate a value of the sequence into an integer valid value.

Parameters: `current_value` is the current value analysed in the sequence that must be translated.
`valid_value_counter` is the number of valid value(s) already counted (the current value is not included).
 This is also the position (beyond the valid values) of the current value analysed (0 <= `valid_value_counter` < n).

Returns: the translation of the current value in the range [0..9].

Throws: `boost::checks::translation_exception` is thrown if the translation of `current_value` failed.
 This will automatically throw if the value is not a digit (0 <= i < 11).

6.

```
static bool validate_checksum(int checksum);
```

Validate the checksum.

This function should be overloaded if you want to check a sequence.

Parameters: checksum is the checksum to validate.

Returns: true always (unless overloaded to check a sequence).

Class Index

Symbols

A

amex_algorithm

Header < boost/checks/amex.hpp >, 19

B

basic_check_algorithm

Header < boost/checks/basic_check_algorithm.hpp >, 23

Header < boost/checks/verhoeff.hpp >, 69

Header < boost/checks/weighted_sum.hpp >, 75

C

checkdigit

Header < boost/checks/basic_check_algorithm.hpp >, 23, 25

I

isbn13_algorithm

Header < boost/checks/isbn.hpp >, 45

iterator

Header < boost/checks/iteration_sense.hpp >, 47, 48, 49, 50

L

leftmost

Header < boost/checks/iteration_sense.hpp >, 47

luhn_algorithm

Header < boost/checks/amex.hpp >, 19

Header < boost/checks/luhn.hpp >, 53

Header < boost/checks/mastercard.hpp >, 56

Header < boost/checks/visa.hpp >, 71

M

make_mod97_weight

Header < boost/checks/modulus97.hpp >, 64

mastercard_algorithm

Header < boost/checks/mastercard.hpp >, 56

modulus10_algorithm

Header < boost/checks/isbn.hpp >, 45

Header < boost/checks/luhn.hpp >, 53

Header < boost/checks/modulus10.hpp >, 59

modulus11_algorithm

Header < boost/checks/modulus11.hpp >, 61

modulus97_algorithm

Header < boost/checks/modulus97.hpp >, 65

N

no_null_size_contract

Header < boost/checks/limits.hpp >, 51

R

rightmost

Header < boost/checks/iteration_sense.hpp >, 49

S

strict_size_contract

Header < boost/checks/limits.hpp >, 52

T

translation_exception

Header < boost/checks/translation_exception.hpp >, 67

V

verhoeff_algorithm

Header < boost/checks/verhoeff.hpp >, 69

visa_algorithm

Header < boost/checks/visa.hpp >, 71

W

weight

Header < boost/checks/weight.hpp >, 74

weighted_sum_algorithm

Header < boost/checks/modulus10.hpp >, 59

Header < boost/checks/modulus11.hpp >, 61

Header < boost/checks/modulus97.hpp >, 65

Header < boost/checks/weighted_sum.hpp >, 75

Typedef Index

Symbols

A

amex_check_algorithm

Header < boost/checks/amex.hpp >, 18

amex_compute_algorithm

Header < boost/checks/amex.hpp >, 18

E

ean_check_algorithm

Header < boost/checks/ean.hpp >, 42

ean_compute_algorithm

Header < boost/checks/ean.hpp >, 42

ean_sense

Header < boost/checks/ean.hpp >, 42

ean_weight

Header < boost/checks/ean.hpp >, 42

I

initial_mod97_weight

Header < boost/checks/modulus97.hpp >, 63
isbn13_check_algorithm
Header < boost/checks/isbn.hpp >, 44
isbn13_compute_algorithm
Header < boost/checks/isbn.hpp >, 44
iteration_sense
Header < boost/checks/basic_check_algorithm.hpp >, 23

L

luhn_check_algorithm
Header < boost/checks/luhn.hpp >, 52
luhn_compute_algorithm
Header < boost/checks/luhn.hpp >, 52
luhn_sense
Header < boost/checks/luhn.hpp >, 52
luhn_weight
Header < boost/checks/luhn.hpp >, 52

M

mastercard_check_algorithm
Header < boost/checks/mastercard.hpp >, 55
mastercard_compute_algorithm
Header < boost/checks/mastercard.hpp >, 55
mod11_check_algorithm
Header < boost/checks/modulus11.hpp >, 60
mod11_compute_algorithm
Header < boost/checks/modulus11.hpp >, 60
mod11_sense
Header < boost/checks/modulus11.hpp >, 60
mod11_weight
Header < boost/checks/modulus11.hpp >, 60
mod97_10_check_algorithm
Header < boost/checks/modulus97.hpp >, 63
mod97_10_compute_algorithm
Header < boost/checks/modulus97.hpp >, 63
mod97_10_sense
Header < boost/checks/modulus97.hpp >, 63
mod97_10_weight
Header < boost/checks/modulus97.hpp >, 63

N

next
Header < boost/checks/modulus97.hpp >, 64

T

type
Header < boost/checks/basic_check_algorithm.hpp >, 23, 25
Header < boost/checks/iteration_sense.hpp >, 47, 48, 49, 50
Header < boost/checks/modulus97.hpp >, 64

U

upc_check_algorithm
Header < boost/checks/upc.hpp >, 67
upc_compute_algorithm
Header < boost/checks/upc.hpp >, 67
upc_sense

Header < boost/checks/upc.hpp >, 67
 upc_weight
 Header < boost/checks/upc.hpp >, 67

V

verhoeff_check_algorithm
 Header < boost/checks/verhoeff.hpp >, 68
 verhoeff_compute_algorithm
 Header < boost/checks/verhoeff.hpp >, 68
 verhoeff_iteration_sense
 Header < boost/checks/verhoeff.hpp >, 68
 visa_check_algorithm
 Header < boost/checks/visa.hpp >, 70
 visa_compute_algorithm
 Header < boost/checks/visa.hpp >, 70

