Types and Byte Order

Types for fixed-size integers, byte order conversions and the Any/DynamicAny types.
Overview

- Fixed-Size Integer Types
- Byte Order (Conversions)
- The Any Type
- The DynamicAny Type
POCO defines types for fixed-size integers

#include "Poco/Types.h"
(automatically included by Poco/Foundation.h)

Poco::Int8, Poco::Int16, Poco::Int32, Poco::Int64
Poco::UInt8, Poco::UInt16, Poco::UInt32, Poco::UInt64

Poco::IntPtr, Poco::UIntPtr: integer with same size as a pointer type (32 or 64 bits)

For portable code, always use these types if you need a fixed-size integer.
POCO has two macros to determine the size of the long type and of pointer types.

- `POCO_PTR_IS_64_BIT` macro is defined if pointers are 64 bits.
- `POCO_LONG_IS_64_BIT` macro is defined if long's are 64 bits.
Byte Order

- POCO has facilities to deal with byte order issues.
- Macros to determine the current host's byte order:
  - `POCO_ARCH_LITTLE_ENDIAN`
    macro is defined if architecture is little endian
  - `POCO_ARCH_BIG_ENDIAN`
    macro is defined if architecture is big endian
Byte Order Conversions

- Class `Poco::ByteOrder` provides static methods for byte order conversions.
- `#include "Poco/ByteOrder.h"
- All functions are available for `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64` and `UInt64`
- `IntXX flipBytes(IntXX value)` changes byte order from big to little endian and vice versa
Byte Order Conversions (cont'd)

- `IntXX toBigEndian(IntXX value)` converts from host byte order to big endian
- `IntXX toLittleEndian(IntXX value)` converts from host byte order to little endian
- `IntXX fromBigEndian(IntXX value)` converts from big endian to host byte order
- `IntXX fromLittleEndian(IntXX value)` converts from little endian to host byte order
 Byte Order Conversions (cont'd)

- `IntXX toNetwork(IntXX value)` converts from host byte order to network byte order.
- `IntXX fromNetwork(IntXX value)` converts from network byte order to host byte order.
- Network byte order is big endian.
- All methods are defined as inline functions and are very efficient. Unnecessary conversions will be optimized away by the compiler.
#include "Poco/ByteOrder.h"
#include <iostream>

using Poco::ByteOrder;
using Poco::UInt16;

int main(int argc, char** argv)
{
    #ifdef POCO_ARCH_LITTLE_ENDIAN
        std::cout << "little endian" << std::endl;
    #else
        std::cout << "big endian" << std::endl;
    #endif

    UInt16 port = 80;
    UInt16 networkPort = ByteOrder::toNetwork(port);

    return 0;
}
The Any Type

- #include "Poco/Any.h"
- An instance of Poco::Any can hold a value of any built-in or user-defined type.
- Poco::Any supports value semantics.
- The value can be extracted in a type-safe manner.
- The type of the value must be known in order to extract it.
- The Poco::AnyCast() and Poco::RefAnyCast() function templates are used to extract values.
```cpp
#include "Poco/Any.h"
#include "Poco/Exception.h"

using Poco::Any;
using Poco::AnyCast;
using Poco::RefAnyCast;

int main(int argc, char** argv) {
    Any any(42);

    int i = AnyCast<int>(any); // okay
    int& ri = RefAnyCast<int>(any); // okay

    try {
        short s = AnyCast<short>(any); // throws BadCastException
    } catch (Poco::BadCastException&) {
    }
    return 0;
}
```
The DynamicAny Type

> #include "Poco/DynamicAny.h"

> An instance of Poco::DynamicAny can hold a value of any type for which a DynamicAnyHolder specialization is available.

> Poco::DynamicAny supports value semantics.

> The value can be extracted in a type-safe manner.

> Safe implicit and explicit conversions to various types (standard types, std::string) are supported (ranges are checked).
DynamicAny: convert() vs. extract()

- T convert();
- void convert(T& val);
- operator T ()
  - all return a copy
  - autoconversion
  - slower than Any
- const T& extract();
- returns a const ref
- no autoconversion
- as fast as Any
DynamicAny – Conversion Rules

- Data loss is forbidden for numeric values:
  - value < 0 will never be converted to an unsigned
  - value needing x bits, will never be converted to a smaller bit range (e.g.: value = 2000, needs 16bits, conversion to 8 bit is forbidden)

- Precision loss from int to float and back is allowed

- String truncation is allowed (string to single char)
```cpp
#include "Poco/DynamicAny.h"
#include "Poco/Exception.h"

using Poco::DynamicAny;

int main(int argc, char** argv)
{
    DynamicAny any(42);

    int i = any;
    std::string s(any.convert<std::string>());
    any.convert(s); // or without the need to cast
    const int& ri(any.extract<int>());
    short s = any;

    try
    {
        short s = any.extract<short>(); // throws BadCastException
    }
    catch (Poco::BadCastException&)
    {
    }
    return 0;
}
```
Any vs. DynamicAny

- **Any** can hold any type, but you have to know the type to get it out again.

- **DynamicAny** can hold any type for which a `DynamicAnyHolder` specialization exists.

- Implicit and explicit conversions are restricted to a fixed set of types (standard types plus `std::string`).