C++ Internet Protocol Classes

Document Number: N3477=12-0167
Date: 2012-10-28
Authors: Aleksandar Fabijanic <alex@pocoproject.org>
Günter Obiltschnig <guenter.obiltschnig@appinf.com>

Content

1. Revision History

- Oct 10 2012: created
- Oct 28 2012: apply suggested changes from SG4 meeting in Portland, OR

2. Motivation and Scope

Given the pervasiveness of networking and Internet, there is a pressing need for standard networking support in C++ language. This proposal addresses the concerns of IP addresses; it is aimed to be one of the fundamental building blocks and steps toward the Standard C++ Network Library goal.

This proposal is loosely based on IP address implementation in the Net library of the C++ Portable Components [POCO] framework. POCO framework has been open sourced in 2005 by its principal sponsor (Applied Informatics Software Engineering GmbH); it is currently licensed under Boost Software License and used in production by numerous commercial entities as well as other open source projects [POCOU].

The scope of the proposal is description of functionality, interfaces and usage examples for a class storing and manipulating Internet Protocol (IPv4 and v6) addresses.

At the time of this proposal writing, there are two Internet Protocol versions in practical use: IPv4 and IPv6. Rationale for a single class is based on [RFC4038-6.1] recommendation and SG4 (Networking) discussions.

Additionally, since CIDR [RFC4632] has superseded classful IPv4 address space definition, it is used in this proposal as basis for IPv4 portion interface and functionality.

3. Prefix

Prefix represents the length of the bit pattern used to mask portion of the address. IPv6 prefixes are contiguous and there is no concept of subnet mask like the one in IPv4 specification [RFC950]. Since the prefix concept is common to both IPv6 and IPv4 CIDR notation, only the prefix class is included in this proposal. Omission of the subnet mask concept precludes support for non-contiguous IPv4 subnet masks which, although possible [RFC950], are practically non-existent and therefore not included in the scope of this proposal. With this proposal, it is still possible to obtain IPv4 (contiguous) subnet mask using prefix string conversion with appropriate formatting settings (see 3.1.1. and 3.2. for details).

3.1. Usage Examples

3.1.1. IPv4

```
prefix pref(24);
assert(pref.length() == 16);
assert (pref.to_string(prefix::dec, '.', 32, 8) == "255.255.0.0");
```

3.1.2. IPv6

```
prefix pref(64);
assert (pref.length() == 64);
assert (pref.to_string() ==
"ffff ffff ffff ffff 0000 0000 0000 0000");
pref = 48;
assert (pref.to_string(prefix::bin) ==
"1111111111111111 1111111111111111 1111111111111111 0000000000000000 "
"0000000000000000 0000000000000000 0000000000000000 0000000000000000");
```

3.2. Header <prefix> synopsis

```
namespace std {
 namespace net {
 class prefix
 {
 public:
   static const unsigned int size = 128;
     /// Size in bits.

   typedef std::bitset<size> type;
     /// Underlying storage type.

   enum notation
     /// Prefix display notation.
     /// Used for conversions to string.
   {
     bin,
     oct,
     dec,
     hex
   };

   prefix(unsigned int length);
     /// Creates subnet prefix.

   prefix(const prefix& pref);
     /// Creates an ip_address by copying another one.

   prefix(prefix&& pref) noexcept;
     /// Creates an ip_address by moving another one.

   prefix& operator=(prefix pref);
     /// Copies pref to this address.

   prefix& operator=(prefix&& pref) noexcept;
     /// Moves pref to this address.

   ~prefix();
     /// Destroys subnet prefix.

   prefix& operator = (unsigned int length);
     /// Assigns the length to the prefix.

   unsigned int length() const;
  /// Returns prefix length.

   std::string to_string(notation n = hex,
     char separator = ' ',
     unsigned int bits = size,
     unsigned int grouping = 16) const;
```

```
      /// Returns string representation of subnet prefix.
      /// Depending on notation, representation can be in binary,
      /// decimal or hexadecimal notation. Using decimal notation
      /// with '.' as separator and 32 as size, it is possible to
      /// obtain IPv4 subnet mask representation (e.g. "255.255.0.0")

  std::wstring to_string() const;
    /// Returns wide string representation of subnet prefix.

private:
  type _prefix;
};
}} // namespace std::net
```

4. IP Address

The ip_address class represents an Internet Protocol address.
The address can belong to either IPv4 or IPv6 address family.

