Reference Variables for Dynamic, Reliable Packet Operations

Ralph Duncan, Peder Jungck, Kenneth Ross, and Dwight Mulcahy

CloudShield Technologies, A Science Applications International Corporation (SAIC) Company
212 Gibraltar Drive, Sunnyvale, CA 94089 USA
rduncan@cloudshield.com

Abstract. A classic 'reference' variable provides an indirect way to access a variable or aggregate. packetC, [1] a language for network packet processing, has specialized requirements for references that apply to aggregates, based on domain-specific, extended data types. The primary functional requirement is to defer selecting particular aggregates until runtime. In addition, requirements for high program reliability and security are paramount. Thus, packetC reference constructs must guarantee that a selected aggregate (i.e., the value of a runtime dereference) always constitutes a legal aggregate for the involved operation. Both reliability concerns and current domain implementation practice discourage references based on addresses (detailed below). A secondary requirement is to support chaining aggregate operations, where the aggregate used in an operation depends on the result of the previous operation. Our design and implementation of packetC references provides a useful case study in how secure, reliable references can meet these requirements by combining strong typing features (e.g., declaration rules), simple mechanics (encoded ordinal values) and appropriate technical attributes for references, such as reseatability and non-nullability.

1 Introduction

In its most general sense, a programming language reference is a construct that provides an indirect means for referring to a variable, aggregate or object's values. Thus, the reference's actual value, such as a memory address, is distinct from the value(s) to which it provides access.

Key reference characteristics include whether a variable with a reference type can

- Be set to a null value or nullability (be in a state where it provides no indirect access to underlying values)
- Be assigned a new value after its declaration or reseatability
- Expose its actual values to the user.

The particular set of characteristics a language's reference types and variables possess depends on the primary roles the constructs play in the language.

J.J. Park et al. (Eds.): NPC 2012, LNCS 7513, pp. 48–60, 2012.

packetC [1], is a heavily extended C dialect for network packet processing. It is being employed to develop commercial and military systems by users that include the U.S. Air Force, IBM and Harris Corporation. packetC uses references to defer until runtime specifying the specific aggregate (extended-type data structure) on which classic packet searching or matching actions will operate. A major use case involves chaining, such that the identity of the aggregate used in one operation depends on the result of a previous one. A packetC user can achieve this by exploiting a simple scheme of reference array indexing. Otherwise, users would have to explicitly code each possible combination of secondary operation and primary operation result value (e.g., in a C-style switch statement), a process prone to errors and omissions.

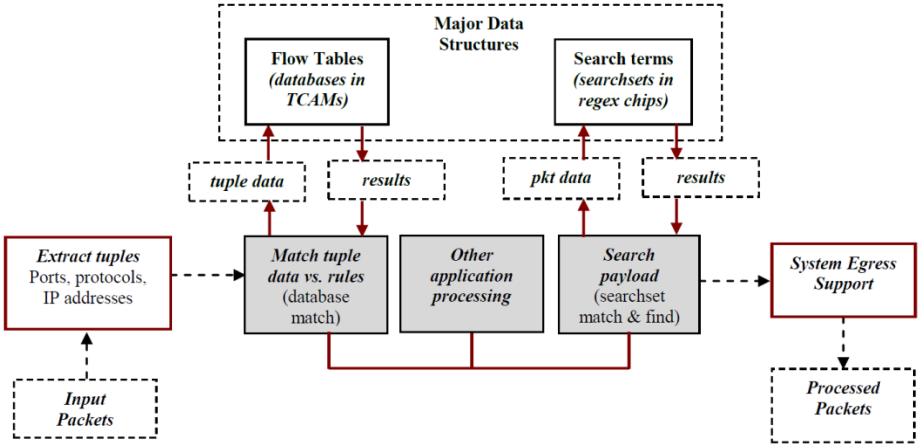


Fig. 1. Matching and searching operations on network packets
©CloudShield Technologies, 2011

Packet processing is an unforgiving application domain. It rewards processing at wire-speed, i.e., processing and outputting packets at the same 1-40 gigabits per second (gbps) speed at which the packets are arriving. Thus, time-consuming exception handling is infeasible and error-free application logic is paramount. This need for reliability requires two technical characteristics of packetC references:

- Strong typing, since trying to apply an operation to a data structure of the wrong type is a major runtime fault.
- Non-nullability, since trying to apply an operation to no data structure at all is prohibitively unreliable.
- In addition, the need to determine at runtime which aggregate will be used for certain operations requires packetC references to have the attribute of
- Reseatability (assignability), especially when references need to be assigned values according to an operation result.

