pacific++

Sponsored by:

Gold

optiver

WARGAMING SYDNEY

Bronze

CROWN

JET BRAINS

Community

CppCast

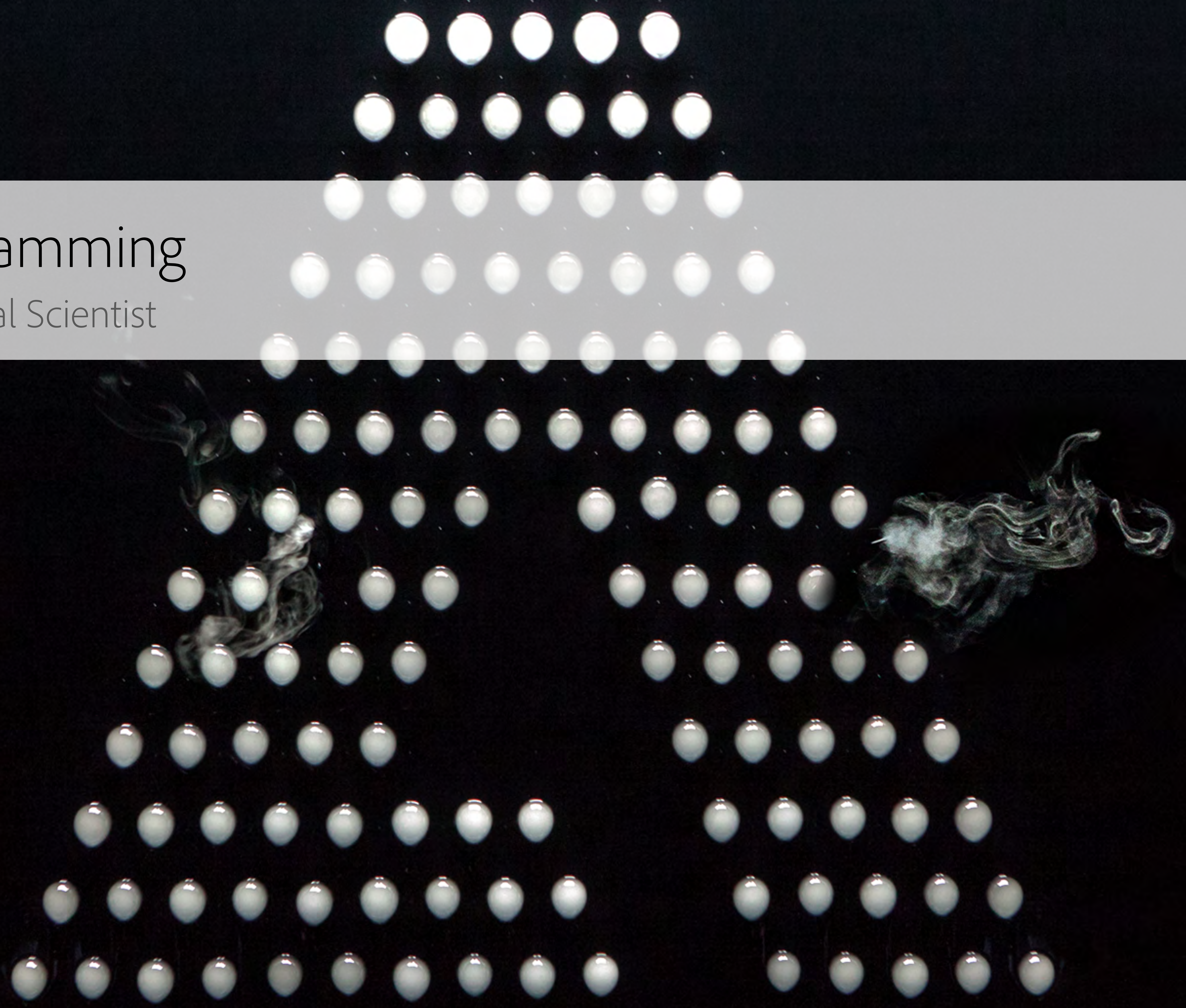
cppcon the c++ conference

Meeting C++

SYDNEY C++ MEETUP

Generic Programming

Sean Parent | Principal Scientist



#AdobeRemix
Thomas Wirtz

"You cannot fully grasp mathematics until you understand its historical context." – Alex Stepanov

