Boost.Checks

Pierre Talbot

Table of Contents

Boost.Checks 2
- Preface 2
- Overview 2
  - Document Conventions 3
- Tutorial 4
  - Starting with Checks 5
  - Extending the library 7
- Common check algorithms 11
- Type of errors 12
  - Alteration 12
  - Transposition 12
  - Length 12
  - Shift 12
  - Phonetic 12
- Modular sum algorithms 12
  - Luhn algorithm 13
  - Modulus 10 algorithm 13
  - Modulus 11 algorithm 13
  - Summary 13
- Acknowledgements 14
- FAQs 14
- References 14
- Rationale 14
- History 15
- Version Info 15

Checks Reference 15
- Header <boost/checks/amex.hpp> 15
- Header <boost/checks/basic_check_algorithm.hpp> 21
- Header <boost/checks/basic_checks.hpp> 25
- Header <boost/checks/checks_fwd.hpp> 33
- Header <boost/checks/ean.hpp> 65
- Header <boost/checks/isbn.hpp> 69
- Header <boost/checks/iteration_sense.hpp> 73
- Header <boost/checks/limits.hpp> 77
- Header <boost/checks/luhn.hpp> 79
- Header <boost/checks/mastercard.hpp> 82
- Header <boost/checks/modulus10.hpp> 87
- Header <boost/checks/modulus11.hpp> 89
- Header <boost/checks/modulus97.hpp> 93
- Header <boost/checks/translation_exception.hpp> 99
- Header <boost/checks/upc.hpp> 100
- Header <boost/checks/verhoeff.hpp> 102
- Header <boost/checks/visa.hpp> 104
- Header <boost/checks/weight.hpp> 109

Header <boost/checks/weighted_sum.hpp> 112
Class Index 114
Typedef Index 116
Function Index 118
Macro Index 118
Index 119

Boost.Checks

Preface

The checks are required in a numerous kind of domains such as the distribution chain (bar codes), the cards number (bank, fidelity cards, ...) and many others. These codes and numbers are often copied or scanned by humans or machines, and both make errors. We need a way to control it and this is why the check digit has been designed. A check digit is aimed to control the validity of a number and catch the maximum mismatched input (we'll detail further the different errors).

Overview

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

Scott McMurray has identifed four fairly distinct types of check:

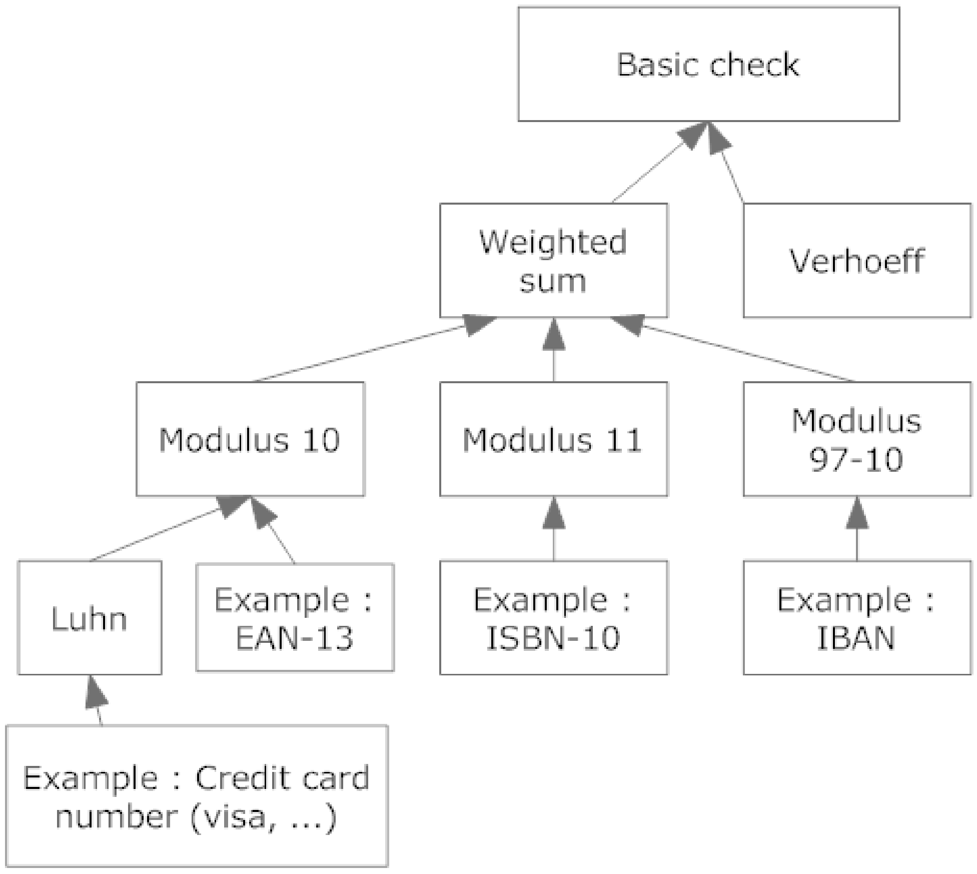
1. ISBN/ISSN/UPC/EAN/VISA/etc, for catching human-entry errors.

2. hash functions as in hash tables, which only care about distribution.

3. checksums like CRC32, for catching data transmission errors.

4. and cryptographic hash functions, the only ones useful against malicious adversaries.

These are primarily for the first category : catching human-entry errors (though it obviously also provides against a mis-scan by a device like a bar code or card reader.) Actually, this library supports four families of check : Modulus 10, Modulus 11, Modulus 97-10 and Verhoeff. A lot of other checks are inherited from these families, the following diagram shows the hierachy used in Boost.Checks:



You can find numerous check algorithm, and this is why this library is design to help you to create your own check. If you are interested, please consult extending the library.

Important

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

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

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

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

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

Document Conventions

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

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

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

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

Admonitions

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 the following 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 learn to extend this library and create our own check functions.