Finally, the demand for a high degree of security, especially for military and telecommunications infrastructure precludes simply using memory addresses as C-style

reference values. Even if this were not the case, current implementation practices with specialized chips often mean that the aggregates of interest are not accessible via classic memory addresses.

We hope that these distinctive, domain-oriented requirements will make packetC reference constructs relevant to students of both language design and network packet processing.

The remainder of the paper is organized as follows. We quickly review classic packet matching and searching operations. We then describe packetC matching and searching aggregate types, showing how reference variables are declared, initialized and dereferenced for each. After describing our implementation mechanics, we present experimental results to show how the scheme plays out in a commercial, parallel execution environment for packet processing.

2 Application Domain Matches and Searches

The packetC reference construct is primarily intended to support two classic network packet operations described below: matching portions of extracted packet header data against large tables and searching packet contents for the presence of particular strings or character patterns (Fig. 1).

Network packet applications often start by extracting a set or tuple of basic data from the packet header (portion containing standard routing data), e.g., the source and destination address, source and destination port and kind of protocol being used. Tuple data is often used to associate a given packet with a current flow, essentially a networking conversation underway between two points of the network. Classifying packets in this fashion is usually done by matching the tuple data against a table of current flows, sometimes termed a table of (matching) rules.

Some applications search the packet payload, the non-header portion containing user-specific data, an operation described as deep packet inspection (DPI). Searches are conducted in terms of matching one or more strings or regular expressions.

As a language for network packet processing packetC supports these classic network operations with extended data types and associated operators. The sections that immediately follow describe these constructs and their relation to references.

3 Preview of ref and deref Operators

Before we encounter the code examples below, we offer descriptions of packetC ref and **deref** operators. We use this explicit syntax to emphasize that, unlike * and & in C, these operators do not use memory addresses or support 'pointer arithmetic.'

The **ref** operator takes a single operand, which must be either:

- The name of a *database* with the same structure base type as the reference variable's base type.
- The name of a *searchset* with a string or regular expression type that matches the **string** or **regex** keyword in the reference variable declaration.

The **deref** operator takes a reference variable operand. It is not as if all the aggregate's values were present at that source code location but, rather, it is as if the aggregate identifier had been hard-coded there. A packetC dereference produces an lvalue that indicates an entire aggregate object. Thus, it acts as a substitute for the lvalue at that source code location. This property can lead to unusual code forms (much as pointer dereferencing does in C); however, it provides capabilities for compact, generic programming, as some code examples and experiments below show.

4 Strong Typing, Reseatability and Non-nullability for Database Matching

In packetC[1] flow tables can be implemented via a database extended type [1]. Each element of a database is a C-style structure. Since network application matching against flow tables often selectively uses only a portion of the tuple data, packetC databases are essentially arrays of symmetrical structures, where each structure is, in turn, composed of a data half and corresponding mask half that have a common structure base type. A field of the data portion is only used in matching operations if the bits of the corresponding mask field are set, rather than zeroed. Example declarations follow.

```
struct stype { short dest; short src;};
database stype virusFlows  [300];
```

Matching operations are performed on a database via a **match** operator, expressed with C++ method-style syntax. When a match's required first argument is a structure, it must have the same type as the database's base type. A successful match returns the matching element's index, while a failure to match any element must be handled by packetC's C++-style system of try and catch constructs. Example match code is shown below.

```
try {

    rownum = virusFlows.match( myStruct );// match my-
Struct vs. entire database }
catch ( ERR_DB_NOMATCH ) {…}
```