1988


T.S.
1989

1988

#1 Song: *Faith*, George Michael

#1 Movie: *Rain Man*

Winter Olympic Games in Calgary, Alberta, Canada

US Senate ratifies INF treaty between US and Soviet Union

    Ronald Regan & Mikhail Gorbachev

George H. W. Bush wins US Presidential Election

1988

#1 Song: *Faith*, George Michael

#1 Movie: *Rain Man*

Winter Olympic Games in Calgary, Alberta, Canada

US Senate ratifies INF treaty between US and Soviet Union

Ronald Regan & Mikhail Gorbachev

George H. W. Bush wins US Presidential Election

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

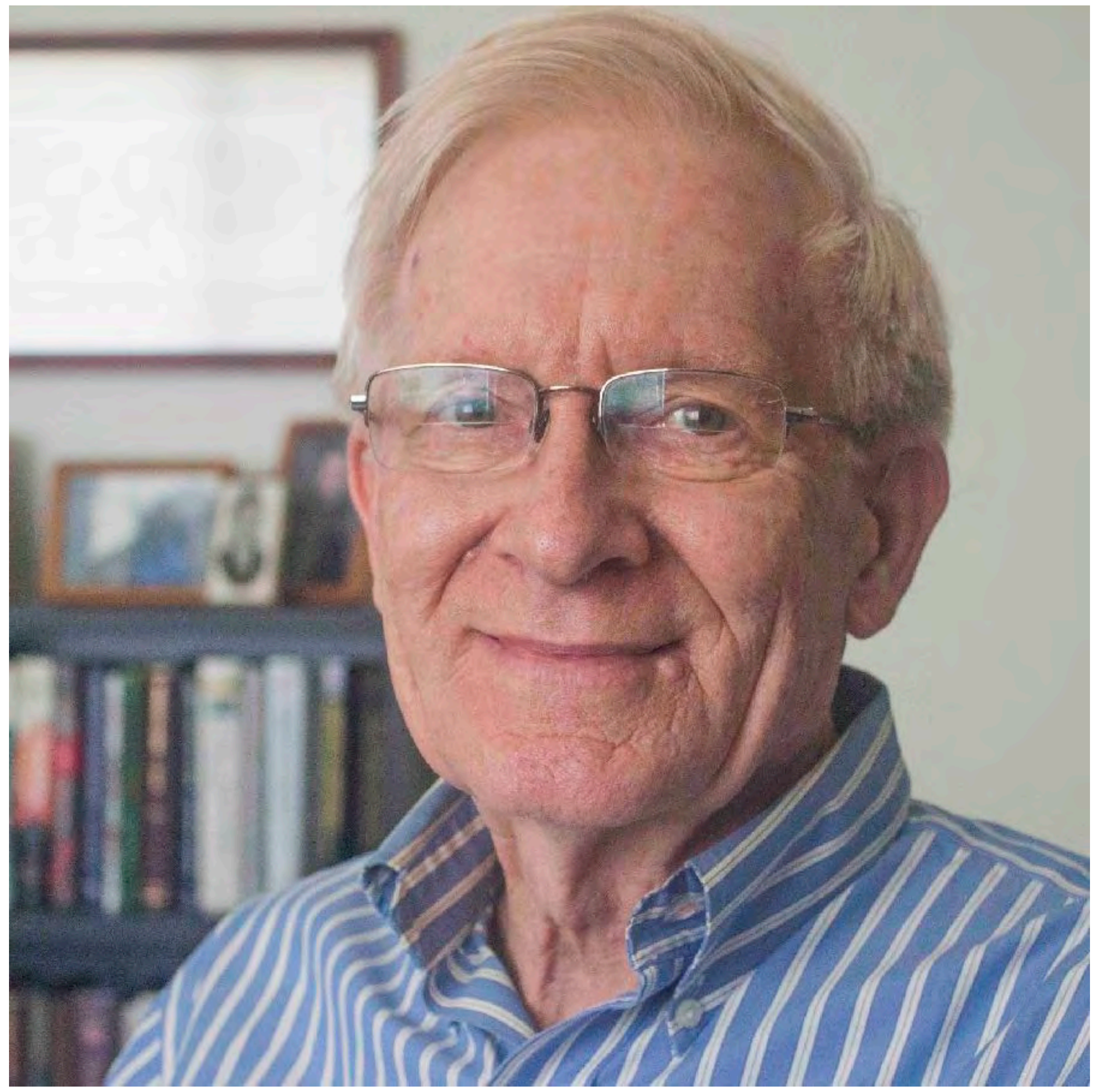
Abstract

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 abstraction—data, algorithmic, structural, and representational—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 component libraries offer important advantages for achieving software productivity and reliability.

*This paper was presented at the First International Joint Conference of ISSAC-88 and AAECC-6, Rome, Italy, July 4-8, 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* 358, Springer-Verlag, 1989, pp. 13-25.

†The first 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 9151. 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

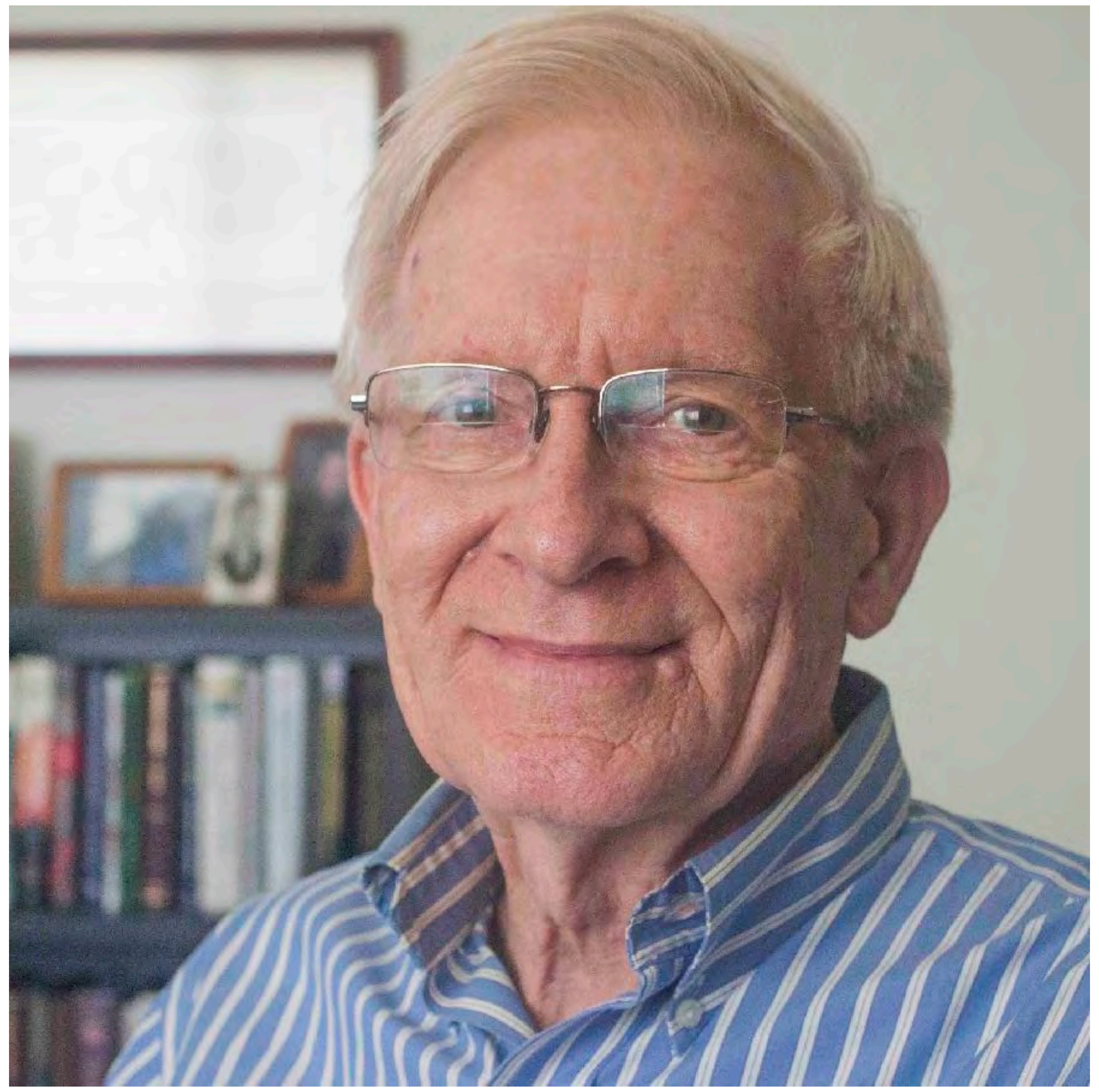
Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work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

Four kinds of abstraction—data, algorithmic, structural, and representational—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 component libraries offer important advantages for achieving software productivity and reliability.

*This paper was presented at the First International Joint Conference of ISSAC-88 and AAECC-6, Rome, Italy, July 4-8, 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* 358, Springer-Verlag, 1989, pp. 13-25.

