```
#ifndef CPP_TOOLS_POD_STRING_H
#define CPP_TOOLS_POD_STRING_H

#include <string>
#include <cstring>
#include <cstdint>
#include <algorithm>
#include <type_traits>
#include <array>

namespace cpp_tools
{
#if(  __SIZEOF_INT128__  == 16)
        using int128_t =  __int128_t;
        using uint128_t =  __uint128_t;
#else
        using int128_t =  std::array<std::int64_t, 2>;
        using uint128_t = std::array<, 2>;
#endif
namespace PodStringImpl{

template<std::size_t MaxSize > struct ImplType;
template<> struct ImplType<1> {using type =  std::uint16_t;};
template<> struct ImplType<3> {using type =  std::uint32_t;};
template<> struct ImplType<7> {using type =  std::uint64_t;};
template<> struct ImplType<15> {using type =  uint128_t;};
template<> struct ImplType<23> {using type =  std::array<std::int64_t, 3>;};
template<> struct ImplType<31> {using type =  std::array<std::int64_t, 4>;};



template<std::size_t MaxSize>  struct PodString {

 using value_type = typename ImplType<MaxSize>::type;
 static_assert(MaxSize < sizeof(value_type), " Coding error. maxSize >=
 sizeof(value_type)" );
 static constexpr bool  IsArithmetic() { return (alignof(value_type)
 ==sizeof(value_type))   ; }
 enum{MAX_SIZE = MaxSize};


 PodString(value_type value)  noexcept  : m_value{value} {}
 PodString(const  char* value, std::size_t len) noexcept{
  if(len < sizeof(value_type)) {
              memcpy(&m_value, value, len);
        }
  }

 PodString(const  char* c_str) noexcept : PodString(c_str,  strlen(c_str)  ) { }
 PodString(const  std::string &str) noexcept : PodString(str.c_str(), str.length()){ }

 bool IsValid() const noexcept{
        return ( m_value != Invalid()  );
  }

 operator  bool()  const noexcept {
        return   IsValid();
 }

 operator  value_type()  const noexcept {
        return  m_value;
 }
 operator  const char* () const noexcept {
  return reinterpret_cast<const char *>  (&m_value) ;
 }
 operator std::string () const noexcept {
       return reinterpret_cast<const char *>  (&m_value) ;
 }
private:
  static constexpr value_type  NullValue()  { return {} ;  }
  static constexpr value_type  Invalid() { return NullValue(); }
private:
  value_type  m_value{ };

};
}//PodStringImpl
using String_7 = PodStringImpl::PodString<7> ;
using String_15 = PodStringImpl::PodString<15> ;
using String_23 = PodStringImpl::PodString<23> ;
using String_31 = PodStringImpl::PodString<31> ;

}
#endif  /* CPP_TOOLS_POD_STRING_H */

/*
using namespace cpp_tools;
int main()
{

    String_7 value24 ("12345678");
  std::string v24 = value24;
  bool bv24 = value24;

 String_7 value1 ("SPY", 3);
 std::string v11 = value1;
  const char* v12 = value1;
  std::uint64_t v13 = value1;

  String_7 value2 ("SPY");
  String_7 value3 (std::string("SPY"));

  bool b1 = value1.IsValid(); bool b2 = value2.IsValid(); bool b3 = value3.IsValid();
  bool isArv1 = value1.IsArithmetic();

  String_15 value8("1"); String_15 value9("3");
  bool isArv8 = value8.IsArithmetic();
  String_15::value_type  im1 = value8;
  String_15::value_type im2 = value9;
  bool con12 = (im1 > im2);
  auto im3 = im1 + im2;

  String_23 value23("123456789123456789");
 std::string v23 = value23;
 bool b23 = value23;
bool isArv7 = String_7::IsArithmetic();
bool isArv15 = String_15::IsArithmetic();
bool isArv23 = String_23::IsArithmetic();


 return 0;
}
*/

#include <type_traits>
#include <string>
#include <exception>
#include <boost/lexical_cast.hpp>
#include "boost/date_time/gregorian/gregorian.hpp"
#include "boost/date_time/posix_time/posix_time.hpp"


#ifndef CPP_TOOLS_CAST_H_
#define CPP_TOOLS_CAST_H_

namespace cpp_tools
{
   enum class CastStatus {
      OK,
      UNNOWN_ERROR
   };

  enum class OnCast {
     DO_CAST,
     DONT_DO_CAST,
     SET_DEFAULT,
     THROW
   };
   /*
    using ReturnT = std::pair<To, std::exception_ptr>
    template <typename To,  typename From> ReturnT Cast( From from )

    cast
    1. from arithmetic to arithmetic,
    2. from arithmetic to std::string,
    3. from string (cold be const char*, char* or std::string) to arithmetic

    ReturnT rt;
    if valid -   rt.first valid value and rt.second is nullptr
    if invalid - rt.first equals To() and  rt.second pointer to the cast exception
 examples
std::string s1("25.3");   std::string s2("25.x");
// std::pair<To, std::exception_ptr>
auto u1 = Cast<double, std::string>(s1);
auto u2 = Cast<double, std::string>(s2);
auto u3 = Cast<std::string, int>(222);
auto u4 = Cast<int, double>(5.2);
auto u5 = Cast<unsigned int, double>(-5.2);
 */

   // arithmetic to arithmetic
template <typename To,  typename From> inline
    typename std::enable_if<
    std::is_arithmetic<To>::value &&
    std::is_arithmetic<From>::value,
    std::pair<To, std::exception_ptr> >::type
Cast( From from )
{
   try{
      return { boost::numeric_cast<To>(from),  nullptr };
   }catch(...)