Because database match operators only make sense if the match operand and database have the same structure base type, packetC reference variables for databases are declared in terms of a base type and can only reference databases with that base type. To further ensure that dereferencing such a variable always produces a legal runtime value, the declaration must set the reference variable to a legal, non-null value (see below). (A packetC structure tag names a type without needing a typedef).

```
struct stype { short dest; short src;};
database stype malwareFlows[500];
database stype virusFlows  [300];
```

```
// declare a database reference var of 'stype' base type
reference db : stype refDB = ref(virusFlows);
…
// 'reseat' reference to DB of 'stype'
refDB = ref(malwareFlows); // legal

// ERROR: try to reseat to DB of another type
database tuple5Type currentFlows[4000];
refDB = ref(currentFlows); // ERROR
A dref operation on a database reference variable can be
used anywhere that a database identifier could be used:
// Using types from above, deref a database
structVar.dest = deref( refDB )[2].dest;
// At runtime the above construct equals StructVar.dest =
malwareFlows[2].dest
```

5 Strong Typing, Reseatability and Non-nullability for Searchset (Payload) Matching

packetC searchsets [1] supply terms for payload searching in the form of an ordered set of strings or set of regular expressions. Because these two forms have different restrictions on what methods can be applied to them, a searchset's elements cannot mix strings and regular expressions. The code example below shows string and regex searchset declarations.

```
searchset sSet[3][3]   = {"dog","cat","bat"};
regex searchset regSet[2][7] = {".*?from",".*?mail"};
```

The packetC find method searches for each searchset element, s, starting anywhere within the argument (which is often the packet payload). When a searchset is declared without the regex keyword qualifier, attempts to find searchset string elements are made in the same order as their declaration: searching terminates when a match is found. When a regex qualifier is used, the matching sequence and behavior depends on the characteristics of the regular expressions involved. Results are returned with the predefined structure type shown below.

```
struct SearchResult {
  int index;   // searchset elem matched
  int position;// search area where match ends
};
```

The code example below shows the appearance of a find operator expression and the required use of an associated try/catch construct to handle the appropriate exception if no search term is found.

```
searchset petSset[3][3] = {"dog","cat","bat"};
SearchResult ansStruct;
  try { // search the entire packet
     ansStruct  =  petSset.find( pkt );
  }
  catch ( ERR_SET_NOTFOUND ) {…}
```

Searchsets also have a match operator, which compares a searchset element of length n to the first n bytes of the match operand. Because regular expressions cannot be used in this way, only searchsets composed of strings can be used with the searchset match operator. Thus, to ensure that a searchset reference variable can be legally dereferenced in any context where the referent could be used, searchset reference variables are declared in terms of being string or regex searchsets.

```
reference set: string refStr = ref(petSset);
//
searchset regSet[2][7] ={".*?from",".*?mail"};
reference set: regex  refReg = ref(regSet);
```

Using the types defined above the code below shows dereferencing applied to both kinds of searchset reference variable.

```
// reference a string searchset for match, using an array
slice operand
result = deref(refStr).match(pkt[64:66]);
// reference a regex searchset for find, using an array
slice operand
// result = regSet.find(pkt[0:end]);
result = deref(refReg).find(pkt[0:end]);
```

The packetC language is agnostic about how references are implemented in that it does not prescribe what kind of values are used to indicate particular databases or searchsets. The language simply specifies that the implementation values cannot be exposed to user examination or manipulation. The next two sections describe our implementation mechanics and how our host platform influences them.

6 Host System Impacts on Reference Implementation

The following aspects of current CloudShield Technologies' platforms [2] affect how we implement references.

- The packetC tool-chain is built atop an interpreted program scheme; thus, the packetC compiler emits bytecodes for an interpreted virtual machine, rather than assembler code.

- CloudShield Technologies' platforms are heterogeneous architectures that use multi-core network processing units (NPUs); thus, dereferencing operations are ultimately executed by NPU microcode.
- Databases and Searchsets are implemented on Ternary Content Addressable Memory (TCAM) chips and regex processors respectively.
- Our bytecodes identify specific databases or searchsets by an integer value assigned on a per-application basis.

Taken together, these characteristics encourage using straightforward mechanics to implement reference values and dereferencing operations. Both are described in the section that follows.

7 Reference Implementation

Our basic approach is to encode the unique integer value associated with databases or searchsets within a 32-bit integer to serve as the basis for reference variables. However, there are two complications:

- Searchsets can be implemented by both a classic searchset table (for the find operation) and a compiler-generated database (for match operations on string searchsets); thus, our reference variables must be able to encode more than just a single compiler-assigned integer identifier.
- The need to keep some additional house-keeping data adds a few additional bits to the 32-bits of encoded data.