Starting with Checks

There are two main functions for each checks, the first is to validate a sequence: `check_<number>`. The second provides a check digit: `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, Mastercard, 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 val↲
id." << 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_num↲
ber << " 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_num↲
ber );
std::cout << "This is a valid Mastercard number : " << mastercard_credit_card_number << std::endl ;
```

This one 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, such as the mod97-10 algorithm use 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 = "12345678901234567890123456789" ;
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_check↵
digits << "." << 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 : 12345678901234567890123456789 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 a number's size that 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 throwed if one of these errors are encountered. We will use the EAN and ISBN headers:

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

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

The 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 detailled message the exception provided:

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

And with integer array

The old C-array 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 Pro↵
grams 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::com↵
pute_isbn10( isbn10_number ) << std::endl ;
std::cout << "ISBN10 integer version. Check digit : " << boost::checks::compute_isbn10( isbn10_in↵
teger_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 a library is an important factor, especially with this library. In fact, we can't code every existing check functions, this is why we will learn how to extend this library and create your own check functions.

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

We can create the rtn.hpp file. The library support the weighted sum and the modulus 10 algorithm so the work will be ea)sy. We can run through the number from right to left or left to right depending on the weight sequence. We will begin with the leftmost digit because it's more "readable".

```
#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 ;
```

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 com↵
pute_rtn (const check_range& check_seq)
{
  return boost::checks::compute_checkdigit<rtn_compute_algorithm, RTN_SIZE_WITHOUT_CHECKDI↵
GIT> ( check_seq ) ;
}
```

And that's all!

> **Note**
>
> `boost::checks::compute_checkdigit` and `boost::checks::check_sequence` are 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_num↵
ber ) << "." << 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

This second example is quite more complete 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, we can extract few elements:

- The number contains letter 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 valids.
- This use a custom modulus 11 algorithm, so the check digit range is [0..9,X].

The library already have 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. Let's 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_vir↲
tual_value_skipped>
```

The classic modulus 11 algorithm doesn't permit the translation of letters (but the 'x' if it's the check digit). But this number use 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 have the effect of stopping the algorithm) if one of these letter is encountered. The other letter must be transformed following 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 convert a letter into its conversion value, the following function do 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_virtu↵
al_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 calcul 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_virtu↵
al_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 com↵
pute_vin (const check_range& check_seq)
{
  return boost::checks::compute_checkdigit<vin_compute_algorithm, VIN_SIZE_WITHOUT_CHECKDI↵
GIT> ( check_seq ) ;
}
```

Note

This algorithm doesn't support full integer array that are not pre-computed (Ex: (A) 10 -> 1 ; (M) 13 -> 4). It can be an excercise 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_num↵
ber) << std::endl ;
return 0;
```

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 describe the algorithms used in Boost.Checks. A check algorithm is firstly design to:

1. Catch the most errors as possible that a human can do.

2. Minimize the cost of the check digit - fast computation - and the size of the check digit.

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 algorithm and the other checksum algorithms such as CRC or cryptographic hashes, is that we don't analyse the binary content of the number of the lexical values meaning, so "123" is equal 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 catched if the two digits transposed have a different weight and if their values with their weight or the weight of the other digit are not the same.

Length

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

Shift

Phonetic

Modular sum algorithms

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

Note

We may add other algorithms later

Luhn algorithm

Description

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

Note

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

Errors

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

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

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

Modulus 10 algorithm

Description

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

Modulus 11 algorithm

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

Summary

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

• Thanks to Paul A. Bristow who is the mentor of this project for her infinite patience and his wise advices.

FAQs

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

References

1. Routing transit number (RTN)
2. Vehicle Identification Number (VIN)
3. Code 39

Rationale

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

Function parameter

- For more flexibility, this library use 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.

Scope of the project

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

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

Performance

- Performance is not a major objective, but all the current algorithms are implemented with a O(n) complexity.

History

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

Version Info

Last edit to Quickbook file D:\boost-sandbox\SOC\2011\checks\libs\checks\doc\checks.qbk was at 12:25:08 AM on 2011-Aug-22.

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 al↵
gorithm 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 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

`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 overload if you want your algorithm compute more than one check digit (through it works for one check digit too).

Parameters:	`checkdigits`	is the iterator in 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:	`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 <= valid_value_counter < n).
Throws:	std::invalid_argument 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's 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 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).
Returns:	the translation of the current value in the range [0..9].	
Throws:	boost::checks::translation_exception is throwed if the translation of current_value failed. This will automaticaly throws 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.	

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 number.

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 number without its check digit.

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 often 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 functions 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  // Help functions
                                                           // to provide 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 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 overload 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 overload if you want your algorithm compute more than one check digit (through it works for one check digit too).

Parameters:	`checkdigits`	is the iterator in 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 overload 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.

This function should be overload 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 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:	Do nothing. The checksum is inchanged.	

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 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).
Returns:	the translation of the current value in the range [0..9].	
Throws:	boost::checks::translation_exception is throwed if the translation of current_value failed. This will automaticaly throws if the value is not a digit (0 <= i < 11).	

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

Validate the checksum.

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

Parameters:	`checksum`	is the checksum to validate.
Returns:	true.	

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 overload 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, size_t size_expected, typename check_range>
      algorithm::checkdigit< check_range >::type
      compute_checkdigit(const check_range &);
    template<typename algorithm, typename check_range>
      algorithm::checkdigit< check_range >::type
      compute_checkdigit(const check_range &);
    template<typename algorithm, typename size_contract, typename check_range>
      int compute_checksum(const check_range &);
    template<typename algorithm, typename size_contract, typename iterator>
      int compute_checksum(iterator, iterator);
    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, 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_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_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_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_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<typename check_range> bool check_verhoeff(const check_range &);
    template<size_t size_expected, 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<typename check_range, typename checkdigits_iter>
      checkdigits_iter compute_mod97_10(const check_range &, checkdigits_iter);
    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>
      boost::checks::mod11_compute_algorithm::checkdigit< check_range >::type
      compute_modulus11(const check_range &);
    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::upc_compute_algorithm::checkdigit< check_range >::type
      compute_upca(const check_range &);
    template<typename check_range>
      boost::checks::verhoeff_compute_algorithm::checkdigit< check_range >::type
      compute_verhoeff(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::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<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_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_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) don't 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<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_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_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_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_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<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_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_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 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 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.

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.

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.

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.

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 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 overload if you want your algorithm compute more than one check digit (through it works for one check digit too).

Parameters:	`checkdigits`	is the iterator in 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:	`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 <= 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 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:	The current weight multiply 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 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).
Returns:	the translation of the current value in the range [0..9].	
Throws:	boost::checks::translation_exception is throwed if the translation of current_value failed. This will automaticaly throws 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.	

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 of iteration to run through the sequence 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 pass 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 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: A reverse iterator represents one pass 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 have not 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
                                                                  // throwed.
                                                                  // 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 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:	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 == 0).	

Class template strict_size_contract

boost::checks::strict_size_contract — This is a contract class used to verify that a sequence have 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
                                                                  // a
                                                                  // exception
                                                                  // _size_fai
                                                                  // lure
                                                                  // exception
                                                                  // will be
                                                                  // throwed.
                                                                  // 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, false otherwise.	

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 al↵
gorithm 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, num↲
ber_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 overload if you want your algorithm compute more than one check digit (through it works for one check digit too).
   Parameters: `checkdigits` is the iterator in 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 overload 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's 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 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).
Returns:	the translation of the current value in the range [0..9].	
Throws:	boost::checks::translation_exception is throwed if the translation of current_value failed. This will automaticaly throws 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>

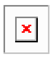
This file provides tools to compute and validate a Mastercard credit card number.