†The first 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 9151. 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.

i



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

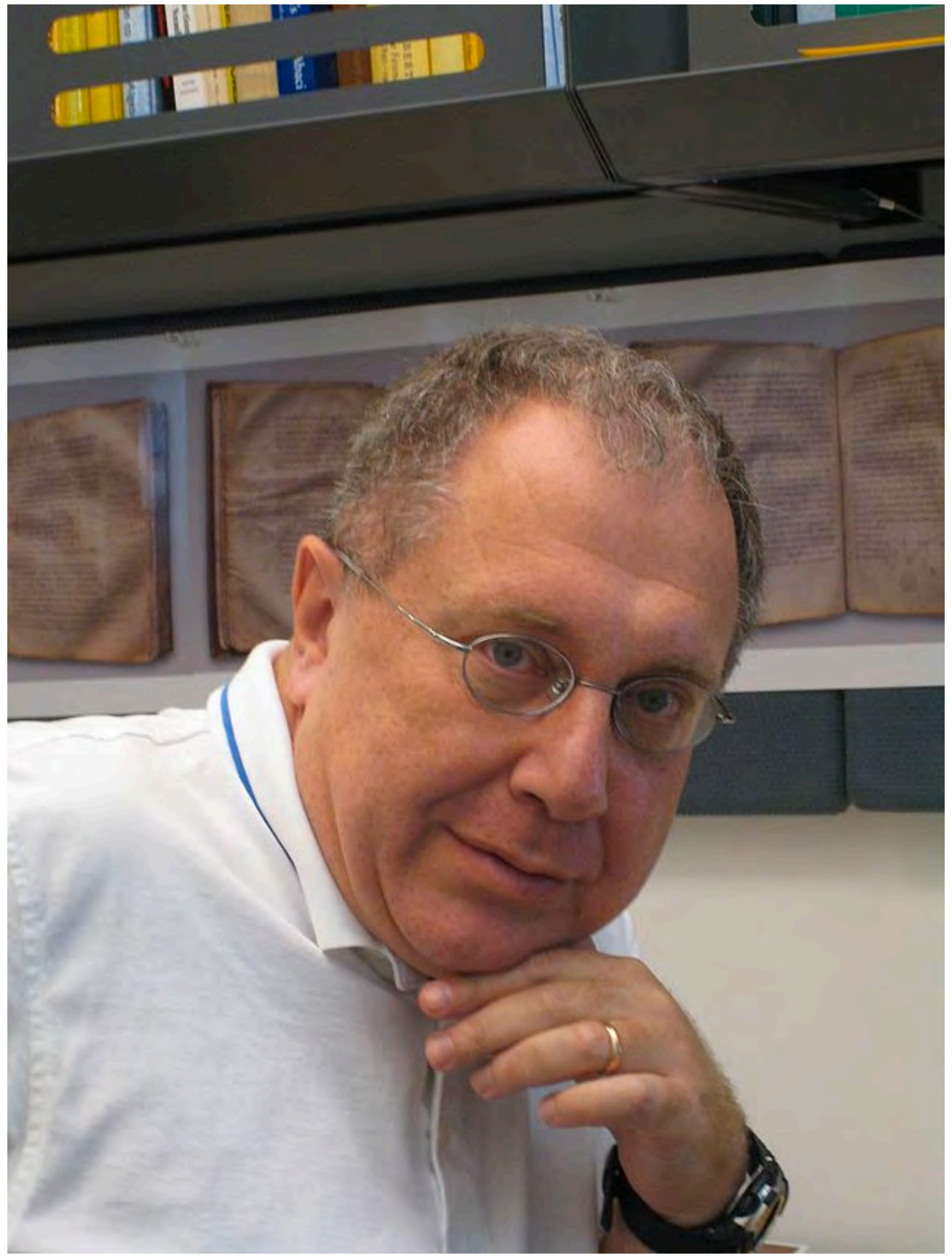
Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work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

Four kinds of abstraction—data, algorithmic, structural, and representational—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 component libraries offer important advantages for achieving software productivity and reliability.

*This paper was presented at the First International Joint Conference of ISSAC-88 and AAECC-6, Rome, Italy, July 4-8, 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* 358, Springer-Verlag, 1989, pp. 13-25.

†The first 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 9151. 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.

i



"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

1976 Parallel Computation and Associative Property

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

$$(x \bullet y) \bullet z = x \bullet (y \bullet z) \text{ for all } x, y, z \text{ in } S.$$

1976 Parallel Computation and Associative Property

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

$(x \bullet y) \bullet z = x \bullet (y \bullet z)$ for all x, y, z in S.

Parallel reduction is associated with monoids

Software is associated with Algebraic Structures

1981 Tecton

The Tecton language

REPRINT 9681

GENERAL ELECTRIC

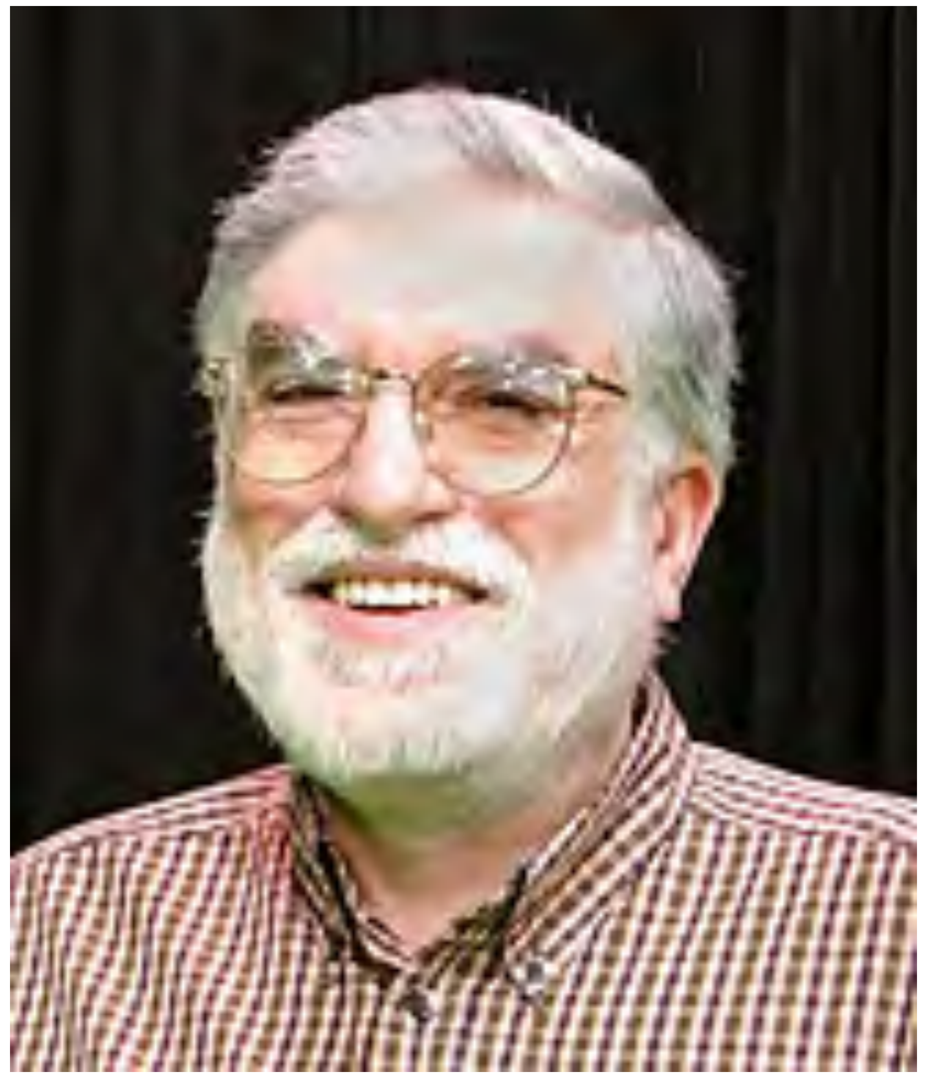
GENERAL ELECTRIC COMPANY
CORPORATE RESEARCH AND DEVELOPMENT
P.O. Box 43, Schenectady, N.Y. 12301 U.S.A.