Relational operators (==, !=, <, <=, >, >=) are supported. Relational operators compare address bytes, so an IPv4 address is never equal to an IPv6 address, even if the IPv6 address is IPv4 compatible and the addresses are logically the same.

For underlying IP address bit manipulation, bitwise operators (&, |, ^, ~) are supported.

4.1 Usage Examples

4.1.1. IPv4:

```
ip_address ip4("127.0.0.1");
assert (ip4.family() == ip_address::ipv4);
assert (ip4.to_string() == "127.0.0.1");
assert (ip4.is_loopback());

ip_address ip4("192.168.0.51");
ip_address::addr addr = ip4.address();
assert (addr.v4[0] == 192);
assert (addr.v4[1] == 168);
assert (addr.v4[2] == 0);
assert (addr.v4[3] == 51);

prefix pref(24);
ip_address net = ip4 & pref;
assert(net.to_string() == "192.0.0.0");
ip_address host("0.0.0.51");
assert((net | host) == addr);
assert((~mask).to_string() == "0.0.0.255");
```

4.1.2. IPv6

```
ip_address ip6("2001:0DB8:0123:1:0:0:0:1");
assert (ip6.family() == ip_address::ipv6);
// note compression and lowercasing:
assert (ip6.to_string() == "2001:db8:123:1::1");
ip_address::addr addr6 = ip6.address();
assert (addr6.v6[0x0] == 0x20);
assert (addr6.v6[0x1] == 0x01);
assert (addr6.v6[0x2] == 0x0D);
assert (addr6.v6[0x3] == 0xB8);
assert (addr6.v6[0x4] == 0x01);
assert (addr6.v6[0x5] == 0x23);
```

```
 assert (addr6.v6[0x6] == 0x00);
 assert (addr6.v6[0x6] == 0x01);
 assert (addr6.v6[0x7] == 0x00);
 // ... zeros
 assert (addr6.v6[0xE] == 0x00);
 assert (addr6.v6[0xF] == 0x01);
```