```


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 overload if you want your algorithm compute more than one check digit (through it works for one check digit too).

Parameters:	`checkdigits`	is the iterator in 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:	`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 <= valid_value_counter < n).
Throws:	std::invalid_argument 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's 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 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).
Returns:	the translation of the current value in the range [0..9].	
Throws:	boost::checks::translation_exception is throwed if the translation of current_value failed. This will automaticaly throws 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.	

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, itera↲
tion_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 overload if you want your algorithm compute more than one check digit (through it works for one check digit too).

Parameters:	`checkdigits`	is the iterator in 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 overload 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 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:	The current weight multiply 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 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).
Returns:	the translation of the current value in the range [0..9].	
Throws:	boost::checks::translation_exception is throwed if the translation of current_value failed. This will automaticaly throws 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/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 used with ↵
a modulus 11 algorithm.
    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
                                  // 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 modulus11_algorithm : public boost::checks::weighted_sum_algorithm< mod11_weight, itera↲
tion_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);
};
```

Description

The range of the check digit is [0..10], the tenth element is translated as the letter 'X'.

`modulus11_algorithm` public static functions

1.
```
template<typename checkdigit>
  static checkdigit compute_checkdigit(int checksum);
```

Compute the check digit with a simple modulus 11.

Parameters:	`checksum`	is the checksum used to extract the check digit.
Returns:	The modulus 11 check digit of checksum. 'X' is returned if the check digit is equal to 10.	
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 overload if you want your algorithm compute more than one check digit (through it works for one check digit too).

Parameters:	`checkdigits`	is the iterator in 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 overload 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 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:	The current weight multiply by the current value is added to the checksum.	

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

translate the current value 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 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).
Returns:	the translation of the current value in the range [0..10].	
Throws:	boost::checks::translation_exception is throwed if the translation of current_value failed. The translation will fail if the current value is not a digit (0 <= i < 10). If it's the rightmost digit 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 classic modulus 97 checksum. It provides function 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 serie.
   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 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-computed 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

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 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, itera↵
tion_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

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 overload 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.
	`checksum`	is the checksum used to extract the check digit.
Requires:	checkdigits should have enough reserved place to store the two check digits.	
Postconditions:	The two check digits are stored into checkdigits.	
Returns:	An iterator initialized at one pass 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 overload 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 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:	The current weight multiply 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 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).
Returns:	the translation of the current value in the range [0..9].	
Throws:	boost::checks::translation_exception is throwed if the translation of current_value failed. This will automaticaly throws 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 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.

```
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 overload if you want your algorithm compute more than one check digit (through it works for one check digit too).

Parameters:	`checkdigits`	is the iterator in 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 overload 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 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 <= 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 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).
Returns:	the translation of the current value in the range [0..9].	
Throws:	boost::checks::translation_exception is throwed if the translation of current_value failed. This will automaticaly throws if the value is not a digit (0 <= 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 al↵
gorithm 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  // Help functions
                                                           // to provide same
                                                           // behavior on
                                                           // 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 overload if you want your algorithm compute more than one check digit (through it works for one check digit too).

Parameters:	`checkdigits`	is the iterator in 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 <= 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's 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 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 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).
Returns:	the translation of the current value in the range [0..9].	
Throws:	boost::checks::translation_exception is throwed if the translation of current_value failed. This will automaticaly throws 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.	

Macro VISA_SIZE

VISA_SIZE — This macro defines the size of a Visa number.

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.