Function Index

Symbols

B

begin
 Header < boost/checks/iteration_sense.hpp >, 47, 48, 49

C

check_amex
 Header < boost/checks/amex.hpp >, 18, 20
 check_ean13
 Header < boost/checks/checks_fwd.hpp >, 29, 31
 check_ean8
 Header < boost/checks/checks_fwd.hpp >, 29, 31
 check_isbn10
 Header < boost/checks/checks_fwd.hpp >, 29, 32
 check_isbn13
 Header < boost/checks/checks_fwd.hpp >, 29, 32
 check_luhn
 Header < boost/checks/checks_fwd.hpp >, 29, 32, 33
 check_mastercard
 Header < boost/checks/checks_fwd.hpp >, 29, 33
 check_mod97_10
 Header < boost/checks/checks_fwd.hpp >, 29, 33, 34
 check_modulus11
 Header < boost/checks/checks_fwd.hpp >, 29, 34, 35
 check_sequence
 Header < boost/checks/basic_checks.hpp >, 25, 26, 27
 check_upca
 Header < boost/checks/checks_fwd.hpp >, 29, 35
 check_verhoeff
 Header < boost/checks/checks_fwd.hpp >, 29, 35, 36
 check_visa
 Header < boost/checks/checks_fwd.hpp >, 29, 36
 compute_amex
 Header < boost/checks/amex.hpp >, 18, 21
 compute_checkdigit
 Extending the library, 9

Header < boost/checks/amex.hpp >, 19
 Header < boost/checks/basic_checks.hpp >, 25, 27
 Header < boost/checks/basic_check_algorithm.hpp >, 23
 Header < boost/checks/isbn.hpp >, 45
 Header < boost/checks/luhn.hpp >, 53
 Header < boost/checks/mastercard.hpp >, 56
 Header < boost/checks/modulus10.hpp >, 59
 Header < boost/checks/modulus11.hpp >, 61, 62
 Header < boost/checks/modulus97.hpp >, 65
 Header < boost/checks/verhoeff.hpp >, 69
 Header < boost/checks/visa.hpp >, 71
 Header < boost/checks/weighted_sum.hpp >, 75
 compute_checksum
 Header < boost/checks/basic_checks.hpp >, 25, 28
 compute_ean13
 Header < boost/checks/checks_fwd.hpp >, 29, 36
 compute_ean8
 Header < boost/checks/checks_fwd.hpp >, 29, 37
 compute_isbn10
 Header < boost/checks/checks_fwd.hpp >, 29, 37
 compute_isbn13
 Header < boost/checks/checks_fwd.hpp >, 29, 38
 compute_luhn
 Header < boost/checks/checks_fwd.hpp >, 29, 38
 compute_mastercard
 Header < boost/checks/checks_fwd.hpp >, 29, 39
 compute_mod97_10
 Header < boost/checks/checks_fwd.hpp >, 29, 39, 40
 compute_modulus11
 Header < boost/checks/checks_fwd.hpp >, 29, 40, 41
 compute_multicheckdigit
 Header < boost/checks/amex.hpp >, 19
 Header < boost/checks/basic_checks.hpp >, 25, 29
 Header < boost/checks/basic_check_algorithm.hpp >, 23, 24
 Header < boost/checks/isbn.hpp >, 45
 Header < boost/checks/luhn.hpp >, 53, 54
 Header < boost/checks/mastercard.hpp >, 56
 Header < boost/checks/modulus10.hpp >, 59
 Header < boost/checks/modulus11.hpp >, 61, 62
 Header < boost/checks/modulus97.hpp >, 65
 Header < boost/checks/verhoeff.hpp >, 69
 Header < boost/checks/visa.hpp >, 71, 72
 Header < boost/checks/weighted_sum.hpp >, 75, 76
 compute_upca
 Header < boost/checks/checks_fwd.hpp >, 29, 41
 compute_verhoeff
 Header < boost/checks/checks_fwd.hpp >, 29, 41, 42
 compute_visa
 Header < boost/checks/checks_fwd.hpp >, 29, 42

E

end

Header < boost/checks/iteration_sense.hpp >, 47, 48, 49

F

filter_valid_value_with_pos

Header < boost/checks/amex.hpp >, 19

Header < boost/checks/basic_check_algorithm.hpp >, 23, 24
Header < boost/checks/isbn.hpp >, 45
Header < boost/checks/luhn.hpp >, 53, 54
Header < boost/checks/mastercard.hpp >, 56
Header < boost/checks/modulus10.hpp >, 59, 60
Header < boost/checks/modulus11.hpp >, 61, 62
Header < boost/checks/modulus97.hpp >, 65, 66
Header < boost/checks/verhoeff.hpp >, 69, 70
Header < boost/checks/visa.hpp >, 71, 72
Header < boost/checks/weighted_sum.hpp >, 75, 76

O

operate_on_valid_value

Extending the library, 9

Header < boost/checks/amex.hpp >, 19, 20
Header < boost/checks/basic_check_algorithm.hpp >, 23, 24
Header < boost/checks/isbn.hpp >, 45, 46
Header < boost/checks/luhn.hpp >, 53, 54
Header < boost/checks/mastercard.hpp >, 56, 57
Header < boost/checks/modulus10.hpp >, 59, 60
Header < boost/checks/modulus11.hpp >, 61, 62
Header < boost/checks/modulus97.hpp >, 65, 66
Header < boost/checks/verhoeff.hpp >, 69, 70
Header < boost/checks/visa.hpp >, 71, 72
Header < boost/checks/weighted_sum.hpp >, 75, 76

R

reach_one_past_the_end

Header < boost/checks/limits.hpp >, 51, 52

respect_size_contract

Header < boost/checks/limits.hpp >, 51, 52

T

translate_checkdigit

Header < boost/checks/modulus11.hpp >, 61, 63

translate_to_valid_value

Extending the library, 9

Header < boost/checks/amex.hpp >, 19, 20
Header < boost/checks/basic_check_algorithm.hpp >, 23, 24
Header < boost/checks/isbn.hpp >, 45, 46
Header < boost/checks/luhn.hpp >, 53, 54
Header < boost/checks/mastercard.hpp >, 56, 57
Header < boost/checks/modulus10.hpp >, 59, 60
Header < boost/checks/modulus11.hpp >, 61, 62
Header < boost/checks/modulus97.hpp >, 65, 66
Header < boost/checks/verhoeff.hpp >, 69, 70
Header < boost/checks/visa.hpp >, 71, 72
Header < boost/checks/weighted_sum.hpp >, 75, 76

