Generic Programming
Sean Parent | Principal Scientist
“You cannot fully grasp mathematics until you understand its historical context.” – Alex Stepanov
1988
Generic Programming^a

David R. Musser
Rensselaer Polytechnic Institute
Computer Science Department
Troy, New York 12180

Alexander A. Stepanov
Hewlett-Packard Laboratories
Software Technology Laboratory
Post Office Box 3040
Palo Alto, California 94309-0009

Abstract

Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

Four kinds of abstractions—data, algorithms, structural, and representations—are discussed, with examples of their use in building an Ada library of software components. The main topic discussed is generic algorithms and an approach to their formal specification and verification, with illustration in terms of a partitioning algorithm such as is used in the quicksort algorithm. It is argued that generically programmed software components (libraries) offer important advantages for achieving software productivity and reliability.

^aThis paper was presented at the First International Conference on ISAAC'98 and AACCD'98, Rome, July 24-26, 1998. (ISAAC stands for International Symposium on Symbolic and Algebraic Computation, and AACCD for Applied Algebra, Algebraic Algorithms, and Number Computing Conferences.) It was published in Lecture Notes in Computer Science (Springer-Verlag, 1998), pp. 145-159.

The first author's work was sponsored in part through a subcontract from Computational Logic Inc. This work was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order 912. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic Inc.
Generic Programming*

David R. Musser
Rensselaer Polytechnic Institute
Computer Science Department
Amos Eaton Hall
Troy, New York 12180

Alexander A. Stepanov
Hewlett-Packard Laboratories
Software Technology Laboratory
Post Office Box 10490
Palo Alto, California 94303-0969

Abstract

Generic programming centers around the idea of abstracting from concrete efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

Four kinds of abstractions—data, algorithmic, structural, and representations—are discussed, with examples of their use in building an Ada library of software components. The main topic discussed is generic algorithms and an approach to their formal specification and verification, with illustration in terms of a partitioning algorithm such as is used in the quicksort algorithm. It is argued that generically programmed software components奈何an offer important advantages for achieving software productivity and reliability.

*This paper was presented at the First International Joint Conference of PPSA’88 and AAECC-8, Rome, Italy, July 4-8, 1988. (PPSA stands for International Symposium on Symbolic and Algebraic Computation and AAECC for Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes. It was published in Lecture Notes in Computer Science 317: Springer-Verlag, 1988, pp. 152-160.)

The first author’s work was sponsored in part through a subcontract from Computational Logic Inc. This work was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order 1153. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic Inc.
Generic Programming*

David R. Musser
Rensselaer Polytechnic Institute
Computer Science Department
Amos Eaton Hall
Troy, New York 12180

Alexander A. Stepanov
Hewlett-Packard Laboratories
Software Technology Laboratory
Post Office Box 10490
Palo Alto, California 94303-0969

Abstract

Generic programming centers around the idea of abstracting from concrete efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

Four kinds of abstractions—data, algorithms, structural, and representation—are discussed, with examples of their use in building an Ada library of software components. The main topic discussed is generic algorithms and an approach to their formal specification and verification, with illustration in terms of a partitioning algorithm such as is used in the quicksort algorithm. It is argued that generally programmed software components inherently offer important advantages for achieving software productivity and reliability.

* This paper was presented at the First International Joint Conference of ISSAC'88 and AAECS'88, Rome, Italy, July 28-30, 1988. (ISSAC stands for International Symposium on Symbolic and Algebraic Computation and AAECC for Applied Algebra, Algebraic Algorithms, and Error Correcting Codes.) It was published in Lecture Notes in Computer Science (58: Springer-Verlag, 1988, pp. 152-162).

The line author's work was sponsored in part through a subcontract from Computational Logic Inc., which was sponsored in turn by the Defense Advanced Research Projects Agency, ARPA Order 925. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic Inc.
“By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms.”
“By **generic programming** we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms.”
“By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms.”
“By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms.”
“By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms.”
“By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms.”
“By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms.”
“By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms.”
“By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms.”
1976-1987
1976 Parallel Computation and Associative Property
A binary operation $\cdot$ on a set $S$ is called associative if it satisfies the associative law:

$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \text{ for all } x, y, z \text{ in } S.$$
Parallel reduction is associated with monoids

A binary operation $\bullet$ on a set $S$ is called associative if it satisfies the associative law:

$$(x \bullet y) \bullet z = x \bullet (y \bullet z) \text{ for all } x, y, z \text{ in } S.$$
Software is associated with Algebraic Structures
The 1977 ACM Turing Award was presented to John Backus at the ACM Annual Conference in Seattle, October 17. In introducing the recipient, John E. Searle, Chairman of the Awards Committee, made the following comments and read a portion of the final citation. The full announcement is in the September 1977 issue of Communications, page 482.

"...probably there is nobody in the room who has not heard of Fortran and most of you have probably used it at least once, or at least looked over the shoulder of someone who was writing a Fortran program. There are probably almost as many people who have heard the letters BNF but don't necessarily know what they stand for. Well, the B in this case stands for Backus, as is explained in the formal citation. These two contributions, in my opinion, are among the half dozen most important technical contributions to the computer field and both were made by John Backus (which in the Fortran case also involved some colleagues). It is for these contributions that he is receiving this year's Turing award.

The short form of his citation is for profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran, and for seminal publication of formal procedures for the specification of programming languages.

The most significant part of the full citation is as follows: '...Backus headed a small IBM group in New York City during the early 1950s. The earliest product of this group's efforts was a high-level language for scientific and technical computing inherited from their common ancestor—the von Neumann computer, its close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for creating new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming languages are now described with some type of formal syntactic definition. "Probably there is nobody in the room who has not heard of Fortran and most of you have probably used it at least once, or at least looked over the shoulder of someone who was writing a Fortran program. There are probably almost as many people who have heard the letters BNF but don't necessarily know what they stand for. Well, the B in this case stands for Backus, as is explained in the formal citation. These two contributions, in my opinion, are among the half dozen most important technical contributions to the computer field and both were made by John Backus (which in the Fortran case also involved some colleagues). It is for these contributions that he is receiving this year's Turing award.

The short form of his citation is for profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran, and for seminal publication of formal procedures for the specification of programming languages.

The most significant part of the full citation is as follows: '...Backus headed a small IBM group in New York City during the early 1950s. The earliest product of this group's efforts was a high-level language for scientific and technical computing inherited from their common ancestor—the von Neumann computer, its close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for creating new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming languages are now described with some type of formal syntactic definition. "Probably there is nobody in the room who has not heard of Fortran and most of you have probably used it at least once, or at least looked over the shoulder of someone who was writing a Fortran program. There are probably almost as many people who have heard the letters BNF but don't necessarily know what they stand for. Well, the B in this case stands for Backus, as is explained in the formal citation. These two contributions, in my opinion, are among the half dozen most important technical contributions to the computer field and both were made by John Backus (which in the Fortran case also involved some colleagues). It is for these contributions that he is receiving this year's Turing award.

The short form of his citation is for profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran, and for seminal publication of formal procedures for the specification of programming languages.

The most significant part of the full citation is as follows: '...Backus headed a small IBM group in New York City during the early 1950s. The earliest product of this group's efforts was a high-level language for scientific and technical computing inherited from their common ancestor—the von Neumann computer, its close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for creating new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming languages are now described with some type of formal syntactic definition. "Probably there is nobody in the room who has not heard of Fortran and most of you have probably used it at least once, or at least looked over the shoulder of someone who was writing a Fortran program. There are probably almost as many people who have heard the letters BNF but don't necessarily know what they stand for. Well, the B in this case stands for Backus, as is explained in the formal citation. These two contributions, in my opinion, are among the half dozen most important technical contributions to the computer field and both were made by John Backus (which in the Fortran case also involved some colleagues). It is for these contributions that he is receiving this year's Turing award.

The short form of his citation is for profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran, and for seminal publication of formal procedures for the specification of programming languages.

The most significant part of the full citation is as follows: '...Backus headed a small IBM group in New York City during the early 1950s. The earliest product of this group's efforts was a high-level language for scientific and technical computing inherited from their common ancestor—the von Neumann computer, its close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for creating new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming languages are now described with some type of formal syntactic definition. "Probably there is nobody in the room who has not heard of Fortran and most of you have probably used it at least once, or at least looked over the shoulder of someone who was writing a Fortran program. There are probably almost as many people who have heard the letters BNF but don't necessarily know what they stand for. Well, the B in this case stands for Backus, as is explained in the formal citation. These two contributions, in my opinion, are among the half dozen most important technical contributions to the computer field and both were made by John Backus (which in the Fortran case also involved some colleagues). It is for these contributions that he is receiving this year's Turing award.

The short form of his citation is for profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran, and for seminal publication of formal procedures for the specification of programming languages.

The most significant part of the full citation is as follows: '...Backus headed a small IBM group in New York City during the early 1950s. The earliest product of this group's efforts was a high-level language for scientific and technical computing inherited from their common ancestor—the von Neumann computer, its close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for creating new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative, functional style of programming is...
The 1979 ACM Award was presented to Kenneth E. Iverson by Walter Carlson, Chairman of the Awards Committee, at the ACM Annual Conference in Detroit, Michigan, October 29, 1979.

In making its selection, the General Technical Achievement Award Committee cited Iverson for his pioneering effort in programming languages and mathematical notation resulting in what the computing field now knows as APL. Iverson's contributions to the implementation of interactive systems, to the educational uses of APL, and to programming language theory and practice were also noted.

Born and raised in Canada, Iverson received his doctorate in 1954 from Harvard University. There he served as Assistant Professor of Applied Mathematics from 1955-1960. He then joined International Business Machines Corp. and in 1970 was named an IBM Fellow in honor of his contribution to the development of APL.