Synopsis

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

VISA_SIZE_WITHOUT_CHECKDIGIT
```

Header <boost/checks/weight.hpp>

Provides a template overriden 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 weight 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 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 permit to add to the current checksum the weight multiply 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  // Help functions
                                                           // 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 overload 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.
   ```
   static checkdigits_iter
   compute_multicheckdigit(int checksum, checkdigits_iter checkdigits);
   ```

   Compute the check digit(s) of a sequence.

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

Parameters:	`checkdigits`	is the iterator in 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 overload 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 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:	The current weight multiply 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 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).
Returns:	the translation of the current value in the range [0..9].	
Throws:	boost::checks::translation_exception is throwed if the translation of current_value failed. This will automaticaly throws if the value is not a digit (0 <= i < 11).	

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

Validate the checksum.

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

Parameters:	`checksum`	is the checksum to validate.
Returns:	true.	

Class Index

Symbols

A

amex_algorithm

Header < boost/checks/amex.hpp >, 16

B

basic_check_algorithm
    Header < boost/checks/basic_check_algorithm.hpp >, 22
    Header < boost/checks/verhoeff.hpp >, 103
    Header < boost/checks/weighted_sum.hpp >, 113

C

checkdigit
    Header < boost/checks/basic_check_algorithm.hpp >, 22, 25

I

isbn13_algorithm
    Header < boost/checks/isbn.hpp >, 70
iterator
    Header < boost/checks/iteration_sense.hpp >, 74, 75, 76, 77

L

leftmost
    Header < boost/checks/iteration_sense.hpp >, 74
luhn_algorithm
    Header < boost/checks/amex.hpp >, 16
    Header < boost/checks/luhn.hpp >, 81
    Header < boost/checks/mastercard.hpp >, 84
    Header < boost/checks/visa.hpp >, 106

M

make_mod97_weight
    Header < boost/checks/modulus97.hpp >, 94, 95
mastercard_algorithm
    Header < boost/checks/mastercard.hpp >, 84
modulus10_algorithm
    Header < boost/checks/isbn.hpp >, 70
    Header < boost/checks/luhn.hpp >, 81
    Header < boost/checks/modulus10.hpp >, 88
modulus11_algorithm
    Header < boost/checks/modulus11.hpp >, 91
modulus97_algorithm
    Header < boost/checks/modulus97.hpp >, 96

N

no_null_size_contract
    Header < boost/checks/limits.hpp >, 78

R

rightmost
    Header < boost/checks/iteration_sense.hpp >, 76

S

strict_size_contract
    Header < boost/checks/limits.hpp >, 79

T

translation_exception

Header < boost/checks/translation_exception.hpp >, 100

V
verhoeff_algorithm
    Header < boost/checks/verhoeff.hpp >, 103
visa_algorithm
    Header < boost/checks/visa.hpp >, 106

W
weight
    Header < boost/checks/weight.hpp >, 110
weighted_sum_algorithm
    Header < boost/checks/modulus10.hpp >, 88
    Header < boost/checks/modulus11.hpp >, 91
    Header < boost/checks/modulus97.hpp >, 96
    Header < boost/checks/weighted_sum.hpp >, 113

Typedef Index

Symbols

A
amex_check_algorithm
    Header < boost/checks/amex.hpp >, 15
amex_compute_algorithm
    Header < boost/checks/amex.hpp >, 15

E
ean_check_algorithm
    Header < boost/checks/ean.hpp >, 65
ean_compute_algorithm
    Header < boost/checks/ean.hpp >, 65
ean_sense
    Header < boost/checks/ean.hpp >, 65
ean_weight
    Header < boost/checks/ean.hpp >, 65

I
initial_mod97_weight
    Header < boost/checks/modulus97.hpp >, 93
isbn13_check_algorithm
    Header < boost/checks/isbn.hpp >, 69
isbn13_compute_algorithm
    Header < boost/checks/isbn.hpp >, 69
iteration_sense
    Header < boost/checks/basic_check_algorithm.hpp >, 22

L
luhn_check_algorithm
    Header < boost/checks/luhn.hpp >, 79
luhn_compute_algorithm
    Header < boost/checks/luhn.hpp >, 79
luhn_sense
    Header < boost/checks/luhn.hpp >, 79

luhn_weight
    Header < boost/checks/luhn.hpp >, 79

M

mastercard_check_algorithm
    Header < boost/checks/mastercard.hpp >, 82
mastercard_compute_algorithm
    Header < boost/checks/mastercard.hpp >, 82
mod11_check_algorithm
    Header < boost/checks/modulus11.hpp >, 89
mod11_compute_algorithm
    Header < boost/checks/modulus11.hpp >, 89
mod11_sense
    Header < boost/checks/modulus11.hpp >, 89
mod11_weight
    Header < boost/checks/modulus11.hpp >, 89
mod97_10_check_algorithm
    Header < boost/checks/modulus97.hpp >, 93
mod97_10_compute_algorithm
    Header < boost/checks/modulus97.hpp >, 93
mod97_10_sense