V

validate_checksum

Header < boost/checks/amex.hpp >, 19, 20
Header < boost/checks/basic_check_algorithm.hpp >, 23, 24
Header < boost/checks/isbn.hpp >, 45, 46
Header < boost/checks/luhn.hpp >, 53, 54
Header < boost/checks/mastercard.hpp >, 56, 57
Header < boost/checks/modulus10.hpp >, 59, 60

Header < boost/checks/modulus11.hpp >, 61, 63
Header < boost/checks/modulus97.hpp >, 65, 66
Header < boost/checks/verhoeff.hpp >, 69, 70
Header < boost/checks/visa.hpp >, 71, 72
Header < boost/checks/weighted_sum.hpp >, 75, 76

W

weight_associated_with_pos
Header < boost/checks/weight.hpp >, 74

Macro Index

Symbols

_WEIGHT_factory
Header < boost/checks/weight.hpp >, 73, 74

A

AMEX_SIZE
Header < boost/checks/amex.hpp >, 18, 19, 21
AMEX_SIZE_WITHOUT_CHECKDIGIT
Header < boost/checks/amex.hpp >, 18, 21

B

BOOST_CHECK_LIMIT_WEIGHTS
Header < boost/checks/weight.hpp >, 73, 74

E

EAN13_SIZE
Header < boost/checks/checks_fwd.hpp >, 31, 32
Header < boost/checks/ean.hpp >, 42, 43
Header < boost/checks/isbn.hpp >, 45
EAN13_SIZE_WITHOUT_CHECKDIGIT
Header < boost/checks/checks_fwd.hpp >, 37, 38
Header < boost/checks/ean.hpp >, 42, 43
EAN8_SIZE
Header < boost/checks/checks_fwd.hpp >, 31
Header < boost/checks/ean.hpp >, 42, 43
EAN8_SIZE_WITHOUT_CHECKDIGIT
Header < boost/checks/checks_fwd.hpp >, 37
Header < boost/checks/ean.hpp >, 42, 43, 44

I

ISBN10_SIZE
Header < boost/checks/checks_fwd.hpp >, 32
Header < boost/checks/isbn.hpp >, 44, 46
ISBN10_SIZE_WITHOUT_CHECKDIGIT
Header < boost/checks/checks_fwd.hpp >, 37
Header < boost/checks/isbn.hpp >, 44, 47

M

MASTERCARD_SIZE
Header < boost/checks/checks_fwd.hpp >, 33
Header < boost/checks/mastercard.hpp >, 55, 56, 57
MASTERCARD_SIZE_WITHOUT_CHECKDIGIT

Header < boost/checks/checks_fwd.hpp >, 39
 Header < boost/checks/mastercard.hpp >, 55, 58
 MOD97_weight_maker
 Header < boost/checks/modulus97.hpp >, 63, 66, 67

N

NEXT

Header < boost/checks/modulus97.hpp >, 63, 67

U

UPCA_SIZE

Header < boost/checks/checks_fwd.hpp >, 35

Header < boost/checks/upc.hpp >, 67, 68

UPCA_SIZE_WITHOUT_CHECKDIGIT

Header < boost/checks/checks_fwd.hpp >, 41

Header < boost/checks/upc.hpp >, 67, 68

V

VISA_SIZE

Header < boost/checks/checks_fwd.hpp >, 36

Header < boost/checks/visa.hpp >, 70, 72, 73

VISA_SIZE_WITHOUT_CHECKDIGIT

Header < boost/checks/checks_fwd.hpp >, 42

Header < boost/checks/visa.hpp >, 70, 73

Index

Symbols

_WEIGHT_factory

Header < boost/checks/weight.hpp >, 73, 74

A

acknowledgements

Acknowledgements, 16

Acknowledgements

acknowledgements, 16

C++, 16

Verhoeff, 16

Alteration

C++, 14

example, 14

amex_algorithm

Header < boost/checks/amex.hpp >, 19

amex_check_algorithm

Header < boost/checks/amex.hpp >, 18

amex_compute_algorithm

Header < boost/checks/amex.hpp >, 18

AMEX_SIZE

Header < boost/checks/amex.hpp >, 18, 19, 21

AMEX_SIZE_WITHOUT_CHECKDIGIT

Header < boost/checks/amex.hpp >, 18, 21

B

basic_check_algorithm

Header < boost/checks/basic_check_algorithm.hpp >, 23

Header < boost/checks/verhoeff.hpp >, 69
Header < boost/checks/weighted_sum.hpp >, 75
begin
Header < boost/checks/iteration_sense.hpp >, 47, 48, 49
book
Header < boost/checks/isbn.hpp >, 44
Starting with Checks, 6
BOOST_CHECK_LIMIT_WEIGHTS
Header < boost/checks/weight.hpp >, 73, 74

C

C++

Acknowledgements, 16
Alteration, 14
Checks, 1, 2, 3
Checks Reference, 18
Common check algorithms, 13, 14
Document Conventions, 4, 5
Error catching summary, 13
Extending the library, 9, 10
Header < boost/checks/amex.hpp >, 18, 19, 20, 21
Header < boost/checks/basic_checks.hpp >, 25, 26, 27, 28, 29
Header < boost/checks/basic_check_algorithm.hpp >, 22, 23, 24, 25
Header < boost/checks/checks_fwd.hpp >, 29, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42
Header < boost/checks/ean.hpp >, 42, 43, 44
Header < boost/checks/isbn.hpp >, 44, 45, 46, 47
Header < boost/checks/iteration_sense.hpp >, 47, 48, 49, 50
Header < boost/checks/limits.hpp >, 50, 51, 52
Header < boost/checks/luhn.hpp >, 52, 53, 54, 55
Header < boost/checks/mastercard.hpp >, 55, 56, 57, 58
Header < boost/checks/modulus10.hpp >, 58, 59, 60
Header < boost/checks/modulus11.hpp >, 60, 61, 62, 63
Header < boost/checks/modulus97.hpp >, 63, 64, 65, 66, 67
Header < boost/checks/translation_exception.hpp >, 67
Header < boost/checks/upc.hpp >, 67, 68
Header < boost/checks/verhoeff.hpp >, 68, 69, 70
Header < boost/checks/visa.hpp >, 70, 71, 72, 73
Header < boost/checks/weight.hpp >, 73, 74
Header < boost/checks/weighted_sum.hpp >, 75, 76, 77
History, 17
Length, 14
Letter to digit VIN conversion table, 9
Luhn algorithm, 15
Modular sum algorithms, 14
Modulus 10 algorithm, 15
Modulus 11 algorithm, 15
Phonetic, 14
Preface, 3
Rationale, 17
References, 16, 17
Starting with Checks, 6
Status, 4
Summary of the modular sum algorithms, 16
Transposition, 14
Tutorial, 6
Type of errors, 14
Verhoeff Algorithm, 15