Dr. Iverson is presently with J.P. Sharp Associates in Toronto. He has published numerous articles on programming languages and has written four books about programming and mathematics:

- *A Programming Language* (1962),
- *Elementary Functions* (1966),
- *Algebra: An Algorithmic Treatment* (1972), and

Notation as a Tool of Thought
Kenneth E. Iverson
IBM Thomas J. Watson Research Center

The importance of nomenclature, notation, and language as tools of thought has long been recognized. In chemistry and in botany, for example, the establishment of systems of nomenclature by Lavoisier and Linnaeus did much to stimulate and to channel later investigation. Concerning language, George Boole in his *Laws of Thought* [1, p.243] asserted "That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted."

Mathematical notation provides perhaps the best-known and best-developed example of language used consciously as a tool of thought. Recognition of the important role of notation in mathematics is clear from the quotations from mathematicians given in Cajori's *A History of Mathematical Notations* [2, pp.321-322]. They are well worth reading in full, but the following excerpts suggest the tone:

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.

A. N. Whitehead

Key Words and Phrases: APL, mathematical notation

CR Category: 4.2
The importance of nomenclature, notation, and language as tools of thought has long been recognized. In chemistry and in botany, for example, the establishment of systems of nomenclature by Lavoisier and Linnaeus did much to stimulate and to channel later investigation. Concerning language, George Boole in his Laws of Thought [1, p.24] asserted "That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted."

Mathematical notation provides perhaps the best-known and best-developed example of language used consciously as a tool of thought. Recognition of the important role of notation in mathematics is clear from the quotations from mathematicians given in Cajori's A History of Mathematical Notations [2, pp.332,331]. They are well
\textit{life} \leftarrow \{ \uparrow 1 \circ \omega. \land 3 \ 4=+/,-1\ 0\ 1\circ.\ominus -1\ 0\ 1\circ.\ominus \circ \omega \}$
TECTON: A LANGUAGE FOR MANIPULATING GENERIC OBJECTS

D. Kapur, D.R. Musser, and A.A. Stepanov
1981 Tecton

The Tecton language

TECTON: A LANGUAGE FOR MANIPULATING GENERIC OBJECTS

D. Kapur, D.R. Musser, and A.A. Stepanov
Higher Order Programming

Copyright ©1986 by Alexander A. Stepanov, Aaron Kershenbaum and David R. Musser

March 5, 1987

USING TOURNAMENT TREES TO SORT
ALEXANDER STEPANOV AND AARON KERSHENBAUM

Polytechnic University
333 Jay Street
Brooklyn, New York 11201

Center for Advanced Technology in Telecommunications
Higher Order Programming

Copyright ©1986 by
Alexander A. Stepanov, Aaron Kershenbaum and David R. Musser

March 5, 1987
1987

Alex works briefly at Bell Labs
1987

Alex works briefly at Bell Labs
Starts a friendship with Bjarne Stroustrup
1987

Alex works briefly at Bell Labs
Starts a friendship with Bjarne Stroustrup
Andrew Koenig explains the C machine
1987

Alex works briefly at Bell Labs
Starts a friendship with Bjarne Stroustrup
Andrew Koenig explains the C machine
Reads Ken Thompson’s and Rob Pike’s code for Unix and Plan 9
1987

Leonhard Euler
1987

Leonhard Euler
“De-Bourbakized”
1987

Leonhard Euler
“De-Bourbakized”
Nicolas Bourbaki
1987

Leonhard Euler
“De-Bourbakized”
Nicolas Bourbaki
Knowledge is founded on the basis of precise, quantitative laws
Knowledge is founded on the basis of precise, quantitative laws.
Mathematics is discovery, not invention.
Software is defined on Algebraic Structures
Generic Programming

David R. Musser
Rensselaer Polytechnic Institute
Computer Science Department
Troy, New York 12180

Alexander A. Stepanov
Hewlett-Packard Laboratories
Software Technology Laboratory
Post Office Box 10400
Palo Alto, California 94304-0400

Abstract

Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which works with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

Four kinds of abstractions—data, algorithms, structural, and representations—are discussed, with examples of their use in building an Ada library of software components. The main topic discussed is generic algorithms and an approach to their formal specification and verification, with illustration in terms of a partitioning algorithm such as is used in the quicksort algorithm. It is argued that generically programmed software components inherently offer important advantages for achieving software productivity and reliability.

---

*This paper was presented at the First International Joint Conference of PPSAC'88 and AAMC'88, Rome, July 4-8, 1988. (PPSAC stands for International Symposium on Symbolic and Algebraic Computation, and AAMC for Applications of Algebraic Algorithms and Their Computer Coding.) It was published in Lecture Notes in Computer Science #33, Springer-Verlag, 1988, pp. 15-26.

The first author's work was sponsored in part through a subcontract from Computational Logic Inc. This work was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order 183. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic Inc.
Generic Programming

David R. Musser
Rensselaer Polytechnic Institute
Computer Science Department
Amos Eaton Hall
Troy, New York 12180

Alexander A. Stepanov
Hewlett-Packard Laboratories
Software Technology Laboratory
Post Office Box 10490
Palo Alto, California 94303-0969

Abstract

Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

Four kinds of abstractions—data, algorithmic, structural, and representation—are discussed, with examples of their use in building an Ada library of software components. The main topic discussed is generic algorithms and an approach to their formal specification and verification, with illustration in terms of a partitioning algorithm such as is used in the quicksort algorithm. It is argued that generically programmed software components (libraries) offer important advantages for achieving software productivity and reliability.

---


The first author’s work was sponsored in part through a subcontract from Computational Logic, Inc. which was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order 5149. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic, Inc.
procedure Partition(S : in out Sequence; F, L : in Coordinate; Middle, Middle_OK : out Coordinate;)
begin
  First := Coordinate := F;
  Last := Coordinate := L;
  loop
    loop
      if First = Last then
        Middle := First;
        Middle_OK := Test(S, First);
        return;
      end if;
      exit when not Test(S, First);
      First := Next(First);
    end loop;
    loop
      exit when Test(S, Last);
      Last := Prev(Last);
      if First = Last then
        Middle := First;
        Middle_OK := False;
        return;
      end if;
    end loop;
    Step(S, First, Last);
    First := Next(First);
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Last := Prev(Last);
  end loop;
end Partition;

Figure 1: Body of Partition Algorithm
procedure Partition(S : in out Sequence;
    F, L : in Coordinate;
    Middle : out Coordinate;
    Middle_OK : out Boolean) is

    First : Coordinate := F;
    Last : Coordinate := L;
begin
loop
    loop
        if First = Last then
            Middle := First;
            Middle_OK := Test(S, First);
            return;
        end if;
        exit when not Test(S, First);
        First := Next(First);
    end loop;
end loop;

loop
    exit when Test(S, Last);
    Last := Prev(Last);
    if First = Last then
        Middle := First;
        Middle_OK := False;
        return;
    end if;
end loop;
Swap(S, First, Last);
First := Next(First);
if First = Last then
    Middle := First;
    Middle_OK := False;
    return;
end if;
Last := Prev(Last);
end loop;
end Partition;
procedure Partition(S : in out Sequence; F, L : in Coordinate; Middle : out Coordinate; Middle_OK : out Boolean) is

    First : Coordinate := F;
    Last  : Coordinate := L;
begin
    loop
        loop
            if First = Last then
                Middle := First;
                Middle_OK := Test(S, First);
                return;
            end if;

            exit when not Test(S, First);
            First := Next(First);
        end loop;
    end loop;
end Partition;

loop
    exit when Test(S, Last);
    Last := Prev(Last);
    if First = Last then
        Middle := First;
        Middle_OK := False;
        return;
    end if;
end loop;

Swap(S, First, Last);
First := Next(First);
if First = Last then
    Middle := First;
    Middle_OK := False;
    return;
end if;
Last := Prev(Last);
end loop;
end Partition;
procedure Partition(S : in out Sequence;
    F, L : in Coordinate;
    Middle : out Coordinate;
    Middle_OK : out Boolean) is

    First : Coordinate := F;
    Last  : Coordinate := L;

begin
    loop
        loop
            if First = Last then
                Middle := First;
                Middle_OK := Test(S, First);
                return;
            end if;
            exit when not Test(S, First);
            First := Next(First);
        end loop;
    end loop;

do while (F <= L)
        exit when Test(S, Last);
        Last := Prev(Last);
        if First = Last then
            Middle := First;
            Middle_OK := False;
            return;
        end if;
    end do while (F <= L)

Swap(S, First, Last);
First := Next(First);
if First = Last then
    Middle := First;
    Middle_OK := False;
    return;
end if;
Last := Prev(Last);
end loop;
end Partition;
procedure Partition(S : in Sequence; F, L : in out Coordinate; Middle, Middle_OK : out Boolean) is
  begin
    Loop
    exit when not Test(S, First);
    exit if First = Last then
      Middle := Next(First);
      First := Prev(First);
      Middle := Last;
      Last := Prev(Last);
    end if;
    return;
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
    Loop
    if First = Last then
      Middle := First;
      Middle_OK := False;
      return;
    end if;
    Middle := Next(Middle);
    First := Last;
    Last := Prev(Last);
  end Loop
end Partition;
David R. Musser
Alexander A. Stepanov

The Ada®
Generic Library

Linear List Processing Packages

Springer-Verlag
The Mathematical Foundation of QuickDraw

To create graphics that are both precise and pretty requires not supercharged features but a firm mathematical foundation for the graphic package features you have. If the mathematics that underlie a graphic package are imprecise or fuzzy, the graphic will be, too. QuickDraw defines some clear mathematical constructs that are widely used in its procedures, functions, and data types: the coordinate plane, the point, the rectangle, and the region.

The Coordinate Plane

All information about location, placement, or movement that you give to QuickDraw is in terms of coordinates on a plane. The coordinate plane is a two-dimensional grid, as illustrated in Figure 2.