    Header < boost/checks/modulus97.hpp >, 93
mod97_10_weight
    Header < boost/checks/modulus97.hpp >, 93

N

next
    Header < boost/checks/modulus97.hpp >, 94

T

type
    Header < boost/checks/basic_check_algorithm.hpp >, 22, 25
    Header < boost/checks/iteration_sense.hpp >, 74, 75, 76, 77
    Header < boost/checks/modulus97.hpp >, 95

U

upc_check_algorithm
    Header < boost/checks/upc.hpp >, 100
upc_compute_algorithm
    Header < boost/checks/upc.hpp >, 100
upc_sense
    Header < boost/checks/upc.hpp >, 100
upc_weight
    Header < boost/checks/upc.hpp >, 100

V

verhoeff_check_algorithm
    Header < boost/checks/verhoeff.hpp >, 102
verhoeff_compute_algorithm
    Header < boost/checks/verhoeff.hpp >, 102
verhoeff_iteration_sense
    Header < boost/checks/verhoeff.hpp >, 102
visa_check_algorithm
    Header < boost/checks/visa.hpp >, 104
visa_compute_algorithm
    Header < boost/checks/visa.hpp >, 104

Function Index

Symbols

Macro Index

Symbols

_WEIGHT_factory
    Header < boost/checks/weight.hpp >, 109, 111

A

AMEX_SIZE
    Header < boost/checks/amex.hpp >, 15, 17, 18, 20
AMEX_SIZE_WITHOUT_CHECKDIGIT
    Header < boost/checks/amex.hpp >, 15, 19, 21

B

BOOST_CHECK_LIMIT_WEIGHTS
    Header < boost/checks/weight.hpp >, 109, 110, 112

E

EAN13_SIZE
    Header < boost/checks/checks_fwd.hpp >, 36, 39
    Header < boost/checks/ean.hpp >, 65, 66
    Header < boost/checks/isbn.hpp >, 71
EAN13_SIZE_WITHOUT_CHECKDIGIT
    Header < boost/checks/checks_fwd.hpp >, 51, 54
    Header < boost/checks/ean.hpp >, 65, 67
EAN8_SIZE
    Header < boost/checks/checks_fwd.hpp >, 37
    Header < boost/checks/ean.hpp >, 65, 68
EAN8_SIZE_WITHOUT_CHECKDIGIT
    Header < boost/checks/checks_fwd.hpp >, 52
    Header < boost/checks/ean.hpp >, 65, 69

I

ISBN10_SIZE
    Header < boost/checks/checks_fwd.hpp >, 38
    Header < boost/checks/isbn.hpp >, 69, 72
ISBN10_SIZE_WITHOUT_CHECKDIGIT
    Header < boost/checks/checks_fwd.hpp >, 53
    Header < boost/checks/isbn.hpp >, 69, 73

M

MASTERCARD_SIZE
    Header < boost/checks/checks_fwd.hpp >, 42
    Header < boost/checks/mastercard.hpp >, 82, 85, 86
MASTERCARD_SIZE_WITHOUT_CHECKDIGIT
    Header < boost/checks/checks_fwd.hpp >, 57
    Header < boost/checks/mastercard.hpp >, 82, 87
MOD97_weight_maker
    Header < boost/checks/modulus97.hpp >, 93, 98

N

NEXT
Header < boost/checks/modulus97.hpp >, 93, 99

U

UPCA_SIZE
Header < boost/checks/checks_fwd.hpp >, 47
Header < boost/checks/upc.hpp >, 100, 101
UPCA_SIZE_WITHOUT_CHECKDIGIT
Header < boost/checks/checks_fwd.hpp >, 62
Header < boost/checks/upc.hpp >, 100, 102

V

VISA_SIZE
Header < boost/checks/checks_fwd.hpp >, 50
Header < boost/checks/visa.hpp >, 104, 107, 108
VISA_SIZE_WITHOUT_CHECKDIGIT
Header < boost/checks/checks_fwd.hpp >, 65
Header < boost/checks/visa.hpp >, 104, 109

Index

Symbols

_WEIGHT_factory
Header < boost/checks/weight.hpp >, 109, 111

A

acknowledgements
Acknowledgements, 14
Acknowledgements
acknowledgements, 14
Alteration
C++, 12
example, 12
amex_algorithm
Header < boost/checks/amex.hpp >, 16
amex_check_algorithm
Header < boost/checks/amex.hpp >, 15
amex_compute_algorithm
Header < boost/checks/amex.hpp >, 15
AMEX_SIZE
Header < boost/checks/amex.hpp >, 15, 17, 18, 20
AMEX_SIZE_WITHOUT_CHECKDIGIT
Header < boost/checks/amex.hpp >, 15, 19, 21

B

basic_check_algorithm
Header < boost/checks/basic_check_algorithm.hpp >, 22
Header < boost/checks/verhoeff.hpp >, 103
Header < boost/checks/weighted_sum.hpp >, 113
book
Header < boost/checks/isbn.hpp >, 69
Boost.Checks
C++, 1, 2
index, 2

    version, 1
BOOST_CHECK_LIMIT_WEIGHTS
    Header < boost/checks/weight.hpp >, 109, 110, 112

C

C++
    Alteration, 12
    Boost.Checks, 1, 2
    Checks Reference, 15
    Common check algorithms, 11, 12
    Document Conventions, 3, 4
    Error catching summary, 11
    Extending the library, 7, 8
    FAQs, 14
    Header < boost/checks/amex.hpp >, 15, 16, 17, 18, 19, 20, 21
    Header < boost/checks/basic_checks.hpp >, 25, 26, 27, 28, 29, 30, 31, 32, 33
    Header < boost/checks/basic_check_algorithm.hpp >, 21, 22, 23, 24, 25
    Header < boost/checks/checks_fwd.hpp >, 33, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65
    Header < boost/checks/ean.hpp >, 65, 66, 67, 68, 69
    Header < boost/checks/isbn.hpp >, 69, 70, 71, 72, 73
    Header < boost/checks/iteration_sense.hpp >, 73, 74, 75, 76, 77
    Header < boost/checks/limits.hpp >, 77, 78, 79
    Header < boost/checks/luhn.hpp >, 79, 81, 82
    Header < boost/checks/mastercard.hpp >, 82, 84, 85, 86, 87
    Header < boost/checks/modulus10.hpp >, 87, 88, 89
    Header < boost/checks/modulus11.hpp >, 89, 91, 92, 93
    Header < boost/checks/modulus97.hpp >, 93, 94, 95, 96, 97, 98, 99
    Header < boost/checks/translation_exception.hpp >, 99, 100
    Header < boost/checks/upc.hpp >, 100, 101, 102
    Header < boost/checks/verhoeff.hpp >, 102, 103, 104
    Header < boost/checks/visa.hpp >, 104, 106, 107, 108, 109
    Header < boost/checks/weight.hpp >, 109, 110, 111, 112