4.2. Header <ip_address> synopsis

```
namespace std {
 namespace net {

 class ip_address
 {
 public:
   union addr
   {
     struct
     {
       const unsigned char& operator [] (unsigned int index);
         /// Access address byte at position index.
         /// Throws std::range_error if index is larger than 3.

     private:
       unsigned char byte[4];
     } v4;
     struct
     {
       const unsigned char& operator [] (unsigned int index)
         /// Access address byte at position index.
         /// Throws std::range_error if index is larger than 15.

     private:
       unsigned char byte[16];
     } v6;
   };
   enum family
     /// Possible address families for IP addresses.
   {
     ipv4,
     ipv6
   };

   enum ipv6_scope
     /// IPv6 scopes. For unicast and anycast addresses,
     /// only link_local, site_local and global apply.
     /// All values apply to multicast addresses.
     /// Unspecified (wildcard) IP address has no scope.
   {
     // reserved           = 0x0,
        interface_local    = 0x1,
        link_local         = 0x2,
     // reserved           = 0x3,
```

```
    admin_local        = 0x4,
    site_local         = 0x5,
  // unassigned         = 0x6,
  // unassigned         = 0x7,
    organization_local = 0x8,
  // unassigned         = 0x9,
  // unassigned         = 0xA,
  // unassigned         = 0xB,
  // unassigned         = 0xC,
  // unassigned         = 0xD,
    global             = 0xE,
  // reserved           = 0xF
};

ip_address();
  /// Creates an unspecified (wildcard, zero) IPv6 ip_address.

explicit ip_address(family family);
  /// Creates a unspecified (wildcard, zero) ip_address for the given
  /// address family.

ip_address(const std::string& addr);
  /// Creates an IPAddress from the string containing
  /// an IP address in presentation format (dotted decimal
  /// for IPv4, hex string for IPv6).
  /// Depending on the format of addr, IPv4 or an IPv6 address is
  /// created. See to_string() for details on the supported formats.
  /// Throws std::invalid_argument if the address cannot be parsed.

ip_address(const ip_address& addr);
  /// Creates an ip_address by copying another one.

ip_address(ip_address&& addr) noexcept;
  /// Creates an ip_address by moving another one.

ip_address& operator=(const ip_address& addr);
  /// Assigns addr to this address by copying.

ip_address& operator=(ip_address&& addr) noexcept;
  /// Assigns addr to this address by moving.

~ip_address() noexcept;
  /// Destroys the IPAddress.

ip_address& operator = (const ip_address& addr);
  /// Assigns an IPAddress.

void swap(ip_address& addr);
  /// Swaps the IPAddress with another one.

const addr& address() const;
  /// Returns const reference to address bytes;
  /// addr::v4 and addr::v6:: operator[] is used to access address bytes.
```

```
family family() const;
  /// Returns the address family (ipv4 or ipv6) of the address.

ipv6_scope scope() const;
  /// Returns the IPv6 scope identifier of the address. Returns 0 if
  /// the address is an IPv4 address, or the address is an IPv6 address
  /// but does not have a scope identifier (such as e.g. unspecified
  /// address in most cases).
  /// For unicast and anycast addresses, scope is determined from
  /// dedicated prefixes (fe80::/10 is link_local,
  /// fec0::/10 is site_local,
  /// everything else is global. Loopback address (::1) is a special
  /// link-local address. Unspecified address (::) typically does not
  /// have scope (although, implementation is not prohibited from setting
  /// it in the sense of "any address in a scope"). IPv4-mapped IPv6
  /// addresses have global scope. However, an implementation is not
  /// prohibited from using them as if they had other scope (e.g.
  /// 169.254.0.0/16 being considered link-local when IPv6-mapped).
  /// For details, see [RFC4007].

static std::string zone(unsigned int index) const;
  /// Returns the zone for the link-local ip_address.
  /// Index parameter represents the network interface index.
  /// Usually, the zone is numeric interface index (Windows)
  /// or the name of the interface (POSIX).

std::string to_string(int zone = -1) const;
  /// Returns a string containing a representation of the address
  /// in presentation format.
  /// For IPv4 addresses the result will be in dotted-decimal (d.d.d.d)
  /// notation.
  /// The preferred form is x:x:x:x:x:x:x:x, where the 'x's are the
  /// hexadecimal  values of the eight 16-bit pieces of the address. This
  /// is the full form.
  /// Example: 1080:0:0:0:8:600:200a:425c
  /// It is not necessary to write the leading zeros in an individual
  /// field.  However, there must be at least one numeral in every field,
  /// except as described below.
  /// It is common for IPv6 addresses to contain long strings of zero
  /// bits. In order to make writing addresses containing zero bits
  /// easier, a special syntax is  available to compress the zeros. The
  /// use of "::" indicates multiple groups of 16-bits of zeros.
  /// The "::" can only appear once in an address. The "::" can also be
  /// used to compress the leading and/or trailing zeros in an address.
  /// Example: 1080::8:600:200a:425c
  /// For dealing with IPv4 compatible addresses in a mixed environment,
  /// a special syntax is available: x:x:x:x:x:x:d.d.d.d, where the 'x's
  /// are the hexadecimal values of the six high-order 16-bit pieces of
  /// the address, and the 'd's are the decimal values of the four low-
  /// order 8-bit pieces of the standard IPv4 representation address.
  /// Example: ::ffff:192.168.1.120
  /// If zone identifier is a positive number, it is considered to
  /// represent a network interface index, properly formatted through
  /// call to zone(unsigned int) and added to the string, delimited by a
```

```
  /// percent character. On Windows platforms, the interface index
  /// numeric value is directly appended. On Unix platforms, the name of
  /// the interface corresponding to the index (e.g. "eth0") is appended.

std::wstring to_string(int zone = -1) const;
  /// Wide string version of to_string(int).

bool is_unspecified() const;
  /// Returns true iff the address is unspecified (wildcard, all zero)
  /// address.

bool is_broadcast() const;