TECTON: A LANGUAGE FOR MANIPULATING GENERIC OBJECTS

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

1981 Tecton

The Tecton language



REPRINT 9681

GENERAL ELECTRIC

GENERAL ELECTRIC COMPANY
CORPORATE RESEARCH AND DEVELOPMENT
P.O. Box 43, Schenectady, N.Y. 12301 U.S.A.

TECTON: A LANGUAGE FOR MANIPULATING GENERIC OBJECTS

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

1986-87 Libraries

Higher Order Programming

Higher Order Programming

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

March 5, 1987

Polytechnic Institute of New York

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

C.A.T.T. Technical Report 86-13

CENTER FOR
ADVANCED
TECHNOLOGY IN
TELECOMMUNICATIONS

1986-87 Libraries

Higher Order Programming

Higher Order Programming

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

March 5, 1987

Polytechnic Institute of New York

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

C.A.T.T. Technical Report 86-13

CENTER FOR
ADVANCED
TECHNOLOGY IN
TELECOMMUNICATIONS

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

1987

Leonhard Euler

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

1988

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 abstraction—data, algorithmic, structural, and representational—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 component libraries offer important advantages for achieving software productivity and reliability.

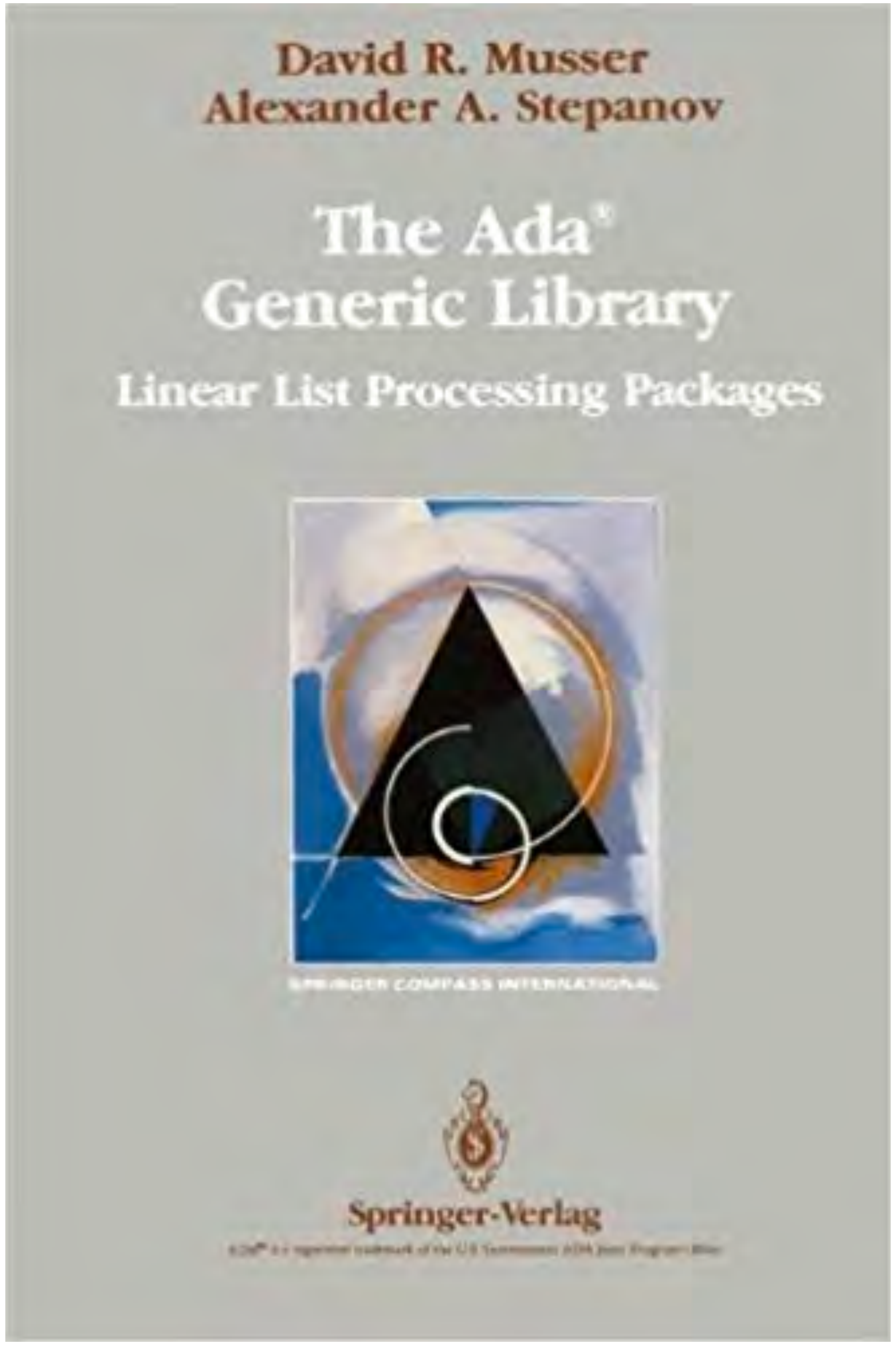
---

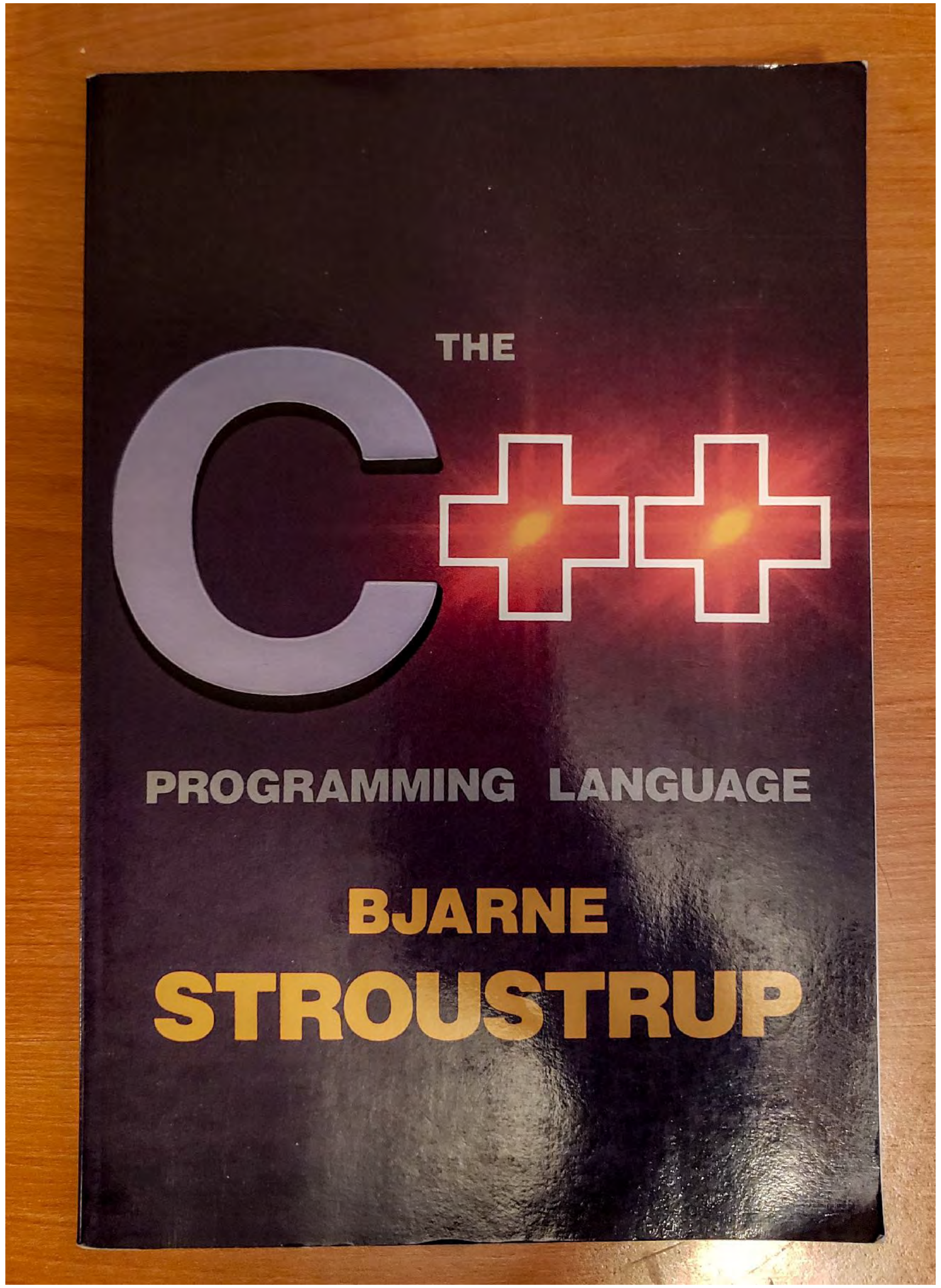
*This paper was presented at the First International Joint Conference of ISSAC-88 and AAECC-6, Rome, Italy, July 4-8, 1988. (ISSAC stands for International Symposium on Symbolic and Algebraic Computation and AAECC for Applied Algebra, Algebraic Algorithms, and Error Correcting Codes). It was published in *Lecture Notes in Computer Science* 358, Springer-Verlag, 1989, pp. 13-25.

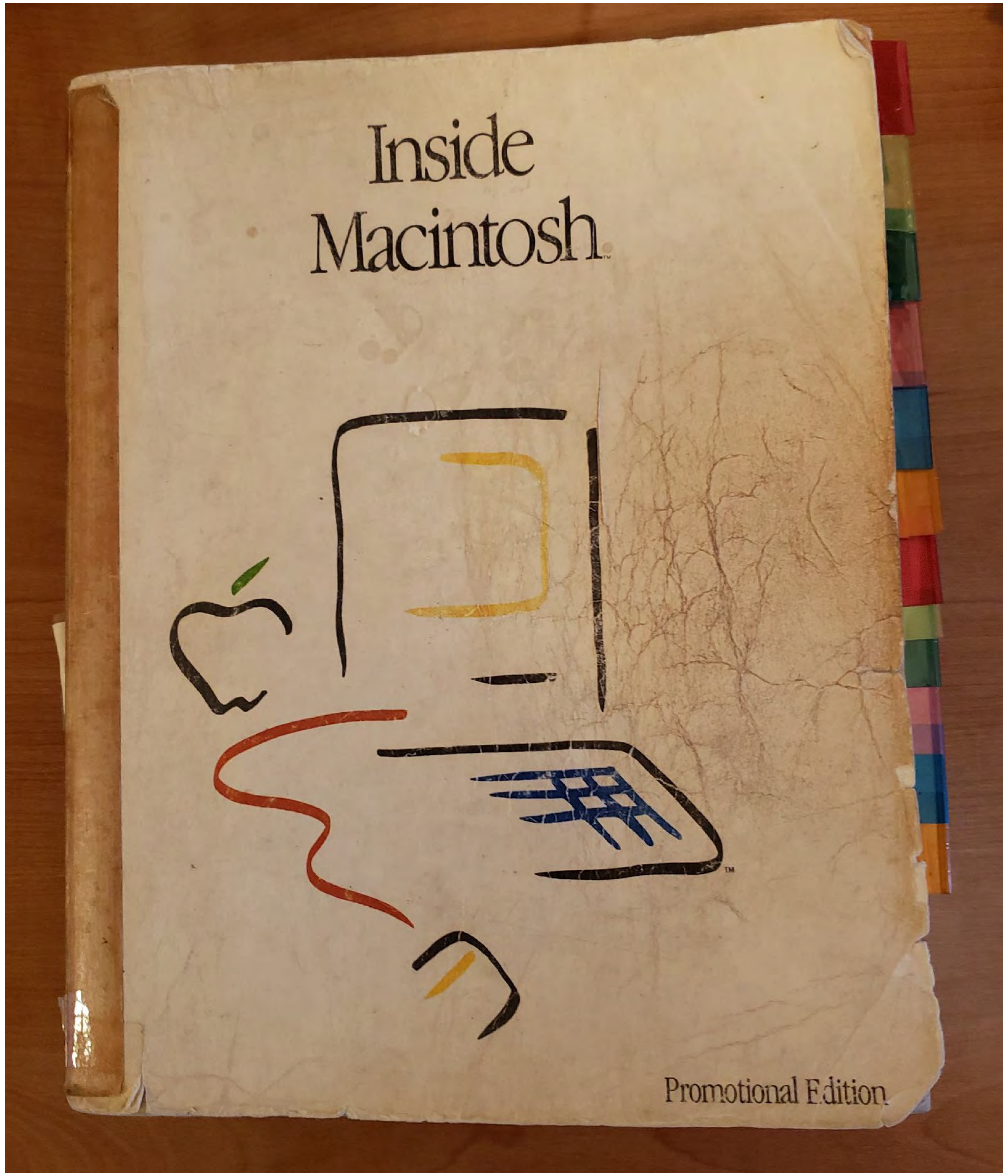
†The first author's work was sponsored in part through a subcontract from Computational Logic, Inc., which was sponsored in turn by the Defense Advanced Research Projects Agency, ARPA order 9151. The views and conclusions contained in this document are those of the authors and should not be interpreted 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    : 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;
    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;
```

Figure 1: Body of Partition Algorithm


David R. Musser
Alexander A. Stepanov
The Ada®
Generic Library
Linear List Processing Packages
SPRINGER COMPASS INTERNATIONAL
Springer-Verlag


THE
C++
PROGRAMMING LANGUAGE
BJARNE
STROUSTRUP


Inside
Macintosh
Promotional Edition

