```
#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
{
  template<typename T, std::size_t NUM, typename ValueType,  typename EnabledType = void>
  struct PodString;

template<typename T,  std::size_t NUM > struct PodString <T, NUM, std::array<T, NUM>,
          typename std::enable_if<
              std::is_pod< std::array<T, NUM> >::value
               && (NUM > 1)
             >::type
        >
{
    using element_type = T;
    using value_type = std::array<T, NUM>;
    enum {
        MaxSize = sizeof(value_type) -1,
        Rank = std::rank<value_type>::value,
        Alignment = alignof(value_type)
    };

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

template<typename T> struct PodString <T, 1, T,
          typename std::enable_if<
               std::is_pod<T>::value
              >::type
        >
{
    using element_type = T;
    using value_type = T;
    enum {
        MaxSize = sizeof(value_type) -1,
        Rank = std::rank<value_type>::value,
        Alignment = alignof(value_type)
    };

    PodString(value_type value) noexcept : m_value{value} {}
    PodString(const char* value, std::size_t len) noexcept{
            if(len <= MaxSize) {
                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 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 {  };
};
```

```
template<typename T> using ShortPodString = PodString<T, 1, T>;

// Pod string types implemented by uint_xx_t types.  The strings types can be implicitly
cast to and from  uint_xx_t types.
// The strings types have the same size and alignment as corresponding uint_xx_t types.
using String_7 =  PodString<std::uint8_t, 1, std::uint8_t>;
using String_15 = PodString<std::uint16_t, 1, std::uint16_t>;
using String_31 = PodString<std::uint32_t, 1, std::uint32_t>;
using String_63 = PodString<std::uint64_t, 1, std::uint64_t>;

//gcc specific __int128_t
//Pod string types implemented by std::array<uint_xx_t, n> types.  The strings types can
be implicitly cast to and from  std::array<uint_xx_t, n> types.
// The strings types have size = sizeof(uint_xx_t) * n.   as corresponding uint_xx_t
types.
/// The strings types have the same alignment as corresponding uint_xx_t types.
#if(__SIZEOF_INT128__ == 16)
using int128_t = __int128_t;
using uint128_t = __uint128_t;
using String_127 = PodString<uint128_t, 1, uint128_t >;
#else
using String_127 = PodString<std::uint64_t, 2, std::array<std::uint64_t, 2> >;
#endif



}// namespace cpp_tools


#endif  /* CPP_TOOLS_POD_STRING_H */


/*
 int main(int argc, char *argv[]) {


    using namespace cpp_tools;
    String_63 value1 ("SPY", 3);
    String_63 value2 ("SPY");
    String_63 value3 (std::string("SPY"));
    bool b1 = value1.IsValid();  bool b2 = value2.IsValid();  bool b3 = value3.IsValid();

    std::string v11 = value1; const char* v12 = value1; uint64_t v13 = value1;

    std::string v21=value2;
    std::string v31=value3;


    ShortPodString<std::uint64_t> value4("1234567");
    std::string v4 = value4;
    ShortPodString<std::uint64_t> value5("12345678");
    std::string v5 = value5;
    bool b4 = value4.IsValid(); bool b5 = value5.IsValid();
    String_127  value6("123456789");
    std::string v6 = value6;
    String_127  value7("12345678123456789");
    bool b6 = value6.IsValid();
    bool b7 = value7.IsValid();
    String_127  value8("1"); String_127  value9("3");
    int max_s = String_127::MaxSize;
    int Rank = String_127::Rank;
    int Alignment = String_127::Alignment;

    String_127::value_type im1 = value8;
    String_127::value_type im2 = value9;
    bool con12 = (im1 >  im2 );
    auto im3 = im1 + im2;


    return 0;
}
*/
```