  /// Returns true iff the address is a broadcast address.
  /// Only IPv4 addresses can be broadcast addresses. In a broadcast
  /// address, all bits are one. For an IPv6 address, returns always
  /// false.

bool is_loopback() const;
  /// Returns true iff the address is a loopback address.
  /// For IPv4, the loopback address is 127.0.0.1.
  /// For IPv6, the loopback address is ::1.

bool is_multicast() const;
  /// Returns true iff the address is a multicast address.
  /// IPv4 multicast addresses are in the
  /// 224.0.0.0 to 239.255.255.255 range
  /// (the first four bits have the value 1110).
  /// IPv6 multicast addresses are in the ffxx:x:x:x:x:x:x:x range.

bool is_unicast() const;
  /// Returns true iff the address is a unicast address.
  /// An address is unicast if it is neither a wildcard,
  /// broadcast or multicast address.

bool is_link_local() const;
  /// Returns true iff the address is a link local unicast address.
  /// IPv4 link local addresses are in the 169.254.0.0/16 range,
  /// according to RFC 3927.
  /// IPv6 link local addresses have 1111 1110 10 as the first
  /// 10 bits, followed by 54 zeros.

bool is_site_local() const;
  /// Returns true iff the address is a site local unicast address.
  /// IPv4 site local addresses are in on of the 10.0.0.0/24,
  /// 192.168.0.0/16 or 172.16.0.0 to 172.31.255.255 ranges.
  /// Originally, IPv6 site-local addresses had FEC0/10 (1111 1110 11)
  /// prefix (RFC 4291), followed by 38 zeros. Interfaces using
  /// this prefix are supported, but obsolete; [RFC4193] prescribes
  /// fc00::/7 (1111 110) as globally-unique local unicast prefix.

bool is_ipv4_compatible() const;
  /// Returns true iff the address is IPv4 compatible.
  /// For IPv4 addresses, this is always true.
  /// For IPv6, the address must be in the ::x:x range (the
```

```
  /// first 96 bits are zero). Note that IPv4 compatible IPv6
  /// addresses have been obsoleted by [RFC4291].

bool is_ipv4_mapped() const;
  /// Returns true iff the address is an IPv4 mapped IPv6 address.
  /// For IPv4 addresses, this is always true.
  /// For IPv6, the address must be in the ::FFFF:x:x range.

bool is_well_known_mcast() const;
  /// Returns true iff the address is a well-known multicast address.
  /// For IPv4, well-known multicast address range is 224.0.0.0/8.
  /// For IPv6, well-known multicast addresses are in the
  /// ff0x:x:x:x:x:x:x:x range.

bool is_node_local_mcast() const;
  /// Returns true iff the address is a node-local multicast address.
  /// IPv4 does not support node-local addresses, thus the result is
  /// always false for an IPv4 address.
  /// For IPv6, node-local multicast addresses are in the
  /// ffx1:x:x:x:x:x:x:x range.

bool is_link_local_mcast() const;
  /// Returns true iff the address is a link-local multicast address.
  /// For IPv4, link-local multicast addresses are in the
  /// 224.0.0.0/24 range. Note that this overlaps with well-known
  /// multicast addresses range.
  /// For IPv6, link-local multicast addresses are in the
  /// ffx2:x:x:x:x:x:x:x range.

bool is_site_local_mcast() const;
  /// Returns true iff the address is a site-local multicast address.
  /// For IPv4, site local multicast addresses range is 239.255.0.0/16.
  /// For IPv6, site-local multicast addresses are in the
  /// ffx5:x:x:x:x:x:x:x range.

bool is_org_local_mcast() const;
  /// Returns true iff the address is organization-local multicast.
  /// For IPv4, organization-local multicast addresses are in the
  /// 239.192.0.0/16 range.
  /// For IPv6, organization-local multicast addresses are in the
  /// ffx8:x:x:x:x:x:x:x range.

bool is_global_mcast() const;
  /// Returns true iff the address is a global multicast address.
  /// For IPv4, global multicast addresses are in the
  /// 224.0.1.0 to 238.255.255.255 range.
  /// For IPv6, global multicast addresses are in the
  /// ffxf:x:x:x:x:x:x:x range.

bool operator == (const ip_address& addr) const;
  /// Returns true if IP addresses are equal.

bool operator != (const ip_address& addr) const;
  /// Returns true if IP addresses are not equal.
```

```
  bool operator <  (const ip_address& addr) const;
    /// Returns true if this IP address is lesser than addr.

  bool operator <= (const ip_address& addr) const;
  /// Returns true if this IP address is lesser than or equal to addr.

  bool operator >  (const ip_address& addr) const
    /// Returns true if this IP address is greater than addr.

  bool operator >= (const ip_address& addr) const;
    /// Returns true if this IP address is greater than or equal to addr.

  ip_address operator & (const ip_address& addr) const;
    /// Performs a bitwise AND operation and returns result.

  ip_address operator | (const ip_address& addr) const;
    /// Performs a bitwise OR operation and returns result.

  ip_address operator ^ (const ip_address& addr) const;
    /// Performs a bitwise XOR operation and returns result.

  ip_address operator ~ () const;
    /// Performs a bitwise NOT operation on this IP address and
    /// returns result.

  ip_address operator & (const prefix& addr) const;
    /// Performs a bitwise AND operation and returns result.

  ip_address operator | (const prefix& addr) const;
    /// Performs a bitwise OR operation and returns result.

  ip_address operator ^ (const prefix& addr) const;
    /// Performs a bitwise XOR operation and returns result.

  static ip_address parse(const std::string& addr);
    /// Creates an IPAddress from the string containing
    /// an IP address in presentation format (dotted decimal
    /// for IPv4, hex string for IPv6).
    /// Depending on the format of addr, either an IPv4 or
    /// an IPv6 address is created.
    /// See to_string() for details on the supported formats.
    /// Throws std::runtime_exception if the address cannot be parsed.

  static bool try_parse(const std::string& addr, IPAddress& result);
    /// Tries to interpret the given address string as an
    /// IP address in presentation format (dotted decimal
    /// for IPv4, hex string for IPv6).
    /// Returns true and stores the IPAddress in result if the
    /// string contains a valid address.
    /// Returns false with result content unspecified if string
    /// was not valid.
};
}} // namespace std::net
```