Version Info, 17

card

- Header < boost/checks/amex.hpp >, 18, 21
- Header < boost/checks/mastercard.hpp >, 55
- Header < boost/checks/visa.hpp >, 70
- Luhn algorithm, 15
- Preface, 3
- Starting with Checks, 6

checkdigit

- Header < boost/checks/basic_check_algorithm.hpp >, 23, 25

Checks

- C++, 1, 2, 3
- index, 2
- version, 1

Checks Reference

- C++, 18

check_amex

- Header < boost/checks/amex.hpp >, 18, 20

check_ean13

- Header < boost/checks/checks_fwd.hpp >, 29, 31

check_ean8

- Header < boost/checks/checks_fwd.hpp >, 29, 31

check_isbn10

- Header < boost/checks/checks_fwd.hpp >, 29, 32

check_isbn13

- Header < boost/checks/checks_fwd.hpp >, 29, 32

check_luhn

- Header < boost/checks/checks_fwd.hpp >, 29, 32, 33

check_mastercard

- Header < boost/checks/checks_fwd.hpp >, 29, 33

check_mod97_10

- Header < boost/checks/checks_fwd.hpp >, 29, 33, 34

check_modulus11

- Header < boost/checks/checks_fwd.hpp >, 29, 34, 35

check_sequence

- Header < boost/checks/basic_checks.hpp >, 25, 26, 27

check_upca

- Header < boost/checks/checks_fwd.hpp >, 29, 35

check_verhoeff

- Header < boost/checks/checks_fwd.hpp >, 29, 35, 36

check_visa

- Header < boost/checks/checks_fwd.hpp >, 29, 36

Common check algorithms

- C++, 13, 14
- example, 13

compute_amex

- Header < boost/checks/amex.hpp >, 18, 21

compute_checkdigit

- Extending the library, 9
- Header < boost/checks/amex.hpp >, 19
- Header < boost/checks/basic_checks.hpp >, 25, 27
- Header < boost/checks/basic_check_algorithm.hpp >, 23
- Header < boost/checks/isbn.hpp >, 45
- Header < boost/checks/luhn.hpp >, 53
- Header < boost/checks/mastercard.hpp >, 56
- Header < boost/checks/modulus10.hpp >, 59
- Header < boost/checks/modulus11.hpp >, 61, 62
- Header < boost/checks/modulus97.hpp >, 65

Header < boost/checks/verhoeff.hpp >, 69
Header < boost/checks/visa.hpp >, 71
Header < boost/checks/weighted_sum.hpp >, 75
compute_checksum
Header < boost/checks/basic_checks.hpp >, 25, 28
compute_ean13
Header < boost/checks/checks_fwd.hpp >, 29, 36
compute_ean8
Header < boost/checks/checks_fwd.hpp >, 29, 37
compute_isbn10
Header < boost/checks/checks_fwd.hpp >, 29, 37
compute_isbn13
Header < boost/checks/checks_fwd.hpp >, 29, 38
compute_luhn
Header < boost/checks/checks_fwd.hpp >, 29, 38
compute_mastercard
Header < boost/checks/checks_fwd.hpp >, 29, 39
compute_mod97_10
Header < boost/checks/checks_fwd.hpp >, 29, 39, 40
compute_modulus11
Header < boost/checks/checks_fwd.hpp >, 29, 40, 41
compute_multicheckdigit
Header < boost/checks/amex.hpp >, 19
Header < boost/checks/basic_checks.hpp >, 25, 29
Header < boost/checks/basic_check_algorithm.hpp >, 23, 24
Header < boost/checks/isbn.hpp >, 45
Header < boost/checks/luhn.hpp >, 53, 54
Header < boost/checks/mastercard.hpp >, 56
Header < boost/checks/modulus10.hpp >, 59
Header < boost/checks/modulus11.hpp >, 61, 62
Header < boost/checks/modulus97.hpp >, 65
Header < boost/checks/verhoeff.hpp >, 69
Header < boost/checks/visa.hpp >, 71, 72
Header < boost/checks/weighted_sum.hpp >, 75, 76
compute_upca
Header < boost/checks/checks_fwd.hpp >, 29, 41
compute_verhoeff
Header < boost/checks/checks_fwd.hpp >, 29, 41, 42
compute_visa
Header < boost/checks/checks_fwd.hpp >, 29, 42
credit
Header < boost/checks/amex.hpp >, 18
Header < boost/checks/mastercard.hpp >, 55
Header < boost/checks/visa.hpp >, 70
Luhn algorithm, 15
Preface, 3
Starting with Checks, 6

D

Document Conventions

C++, 4, 5
Doxygen, 5
example, 4, 5
index, 5
italic, 5
pre-conditions, 5
snippet, 5