There are two distinctive features of the QuickDraw coordinate plane:

- All grid coordinates are integers.
- All grid lines are infinitely thin.

These concepts are important: first, they mean that the QuickDraw plane is finite, not infinite (although it's very large). Horizontal coordinates range from -32768 to +32767, and vertical coordinates have a range of -32768. An auxiliary package is available that maps real Cartesian space, with x, y, and z coordinates, onto QuickDraw's twodimensional integer coordinate system.

Second, they mean that all elements represented on the coordinate plane are mathematically pure. Mathematical calculations using integer arithmetic will produce intuitively correct results. If you keep in mind that grid lines are infinitely thin, you'll never have "endpoint problems" -- the confusion that results from not knowing whether that last dot is included in the line.

Points

On the coordinate plane are 4,294,967,296 unique points. Each point is at the intersection of a horizontal grid line and a vertical grid line. As the grid lines are infinitely thin, a point is infinitely small. Of course there are more points on this grid than there are dots on the Macintosh screen when using QuickDraw you associate small parts of the grid with areas on the screen, so that you aren't bound into an arbitrary, limited coordinate space.

The coordinate origin (0,0) is in the middle of the grid. Horizontal coordinates increase as you move from left to right, and vertical coordinates increase as you move from top to bottom. This is the way both a TV screen and a page of English text are scanned: from the top left to the bottom right.

You can store the coordinates of a point into a Pascal variable whose type is defined by QuickDraw. The type Point is a record of two integers, and has this structure:

```pascal
  TYPE TSelect = (V,N);  
  POINT = RECORD CASE TSelect OF  
    V: (x: INTEGER;  
       y: INTEGER);  
    N: (x: ARRAY [0..7] OF INTEGER));  
END;
```

The variant part allows you to access the vertical and horizontal components of a point either individually or as an array. For example, if the variable point were declared to be of type Point, the following would all refer to the coordinate parts of the point:

```pascal
point.X  
point.Y  
point[V]  
point[1]`
THE MATHEMATICAL FOUNDATION OF QUICKDRAW

To create graphics that are both precise and pretty requires not supercharged features but a firm mathematical foundation for the procedures, functions, and data types: the coordinate plane, the procedures, functions, and data types: the coordinate plane, the

movement, or movement that you give to graphics on a plane. The coordinate plane is illustrated in Figure 2.

The coordinate plane of the QuickDraw coordinate plane:

- All grid coordinates are integers.
- All grid lines are infinitely thin.

These concepts are important: First, they mean that the QuickDraw plane is finite, not infinite (although it's very large). Horizontal coordinates range from -32768 to +32767, and vertical coordinates have the same range. (An auxiliary package is available that maps real Cartesian space, with x, y, and z coordinates, onto QuickDraw's two-dimensional integer coordinate system.)

Second, they mean that all elements represented on the coordinate plane are mathematically pure. Mathematical calculations using integer arithmetic will produce definitely correct results. If you keep in mind that grid lines are infinitely thin, you'll never have "endpoint paranoia", the confusion that results from not knowing whether that last dot is included in the line.

Points

On the coordinate plane are 4,294,967,296 unique points. Each point is at the intersection of a horizontal grid line and a vertical grid line. As the grid lines are infinitely thin, a point is infinitely small. Of course there are more points on this grid than there are dots on the Macintosh screen: when using QuickDraw you associate small parts of the grid with areas on the screen, so that you aren't bound into an arbitrary, limited coordinate space.

The coordinate origin (0,0) is in the middle of the grid. Horizontal coordinates increase as you move from left to right, and vertical coordinates increase as you move from top to bottom. This is the way both a TV screen and a page of English text are numbered: from the top left to the bottom right.

You can store the coordinates of a point into a Pascal variable whose type is defined by QuickDraw. The type Point is a record of two integers, and has this structure:

```
TYPE VWSelect = (V, W);
Point = RECORD CASE INTEGER OF
  0: (vo INTEGER);
  1: (hi INTEGER);
  2: (ho ARRAY [VWSelect] OF INTEGER)
END;
```

The variant part allows you to access the vertical and horizontal components of a point either individually or as an array. For example, if the variable point were declared to be of type Point, the following would all refer to the coordinate parts of the point:

```
3/12/83 Espinosas-Rosco
```
- All grid coordinates are integers.
- All grid lines are infinitely thin.

These concepts are important! ...they mean that all elements represented on the coordinate plane are mathematically pure. Mathematical calculations using integer arithmetic will produce intuitively correct results. If you keep in mind that the grid lines are infinitely thin, you’ll never have “endpoint paranoia” — the confusion that results from not knowing whether that last dot is included in the line.
QuickDraw Routines to operate on Regions.

.PROC StdRgn,2
.REF CheckPic,PutPicVerb,DPutPicByte,PutPicRgn
.REF PutPic,Rgn,PushVerb,DrawRgn

PROCEDURE StdRgn(verb: GrafVerb; rgn: RgnHandle);

A6 OFFSETS OF PARAMS AFTER LINK:

PARAMSIZE .EQU 6
VERB .EQU PARAMSIZE+8-2 ;GRAFVERB
RGN .EQU VERB-4 ;LONG, RGNHANDLE

LINK A6,#0 ;NO LOCALS
MOVEM.L D6-D7/A2-A4,-(SP) ;SAVE REGS
MOVE.B VERB(A6),D7 ;GET VERB
JSR CHECKPIC ;SET UP A4,A3 AND CHECK PICSAVE
BLE.S NOTPIC ;BRANCH IF NOT PICSAVE

MOVE.B D7,-(SP) ;PUSH VERB
JSR PutPicVerb ;PUT ADDITIONAL PARAMS TO THEPIC
MOVE #$80,D0 ;PUT RGNNOUN IN HI NIBBLE
ADD D7,D0 ;PUT VERB IN LO NIBBLE
JSR DPutPicByte ;PUT OPCODE TO THEPIC
MOVE.L RGN(A6),-(SP) ;PUT RGNHANDLE
JSR PutPicRgn ;PUT REGION TO THEPIC

NOTPIC MOVE.L RGN(A6),-(SP) ;PUSH RGNHANDLE
JSR PushVerb ;PUSH MODE AND PATTERN
TST.B D7 ;IS VERB FRAME ?
BNE.S NOTFR ;NO, CONTINUE
Gather

template <typename I, // I models BidirectionalIterator
typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
}
template <typename I, // I models BidirectionalIterator
typename S> // S models UnaryPredicate
auto gather(I f, I l, I p, S s) -> pair<I, I>
{
    return { stable_partition(f, p, not1(s)),
             stable_partition(p, l, s) };
}
For a sequence of $n$ elements there are $n + 1$ positions
1993

Alex resumes work on Generic Programming
Andrew Koenig suggests writing a standard library proposal
The Standard Template Library

Alexander Stepanov
Silicon Graphics Inc.
2011 N. Shoreline Blvd.
Mt. View, CA 94043
stepanov@mti.sgi.com

Meng Lee
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
lee@hpl.hp.com

October 31, 1995
The Standard Template Library

Alexander Stepanov

Silicon Graphics Inc.
2011 N. Shoreline Blvd.
Mt. View, CA 94043
stepanov@mti.sgi.com

Meng Lee

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
lee@hpl.hp.com
1983
WRITING CORRECT PROGRAMS

In the late 1960s people were talking about the promise of programs that verify the correctness of other programs. Unfortunately, it is now the middle of the 1980s, and, with precious few exceptions, there is still little more than talk about automated verification systems. Despite unrealized expectations, however, the research on program verification has given us something more valuable than a black box that gives programs and flails "good" or "bad"—we now have a fundamental understanding of computer programming.

The purpose of this column is to show how that fundamental understanding can help programmers write correct programs and flashes "good" or "bad"—in the mathematical sense of those terms. (We'll use "good" and "bad" to represent the range of values of a variable.)

Even with the best of designs, every now and then a program and data structure selection. If you perform those tasks well, then writing correct code is usually easy.

The Challenge of Binary Search

Even with the best of designs, every now and then a program has to write working code. This column is about one problem that requires particularly careful code binary search. After defining the problem and sketching an algorithm to solve it, we'll use principles of program verification in several stages as we develop the program.

The problem is to determine whether the sorted array $X[1..N]$ contains the element $T$. Precisely, we know that $N$ is not 0 and that $X[1] < X[2] < \ldots < X[N]$. The types of $T$ and the elements of $X$ are (the same; the pseudocode should work equally well for integers, reals or strings. The answer is stored in a variable $Found$.

The key idea of binary search is that we always know that if $T$ is anywhere in $X[1..N]$, then it must be in a certain range of the array. Initially, the range is the entire array. The range is known to be empty. The process makes roughly $\log N$ comparisons.

Binary search solves the problem by keeping track of a range within the array in which $T$ must be if it is anywhere in the array. Initially, the range is the entire array. The range is diminished by comparing its middle element to $T$ and discarding half the range. This process continues until $T$ is discovered in the array or until the range in which it must lie is known to be empty. The process makes roughly $\log N$ comparisons.

Most programmers think that with the above description in hand, writing the code is easy; they're wrong. The only way you'll believe this is by putting down this column right now, and writing the code yourself. Try it.

I've given this problem as an in-class assignment in courses at Bell Labs and IBM. The professional programmers had one hour (approximately enough to convert the above description into a program in the language of their choice; a high-level pseudocode was fine.) At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take 30 minutes to examine their code, which the programmers did with test cases. In many different classes and with over a hundred programmers, the results varied little: 90 percent of the programmers found bugs in their code (I've always considered the correctness of the problem specifications to be a matter of design, not verification) and 5 percent of the programmers had bugs in their code. The only way we know this is by putting down this column right now, and writing the code ourselves. Try it.

The Challenge of Binary Search

The problem is to determine whether the sorted array $X[1..N]$ contains the element $T$. Precisely, we know that $N$ is not 0 and that $X[1] < X[2] < \ldots < X[N]$. The types of $T$ and the elements of $X$ are (the same; the pseudocode should work equally well for integers, reals or strings. The answer is stored in a variable $Found$.

The key idea of binary search is that we always know that if $T$ is anywhere in $X[1..N]$, then it must be in a certain range of the array. Initially, the range is the entire array. The range is known to be empty. The process makes roughly $\log N$ comparisons.

Binary search solves the problem by keeping track of a range within the array in which $T$ must be if it is anywhere in the array. Initially, the range is the entire array. The range is diminished by comparing its middle element to $T$ and discarding half the range. This process continues until $T$ is discovered in the array or until the range in which it must lie is known to be empty. The process makes roughly $\log N$ comparisons.

Most programmers think that with the above description in hand, writing the code is easy; they're wrong. The only way you'll believe this is by putting down this column right now, and writing the code yourself. Try it.

Writing the Program

The key idea of binary search is that we always know that if $T$ is anywhere in $X[1..N]$ then it must be in a certain range of $X$. We'll use the shorthand $X[\text{low}..\text{high}]$ to mean that if $T$ is anywhere in the array, then it must be in range. With this notation, it's easy to convert the above description of binary search into a program sketch.