```
TYPE QDByte    = -128..127;
     QDPtr     = ^QDByte;
     QDHandle  = ^QDPtr;
```

QuickDraw includes only the graphics and utility procedures and functions you'll need to create graphics on the screen. Keyboard input, mouse input, and larger user-interface constructs such as windows and menus are implemented in separate packages that use QuickDraw but are linked in as separate units. You don't need these units in order to use QuickDraw; however, you'll probably want to read the documentation for windows and menus and learn how to use them with your Macintosh programs.

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 features you have. If the mathematics that underlie a graphics package are imprecise or fuzzy, the graphics 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.

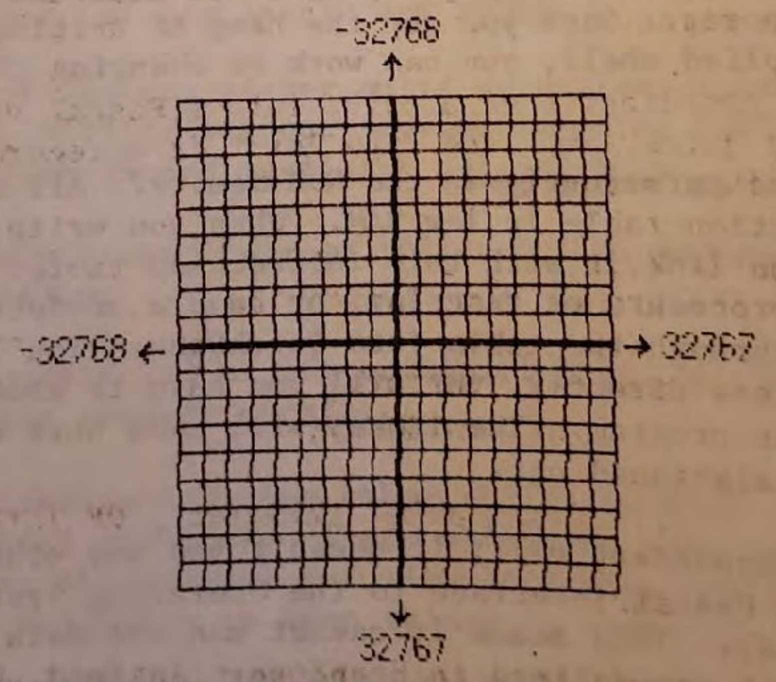


Figure 2. The Coordinate Plane

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 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 intuitively 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 system.

The coordinate origin (Ø,Ø) 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:

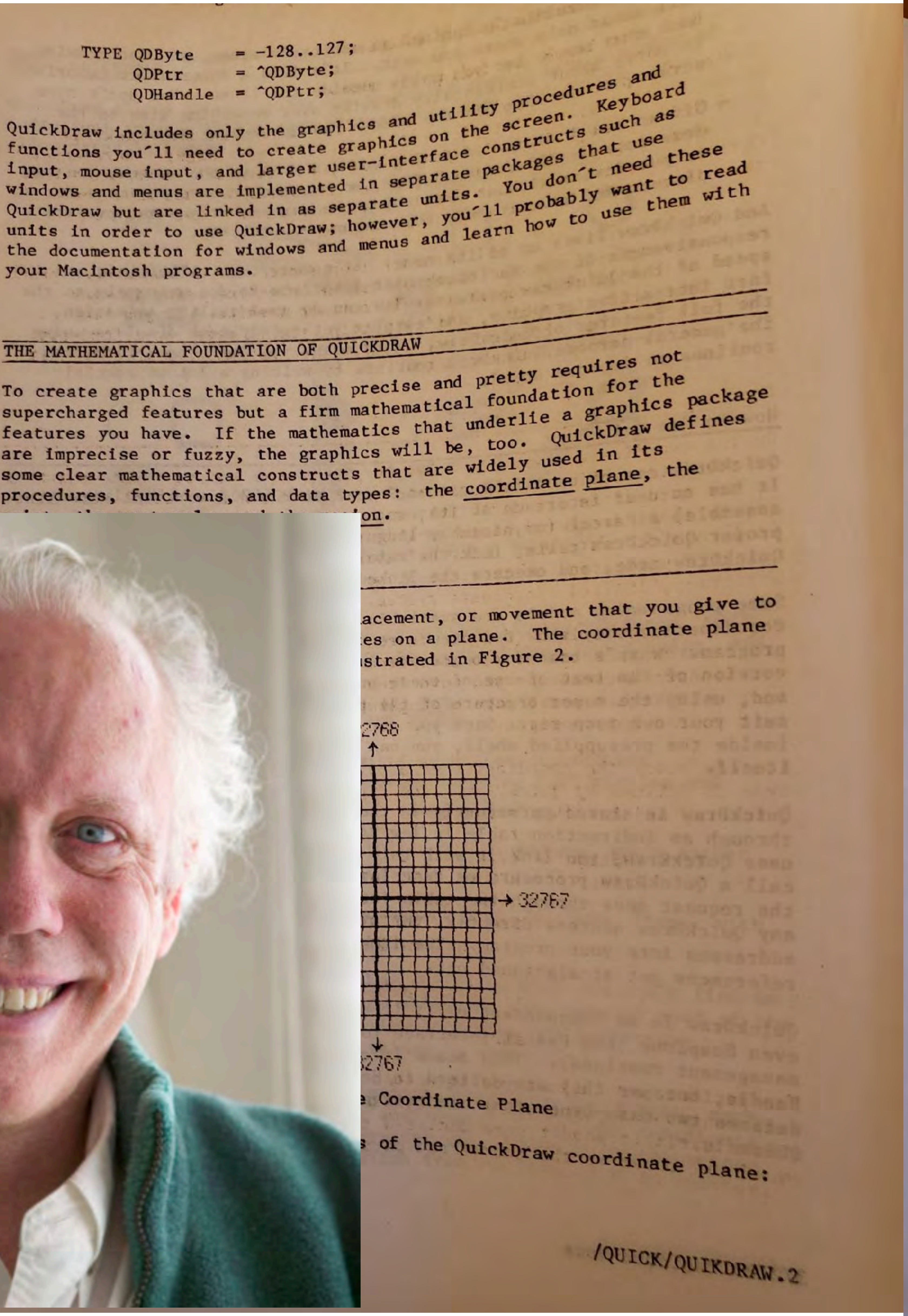
```
TYPE VHSelect = (V,H);
     Point    = RECORD CASE INTEGER OF