    Header < boost/checks/weighted_sum.hpp >, 112, 113, 114
    History, 15
    Length, 12
    Letter to digit VIN conversion table, 7
    Luhn algorithm, 13
    Modular sum algorithms, 12
    Modulus 10 algorithm, 13
    Modulus 11 algorithm, 13
    Overview, 2
    Phonetic, 12
    Preface, 2
    Rationale, 14, 15
    References, 14
    Starting with Checks, 5
    Summary of the modular sum algorithms, 13
    Transposition, 12
    Tutorial, 4
    Type of errors, 12
    Version Info, 15
card
    Header < boost/checks/amex.hpp >, 15
    Header < boost/checks/mastercard.hpp >, 82
    Header < boost/checks/visa.hpp >, 104
    Luhn algorithm, 13

Overview, 2
Starting with Checks, 5
checkdigit
Header < boost/checks/basic_check_algorithm.hpp >, 22, 25
Checks Reference
C++, 15
Common check algorithms
C++, 11, 12
example, 11
credit
Header < boost/checks/amex.hpp >, 15
Header < boost/checks/mastercard.hpp >, 82
Header < boost/checks/visa.hpp >, 104
Luhn algorithm, 13
Starting with Checks, 5

D

Document Conventions
C++, 3, 4
Doxygen, 3
example, 3, 4
index, 3
italic, 4
pre-conditions, 3
snippet, 4
Doxygen
Document Conventions, 3

E

EAN13_SIZE
Header < boost/checks/checks_fwd.hpp >, 36, 39
Header < boost/checks/ean.hpp >, 65, 66
Header < boost/checks/isbn.hpp >, 71
EAN13_SIZE_WITHOUT_CHECKDIGIT
Header < boost/checks/checks_fwd.hpp >, 51, 54
Header < boost/checks/ean.hpp >, 65, 67
EAN8_SIZE
Header < boost/checks/checks_fwd.hpp >, 37
Header < boost/checks/ean.hpp >, 65, 68
EAN8_SIZE_WITHOUT_CHECKDIGIT
Header < boost/checks/checks_fwd.hpp >, 52
Header < boost/checks/ean.hpp >, 65, 69
ean_check_algorithm
Header < boost/checks/ean.hpp >, 65
ean_compute_algorithm
Header < boost/checks/ean.hpp >, 65
ean_sense
Header < boost/checks/ean.hpp >, 65
ean_weight
Header < boost/checks/ean.hpp >, 65
Error catching summary
C++, 11
Luhn, 11
Verhoeff, 11
example
Alteration, 12
Common check algorithms, 11

  - Document Conventions, 3, 4
  - Extending the library, 7
  - Header < boost/checks/translation_exception.hpp >, 100
  - Starting with Checks, 5
- Extending the library
  - C++, 7, 8
  - example, 7
  - modulus, 7, 8
  - pre-conditions, 7

F

- FAQs
  - C++, 14

H

- Header < boost/checks/amex.hpp >
  - amex_algorithm, 16
  - amex_check_algorithm, 15
  - amex_compute_algorithm, 15
  - AMEX_SIZE, 15, 17, 18, 20
  - AMEX_SIZE_WITHOUT_CHECKDIGIT, 15, 19, 21
  - C++, 15, 16, 17, 18, 19, 20, 21
  - card, 15
  - credit, 15
  - Luhn, 16, 17
  - luhn_algorithm, 16
  - modulus, 16, 17
  - post-conditions, 17
- Header < boost/checks/basic_checks.hpp >
  - C++, 25, 26, 27, 28, 29, 30, 31, 32, 33
  - version, 30
- Header < boost/checks/basic_check_algorithm.hpp >
  - basic_check_algorithm, 22
  - C++, 21, 22, 23, 24, 25
  - checkdigit, 22, 25
  - iteration_sense, 22
  - post-conditions, 23
  - type, 22, 25
- Header < boost/checks/checks_fwd.hpp >
  - C++, 33, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65
  - EAN13_SIZE, 36, 39
  - EAN13_SIZE_WITHOUT_CHECKDIGIT, 51, 54
  - EAN8_SIZE, 37
  - EAN8_SIZE_WITHOUT_CHECKDIGIT, 52
  - ISBN10_SIZE, 38
  - ISBN10_SIZE_WITHOUT_CHECKDIGIT, 53
  - Mastercard, 42, 57
  - MASTERCARD_SIZE, 42
  - MASTERCARD_SIZE_WITHOUT_CHECKDIGIT, 57
  - UPCA_SIZE, 47
  - UPCA_SIZE_WITHOUT_CHECKDIGIT, 62
  - VISA, 50, 65
  - VISA_SIZE, 50
  - VISA_SIZE_WITHOUT_CHECKDIGIT, 65
- Header < boost/checks/ean.hpp >
  - C++, 65, 66, 67, 68, 69
  - EAN13_SIZE, 65, 66

  - EAN13_SIZE_WITHOUT_CHECKDIGIT, 65, 67
  - EAN8_SIZE, 65, 68
  - EAN8_SIZE_WITHOUT_CHECKDIGIT, 65, 69
  - ean_check_algorithm, 65
  - ean_compute_algorithm, 65
  - ean_sense, 65
  - ean_weight, 65
- Header < boost/checks/isbn.hpp >
  - book, 69
  - C++, 69, 70, 71, 72, 73
  - EAN13_SIZE, 71
  - ISBN, 69, 70, 71, 72, 73
  - ISBN10_SIZE, 69, 72
  - ISBN10_SIZE_WITHOUT_CHECKDIGIT, 69, 73
  - isbn13_algorithm, 70
  - isbn13_check_algorithm, 69
  - isbn13_compute_algorithm, 69
  - modulus, 70, 71
  - modulus10_algorithm, 70
  - post-conditions, 71
  - pre-conditions, 70
- Header < boost/checks/iteration_sense.hpp >
  - C++, 73, 74, 75, 76, 77
  - iterator, 74, 75, 76, 77
  - leftmost, 74
  - rightmost, 76
  - type, 74, 75, 76, 77
- Header < boost/checks/limits.hpp >
  - C++, 77, 78, 79
  - no_null_size_contract, 78
  - strict_size_contract, 79
- Header < boost/checks/luhn.hpp >
  - C++, 79, 81, 82
  - Luhn, 79, 81, 82
  - luhn_algorithm, 81
  - luhn_check_algorithm, 79
  - luhn_compute_algorithm, 79
  - luhn_sense, 79
  - luhn_weight, 79
  - modulus, 81, 82
  - modulus10_algorithm, 81
  - post-conditions, 82
- Header < boost/checks/mastercard.hpp >
  - C++, 82, 84, 85, 86, 87
  - card, 82
  - credit, 82
  - Luhn, 84, 85
  - luhn_algorithm, 84
  - Mastercard, 82, 84, 85, 86, 87
  - mastercard_algorithm, 84
  - mastercard_check_algorithm, 82
  - mastercard_compute_algorithm, 82
  - MASTERCARD_SIZE, 82, 85, 86
  - MASTERCARD_SIZE_WITHOUT_CHECKDIGIT, 82, 87
  - modulus, 84, 85
  - post-conditions, 85