```
write "the middle of the range use M as a probe to shrink the range if T is found during the shrinking process, return its position
```

```
initialize range to designate X[1..N]
```

```
loop
{ invariant: MustBe(range) }
if range is empty, return that T is numbers in the range, compute M, the middle of the range use M as a probe to shrink the range if T is found during the shrinking process, return its position
```

The crucial part of this program is the loop invariant, which is expressed in (1). This is an assertion about the program state that is invariantly true at the beginning and end of each iteration of the loop (hence its name); it formalizes the intuitive notion we had above.

```
if range is empty, return that T is numbers in the range
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
By Jon Bentley

WRITING CORRECT PROGRAMS

The Challenge of Binary Search

Even with the best of design, every now and then a programmer has to write usable code. This column is about one problem that requires particularly careful code: binary search. After defining the problem and sketching an algorithm to solve it, we’ll use principles of program verification in several stages as we develop the program.

The problem is to determine whether the sorted array $X[1..N]$ contains the element $T$. Precisely, we know that $N > 0$ and that $X[1] < X[2] < \ldots < X[N]$. The types of $T$ and the elements of $X$ are the same; the pseudocode should work equally well for integers, reals, or strings. The process is described in the following manner:

- Binary search solves the problem by keeping track of a range within the array in which $T$ must be if it is anywhere in the array. Initially, the range is the entire array. The range is diminished by comparing its middle element to $T$ and discarding half the range. This process continues until $T$ is discovered in the array or until the range in which it must lie is known to be empty. The process makes roughly $\log_2 N$ comparisons.

Most programmers think that with the above description in hand, writing the code is easy; they’re wrong. The only way you’ll believe this is by putting down this column right now, and writing the code yourself. Try it.

To reproduce the above description for all or part of this material is granted provided that the replica or replicas are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the column are included, and credit is given to the author and the Communications of the ACM. For further information, or to reproduce, require a fee for specific permissions. © 1983 ACM 0001-0782/83/1200-1040 75¢

The key idea of binary search is that we always know that if $T$ is anywhere in $X[1..N]$, then it must be in a certain range of $X$. We’ll use the shorthand `$L \leq T \leq U$' to mean that if $T$ is anywhere in the array, then it must lie in range. With this notation, it’s easy to convert the above description of binary search into a program sketch.

```
initialise range to designate $X[1..N]$
loop
  invariant: MustBe(range)
  if range is empty, return that $T$ is nowhere in the array
  compute $L$, the middle of the range, as a probe to shrink the range if $T$ is found during the shrinking process, return its position
endloop
```

The crucial part of this process is the loop invariant, which is enshrined in `MustBe(range)`. This is an assertion about the program state that is invariantly true at the beginning and end of each iteration of the loop (hence its name) if formulated the intuitive notion we had above.

We’ll now refine the program, making sure that all our actions respect the invariant. The first issue we must first is the representation of range: we’ll use two indices $L$ and $U$ (the “lower” and “upper”) to represent the range $[L..U]$. There are other possible representations for a range, such as its length...
“I’ve assigned this problem [binary search] in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert the description into a programming language of their choice; a high-level pseudo code was fine… Ninety percent of the programmers found bugs in their programs (and I wasn’t always convinced of the correctness of the code in which no bugs were found).”
– Jon Bentley, Programming Pearls
“I want to hire the other ten percent.”
– Mark Hamburg, Photoshop Lead
“I want to hire the other ten percent.”
– Mark Hamburg, Photoshop Lead
Jon Bentley's Solution (translated to C++)

```c
int binary_search(int x[], int n, int v) {
    int l = 0;
    int u = n;

    while (true) {
        if (l > u) return -1;

        int m = (l + u) / 2;

        if (x[m] < v) l = m + 1;
        else if (x[m] == v) return m;
        else /* (x[m] > v) */ u = m - 1;
    }
}
```
int binary_search(int x[], int n, int v) {
    int l = 0;
    int u = n;

    while (true) {
        if (l > u) return -1;

        int m = (l + u) / 2;

        if (x[m] < v) l = m + 1;
        else if (x[m] == v) return m;
        else /* (x[m] > v) */ u = m - 1;
    }
}
```c
int binary_search(int x[], int n, int v) {
    int l = 0;
    int u = n;

    while (true) {
        if (l > u) return -1;

        int m = (l + u) / 2;
        if (x[m] < v) l = m + 1;
        else if (x[m] == v) return m;
        else /* (x[m] > v) */ u = m - 1;
    }
}
```
Jon Bentley's Solution (translated to C++)

```c++
int binary_search(int x[], int n, int v) {
    int l = 0;
    int u = n;

    while (true) {
        if (l > u) return -1;

        int m = (l + u) / 2;

        if (x[m] < v) l = m + 1;
        else if (x[m] == v) return m;
        else /* (x[m] > v) */ u = m - 1;
    }
}
```
Jon Bentley’s Solution (translated to C++)

```c
int binary_search(int x[], int n, int v) {
    int l = 0;
    int u = n;

    while (true) {
        if (l > u) return -1;

        int m = (l + u) / 2;

        if (x[m] < v) l = m + 1;
        else if (x[m] == v) return m;
        else /* (x[m] > v) */ u = m - 1;
    }
}
```
STL implementation

template <class I, // I models ForwardIterator
    class T> // T is value_type(I)
I lower_bound(I f, I l, const T& v) {
    while (f != l) {
        auto m = next(f, distance(f, l) / 2);

        if (*m < v) f = next(m);
        else l = m;
    }
    return f;
}
STL implementation

template <class I, // I models ForwardIterator
class T> // T is value_type(I)
I lower_bound(I f, I l, const T& v) {
    while (f != l) {
        auto m = next(f, distance(f, l) / 2);

        if (*m < v) f = next(m);
        else l = m;
    }
    return f;
}
template <class I, // I models ForwardIterator
         class T> // T is value_type(I)
I lower_bound(I f, I l, const T& v) {
    while (f != l) {
        auto m = next(f, distance(f, l) / 2);
        if (*m < v) f = next(m);
        else l = m;
    }
    return f;
}
STL implementation

template <class I, // I models ForwardIterator
class T> // T is value_type(I)
I lower_bound(I f, I l, const T& v) {
    while (f != l) {
        auto m = next(f, distance(f, l) / 2);

        if (*m < v) f = next(m);
        else l = m;
    }
    return f;
}
template <class I, // I models ForwardIterator
         class T> // T is value_type(I)
I lower_bound(I f, I l, const T& v) {
    while (f != l) {
        auto m = next(f, distance(f, l) / 2);
        if (*m < v) f = next(m);
        else l = m;
    }
    return f;
}

STL implementation

template <class I, // I models ForwardIterator
          class T> // T is value_type(I)
I lower_bound(I f, I l, const T& v) {
    while (f != l) {
        auto m = next(f, distance(f, l) / 2);

        if (*m < v) f = next(m);
        else l = m;
    }
    return f;
}
1998
Programming languages — C++

Langages de programmation — C++
Programming languages — C++

Langages de programmation — C++
Exception-Safety in Generic Components
Lessons Learned from Specifying Exception-Safety for the C++ Standard Library

David Abrahams
Dragon Systems
David.Abrahams@dragonsys.com

Abstract. This paper represents the knowledge accumulated in response to a real-world need: that the C++ Standard Template Library exhibit useful and well-defined interactions with exceptions, the error-handling mechanism built-in to the core C++ language. It explores the meaning of exception-safety, reveals surprising myths about exceptions and genericity, describes valuable tools for reasoning about program correctness, and outlines an automated testing procedure for verifying exception-safety.

Keywords: exception-safety, exceptions, STL, C++

1 What Is Exception-Safety?
Informally, exception-safety in a component means that it exhibits reasonable behavior when an exception is thrown during its execution. For most people, the term “reasonable” includes all the usual expectations for error-handling: that resources should not be leaked, and that the program should remain in a well-defined state so that execution can continue. For most components, it also includes the expectation that when an error is encountered, it is reported to the caller.

More formally, we can describe a component as minimally exception-safe if, when exceptions are thrown from within that component, its invariants are intact. Later on we’ll see that at least three different levels of exception-safety can be usefully distinguished. These distinctions can help us to describe and reason about the behavior of large systems.

In a generic component, we usually have an additional expectation of exception-neutrality, which means that exceptions thrown by a component’s type parameters should be propagated, unchanged, to the component’s caller.

2 Myths and Superstitions
Exception-safety seems straightforward so far: it doesn’t constitute anything more than we’d expect from code using more traditional error-handling techniques. It might be worthwhile, however, to examine the term from a psychological viewpoint. Nobody ever spoke of “error-safety” before C++ had exceptions.