                  Ø:  (v:  INTEGER;
                       h:  INTEGER);

                  1:  (vh:  ARRAY [VHSelect] 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 goodPt were declared to be of type Point, the following would all refer to the coordinate parts of the point:



```
TYPE QDByte    = -128..127;
     QDPtr     = ^QDByte;
     QDHandle  = ^QDPtr;
```

QuickDraw includes only the graphics and utility procedures and functions you'll need to create graphics on the screen. Keyboard input, mouse input, and larger user-interface constructs such as windows and menus are implemented in separate packages that use QuickDraw but are linked in as separate units. You don't need these units in order to use QuickDraw; however, you'll probably want to read the documentation for windows and menus and learn how to use them with your Macintosh programs.

THE MATHEMATICAL FOUNDATION OF QUICKDRAW

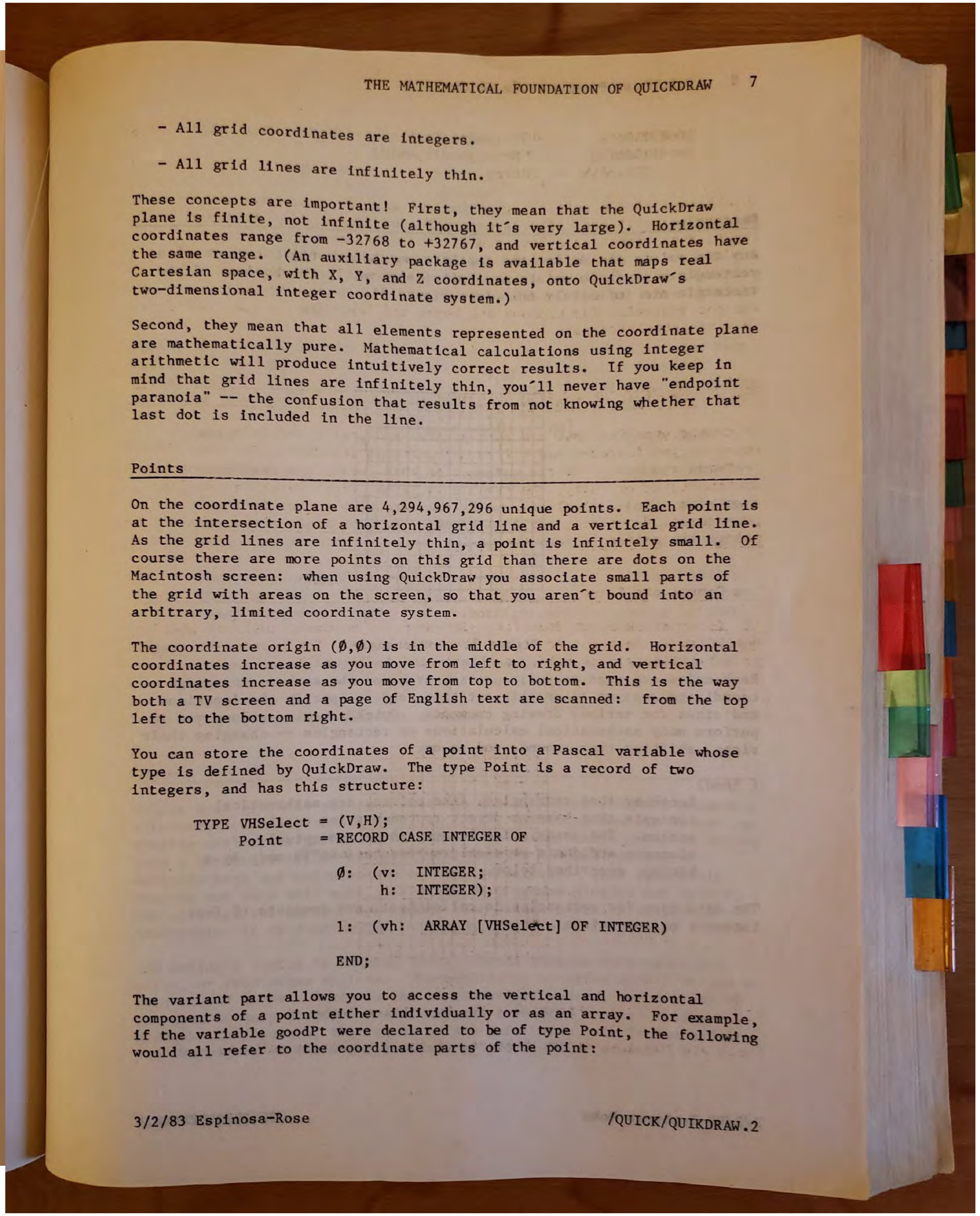
To create graphics that are both precise and pretty requires not supercharged features but a firm mathematical foundation for the features you have. If the mathematics that underlie a graphics package are imprecise or fuzzy, the graphics will be, too. QuickDraw defines some clear mathematical constructs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

acement, or movement that you give to
es on a plane. The coordinate plane
strated in Figure 2.

e Coordinate Plane

s of the QuickDraw coordinate plane:

/QUICK/QUIKDRAW.2



THE MATHEMATICAL FOUNDATION OF QUICKDRAW 7

- 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 intuitively 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 system.