Hence, we implement a reference variable as a 32-bit integer with several smaller integer values packed inside it. At worst, dereferencing a packetC reference variable involves a bitwise AND operation to discard the unwanted values and a SHIFT to reposition the remaining value.

The example below shows a searchset find operation chained to the results of a preceding database match, which selects which of three searchsets to use. In this case, the dereferencing only requires an AND operation.

```
searchset bad[2][8]={"bad.com," "evil.com"};
searchset sly[2][6]= {"mal.com", "bat.com"};
searchset incomp[2][5] = {"@!*&!","%^&*"};
reference set:string refArr[3] = {ref(bad), ref(sly),
ref(incomp)};

// match on packet origins in header info
flowRow = flowTab.match( rec );
// search payload, based on packet origins
loca = deref( refArr[flowRow]).find( pkt );
```

A readable snapshot follows of the bytecodes emitted by the compiler for the packetC code above.

```
// match on packet origins in header info; store matching
row in rowNum
DBMATCH 5, mskND, rec.data, rec.mask. rowNum
// use rowNum to put selected reference values into
ssetNum
MCOPY ssetNum, refArr[rownum]
// discard unwanted bits of reference value; remaining
bits are searchset ID num
AND ssetNum, 65535
// perform the chained search, using searchset indicated
by ssetNum
SEARCH_PKT ssetNum, strtIdx, finIdx, resultLoc
```

Thus, with the simple encoding scheme, we can meet our primary requirements for dynamic selection of extended data type aggregates and do so in a secure, reliable manner. In addition, we can provide dynamic, chained operations that are typical of the domain at the cost of two or three elementary operations. The next section presents performance data for these constructs.

8 Reference Performance in a Parallel Environment: Chained Operation Example

A reference implementation that accesses encoded ordinal values by shifting and masking instructions is obviously unlikely to incur significant performance overhead simply on that account. However, the target domain is high-speed packet processing and it is not self-evident how accessing multiple aggregates (stored in specialty chips) by these means plays out in practice. Thus, this section compares a 'brute-force' source code approach to chaining operations with a simple use of references, both hosted on one of our parallel platforms where 96 contexts are processing packets.

The experiment used a CloudShield PN41 [3] 10 Gigabit Ethernet blade hosting the packetC application. The DPPM blade contains an Intel Corporation® IXP 2800 NPU and custom Xilinx, Inc.® Virtex® 5 FPGAs. Netlogic, Inc.® NSE 5512 TCAM chips implement the databases. Searchsets are implemented by an IDT PAX.port 2500 content inspection engine [4]. We used an IXIA® XM12 traffic generator [5] to generate network traffic at a maximum of 10 gigabits per second (Gbps) and approximately 14 million packets per second (pps).

The experimental application defines a packetC database with each of the four database rows geared to matching one of the following kinds of network traffic: HTTP, VoIP, email, DNS. Based on which of the four database cases a packet matches, the program searches the packet payload for matches with strings in one of four possible searchsets. The database declaration and flow matching is conducted in terms of a structure with packet protocol information, as shown below.

```
struct ipv4Tuple {
  int   scrAddr;    int   destAddr;
```

```
  short srcPort;    short destPort;
  byte  protocol;
};
database ipv4Tuple flowTable[4] = {…};
try { matchRow = flowTable.match( flow ); }
catch ( ERR_DB_NOMATCH ) {…exit; }
// searchset declarations for each kind of network traf-
fic we are handling follow.
searchset httpVerbs[2][4] = {"GET", "POST"};
searchset sipVerbs[2][6]  = {"INVITE","BYE"};
searchset badGuys[5][20]  = {"aca-
pone","jdillinger","pbfloyd","bonnie", "clyde"};
searchset sites[4][20]
={"yahoo.com","google.com","purepeople.com","cnn.com"};
```