   {
      return{To(), std::current_exception()} ;
   }
}

// arithmetic to std::string
template <typename To, typename From> inline
typename std::enable_if<
   std::is_same<To,std::string>::value &&
   std::is_arithmetic<From>::value,
   std::pair<To, std::exception_ptr> >::type
Cast( const From &from )
{
   return { boost::lexical_cast<To>(from),  nullptr };
}

// string (cold be const char*, char* or std::string) to arithmetic
template <typename To, typename From> inline
typename std::enable_if<
   std::is_arithmetic<To>::value &&
   std::is_convertible<From,std::string>::value,
   std::pair<To, std::exception_ptr> >::type
Cast( const From &from )
{
   try{
      return { boost::lexical_cast<To>(from), nullptr };
   }catch(...)
   {
      return { To(), std::current_exception(), } ;
   }
}
```

```
//////

template <typename To> inline  std::exception_ptr Cast(const std::string &from,  To
&to )
{
   static_assert(std::is_arithmetic<To>::value, "Wrong cast. Can be casted only to an
   arithmetic type. ");
   try{
      to = boost::lexical_cast<To>(from);
   }catch(...)
   {
      return std::current_exception();
   }
   return nullptr;
}

 template<typename To>
inline std::exception_ptr  StrToArithmetic(const std::string &from, To &to, OnCast
onEmpty = OnCast:: SET_DEFAULT, OnCast onError = OnCast:: SET_DEFAULT)
{
    if(from.empty() ) // not an error
    {
         switch(onError)
         {
            case cpp_tools::OnCast::SET_DEFAULT:
            {
               to = To();
               break;
            }
         };
         return nullptr;
    }

   std::exception_ptr err = cpp_tools::Cast(from,to);
   if(err)
   {
    switch(onError)
    {
       case cpp_tools::OnCast::SET_DEFAULT:
       {
          to = To();
          break;
       }
       case cpp_tools::OnCast::THROW:
       {
          std::rethrow_exception (err);
       }
    };
   }

   return err;
 }
 template< typename ToType, typename FromType >  inline ToType StorageCast(FromType from)
 {
   static_assert(
                (sizeof(FromType) ==  sizeof(ToType))
            &&  (alignof(FromType) == alignof(ToType))
           ,"Uknown cast");
        return *(reinterpret_cast<ToType *> (&from)) ;
 }


template<typename FromType> struct DateTime_IntTraits;
template<> struct DateTime_IntTraits<boost::gregorian::date >{ using ToType =
std::uint32_t;    };
template<> struct DateTime_IntTraits<std::uint32_t >{ using ToType =
boost::gregorian::date;    };
template<> struct DateTime_IntTraits< boost::posix_time::ptime>{ using ToType =
std::uint64_t;  };
template<> struct DateTime_IntTraits<std::uint64_t  >{ using ToType =
boost::posix_time::ptime; };


 template< typename FromType >  inline typename DateTime_IntTraits<FromType>::ToType
 DateTimeIntCast (FromType from)
 {
     using ToType = typename DateTime_IntTraits<FromType>::ToType;
     return StorageCast<ToType>(from);
 }



}


#endif  /* CPP_TOOLS_CAST_H_*/

/*// testing DateTimeIntCast
 #include "CPPTools/Cast.h"
 #include <string>

int main()
{
    using namespace cpp_tools;
    using namespace boost::posix_time;
    using namespace boost::gregorian;
    date d0(from_undelimited_string("20140307"));
    std::uint32_t  di = DateTimeIntCast(d0);
    date dd =  DateTimeIntCast(di);
    std::string ds = to_iso_string(dd);

     ptime t0(from_iso_string("20140307T235959"));
    std::uint64_t  ti = DateTimeIntCast(t0);
    ptime tt =  DateTimeIntCast(ti);
    std::string ts = to_iso_string(tt);

  return 0;
}
*/
```