Doxygen

Document Conventions, 5

E

EAN13_SIZE

Header < boost/checks/checks_fwd.hpp >, 31, 32

Header < boost/checks/ean.hpp >, 42, 43

Header < boost/checks/isbn.hpp >, 45

EAN13_SIZE_WITHOUT_CHECKDIGIT

Header < boost/checks/checks_fwd.hpp >, 37, 38

Header < boost/checks/ean.hpp >, 42, 43

EAN8_SIZE

Header < boost/checks/checks_fwd.hpp >, 31

Header < boost/checks/ean.hpp >, 42, 43

EAN8_SIZE_WITHOUT_CHECKDIGIT

Header < boost/checks/checks_fwd.hpp >, 37

Header < boost/checks/ean.hpp >, 42, 43, 44

ean_check_algorithm

Header < boost/checks/ean.hpp >, 42

ean_compute_algorithm

Header < boost/checks/ean.hpp >, 42

ean_sense

Header < boost/checks/ean.hpp >, 42

ean_weight

Header < boost/checks/ean.hpp >, 42

end

Header < boost/checks/iteration_sense.hpp >, 47, 48, 49

equations

Verhoeff Algorithm, 15

Error catching summary

C++, 13

Luhn, 13

Verhoeff, 13

example

Alteration, 14

Common check algorithms, 13

Document Conventions, 4, 5

Extending the library, 9

Header < boost/checks/translation_exception.hpp >, 67

Starting with Checks, 6

Extending the library

C++, 9, 10

compute_checkdigit, 9

example, 9

modulus, 9, 10

operate_on_valid_value, 9

pre-conditions, 9, 10

translate_to_valid_value, 9

F

filter_valid_value_with_pos

Header < boost/checks/amex.hpp >, 19

Header < boost/checks/basic_check_algorithm.hpp >, 23, 24

Header < boost/checks/isbn.hpp >, 45

Header < boost/checks/luhn.hpp >, 53, 54

Header < boost/checks/mastercard.hpp >, 56

Header < boost/checks/modulus10.hpp >, 59, 60

Header < boost/checks/modulus11.hpp >, 61, 62
Header < boost/checks/modulus97.hpp >, 65, 66
Header < boost/checks/verhoeff.hpp >, 69, 70
Header < boost/checks/visa.hpp >, 71, 72
Header < boost/checks/weighted_sum.hpp >, 75, 76

G

Gumm

References, 16

H

Header < boost/checks/amex.hpp >
amex_algorithm, 19
amex_check_algorithm, 18
amex_compute_algorithm, 18
AMEX_SIZE, 18, 19, 21
AMEX_SIZE_WITHOUT_CHECKDIGIT, 18, 21
C++, 18, 19, 20, 21
card, 18, 21
check_amex, 18, 20
compute_amex, 18, 21
compute_checkdigit, 19
compute_multicheckdigit, 19
credit, 18
filter_valid_value_with_pos, 19
Luhn, 18, 20
luhn_algorithm, 19
modulus, 19, 20
operate_on_valid_value, 19, 20
post-conditions, 20
translate_to_valid_value, 19, 20
validate_checksum, 19, 20
Header < boost/checks/basic_checks.hpp >
C++, 25, 26, 27, 28, 29
check_sequence, 25, 26, 27
compute_checkdigit, 25, 27
compute_checksum, 25, 28
compute_multicheckdigit, 25, 29
version, 28
Header < boost/checks/basic_check_algorithm.hpp >
basic_check_algorithm, 23
C++, 22, 23, 24, 25
checkdigit, 23, 25
compute_checkdigit, 23
compute_multicheckdigit, 23, 24
filter_valid_value_with_pos, 23, 24
iteration_sense, 23
operate_on_valid_value, 23, 24
post-conditions, 24
translate_to_valid_value, 23, 24
type, 23, 25
validate_checksum, 23, 24
Header < boost/checks/checks_fwd.hpp >
C++, 29, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42
check_ean13, 29, 31
check_ean8, 29, 31
check_isbn10, 29, 32

check_isbn13, 29, 32
check_luhn, 29, 32, 33
check_mastercard, 29, 33
check_mod97_10, 29, 33, 34
check_modulus11, 29, 34, 35
check_upca, 29, 35
check_verhoeff, 29, 35, 36
check_visa, 29, 36
compute_ean13, 29, 36
compute_ean8, 29, 37
compute_isbn10, 29, 37
compute_isbn13, 29, 38
compute_luhn, 29, 38
compute_mastercard, 29, 39
compute_mod97_10, 29, 39, 40
compute_modulus11, 29, 40, 41
compute_upca, 29, 41
compute_verhoeff, 29, 41, 42
compute_visa, 29, 42
EAN13_SIZE, 31, 32
EAN13_SIZE_WITHOUT_CHECKDIGIT, 37, 38
EAN8_SIZE, 31
EAN8_SIZE_WITHOUT_CHECKDIGIT, 37
ISBN10_SIZE, 32
ISBN10_SIZE_WITHOUT_CHECKDIGIT, 37
Mastercard, 33, 39
MASTERCARD_SIZE, 33
MASTERCARD_SIZE_WITHOUT_CHECKDIGIT, 39
UPCA_SIZE, 35
UPCA_SIZE_WITHOUT_CHECKDIGIT, 41
VISA, 36, 42
VISA_SIZE, 36
VISA_SIZE_WITHOUT_CHECKDIGIT, 42
Header < boost/checks/ean.hpp >
C++, 42, 43, 44
EAN13_SIZE, 42, 43
EAN13_SIZE_WITHOUT_CHECKDIGIT, 42, 43
EAN8_SIZE, 42, 43
EAN8_SIZE_WITHOUT_CHECKDIGIT, 42, 43, 44
ean_check_algorithm, 42
ean_compute_algorithm, 42
ean_sense, 42
ean_weight, 42
Header < boost/checks/isbn.hpp >
book, 44
C++, 44, 45, 46, 47
compute_checkdigit, 45
compute_multicheckdigit, 45
EAN13_SIZE, 45
filter_valid_value_with_pos, 45
ISBN, 44, 45, 46, 47
ISBN10_SIZE, 44, 46
ISBN10_SIZE_WITHOUT_CHECKDIGIT, 44, 47
isbn13_algorithm, 45
isbn13_check_algorithm, 44
isbn13_compute_algorithm, 44
modulus, 44, 45, 46
modulus10_algorithm, 45

operate_on_valid_value, 45, 46
post-conditions, 46
pre-conditions, 44
translate_to_valid_value, 45, 46
validate_checksum, 45, 46

Header < boost/checks/iteration_sense.hpp >
begin, 47, 48, 49
C++, 47, 48, 49, 50
end, 47, 48, 49
iterator, 47, 48, 49, 50
leftmost, 47
rightmost, 49
type, 47, 48, 49, 50

Header < boost/checks/limits.hpp >
C++, 50, 51, 52
no_null_size_contract, 51
reach_one_past_the_end, 51, 52
respect_size_contract, 51, 52
strict_size_contract, 52