8.1 Version 1: Without References

The version without references must explicitly code each possible searchset operation that could occur, using the relevant identifier for each one. Since only one of the searchsets could be used after a given database match, the code for the possible searchset operations must be structured conditionally, e.g., by using a switch statement as we did to implement this version.

```
try { matchedRow = flowTable.match(flow); }
catch ( ERR_DB_NOMATCH ) {…}

try { // do the desired chained operation
  switch (matchedRow) {
  case 0:
    result=httpVerbs.find(pkt[0:end]);    // throws
ERR_SET_NOTFOUND if no match
    break;
  case 1:
    result = sipVerbs.find(pkt[0:end]);   // throws
ERR_SET_NOTFOUND if no match
    break;
  case 2:
    result = badGuys.find(pkt[0:end]);    // throws
ERR_SET_NOTFOUND if no match
    break;
  case 3:
    result = sites.find(pkt[0:end]);      // throws
ERR_SET_NOTFOUND if no match
    break;
  default:
```

```
    exit;
}
catch ( ERR_SET_NOTFOUND ) {…}
```

8.2 Version 2: Using References

To exploit the reference construct in this situation, we need an array of references in which the:

- Index values correspond to the database rows (records) to be matched during the initial operation.
- Array element values correspond to the searchsets to be used for the next operation.

The relevant array declaration is shown below.

```
reference set:string refSet[4] ={ref(httpVerbs),
ref(sipVerbs), ref(badGuys), ref(sites)};
```

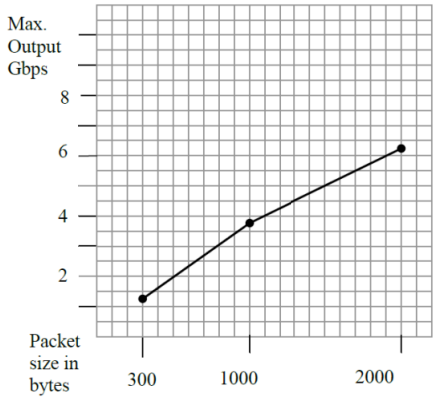
We can now replace coding find operations for each of the possible database match results (i.e., for each possible searchset name) with a single searchset find, abstracting out the individual searchset names and replacing them with a solitary variable holding the reference array's index values. Using the deref operator on an element of that reference array at runtime effectively delivers the referenced aggregate as the searchset upon which the find method will operate (shown below).

```
try { matchedRow = flowTable.match(flow);
      result =
deref(refSet[matchedRow]).find( pkt[0:end] );}
catch (…) {…}
```

8.3 Results: Source Code Reduction and Performance

Our experiments suggest that, for this application domain and hardware, using reference constructs does not cause meaningful performance differences. Fig. 2 shows that, when measuring throughput in gigabits per second (gbps), the application's throughput for packet sizes 300 and 1000 bytes do not vary for hundredths of a gbps. Only at a packet size of 2000 bytes is there a discernable difference, with the using-references version achieving 6.15 gbps and the without-references version running at 6.14 gbps. This small variation could be an artifact of experiment mechanics.

We were concerned that the overhead of moving packet data to the regex chip for the searchset find operation was dwarfing all other effects. To check this, we coded a version of the application in which the second operation was another database match operation, one that involved four alternative databases. The results in Fig. 2(c). show no meaningful performance difference caused by references being used or not.



(a)

Database op-> searchset op		
Packet size in bytes	**Throughput with references (Gbps)**	**Throughput without references (Gbps)**
300	1.16	1.16
1000	3.88	3.88
2000	6.15	6.14

(b)

Database op#1->database op#2		
Packet size in bytes	**Throughput with references (Gbps)**	**Throughput without references (Gbps)**
300	0.582	0.582
1000	1.9	1.9
2000	3.9	3.9

(c)

Fig. 1. Performance with and without references. (a) Throughput sensitivity to packet size (database result drives searchset selection). (b) Database result drives searchset selection. (c) Database result drives 2nd database selection. © CloudShield Technologies, 2011

In these tests the version without references will require comparison and jump bytecodes to execute the switch statement control flow. The versions that use a reference will employ 2-3 instructions to mask off (and sometimes shift) portions of the encoded data. Any performance differences in these two small instruction sets will be dwarfed by the time spent to move data to the TCAM/regex subsystems or to perform classic packet matching and searching operations, whether they are implemented with specialized processors or not. Thus, it is not surprising that the experiments show no meaningful performance increase from using references. Our contention is that the increased code simplicity shown above is a significant benefit and that it does not come at a cost of decreased performance. Before presenting final conclusions, the next section quickly compares packetC references with similar constructs in other languages.