- Header < boost/checks/modulus10.hpp >
  - C++, 87, 88, 89

- modulus, 87, 88, 89
- modulus10_algorithm, 88
- post-conditions, 89
- weighted_sum_algorithm, 88

Header < boost/checks/modulus11.hpp >
- C++, 89, 91, 92, 93
- mod11_check_algorithm, 89
- mod11_compute_algorithm, 89
- mod11_sense, 89
- mod11_weight, 89
- modulus, 89, 91, 92
- modulus11_algorithm, 91
- post-conditions, 92
- weighted_sum_algorithm, 91

Header < boost/checks/modulus97.hpp >
- C++, 93, 94, 95, 96, 97, 98, 99
- initial_mod97_weight, 93
- make_mod97_weight, 94, 95
- mod97_10_check_algorithm, 93
- mod97_10_compute_algorithm, 93
- mod97_10_sense, 93
- mod97_10_weight, 93
- MOD97_weight_maker, 93, 98
- modulus, 93, 96, 97
- modulus97_algorithm, 96
- NEXT, 93, 99
- next, 94
- post-conditions, 97
- pre-conditions, 94
- type, 95
- weighted_sum_algorithm, 96

Header < boost/checks/translation_exception.hpp >
- C++, 99, 100
- example, 100
- translation_exception, 100

Header < boost/checks/upc.hpp >
- C++, 100, 101, 102
- UPCA_SIZE, 100, 101
- UPCA_SIZE_WITHOUT_CHECKDIGIT, 100, 102
- upc_check_algorithm, 100
- upc_compute_algorithm, 100
- upc_sense, 100
- upc_weight, 100

Header < boost/checks/verhoeff.hpp >
- basic_check_algorithm, 103
- C++, 102, 103, 104
- post-conditions, 104
- Verhoeff, 102, 103, 104
- verhoeff_algorithm, 103
- verhoeff_check_algorithm, 102
- verhoeff_compute_algorithm, 102
- verhoeff_iteration_sense, 102

Header < boost/checks/visa.hpp >
- C++, 104, 106, 107, 108, 109
- card, 104
- credit, 104
- Luhn, 106, 107
- luhn_algorithm, 106

- modulus, 106, 107
- post-conditions, 107
- VISA, 104, 106, 107, 108, 109
- visa_algorithm, 106
- visa_check_algorithm, 104
- visa_compute_algorithm, 104
- VISA_SIZE, 104, 107, 108
- VISA_SIZE_WITHOUT_CHECKDIGIT, 104, 109

Header < boost/checks/weight.hpp >
- BOOST_CHECK_LIMIT_WEIGHTS, 109, 110, 112
- C++, 109, 110, 111, 112
- pre-conditions, 112
- weight, 110
- _WEIGHT_factory, 109, 111

Header < boost/checks/weighted_sum.hpp >
- basic_check_algorithm, 113
- C++, 112, 113, 114
- post-conditions, 114
- weighted_sum_algorithm, 113

History
- C++, 15

I

index
- Boost.Checks, 2
- Document Conventions, 3
- Overview, 2

initial_mod97_weight
- Header < boost/checks/modulus97.hpp >, 93

ISBN
- Header < boost/checks/isbn.hpp >, 69, 70, 71, 72, 73
- Overview, 2
- Rationale, 14
- Starting with Checks, 5

ISBN10_SIZE
- Header < boost/checks/checks_fwd.hpp >, 38
- Header < boost/checks/isbn.hpp >, 69, 72

ISBN10_SIZE_WITHOUT_CHECKDIGIT
- Header < boost/checks/checks_fwd.hpp >, 53
- Header < boost/checks/isbn.hpp >, 69, 73

isbn13_algorithm
- Header < boost/checks/isbn.hpp >, 70

isbn13_check_algorithm
- Header < boost/checks/isbn.hpp >, 69

isbn13_compute_algorithm
- Header < boost/checks/isbn.hpp >, 69

ISSN
- Overview, 2
- Rationale, 14

italic
- Document Conventions, 4

iteration_sense
- Header < boost/checks/basic_check_algorithm.hpp >, 22

iterator
- Header < boost/checks/iteration_sense.hpp >, 74, 75, 76, 77

L

leftmost
  Header < boost/checks/iteration_sense.hpp >, 74
Length
  C++, 12
Letter to digit VIN conversion table
  C++, 7
Luhn
  Error catching summary, 11
  Header < boost/checks/amex.hpp >, 16, 17
  Header < boost/checks/luhn.hpp >, 79, 81, 82
  Header < boost/checks/mastercard.hpp >, 84, 85
  Header < boost/checks/visa.hpp >, 106, 107
  Luhn algorithm, 13
  Modulus 10 algorithm, 13
  Summary of the modular sum algorithms, 13
Luhn algorithm
  C++, 13
  card, 13
  credit, 13
  Luhn, 13
  modulus, 13
luhn_algorithm
  Header < boost/checks/amex.hpp >, 16
  Header < boost/checks/luhn.hpp >, 81
  Header < boost/checks/mastercard.hpp >, 84
  Header < boost/checks/visa.hpp >, 106
luhn_check_algorithm
  Header < boost/checks/luhn.hpp >, 79
luhn_compute_algorithm
  Header < boost/checks/luhn.hpp >, 79
luhn_sense
  Header < boost/checks/luhn.hpp >, 79
luhn_weight
  Header < boost/checks/luhn.hpp >, 79

M

make_mod97_weight
  Header < boost/checks/modulus97.hpp >, 94, 95
Mastercard
  Header < boost/checks/checks_fwd.hpp >, 42, 57
  Header < boost/checks/mastercard.hpp >, 82, 84, 85, 86, 87
  Starting with Checks, 5
mastercard_algorithm
  Header < boost/checks/mastercard.hpp >, 84
mastercard_check_algorithm
  Header < boost/checks/mastercard.hpp >, 82
mastercard_compute_algorithm
  Header < boost/checks/mastercard.hpp >, 82
MASTERCARD_SIZE
  Header < boost/checks/checks_fwd.hpp >, 42
  Header < boost/checks/mastercard.hpp >, 82, 85, 86
MASTERCARD_SIZE_WITHOUT_CHECKDIGIT
  Header < boost/checks/checks_fwd.hpp >, 57
  Header < boost/checks/mastercard.hpp >, 82, 87
mod11_check_algorithm
  Header < boost/checks/modulus11.hpp >, 89

mod11_compute_algorithm
Header < boost/checks/modulus11.hpp >, 89
mod11_sense
Header < boost/checks/modulus11.hpp >, 89
mod11_weight
Header < boost/checks/modulus11.hpp >, 89
mod97_10_check_algorithm
Header < boost/checks/modulus97.hpp >, 93
mod97_10_compute_algorithm
Header < boost/checks/modulus97.hpp >, 93