Header < boost/checks/luhn.hpp >
C++, 52, 53, 54, 55
compute_checkdigit, 53
compute_multicheckdigit, 53, 54
filter_valid_value_with_pos, 53, 54
Luhn, 52, 53, 54
luhn_algorithm, 53
luhn_check_algorithm, 52
luhn_compute_algorithm, 52
luhn_sense, 52
luhn_weight, 52
modulus, 53, 54
modulus10_algorithm, 53
operate_on_valid_value, 53, 54
post-conditions, 54
translate_to_valid_value, 53, 54
validate_checksum, 53, 54

Header < boost/checks/mastercard.hpp >
C++, 55, 56, 57, 58
card, 55
compute_checkdigit, 56
compute_multicheckdigit, 56
credit, 55
filter_valid_value_with_pos, 56
Luhn, 55, 57
luhn_algorithm, 56
Mastercard, 55, 56, 57, 58
mastercard_algorithm, 56
mastercard_check_algorithm, 55
mastercard_compute_algorithm, 55
MASTERCARD_SIZE, 55, 56, 57
MASTERCARD_SIZE_WITHOUT_CHECKDIGIT, 55, 58
modulus, 56, 57
operate_on_valid_value, 56, 57
post-conditions, 57
translate_to_valid_value, 56, 57
validate_checksum, 56, 57

Header < boost/checks/modulus10.hpp >
C++, 58, 59, 60

compute_checkdigit, 59
compute_multicheckdigit, 59
filter_valid_value_with_pos, 59, 60
modulus, 58, 59, 60
modulus10_algorithm, 59
operate_on_valid_value, 59, 60
post-conditions, 60
translate_to_valid_value, 59, 60
validate_checksum, 59, 60
weighted_sum_algorithm, 59
Header < boost/checks/modulus11.hpp >
C++, 60, 61, 62, 63
compute_checkdigit, 61, 62
compute_multicheckdigit, 61, 62
filter_valid_value_with_pos, 61, 62
mod11_check_algorithm, 60
mod11_compute_algorithm, 60
mod11_sense, 60
mod11_weight, 60
modulus, 60, 61, 62, 63
modulus11_algorithm, 61
operate_on_valid_value, 61, 62
post-conditions, 62
translate_checkdigit, 61, 63
translate_to_valid_value, 61, 62
validate_checksum, 61, 63
weighted_sum_algorithm, 61
Header < boost/checks/modulus97.hpp >
C++, 63, 64, 65, 66, 67
compute_checkdigit, 65
compute_multicheckdigit, 65
filter_valid_value_with_pos, 65, 66
initial_mod97_weight, 63
make_mod97_weight, 64
mod97_10_check_algorithm, 63
mod97_10_compute_algorithm, 63
mod97_10_sense, 63
mod97_10_weight, 63
MOD97_weight_maker, 63, 66, 67
modulus, 63, 64, 65, 66
modulus97_algorithm, 65
NEXT, 63, 67
next, 64
operate_on_valid_value, 65, 66
post-conditions, 66
pre-conditions, 64
translate_to_valid_value, 65, 66
type, 64
validate_checksum, 65, 66
weighted_sum_algorithm, 65
Header < boost/checks/translation_exception.hpp >
C++, 67
example, 67
translation_exception, 67
Header < boost/checks/upc.hpp >
C++, 67, 68
UPCA_SIZE, 67, 68
UPCA_SIZE_WITHOUT_CHECKDIGIT, 67, 68

upc_check_algorithm, 67
upc_compute_algorithm, 67
upc_sense, 67
upc_weight, 67
Header < boost/checks/verhoeff.hpp >
basic_check_algorithm, 69
C++, 68, 69, 70
compute_checkdigit, 69
compute_multicheckdigit, 69
filter_valid_value_with_pos, 69, 70
operate_on_valid_value, 69, 70
post-conditions, 70
translate_to_valid_value, 69, 70
validate_checksum, 69, 70
Verhoeff, 68, 69, 70
verhoeff_algorithm, 69
verhoeff_check_algorithm, 68
verhoeff_compute_algorithm, 68
verhoeff_iteration_sense, 68
Header < boost/checks/visa.hpp >
C++, 70, 71, 72, 73
card, 70
compute_checkdigit, 71
compute_multicheckdigit, 71, 72
credit, 70
filter_valid_value_with_pos, 71, 72
Luhn, 71, 72
luhn_algorithm, 71
modulus, 71, 72
operate_on_valid_value, 71, 72
post-conditions, 72
translate_to_valid_value, 71, 72
validate_checksum, 71, 72
VISA, 70, 71, 72, 73
visa_algorithm, 71
visa_check_algorithm, 70
visa_compute_algorithm, 70
VISA_SIZE, 70, 72, 73
VISA_SIZE_WITHOUT_CHECKDIGIT, 70, 73
Header < boost/checks/weight.hpp >
BOOST_CHECK_LIMIT_WEIGHTS, 73, 74
C++, 73, 74
pre-conditions, 74
weight, 74
weight_associated_with_pos, 74
_WEIGHT_factory, 73, 74
Header < boost/checks/weighted_sum.hpp >
basic_check_algorithm, 75
C++, 75, 76, 77
compute_checkdigit, 75
compute_multicheckdigit, 75, 76
filter_valid_value_with_pos, 75, 76
operate_on_valid_value, 75, 76
post-conditions, 76
translate_to_valid_value, 75, 76
validate_checksum, 75, 76
weighted_sum_algorithm, 75
History

C++, 17

I

index

- Checks, 2
- Document Conventions, 5
- Status, 4

initial_mod97_weight

- Header < boost/checks/modulus97.hpp >, 63

ISBN

- Header < boost/checks/isbn.hpp >, 44, 45, 46, 47
- Preface, 3
- Rationale, 17
- References, 16
- Starting with Checks, 6

ISBN10_SIZE

- Header < boost/checks/checks_fwd.hpp >, 32
- Header < boost/checks/isbn.hpp >, 44, 46