Exception-Safety in Generic Components
Lessons Learned from Specifying Exception-Safety for the C++ Standard Library

David Abrahams

Dragon Systems
David.Abrahams@dragonsys.com

More formally, we can describe a component as minimally exception-safe if, when exceptions are thrown from within that component, its invariants are intact. Later on we’ll see that at least three different levels of exception-safety can be usefully distinguished. These distinctions can help us to describe and reason about the behavior of large systems.

In a generic component, we usually have an additional expectation of exception-neutrality, which means that exceptions thrown by a component’s type parameters should be propagated, unchanged, to the component’s caller.

2 Myths and Superstitions
Exception-safety seems straightforward so far: it doesn’t constitute anything more than we’d expect from code using more traditional error-handling techniques. It might be worthwhile, however, to examine the term from a psychological viewpoint. Nobody ever spoke of “error-safety” before C++ had exceptions.

Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov
Silicon Graphics, Inc.
dehnert@acm.org, stepanov@attlabs.att.com

Keywords: Generic programming, operator semantics, concept, regular type.

Abstract. Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.

Copyright © Springer-Verlag. Appears in Lecture Notes in Computer Science (LNCS) volume 1766. See http://www.springer.de/computing/index.html
Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov
Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

Abstract.
Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.
"We call the set of axioms satisfied by a data type and a set of operations on it a concept."
“We call the set of axioms satisfied by a data type and a set of operations on it a **concept**.”
“Since we wish to extend semantics as well as syntax from built-in types to user types, we introduce the idea of a regular type, which matches the built-in type semantics, thereby making our user-defined types behave like built-in types as well.”
“Since we wish to extend semantics as well as syntax from built-in types to user types, we introduce the idea of a regular type, which matches the built-in type semantics, thereby making our user-defined types behave like built-in types as well.”
NOTES ON THE FOUNDATIONS OF PROGRAMMING

ALEX STEPANOV AND MAT MARCUS

Disclaimer: Please do not redistribute. Instead, requests for a current draft should go to Mat Marcus. These notes are a work in progress and do not constitute a book. In particular, most of the current effort is directed towards writing up new material. As a consequence little time remains for structuring, refinement, or clean up, so please be patient. Nevertheless, suggestions, comments and corrections are welcomed. Please reply to mmarcus@adobe.com and stepanov@adobe.com.
NOTES ON THE FOUNDATIONS OF PROGRAMMING

ALEX STEPANOV AND MAT MARCUS

Disclaimer: Please do not redistribute. Instead, requests for a current draft should go to Mat Marcus. These notes are a work in progress and do not constitute a book. In particular, most of the current effort is directed towards writing up new material. As a consequence, little time remains for structuring, refinement, or clean up, so please be patient. Nevertheless, suggestions, comments and corrections are welcomed. Please reply to mmarcus@adobe.com and stepanov@adobe.com.
Elements of Programming

Alexander Stepanov
Paul McJones
Elements of Programming

Alexander Stepanov
Paul McJones
template <typename I, typename P>
   requires(Mutable(I) && ForwardIterator(I) &&
            UnaryPredicate(P) && ValueType(I) == Domain(P))
I partition_semistable(I f, I l, P p) {
   // Precondition: mutable_bounded_range(f, l)
   I i = find_if(f, l, p);
   if (i == l) return i;
   I j = successor(i);
   while (true) {
      j = find_if_not(j, l, p);
      if (j == l) return i;
      swap_step(i, j);
   }
}
template <typename I, typename P>
requires(Mutable(I) && ForwardIterator(I) &&
    UnaryPredicate(P) && ValueType(I) == Domain(P))
I partition_semistable(I f, I l, P p) {
    // Precondition: mutable_bounded_range(f, l)
    I i = find_if(f, l, p);
    if (i == l) return i;
    I j = successor(i);
    while (true) {
        j = find_if_not(j, l, p);
        if (j == l) return i;
        swap_step(i, j);
    }
}
Appendix B. Programming Language

Sean Parent and Bjarne Stroustrup

This appendix defines the subset of C++ used in the book. To simplify the syntax, we use a few library facilities as intrinsics. These intrinsics are not written in this subset but take advantage of other C++ features. Section B.1 defines this subset; Section B.2 specifies the implementation of the intrinsics.

B.1 Language Definition

Syntax Notation

An Extended Backus-Naur Form designed by Niklaus Wirth is used. Wirth [1974, pages 822–823] describes it as follows:

The word identifier is used to denote nonterminal symbol, and literal stands for terminal symbol. For brevity, identifier and character are not defined in further detail.

```
syntax = {production}.
production = identifier "=" expression ".",
expression = term {(" term).
term = factor {(" term).
factor = identifier | literal
     | "(" expression ")"
     | "(" expression ")",
literal = """" character {character} """".
```

Repetition is denoted by curly brackets, i.e., {a} stands for a |a|a|a|a|a|a and [a] Optionality is expressed by square brackets, i.e., [a] stands for a | . Parentheses merely serve for grouping, e.g., (a)b|c stands for ac|bc.

Terminal symbols, i.e., literals, are enclosed in quote marks (and, if a quote mark appears as a literal itself, it is written twice). Lexical Conventions

The following productions give the syntax for identifiers and literals:
Appendix B. Programming Language

Sean Parent and Bjarne Stroustrup

8. Language Definition

Syntax Notation

An Extended Backus-Naur Form designed by Niklaus Wirth is used. Wirth [1973, pages 822–823] describes it as follows:

The word *identifier* is used to denote *nonterminal symbol*, and *literal* stands for *terminal symbol*. For brevity, *identifier* and *character* are not defined in further detail.

```plaintext
syntax = {production},
production = identifier "=" expression ";",
expression = term { ( "term" ) },
term = factor {factor},
factor = identifier { literal |
| "(" expression ")" |
| "(" expression ")" |
| "(" expression ")" },
literal = "" character { character } "".
```

Repetition is denoted by curly brackets, i.e., \{a\} stands for \(a|aa|aaa\) \(...\). Optionality is expressed by square brackets, i.e., \[a\] stands for \(a|\epsilon\). Parentheses merely serve for grouping, e.g., \(a|b|c\) stands for \(a|bc\). Terminal symbols, i.e., literals, are enclosed in quote marks (and, if a quote mark appears as a literal itself, it is written twice).

Lexical Conventions

The following productions give the syntax for identifiers and literals:
The while statement repeatedly evaluates the expression and executes the statement as long as the expression is true. The do statement repeatedly executes the statement and evaluates the expression until the expression is false. In either case, the expression must evaluate to a Boolean.

The compound statement executes the sequence of statements in order.

The goto statement transfers execution to the statement following the corresponding label in the current function.

The break statement terminates the execution of the smallest enclosing switch, while, or do statement; execution continues with the statement following the terminated statement.

The typedef statement defines an alias for a type.

**Templates**

A template allows a structure or procedure to be parameterized by one or more types or constants. Template definitions and template names use `<` and `>` as delimiters [2]

[2] To disambiguate between the use of `<` and `>` as relations or as template name delimiters, once a structure_name or procedure_name is parsed as part of a template, it becomes a terminal symbol.

```
template = template_decl | template | procedure | specialization;
specialization = "struct" structure_name "(" additive_list "")" |
[structure_body] ";";
template_decl = "template" "(" parameter_list ")" [constraint];
constraint = "requires" "(" expression ");";
template_name = structure_name | procedure_name;
["(" additive_list ")"];
additive_list = additive ("," additive)*;
```

When a template_name is used as a primary, the template definition is used to generate a structure or procedure with template parameters replaced by corresponding template arguments. These template arguments are either given explicitly as the delimited expression list in the template_name or, for procedures, may be deduced from the procedure argument types.
The while statement repeatedly evaluates the expression and executes the statement as long as the expression is true. The do statement repeatedly executes the statement and evaluates the expression until the expression is false. In either case, the expression must evaluate to a Boolean.

The compound statement executes the sequence of statements in order.

The goto statement transfers execution to the statement following the corresponding label in the current function.

The break statement terminates the execution of the smallest enclosing switch, while, or do statement; execution continues with the statement following the terminated statement.

The typedef statement defines an alias for a type.

Templates

A template allows a structure or procedure to be parameterized by one or more types or constants. Template definitions and template names use `<` and `>` as delimiters.[2]

[2] To disambiguate between the use of `<` and `>` as relations or as template name delimiters, once a structure_name or procedure_name is

```
template_decl = "template" "<" (parameter_list) ">" [constraint].
constraint = "requires" "(" expression ")".
```

When a template_name is used as a primary, the template definition is used to generate a structure or procedure with template parameters replaced by corresponding template arguments. These template arguments are either given explicitly as the delimited expression list in the template_name or, for procedures, may be deduced from the procedure argument types.
This concept describes a homogeneous functional procedure:

\[ \text{HomogeneousFunction}(F) \triangleq \text{FunctionalProcedure}(F) \]
\[ \land \text{Arity}(F) > 0 \]
\[ \land (\forall i, j \in \mathbb{N})(i, j < \text{Arity}(F)) \Rightarrow (\text{InputType}(F, i) = \text{InputType}(F, j)) \]
\[ \land \text{Domain: HomogeneousFunction} \rightarrow \text{Regular} \]
\[ F \mapsto \text{InputType}(F, 0) \]
2006
Concepts: Linguistic Support for Generic Programming in C++