9 Reference Constructs in Other Languages or Contexts

Since our approach emphasizes choosing language characteristics as much as implementation mechanics, this penultimate section provides a concise comparison of such characteristics in packetC references and several similar constructs: C pointers, Java pointers, and C++ references (Table 1). Recall that references play a variety of roles, including the following:

- Facilitating the creation and destruction of dynamically-allocated objects (e.g., Java, C).
- Exposing array layout and indexing mechanics (C).
- Passing a parameter that has an aggregate (composite) data type without copying its contents to call-stack slots (C++).

Table 1. Comparing C pointers, C++ references and packetC references. ©CloudShield Technologies, 2011.

Construct/ Attribute	C pointer/ Java reference	C++ reference	packetC reference
New value can be assigned (reseatable).	Yes	No	Yes
Can be assigned NULL value.	Yes	No	No
Can be referenced as itself in source code.	Yes	No	Yes
Must be assigned at declaration.	No	Yes	Yes

A C pointer holds a memory address in the form of a numeric value. Despite some implementation variability, users can depend on a pointer holding the value of an address (or null value) and being amenable to pointer arithmetic operations [6]. Java references share C pointers reseatability, nullability and independence from their referent; however, Java reference values are not generally user-accessible or available for arithmetic operations [7].

In contrast, a packetC reference is not constrained to be an address; it is simply a designation that uniquely indicates one of a finite set of aggregates, which share a common type signature and are visible from the reference variable's scope. Users cannot assume a particular internal organization for reference variables in a given implementation.

A C++ reference [8] is more restricted than a C pointer or a packetC reference: it must be declared with a non-null value that cannot be changed. Source code cannot refer to a reference identifier as an entity in itself, since an occurrence of the identifier indicates the referenced object, instead.

As Table 1 shows, packetC's collection of classic reference characteristics is similar to that of several familiar reference constructs, without being identical to any of those reviewed here. We call attention to packetC references, however, not because their general characteristics are unusual but, instead, because of their specific role for practical operations in a specialized domain. It is this aspect that the conclusions explore below.

10 Summary

Our primary finding is that a programming language can exploit relatively simple reference declaration constructs and implementation mechanisms to deliver significant practical benefits:

- Strong typing and non-nullability to guarantee a legal operand at run-time in a domain where time pressures make most exception handling infeasible.

- Reseatability to enable dynamic selection of operands: in this case, to select entire aggregates of considerable complexity and to construct complex chains of dependent operations.
- Hidden reference values (and avoidance of memory addresses as reference values) to discourage malicious exploits.
- Source code compaction and increased extensibility by replacing a system of switch statement case alternatives with a single dereferencing expression.

Taken separately, the language constructs and implementation mechanics used to provide each of these advantages are fairly ordinary: using base types, requiring and restricting initial values, encoding ordinal values as identifiers. However, the whole is greater than the sum of its parts and provides practical lessons in applying reference constructs to a domain-specific language, especially to one with high reliability and security requirements.

Acknowledgements. Peder Jungck, Dwight Mulcahy and Ralph Duncan are the co-authors of the packetC language. Andy Norton, Greg Triplett, Kai Chang, Mary Pham, Alfredo Chorro-Rivas and Minh Nguyen provided valuable help in many areas for this effort. Thanks are also due the SAIC individuals who secured export approval 12-SAIC-0305-550 for the paper.

References

1. Jungck, P., Duncan, R., Mulcahy, D.: packetC Programming. Apress, New York (2011)
2. CloudShield Technologies. CS-2000 Technical Specifications. Product datasheet available from CloudShield Technologies, 212 Gibraltar Dr., Sunnyvale, CA, USA 94089 (2006)
3. International Business Machines Corporation. IBM Blade Center PN41. Product datasheet available from IBM Systems and Technology Group, Route 100, Somersm, New York, USA 10589 (2008)
4. Chao, H.J., Liu, B.: High Performance Switches and Routers, pp. 562–564. John Wiley and Sons, Hoboken (2007)
5. IXIA. XM12 High Performance Chassis. Retrieved from, `http://www.ixia.com.com/products/chassis/display?skey=ch_optixia_xm12` (January 24, 2011)
6. ISO/IEC 9899:1999. Standard for the C programming language (May 2005) (version, 'C99')
7. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley (June 2005)
8. ISO/IEC ISO/IEC 14882:2003 (corrected version of the 1998 C++ standard)