ISBN10_SIZE_WITHOUT_CHECKDIGIT

- Header < boost/checks/checks_fwd.hpp >, 37
- Header < boost/checks/isbn.hpp >, 44, 47

isbn13_algorithm

- Header < boost/checks/isbn.hpp >, 45

isbn13_check_algorithm

- Header < boost/checks/isbn.hpp >, 44

isbn13_compute_algorithm

- Header < boost/checks/isbn.hpp >, 44

ISSN

- Preface, 3
- Rationale, 17
- References, 16

italic

- Document Conventions, 5

iteration_sense

- Header < boost/checks/basic_check_algorithm.hpp >, 23

iterator

- Header < boost/checks/iteration_sense.hpp >, 47, 48, 49, 50

L

leftmost

- Header < boost/checks/iteration_sense.hpp >, 47

Length

- C++, 14

Letter to digit VIN conversion table

- C++, 9

Luhn

- Error catching summary, 13
- Header < boost/checks/amex.hpp >, 18, 20
- Header < boost/checks/luhn.hpp >, 52, 53, 54
- Header < boost/checks/mastercard.hpp >, 55, 57
- Header < boost/checks/visa.hpp >, 71, 72
- Luhn algorithm, 15
- Modulus 10 algorithm, 15
- Summary of the modular sum algorithms, 16

Luhn algorithm

- C++, 15
- card, 15

- credit, 15
- Luhn, 15
- modulus, 15
- luhn_algorithm
 - Header < boost/checks/amex.hpp >, 19
 - Header < boost/checks/luhn.hpp >, 53
 - Header < boost/checks/mastercard.hpp >, 56
 - Header < boost/checks/visa.hpp >, 71
- luhn_check_algorithm
 - Header < boost/checks/luhn.hpp >, 52
- luhn_compute_algorithm
 - Header < boost/checks/luhn.hpp >, 52
- luhn_sense
 - Header < boost/checks/luhn.hpp >, 52
- luhn_weight
 - Header < boost/checks/luhn.hpp >, 52

M

- make_mod97_weight
 - Header < boost/checks/modulus97.hpp >, 64
- Mastercard
 - Header < boost/checks/checks_fwd.hpp >, 33, 39
 - Header < boost/checks/mastercard.hpp >, 55, 56, 57, 58
 - Starting with Checks, 6
- mastercard_algorithm
 - Header < boost/checks/mastercard.hpp >, 56
- mastercard_check_algorithm
 - Header < boost/checks/mastercard.hpp >, 55
- mastercard_compute_algorithm
 - Header < boost/checks/mastercard.hpp >, 55
- MASTERCARD_SIZE
 - Header < boost/checks/checks_fwd.hpp >, 33
 - Header < boost/checks/mastercard.hpp >, 55, 56, 57
- MASTERCARD_SIZE_WITHOUT_CHECKDIGIT
 - Header < boost/checks/checks_fwd.hpp >, 39
 - Header < boost/checks/mastercard.hpp >, 55, 58
- mod11_check_algorithm
 - Header < boost/checks/modulus11.hpp >, 60
- mod11_compute_algorithm
 - Header < boost/checks/modulus11.hpp >, 60
- mod11_sense
 - Header < boost/checks/modulus11.hpp >, 60
- mod11_weight
 - Header < boost/checks/modulus11.hpp >, 60
- mod97_10_check_algorithm
 - Header < boost/checks/modulus97.hpp >, 63
- mod97_10_compute_algorithm
 - Header < boost/checks/modulus97.hpp >, 63
- mod97_10_sense
 - Header < boost/checks/modulus97.hpp >, 63
- mod97_10_weight
 - Header < boost/checks/modulus97.hpp >, 63
- MOD97_weight_maker
 - Header < boost/checks/modulus97.hpp >, 63, 66, 67
- Modular sum algorithms
 - C++, 14
 - modulus, 14

- pre-conditions, 14
- modulus
 - Extending the library, 9, 10
 - Header < boost/checks/amex.hpp >, 19, 20
 - Header < boost/checks/isbn.hpp >, 44, 45, 46
 - Header < boost/checks/luhn.hpp >, 53, 54
 - Header < boost/checks/mastercard.hpp >, 56, 57
 - Header < boost/checks/modulus10.hpp >, 58, 59, 60
 - Header < boost/checks/modulus11.hpp >, 60, 61, 62, 63
 - Header < boost/checks/modulus97.hpp >, 63, 64, 65, 66
 - Header < boost/checks/visa.hpp >, 71, 72
 - Luhn algorithm, 15
 - Modular sum algorithms, 14
 - Modulus 10 algorithm, 15
 - Modulus 11 algorithm, 15
 - Preface, 3
 - Summary of the modular sum algorithms, 16
- Modulus 10 algorithm
 - C++, 15
 - Luhn, 15
 - modulus, 15
- Modulus 11 algorithm
 - C++, 15
 - modulus, 15
- modulus10_algorithm
 - Header < boost/checks/isbn.hpp >, 45
 - Header < boost/checks/luhn.hpp >, 53
 - Header < boost/checks/modulus10.hpp >, 59
- modulus11_algorithm
 - Header < boost/checks/modulus11.hpp >, 61
- modulus97_algorithm
 - Header < boost/checks/modulus97.hpp >, 65

N

NEXT

- Header < boost/checks/modulus97.hpp >, 63, 67
- next
 - Header < boost/checks/modulus97.hpp >, 64
- no_null_size_contract
 - Header < boost/checks/limits.hpp >, 51

O

operate_on_valid_value

- Extending the library, 9
- Header < boost/checks/amex.hpp >, 19, 20
- Header < boost/checks/basic_check_algorithm.hpp >, 23, 24
- Header < boost/checks/isbn.hpp >, 45, 46
- Header < boost/checks/luhn.hpp >, 53, 54
- Header < boost/checks/mastercard.hpp >, 56, 57
- Header < boost/checks/modulus10.hpp >, 59, 60
- Header < boost/checks/modulus11.hpp >, 61, 62
- Header < boost/checks/modulus97.hpp >, 65, 66
- Header < boost/checks/verhoeff.hpp >, 69, 70
- Header < boost/checks/visa.hpp >, 71, 72
- Header < boost/checks/weighted_sum.hpp >, 75, 76