5. Conclusion

Class interfaces for IP and socket address as well as rationale for their introduction in C++ Standard and usage examples were described in this proposal. This proposal is one of the starting points and building blocks for C++ Standard Networking Library. Implementation of the described functionality is loosely based on POCO framework [8] implementation; the implementation differs in naming scheme from this proposal. Given a positive feedback on the proposal, implementation will be available in the near future as a standalone implementation in standard-compliant naming convention, with accompanying tests and code examples.

6. Acknowledgements

POCO project contributors and users

References

[RFC791] RFC 791 INTERNET PROTOCOL (http://tools.ietf.org/rfc/rfc791)
[RFC950] RFC 950 Internet Standard Subnetting Procedure (http://tools.ietf.org/rfc/rfc950)
[RFC1817] RFC 1817 CIDR and Classful Routing (http://tools.ietf.org/html/rfc1817)
[RFC2460] RFC 2460 Internet Protocol, Version 6 (IPv6) Specification (http://tools.ietf.org/rfc/rfc2460)
[RFC2553] RFC 2553 Basic Socket Interface Extensions for IPv6 (http://tools.ietf.org/rfc/rfc2553)
[RFC3513] RFC 3513 Internet Protocol Version 6 (IPv6) Addressing Architecture (http://tools.ietf.org/rfc/rfc3513)
[RFC3879] RFC 3879 Deprecating Site Local Addresses (http://tools.ietf.org/rfc/rfc3879)
[RFC4007] RFC 4007 IPv6 Scoped Address Architecture
[RFC4193] RFC 4193 Unique Local IPv6 Unicast Addresses (http://tools.ietf.org/rfc/rfc3879)
[RFC4632] RFC 4632 Classless Inter-domain Routing (CIDR) (http://tools.ietf.org/html/rfc4632)
[RFC4038] RFC 4038 Application Aspects of IPv6 Transition (http://tools.ietf.org/html/rfc4038)
[RFC4038-6.1] RFC 4038 IP versions - Independent structures (http://tools.ietf.org/html/rfc4038#section-6.1)
[IPV6Core] "IPv6 Core Protocols Implementation", Qing Li, Tatuya Jinmei, Keiichi Shima (Morgan Kaufmann Publishers, 2007)
[UndIPv6] "Understanding IPv6" 3rd edition, Joseph Davies (Microsoft Press, 2012)
[POCO] C++ Portable Components (http://pocoproject.org)
[POCOU] List of known POCO users (http://pocoproject.org/forum/viewtopic.php?f=11&t=3826)