mod97_10_sense
Header < boost/checks/modulus97.hpp >, 93
mod97_10_weight
Header < boost/checks/modulus97.hpp >, 93
MOD97_weight_maker
Header < boost/checks/modulus97.hpp >, 93, 98
Modular sum algorithms
C++, 12
modulus, 12
pre-conditions, 12
modulus
Extending the library, 7, 8
Header < boost/checks/amex.hpp >, 16, 17
Header < boost/checks/isbn.hpp >, 70, 71
Header < boost/checks/luhn.hpp >, 81, 82
Header < boost/checks/mastercard.hpp >, 84, 85
Header < boost/checks/modulus10.hpp >, 87, 88, 89
Header < boost/checks/modulus11.hpp >, 89, 91, 92
Header < boost/checks/modulus97.hpp >, 93, 96, 97
Header < boost/checks/visa.hpp >, 106, 107
Luhn algorithm, 13
Modular sum algorithms, 12
Modulus 10 algorithm, 13
Modulus 11 algorithm, 13
Overview, 2
Summary of the modular sum algorithms, 13
Modulus 10 algorithm
C++, 13
Luhn, 13
modulus, 13
Modulus 11 algorithm
C++, 13
modulus, 13
modulus10_algorithm
Header < boost/checks/isbn.hpp >, 70
Header < boost/checks/luhn.hpp >, 81
Header < boost/checks/modulus10.hpp >, 88
modulus11_algorithm
Header < boost/checks/modulus11.hpp >, 91
modulus97_algorithm
Header < boost/checks/modulus97.hpp >, 96

N
NEXT
Header < boost/checks/modulus97.hpp >, 93, 99
next
Header < boost/checks/modulus97.hpp >, 94

no_null_size_contract
  Header < boost/checks/limits.hpp >, 78

O

Overview
  C++, 2
  card, 2
  index, 2
  ISBN, 2
  ISSN, 2
  modulus, 2
  Verhoeff, 2
  version, 2
  VISA, 2

P

Phonetic
  C++, 12
post-conditions
  Header < boost/checks/amex.hpp >, 17
  Header < boost/checks/basic_check_algorithm.hpp >, 23
  Header < boost/checks/isbn.hpp >, 71
  Header < boost/checks/luhn.hpp >, 82
  Header < boost/checks/mastercard.hpp >, 85
  Header < boost/checks/modulus10.hpp >, 89
  Header < boost/checks/modulus11.hpp >, 92
  Header < boost/checks/modulus97.hpp >, 97
  Header < boost/checks/verhoeff.hpp >, 104
  Header < boost/checks/visa.hpp >, 107
  Header < boost/checks/weighted_sum.hpp >, 114
pre-conditions
  Document Conventions, 3
  Extending the library, 7
  Header < boost/checks/isbn.hpp >, 70
  Header < boost/checks/modulus97.hpp >, 94
  Header < boost/checks/weight.hpp >, 112
  Modular sum algorithms, 12
  Preface, 2
Preface
  C++, 2
  pre-conditions, 2

Q

Quickbook
  Version Info, 15

R

Rationale
  C++, 14, 15
  ISBN, 14
  ISSN, 14
  VISA, 14
References
  C++, 14
rightmost
  Header < boost/checks/iteration_sense.hpp >, 76

S

snippet
  Document Conventions, 4
Starting with Checks
  C++, 5
  card, 5
  credit, 5
  example, 5
  ISBN, 5
  Mastercard, 5
  version, 5
  VISA, 5
strict_size_contract
  Header < boost/checks/limits.hpp >, 79
Summary of the modular sum algorithms
  C++, 13
  Luhn, 13
  modulus, 13
  Verhoeff, 13

T

translation_exception
  Header < boost/checks/translation_exception.hpp >, 100
Transposition
  C++, 12
Tutorial
  C++, 4
type
  Header < boost/checks/basic_check_algorithm.hpp >, 22, 25
  Header < boost/checks/iteration_sense.hpp >, 74, 75, 76, 77
  Header < boost/checks/modulus97.hpp >, 95
Type of errors
  C++, 12

U

UPCA_SIZE
  Header < boost/checks/checks_fwd.hpp >, 47
  Header < boost/checks/upc.hpp >, 100, 101
UPCA_SIZE_WITHOUT_CHECKDIGIT
  Header < boost/checks/checks_fwd.hpp >, 62
  Header < boost/checks/upc.hpp >, 100, 102
upc_check_algorithm
  Header < boost/checks/upc.hpp >, 100
upc_compute_algorithm
  Header < boost/checks/upc.hpp >, 100
upc_sense
  Header < boost/checks/upc.hpp >, 100
upc_weight
  Header < boost/checks/upc.hpp >, 100

V

Verhoeff
  Error catching summary, 11
  Header < boost/checks/verhoeff.hpp >, 102, 103, 104
  Overview, 2
  Summary of the modular sum algorithms, 13

verhoeff_algorithm
    Header < boost/checks/verhoeff.hpp >, 103
verhoeff_check_algorithm
    Header < boost/checks/verhoeff.hpp >, 102
verhoeff_compute_algorithm
    Header < boost/checks/verhoeff.hpp >, 102
verhoeff_iteration_sense
    Header < boost/checks/verhoeff.hpp >, 102
version
    Boost.Checks, 1
    Header < boost/checks/basic_checks.hpp >, 30
    Overview, 2
    Starting with Checks, 5
    Version Info, 15
Version Info
    C++, 15
    Quickbook, 15
    version, 15
VISA
    Header < boost/checks/checks_fwd.hpp >, 50, 65
    Header < boost/checks/visa.hpp >, 104, 106, 107, 108, 109
    Overview, 2
    Rationale, 14
    Starting with Checks, 5
visa_algorithm
    Header < boost/checks/visa.hpp >, 106
visa_check_algorithm
    Header < boost/checks/visa.hpp >, 104
visa_compute_algorithm
    Header < boost/checks/visa.hpp >, 104
VISA_SIZE
    Header < boost/checks/checks_fwd.hpp >, 50
    Header < boost/checks/visa.hpp >, 104, 107, 108
VISA_SIZE_WITHOUT_CHECKDIGIT
    Header < boost/checks/checks_fwd.hpp >, 65
    Header < boost/checks/visa.hpp >, 104, 109

W

weight
    Header < boost/checks/weight.hpp >, 110
weighted_sum_algorithm
    Header < boost/checks/modulus10.hpp >, 88
    Header < boost/checks/modulus11.hpp >, 91
    Header < boost/checks/modulus97.hpp >, 96
    Header < boost/checks/weighted_sum.hpp >, 113