P

Phonetic

C++, 14

post-conditions

Header < boost/checks/amex.hpp >, 20

Header < boost/checks/basic_check_algorithm.hpp >, 24

Header < boost/checks/isbn.hpp >, 46

Header < boost/checks/luhn.hpp >, 54

Header < boost/checks/mastercard.hpp >, 57

Header < boost/checks/modulus10.hpp >, 60

Header < boost/checks/modulus11.hpp >, 62

Header < boost/checks/modulus97.hpp >, 66

Header < boost/checks/verhoeff.hpp >, 70

Header < boost/checks/visa.hpp >, 72

Header < boost/checks/weighted_sum.hpp >, 76

pre-conditions

Document Conventions, 5

Extending the library, 9, 10

Header < boost/checks/isbn.hpp >, 44

Header < boost/checks/modulus97.hpp >, 64

Header < boost/checks/weight.hpp >, 74

Modular sum algorithms, 14

Preface, 3

References, 16

Preface

C++, 3

card, 3

credit, 3

ISBN, 3

ISSN, 3

modulus, 3

pre-conditions, 3

Verhoeff, 3

VISA, 3

Q

Quickbook

Version Info, 17

R

Rationale

C++, 17

ISBN, 17

ISSN, 17

VISA, 17

reach_one_past_the_end

Header < boost/checks/limits.hpp >, 51, 52

References

C++, 16, 17

Gumm, 16

ISBN, 16

ISSN, 16

pre-conditions, 16

Verhoeff, 16

respect_size_contract

Header < boost/checks/limits.hpp >, 51, 52

rightmost

Header `< boost/checks/iteration_sense.hpp >`, 49

S

snippet

Document Conventions, 5

Starting with Checks

book, 6

C++, 6

card, 6

credit, 6

example, 6

ISBN, 6

Mastercard, 6

version, 6

VISA, 6

Status

C++, 4

index, 4

version, 4

strict_size_contract

Header `< boost/checks/limits.hpp >`, 52

Summary of the modular sum algorithms

C++, 16

Luhn, 16

modulus, 16

Verhoeff, 16

T

translate_checkdigit

Header `< boost/checks/modulus11.hpp >`, 61, 63

translate_to_valid_value

Extending the library, 9

Header `< boost/checks/amex.hpp >`, 19, 20

Header `< boost/checks/basic_check_algorithm.hpp >`, 23, 24

Header `< boost/checks/isbn.hpp >`, 45, 46

Header `< boost/checks/luhn.hpp >`, 53, 54

Header `< boost/checks/mastercard.hpp >`, 56, 57

Header `< boost/checks/modulus10.hpp >`, 59, 60

Header `< boost/checks/modulus11.hpp >`, 61, 62

Header `< boost/checks/modulus97.hpp >`, 65, 66

Header `< boost/checks/verhoeff.hpp >`, 69, 70

Header `< boost/checks/visa.hpp >`, 71, 72

Header `< boost/checks/weighted_sum.hpp >`, 75, 76

translation_exception

Header `< boost/checks/translation_exception.hpp >`, 67

Transposition

C++, 14

Tutorial

C++, 6

type

Header `< boost/checks/basic_check_algorithm.hpp >`, 23, 25

Header `< boost/checks/iteration_sense.hpp >`, 47, 48, 49, 50

Header `< boost/checks/modulus97.hpp >`, 64

Type of errors

C++, 14

U

UPCA_SIZE

Header < boost/checks/checks_fwd.hpp >, 35

Header < boost/checks/upc.hpp >, 67, 68

UPCA_SIZE_WITHOUT_CHECKDIGIT

Header < boost/checks/checks_fwd.hpp >, 41

Header < boost/checks/upc.hpp >, 67, 68

upc_check_algorithm

Header < boost/checks/upc.hpp >, 67

upc_compute_algorithm

Header < boost/checks/upc.hpp >, 67

upc_sense

Header < boost/checks/upc.hpp >, 67

upc_weight

Header < boost/checks/upc.hpp >, 67

V

validate_checksum

Header < boost/checks/amex.hpp >, 19, 20

Header < boost/checks/basic_check_algorithm.hpp >, 23, 24

Header < boost/checks/isbn.hpp >, 45, 46

Header < boost/checks/luhn.hpp >, 53, 54

Header < boost/checks/mastercard.hpp >, 56, 57

Header < boost/checks/modulus10.hpp >, 59, 60

Header < boost/checks/modulus11.hpp >, 61, 63

Header < boost/checks/modulus97.hpp >, 65, 66

Header < boost/checks/verhoeff.hpp >, 69, 70

Header < boost/checks/visa.hpp >, 71, 72

Header < boost/checks/weighted_sum.hpp >, 75, 76

Verhoeff

Acknowledgements, 16

Error catching summary, 13

Header < boost/checks/verhoeff.hpp >, 68, 69, 70

Preface, 3

References, 16

Summary of the modular sum algorithms, 16

Verhoeff Algorithm, 15

Verhoeff Algorithm

C++, 15

equations, 15

Verhoeff, 15

verhoeff_algorithm

Header < boost/checks/verhoeff.hpp >, 69

verhoeff_check_algorithm

Header < boost/checks/verhoeff.hpp >, 68

verhoeff_compute_algorithm

Header < boost/checks/verhoeff.hpp >, 68

verhoeff_iteration_sense

Header < boost/checks/verhoeff.hpp >, 68

version

Checks, 1

Header < boost/checks/basic_checks.hpp >, 28

Starting with Checks, 6

Status, 4

Version Info, 17

Version Info

C++, 17

Quickbook, 17

version, 17

VISA

Header < boost/checks/checks_fwd.hpp >, 36, 42

Header < boost/checks/visa.hpp >, 70, 71, 72, 73

Preface, 3

Rationale, 17

Starting with Checks, 6

visa_algorithm

Header < boost/checks/visa.hpp >, 71

visa_check_algorithm

Header < boost/checks/visa.hpp >, 70

visa_compute_algorithm

Header < boost/checks/visa.hpp >, 70

VISA_SIZE

Header < boost/checks/checks_fwd.hpp >, 36

Header < boost/checks/visa.hpp >, 70, 72, 73

VISA_SIZE_WITHOUT_CHECKDIGIT

Header < boost/checks/checks_fwd.hpp >, 42

Header < boost/checks/visa.hpp >, 70, 73

W

weight

Header < boost/checks/weight.hpp >, 74

weighted_sum_algorithm

Header < boost/checks/modulus10.hpp >, 59

Header < boost/checks/modulus11.hpp >, 61

Header < boost/checks/modulus97.hpp >, 65

Header < boost/checks/weighted_sum.hpp >, 75

weight_associated_with_pos

Header < boost/checks/weight.hpp >, 74