Douglas Gregor
Indiana University
dgregor@osl.iu.edu

Jaakko Järvi
Texas A&M University
jarvi.cs.tamu.edu

Jeremy Siek
Rice University
jeremy@cs.rice.edu

Bjarne Stroustrup
Texas A&M University
bjarne@cs.tamu.edu

Gabriel Dos Reis
Texas A&M University
gdr@cs.tamu.edu

Andrew Lumadue
Indiana University
lumadue@osl.iu.edu

Abstract
Generic programming has emerged as an important technique for the development of highly reusable and efficient software libraries. In C++, generic programming is enabled by the flexibility of template-based parameterization, which allows templates to perform any operation on their parameters, including compile-time type computations, allowing the simulation of the basic features required for diverse programming paradigms. Another essential part of templates is their ability to provide abstraction without performance degradation: templates provide succinct, reusable information to a compiler's optimizers (especially the inliner) to generate code that is optimal in both time and space. Consequently, templates have become the preferred implementation style for a vast array of reusable, efficient C++ libraries. Templates are unconstrained, and their misuse results in notoriously confusing error messages. Consider:

```cpp
type sort(const list&);
```

...in the upcoming revision of the ISO C++ standard, C++0x. Aided by the discovery of numerous template techniques [28, 46, 56, 66, 67], C++ libraries are becoming more powerful, more flexible, and more expressive. However, these improvements come at the cost of implementation complexity [6, 41]: authors of C++ libraries typically rely on a grab-bag of template tricks, many of which are complex and poorly documented. Where library interfaces are rigorously separated from library implementations, the complexity of implementation of a library is not a problem for its users. However, templates rely on the absence of modular (separate) type-checking for flexibility and performance. Therefore, the complexities of library implementation leak through to library users. This problem manifests itself most visibly as spectacularly poor error messages for simple mistakes. Consider:

```cpp
sort_list.cpp:8: instantiated from here
```

Attempting to compile this code with a recent version of the GNU C++ compiler [37] produces more than two kilobytes of output, containing six different error messages. Worse, the errors reported provide line numbers and file names that point to the implementation of the STL `sort` function and its helper functions. The only clue provided to users that this error was triggered by their own code (rather than by a bug in the STL implementation) is the following innocuous line of output:

```cpp
sort_list.cpp:8: instantiated from here
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or a fee.

October 22-26, 2006, Portland, Oregon, USA.

Permission © 2006 ACM 1-59593-348-4/06/0010.. . . $5.00.

Abstract
Generic programming has emerged as an important technique for the development of highly reusable and efficient software libraries. In C++, generic programming is enabled by the flexibility of templates, the C++ type parameterization mechanism. However, the flexibility of templates comes at the cost of increased complexity: templates are more difficult to use and develop than non-template libraries and their misuse results in notoriously confusing error messages. As currently defined in C++, templates are unconstrained, and type-checking of templates is performed late in the compilation process, i.e., after the use of a template has been combined with its definition. To improve the support for generic programming in C++, we introduce concepts to express the syntactic and semantic behavior of templates and to constrain the type parameters in a C++ template. Using concepts, type-checking of template definitions is separated from their uses, thereby making templates easier to use and easier to compile. These improvements are achieved without limiting the flexibility of templates or decreasing their performance—in fact, their expressive power is increased. This paper describes the language extensions supporting concepts, their use in the expression of the C++ Standard Library, and their implementation in the ConceptGCC compiler. Concepts are candidates for inclusion in the upcoming revision of the ISO C++ standard.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features—Abstract data types; D.3.3 [Programming Languages]: Language Constructs and Features—Type systems; D.3.3 [Programming Languages]: Type/Class Libraries—Standard-Library classes.

General Terms
Design, Languages

Keywords
Generic programming, constrained generics, parameter polymorphism, C++ templates, C++, concepts

1. Introduction
The C++ language [25, 62] supports parameterized types and functions in the form of template. Templates provide a unique combination of features that have allowed them to be used for many different programming paradigms, including Generic Programming [3, 44], Generative Programming [11], and Template Meta-programming [4, 40]. Much of the flexibility of C++ templates comes from their unconstrained nature: a template can perform any operation on its template parameters, including compile-time type computations, allowing the simulation of the basic features required for diverse programming paradigms. Another essential part of templates is their ability to provide abstraction without performance degradation: templates provide succinct, reusable information to a compiler's optimizers (especially the inliner) to generate code that is optimal in both time and space. Consequently, templates have become the preferred implementation style for a vast array of reusable, efficient C++ libraries [2, 6, 14, 26, 32, 54, 68, 83]; many of which are built upon the Generic Programming methods exemplified by the C++ Standard Template Library (STL). In contrast to ad hoc techniques, the templates techniques [28, 46, 56, 66, 67], C++ libraries are becoming more powerful, more flexible, and more expressive. However, these improvements come at the cost of implementation complexity [6, 41]: authors of C++ libraries typically rely on a grab-bag of template tricks, many of which are complex and poorly documented. Where library interfaces are rigorously separated from library implementations, the complexity of implementation of a library is not a problem for its users. However, templates rely on the absence of modular (separate) type-checking for flexibility and performance. Therefore, the complexities of library implementation leak through to library users. This problem manifests itself most visibly as spectacularly poor error messages for simple mistakes. Consider:

```cpp
sort_list.cpp:8: instantiated from here
```

Attempting to compile this code with a recent version of the GNU C++ compiler [37] produces more than two kilobytes of output, containing six different error messages. Worse, the errors reported provide line numbers and file names that point to the implementation of the STL `sort` function and its helper functions. The only clue provided to users that this error was triggered by their own code (rather than by a bug in the STL implementation) is the following innocuous line of output:

```cpp
sort_list.cpp:8: instantiated from here
```

The actual error, in this case, is that the STL `sort` requires a pair of Random Access Iterators, i.e., iterators that can move any number of steps forward or backward in constant time. The STL...
In C++, generic programming is enabled by the flexibility of templates, the C++ type parameterization mechanism. However, the lack of constraint on template type parameters makes it more difficult to use and develops non-template libraries and their misuse results in notoriously confusing error messages. As currently defined in C++, templates are unconstrained, and type-checking of templates is performed late in the compilation process, i.e., after the use of a template has been combined with its definition. To improve the support for generic programming in C++, we introduce concepts to express the synthetic and semantic behavior of types and to constrain the type parameters in a C++ template. Using concepts, type-checking of template definitions is separated from their use, thereby making templates easier to use and easier to compile. These improvements are achieved without limiting the flexibility of templates or decreasing their performance—in fact, their expressive power is increased.

This paper describes the language extensions supporting concepts, their use in the expression of the C++ Standard Template Library, and their implementation in the ConceptGE compiler. Concepts are candidates for inclusion in the upcoming revisions of the ISO C++ standard, C++0x.

Permission is granted to copy, distribute and/or modify this document under the conditions of the GNU Free Documentation License v1.2 or any later version published by the Free Software Foundation. A copy of the license is included in the section entitled "GNU Free Documentation License".

Consequently, templates have become the preferred implementation style for a vast array of reusable, efficient C++ libraries [2, 6, 14, 16, 12, 34, 36, 58, 65], many of which have built upon the Generic Programming methodology exemplified by the C++ Standard Template Library (STL). C++ templates provide (at least) two major advantages: they allow the expression of a large variety of template techniques [28, 46, 56, 67], C++ libraries are becoming more powerful, more flexible, and more expressive.

However, these improvements come at the cost of implementation complexity [11, 63]; authors of C++ libraries typically rely on a grab-bag of template tricks, many of which are complex and poorly documented. When library interfaces are rigidly separated from library implementations, the complexity of implementation leak through to library users. This problem manifests itself most visibly in spectacularly poor error messages for simple mistakes. Consider the following innocuous line of output:

```
sort();
```

Permission is granted to copy, distribute and/or modify this document under the conditions of the GNU Free Documentation License v1.2 or any later version published by the Free Software Foundation. A copy of the license is included in the section entitled "GNU Free Documentation License".

The actual error, in this case, is that the STL `sort()` function and its helper functions. The only clue provided to users that this error was triggered by their own code (rather than by a bug in the STL implementation) is the following message:

```
no match for 'sort'
```

Consequently, templates have become the preferred implementation style for a vast array of reusable, efficient C++ libraries [2, 6, 14, 16, 12, 34, 36, 58, 65], many of which have built upon the Generic Programming methodology exemplified by the C++ Standard Template Library (STL). C++ templates provide (at least) two major advantages: they allow the expression of a large variety of template techniques [28, 46, 56, 67], C++ libraries are becoming more powerful, more flexible, and more expressive.

However, these improvements come at the cost of implementation complexity [11, 63]; authors of C++ libraries typically rely on a grab-bag of template tricks, many of which are complex and poorly documented. When library interfaces are rigidly separated from library implementations, the complexity of implementation leak through to library users. This problem manifests itself most visibly in spectacularly poor error messages for simple mistakes. Consider:

```
int x; sort(x);
```

The actual error, in this case, is that the STL `sort()` requires a pair of Random Access Iterators, i.e., iterators that can move any number of steps forward or backward in constant time. The STL `sort()` function and its helper functions. The only clue provided to users that this error was triggered by their own code (rather than by a bug in the STL implementation) is the following message:

```
no match for 'sort'
```

Consequently, templates have become the preferred implementation style for a vast array of reusable, efficient C++ libraries [2, 6, 14, 16, 12, 34, 36, 58, 65], many of which have built upon the Generic Programming methodology exemplified by the C++ Standard Template Library (STL). C++ templates provide (at least) two major advantages: they allow the expression of a large variety of template techniques [28, 46, 56, 67], C++ libraries are becoming more powerful, more flexible, and more expressive.

However, these improvements come at the cost of implementation complexity [11, 63]; authors of C++ libraries typically rely on a grab-bag of template tricks, many of which are complex and poorly documented. When library interfaces are rigidly separated from library implementations, the complexity of implementation leak through to library users. This problem manifests itself most visibly in spectacularly poor error messages for simple mistakes. Consider:

```
int x; sort(x);
```

The actual error, in this case, is that the STL `sort()` requires a pair of Random Access Iterators, i.e., iterators that can move any number of steps forward or backward in constant time. The STL `sort()` function and its helper functions. The only clue provided to users that this error was triggered by their own code (rather than by a bug in the STL implementation) is the following message:

```
no match for 'sort'
```

Consequently, templates have become the preferred implementation style for a vast array of reusable, efficient C++ libraries [2, 6, 14, 16, 12, 34, 36, 58, 65], many of which have built upon the Generic Programming methodology exemplified by the C++ Standard Template Library (STL). C++ templates provide (at least) two major advantages: they allow the expression of a large variety of template techniques [28, 46, 56, 67], C++ libraries are becoming more powerful, more flexible, and more expressive.

However, these improvements come at the cost of implementation complexity [11, 63]; authors of C++ libraries typically rely on a grab-bag of template tricks, many of which are complex and poorly documented. When library interfaces are rigidly separated from library implementations, the complexity of implementation leak through to library users. This problem manifests itself most visibly in spectacularly poor error messages for simple mistakes. Consider:

```
int x; sort(x);
```

The actual error, in this case, is that the STL `sort()` requires a pair of Random Access Iterators, i.e., iterators that can move any number of steps forward or backward in constant time. The STL `sort()` function and its helper functions. The only clue provided to users that this error was triggered by their own code (rather than by a bug in the STL implementation) is the following message:

```
no match for 'sort'
```

Consequently, templates have become the preferred implementation style for a vast array of reusable, efficient C++ libraries [2, 6, 14, 16, 12, 34, 36, 58, 65], many of which have built upon the Generic Programming methodology exemplified by the C++ Standard Template Library (STL). C++ templates provide (at least) two major advantages: they allow the expression of a large variety of template techniques [28, 46, 56, 67], C++ libraries are becoming more powerful, more flexible, and more expressive.

However, these improvements come at the cost of implementation complexity [11, 63]; authors of C++ libraries typically rely on a grab-bag of template tricks, many of which are complex and poorly documented. When library interfaces are rigidly separated from library implementations, the complexity of implementation leak through to library users. This problem manifests itself most visibly in spectacularly poor error messages for simple mistakes. Consider:

```
int x; sort(x);
```

The actual error, in this case, is that the STL `sort()` requires a pair of Random Access Iterators, i.e., iterators that can move any number of steps forward or backward in constant time. The STL `sort()` function and its helper functions. The only clue provided to users that this error was triggered by their own code (rather than by a bug in the STL implementation) is the following message:

```
no match for 'sort'
```
In C++, generic programming is enabled by the flexibility of templates, the C++ type parameterization mechanism. However, the expressive power of templates comes at a price: their implementation leak through to library users. This problem manifests itself most visibly in spectacularly poor error messages for simple mistakes. Consider:

```
sort_list.cpp:8: instantiated from here
```

The actual error, in this case, is that the STL `sort()` function and its helper functions. The only clue provided to users that this error was triggered by their own use of the STL is the file name and line number that point to the implementation of the STL `sort()` function and its helper functions. The only other clue provided to users that this error was triggered by their own code is the actual error message: an error message that states that a function was tried on an object that does not have the `random_access_iterator` and `bidirectional_iterator` requirements.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Permission to use digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

```
sort(lst.begin(), lst.end());
```

Attempting to compile this code with a recent version of the GNU C++ compiler [17] produces more than two kilobytes of output, containing six different error messages. Worse, the errors reported provide line numbers and file names that point to the implementation of the STL `sort()` function and its helper functions. The only clue provided to users that this error was triggered by their own use of the STL is the file name and line number that point to the implementation of the STL `sort()` function and its helper functions. The only other clue provided to users that this error was triggered by their own code is a poorly worded error message that states that a function was tried on an object that does not have the `random_access_iterator` and `bidirectional_iterator` requirements.

```
sort_list.cpp:8: instantiated from here
```

The actual error, in this case, is that the STL `sort()` requires a pair of Random Access Iterators, i.e., iterators that can move any number of steps forward or backward in constant time. The STL...
In C++, generic programming is enabled by the flexibility of templates. The C++ type parameterization mechanism allows users to declare functions and classes (templates) that can be used with any type of data. However, this flexibility can make it more difficult to use and develop than non-template libraries, and their misuse results in notoriously confusing error messages. As currently defined in C++98, templates are unimpaired, and type-checking of templates is performed late in the compilation process, i.e., after the use of a template has been combined with its definition. To improve the support for generic programming in C++, we introduce concepts to express the syntactic and semantic behavior of types and to constrain the type parameters in a C++ template. Using concepts, type-checking of template definitions is separated from their uses, thereby making templates easier to use and easier to compile. These improvements are achieved without limiting the flexibility of templates or decreasing their performance—in fact, their expressive power is increased. This paper describes the language extensions supporting concepts, their use in the expression of the C++ Standard Template Library, and their implementation in the ConceptGCC compiler. Concepts are candidates for inclusion in the upcoming revision of the ISO C++ standard, C++0x.

1. Introduction

The C++ language [25, 62] supports parameterized types and functions in the form of templates. Templates provide a unique combination of features that have allowed them to be used for many diverse programming paradigms. Another essential part of templates is their ability to provide abstraction without performance degradation: templates provide sufficient information to a compiler’s optimizer (especially the inliner) to generate code that is optimal in both time and space.

Consequently, templates have become the preferred implementation style for a vast array of reusable, compiler-generated C++ libraries (2, 6, 14, 20, 32, 54, 68), many of which are built upon the Generic Programming methodology exemplified by the C++ Standard Template Library (STL) [42, 60]. Aided by the discovery of numerous mistakes. Consider:

```cpp
int x = 0;
type t = int;
void f(t); // a template function
f(x); // a call to f
```

The actual error, in this case, is that the STL`s sort() function sorts lists of integers, and its implementation in the forthcoming revision of the ISO C++ standard, C++0x. However, templates rely on the absence of modular (separate) type-checking for flexibility and performance. Therefore, the complexity of library implementation leaks through to library users. This problem manifests itself most visibly as a spectacularly poor error messages for simple mistakes, Consider:

```cpp
pair<int, int> a;
```

Attempting to compile this code with a recent version of the GNU C++ compiler [17] produces more than two kilobytes of output, containing six different error messages. Worse, the errors reported provide line numbers and file names that point to the implementation of the STL sort() function and its helper functions. The only clue provided to users that this error was triggered by their own use of the STL is that the error message is followed by a message about the implementation of the STL sort() function:

```
sort_list.cpp:8: instantiated from here
```

This example points to a pair of Random Access Iterators, i.e., iterators that can move any number of steps forward or backward in constant time. The STL...
In C++, generic programming is enabled by the flexibility of templates, the C++ type parameterization mechanism. However, the current implementation of templates in C++ makes it more difficult to use and develop than non-template libraries and their misuse results in notoriously confusing error messages. As currently defined in C++98, templates are unconstrained, and type-checking of templates is performed late in the compilation process, i.e., after the use of a template has been combined with its definition. To improve the support for generic programming in C++, we introduce concepts to express the syntax and semantic behavior of types and to constrain the type parameters in a C++ template. Using concepts, type-checking of template definitions is separated from their use, thereby making templates easier to use and easier to compile. These improvements are achieved without limiting the flexibility of templates or lowering their performance—no fact their expressive power is increased. This paper describes the language extensions supporting concepts, their use in the expression of the C++ Standard Template Library, and their implementation in the ConceptC++ compiler. Concepts are candidates for inclusion in the upcoming revision of the ISO C++ standard.

1. Introduction

The C++ language [25, 62] supports parameterized types and functions in the form of templates. Templates provide a unique combination of features that have allowed them to be used for many different programming paradigms, including Generic Programming [1, 66], Generative Programming [11], and Template Metaprogramming [3, 44]. Much of the flexibility of C++ templates comes from their unconstrained nature: a template can perform any operation on its template parameters, including compile-time type computations, allowing the simulation of the basic features required for diverse programming paradigms. Another essential part of templates in their ability to provide abstraction without performance degradation: templates provide sufficient information to a compiler’s optimizer (especially the inliner) to generate code that is optimal in both time and space.

Consequently, templates have become the preferred implementation style for a vast array of reusable, efficient C++ libraries (2, 6, 14, 26, 52, 54, 58, 67), many of which have built upon the Generic Programming methodology exemplified by the C++ Standard Template Library (STL) [42, 60]. Aided by the discovery of numerous template techniques [28, 46, 56, 66, 67], C++ libraries are becoming more powerful, more flexible, and more expressive.

However, these improvements come at the cost of implementation complexity [6, 62]: authors of C++ libraries typically rely on a grab-bag of template tricks, many of which are complex and poorly documented. When library interfaces are rigorously separated from library implementation, the complexity of implementation of a library is not a problem for its users. However, templates rely on the absence of modular (separate) type-checking for flexibility and performance. Therefore, the complexities of library implementation leak through to library users. This problem manifests itself most visibly in spectacularly poor error messages for simple mistakes. Consider:

```cpp
sort(lst); // standard library
```

Attempting to compile this code with a recent version of the GNU C++ compiler [37] produces more than two kilobytes of output, containing six different error messages. Worse, the errors reported provide line numbers and file names that point to the implementation of the STL sort() function and its helper functions. The only clue provided to users that this error was triggered by their own code is a cryptic note stating: “The STL implementation is the following maximum line of output:

```
sort(last);��0: instantiated from here
```

The actual error, in this case, is that the STL sort() requires a pair of Random Access Iterators, i.e., iterators that can move any number of steps forward or backward in constant time. The STL...
Concepts: Linguistic Support for Generic Programming in C++

Douglas Gregor  
Indiana University  
dgregor@osl.iu.edu

Jaakko Järvi  
Texas A&M University  
jarvi@cs.tamu.edu

Jeremy Siek  
Rice University  
Jeremy.G.Siek@rice.edu

Bjarne Stroustrup  
Texas A&M University  
bs@cs.tamu.edu

Gabriel Dos Reis  
Texas A&M University  
gdr@cs.tamu.edu

Andrew Lumsdaine  
Indiana University  
lums@osl.iu.edu

In C++, generic programming is enabled by the flexibility of templates; the C++ type parameterization mechanism. However, this feature comes with a price: generic (template) libraries can be more difficult to use and develop than non-template libraries and their misuse results in notoriously confusing error messages. As currently defined in C++98, templates are unparameterized, and type-checking of templates is performed late in the compilation process, i.e., after the information of a template has been combined with its definition. To improve the support for generic programming in C++, we introduce concepts to express the syntactic and semantic behavior of types and to constrain the type parameters in a C++ template. Using concepts, type-checking of template definitions is separated from their uses, thereby making templates easier to use and easier to compile. These improvements are achieved without limiting the flexibility of templates or degrading their performance—in fact, their expressive power is increased. This paper describes the language extensions supporting concepts, their use in the expression of the C++ Standard Template Library, and their implementation in the ConceptCC compiler. Concepts are candidates for inclusion in the upcoming revision of the ISO C++ standard.

Categories and Subject Descriptors  
D.3.3 [Programming Languages]: Language Constructs and Features—Abstract data types; C.1.4 [Computing methodologies]: Languages and Techniques—Functional, imperative, logic, object-oriented.

General Terms  
Design, Languages

Keywords  
Generic programming, constrained generics, parametric polymorphism, C++ templates, C++ concepts.

1. Introduction

The C++ language [25, 62] supports parameterized types and functions in the form of templates. Templates provide a unique container-generating [1, 89]. Much of the flexibility of C++ templates comes from their unconstrained nature: a template can perform any operation on its template parameters, including compile-time type computations, allowing the simulation of the basic features required for diverse programming paradigms. Another essential part of templates is their ability to provide abstraction without performance degradation: templates provide sufficient information to a compiler’s optimizer (especially the inline) to generate code that is optimal in both time and space.

Consequently, templates have become the preferred implementation style for a vast array of portable, efficient C++ libraries (2, 6, 14, 28, 32, 54, 56, 67), many of which are built upon the Generic Programming methodology exemplified by the C++ Standard Template Library (STL) [42, 60]. Aided by the discovery of numerous template techniques [28, 46, 56, 67], C++ libraries are becoming more powerful, more flexible, and more expressive.

However, these improvements come at the cost of implementation complexity [6, 83]: authors of C++ libraries typically rely on a grab-bag of template tricks, many of which are complex and poorly documented. When library interfaces are rigorously separated from their implementation, the complexity of implementation of a library is not a problem for its users. However, templates rely on the absence of modular (separate) type-checking for flexibility and performance. Therefore, the complexities of library implementation leak through to library users. This problem manifests itself most visibly in spectacularly poor error messages for simple mistakes. Consider:

```cpp
sort(lst.begin(), lst.end());
```

Attempting to compile this code with a recent version of the GNU C++ compiler [17] produces more than two kilobytes of output, containing six different error messages. Worse, the errors reported provide line numbers and file names that point to the implementation of the STL `sort()` function and in helper functions. The only clue provided to users that this error was triggered by their own code (rather than by a bug in the STL implementation) is the following innocuous line of output:

```
sort_list.cpp:8: instantiated from here
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific prior written permission.

Permission to use digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific prior written permission.

The actual error, in this case, is that the STL `sort()` requires a pair of Random Access Iterators, i.e., indices that can move any number of steps forward or backward in constant time. The STL...
[This Slide Intentionally Left Blank]
A Concept Design for the STL
B. Stroustrup and A. Sutton (Editors)
Jan, 2012

Participants:
Ryan Ernst, A9.com, Inc.
Anil Gangolli, A9.com, Inc.
Jon Kalb, A9.com, Inc.
Andrew Lumsdaine, Indiana University (Aug. 1-4)
Paul McJones, independent
Dan Rose, A9.com, Inc.
Alex Stepnov, A9.com, Inc.
Bjarne Stroustrup, Texas A&M University (Aug. 1-3)
Andrew Sutton, Texas A&M University
Larisse Voufo†, Indiana University
Jeremiah Willcock, Indiana University
Marcin Zalewski†, Indiana University

Abstract
This report presents a concept design for the algorithms part of the STL and outlines the design of the supporting language mechanisms. Both are radical simplifications of what was proposed in the C++0x draft. In particular, this design consists of only 41 concepts (including supporting concepts), does not require concept maps, and (perhaps most importantly) does not resemble template metaprogramming.

Contents
1 Introduction 5
  1.1 Motivation ........................................ 5
  1.2 Approach ......................................... 7
  1.3 Design Ideals ...................................... 8
  1.4 Organization ..................................... 9

2 Algorithms 10
  2.1 Non-modifying Sequence Operations ................. 12
    2.1.1 All, Any, and None .......................... 12
    2.1.2 For Each ...................................... 14
    2.1.3 The Find Family .............................. 15
    2.1.4 The Count Family ............................ 18
    2.1.5 Mismatch and Equal ........................... 18
    2.1.6 Permutations ................................. 19

† Participated in editing of this report.
A Concept Design for the STL

B. Stroustrup and A. Sutton (Editors)

Jan, 2012

Participants:
Ryan Ernst, A9.com, Inc.
Anil Gangolli, A9.com, Inc.
Jon Kalb, A9.com, Inc.
Andrew Lumsdaine, Indiana University (Aug. 1-4)
Paul McJones, independent
Dan Rose, A9.com, Inc.
Alex Stepanov, A9.com, Inc.
Bjarne Stroustrup, Texas A&M University (Aug. 1-3)
Andrew Sutton, Texas A&M University
Larisse Voufo †, Indiana University
Jeremiah Willcock, Indiana University
Marcin Zalewski †, Indiana University

Abstract
This report presents a concept design for the algorithms part of the STL and outlines the design of the supporting language mechanism. Both are radical simplifications of what was proposed in the C++0x draft. In particular, this design consists of only 41 concepts (including supporting concepts), does not require concept maps, and (perhaps most importantly) does not resemble template metaprogramming.

Contents
1 Introduction 5
   1.1 Motivation ....................................... 5
   1.2 Approach ........................................ 7
   1.3 Design Ideals ...................................... 8
   1.4 Organization ...................................... 9
2 Algorithms 10
   2.1 Non-modifying Sequence Operations ......................... 12
      2.1.1 All, Any, and None............................... 12
      2.1.2 For Each .................................... 14
      2.1.3 The Find Family ................................ 15
      2.1.4 The Count Family ............................... 18
      2.1.5 Mismatch and Equal .............................. 18
      2.1.6 Permutations .................................. 19
   ❧

† Participated in editing of this report.
2020
17 Templates

A template defines a family of classes, functions, or variables, an alias for a family of types, or a concept.

1. A template defines a family of classes, functions, or variables, an alias for a family of types, or a concept.

- template-declaration:
  - template-head declaration
  - template-head concept-definition

- template-head:
  - template < template-parameter-list > requires-clause opt

- template-parameter-list:
  - template-parameter
  - template-parameter-list , template-parameter

- requires-clause:
  - requires constraint-logical-or-expression

- constraint-logical-or-expression:
  - constraint-logical-and-expression || constraint-logical-and-expression

- constraint-logical-and-expression:
  - primary-expression && primary-expression

- concept-definition:
  - concept concept-name = constraint-expression ;

- concept-name:
  - identifier

[Note: The > token following the template-parameter-list of a template-declaration may be the product of replacing a >> token by two consecutive >> tokens (17.2) — end note]

The declaration in a template-declaration (if any) shall

- (2.1) declare or define a function, a class, or a variable, or
- (2.2) define a member function, a member class, a member enumeration, or a static data member of a class template or of a class nested within a class template, or
- (2.3) define a member template of a class or class template, or
- (2.4) be a deduction-guide, or
- (2.5) be an alias-declaration.

A template-declaration is a declaration. A template-declaration is also a definition if its template-head is followed by either a concept-definition or a declaration that defines a function, a class, a variable, or a static data member. A declaration introduced by a template declaration of a variable is a variable template. A variable-template at class scope is a static data member template.

[Example:

```cpp
template<class T>
constexpr T pi = T(3.1415926535897932385L);

template<class T>
T circular_area(T r) {
    return pi<T> * r * r;
}

struct matrix_constants {
    template<class T>
    using pauli = hermitian_matrix<T, 2>;
    template<class T>
    static constexpr pauli<T> sigma1 = { { 0, 1 }, { 1, 0 } };
    template<class T>
    static constexpr pauli<T> sigma2 = { { 0, -1i }, { 1i, 0 } };
};
```

Templates 306
17 Templates

requires-clause:
  requires constraint-logical-or-expression

constraint-logical-or-expression:
  constraint-logical-and-expression
  constraint-logical-or-expression || constraint-logical-and-expression

constraint-logical-and-expression:
  primary-expression
  constraint-logical-and-expression && primary-expression

concept-definition:
  concept concept-name = constraint-expression ;

concept-name:
  identifier
“Generic programming is about abstracting and classifying algorithms and data structures."
It gets its inspiration from Knuth and not from type theory.
Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures.
Such an undertaking is still a dream.”

– Alex Stepanov
References

Much of the material in this talk can be found at http://stepanovpapers.com/

A special thanks to Paul McJones for organizing this site

*Can Programming Be Liberated from the von Neumann Style?*
  
  https://www.thocp.net/biographies/papers/backus_turingaward_lecture.pdf

*Notation as a Tool of Thought*
  
  https://amturing.acm.org/award_winners/iverson_9147499.cfm

*Writing Correct Programs*
  

*Exception-Safety in Generic Components*
  
  https://dl.acm.org/citation.cfm?id=724067
References

Concepts: Linguistic Support for Generic Programming in C++
http://www.stroustrup.com/oopsla06.pdf

A Concept Design for The STL
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf

Sincere apologies to anyone I left out, your contribution was important.