The coordinate origin (Ø,Ø) 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:

```
TYPE VHSelect = (V,H);
     Point    = RECORD CASE INTEGER OF

                  Ø:  (v:  INTEGER;
                       h:  INTEGER);

                  1:  (vh:  ARRAY [VHSelect] OF INTEGER)

                END;
```

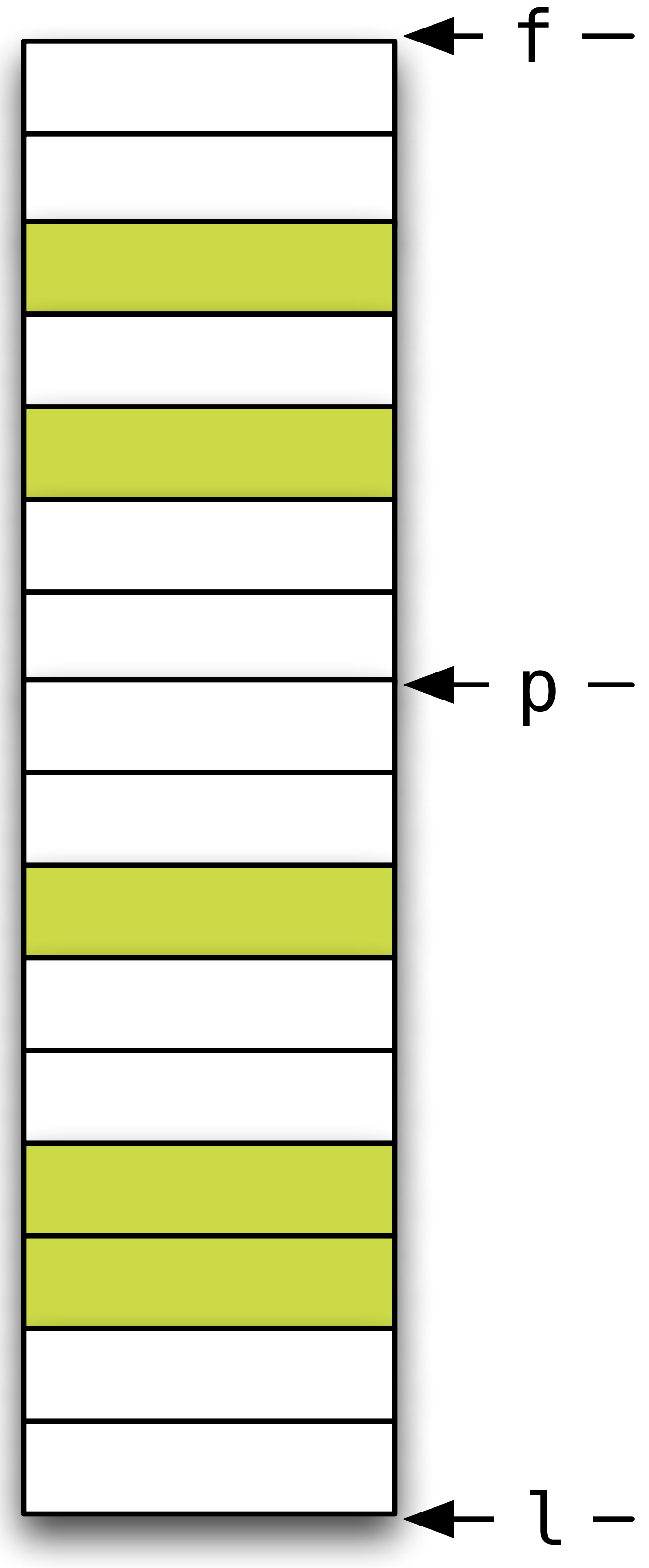
The variant part allows you to access the vertical and horizontal components of a point either individually or as an array. For example, if the variable goodPt were declared to be of type Point, the following would all refer to the coordinate parts of the point:

3/2/83 Espinosa-Rose /QUICK/QUIKDRAW.2

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

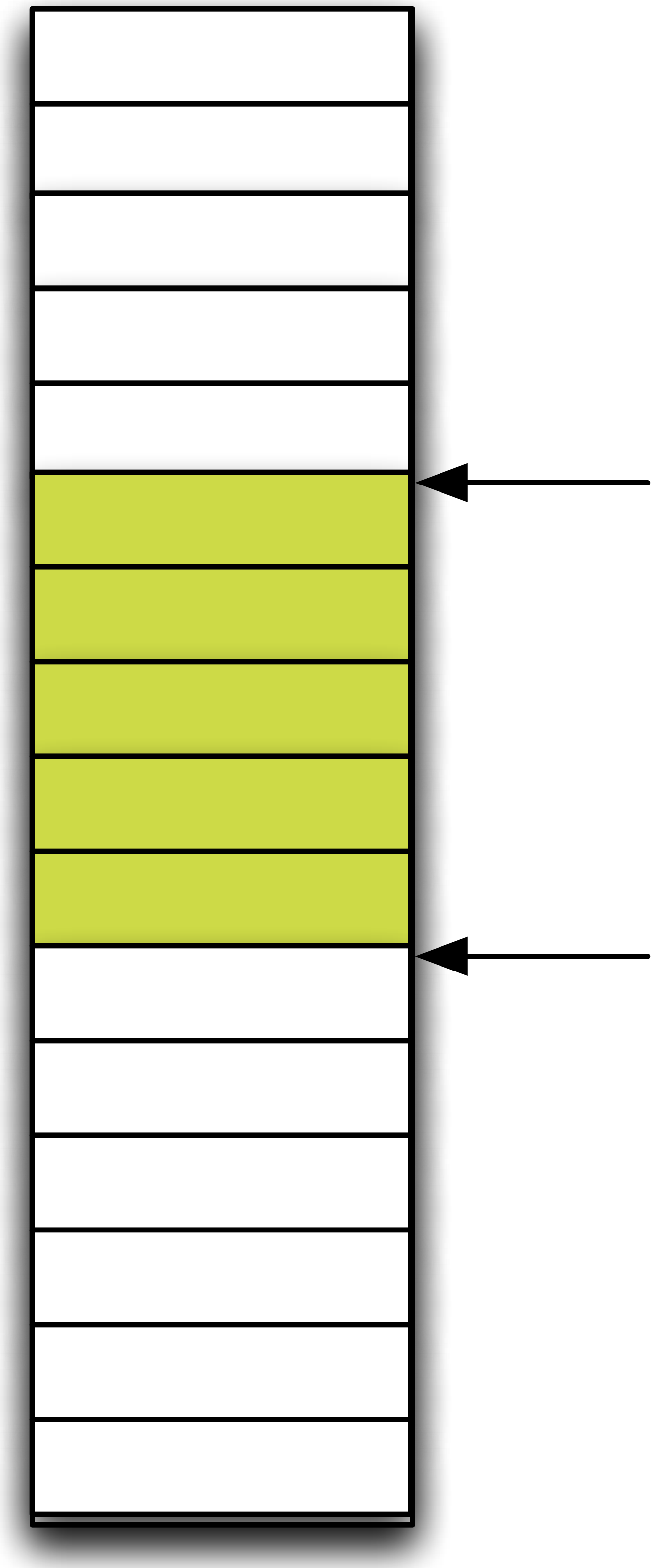
These concepts are important! …they mean that all elements represented on the coordinate plane are mathematically pure. Mathematical calculations using integer arithmetic will produce intuitively correct results. If you keep in mind that the grid lines are infinitely thin, you'll never have "endpoint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

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) };
}
```

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) };
}
```

For a sequence of *n* elements there are *n + 1* positions

1993

1993

Movie: Jurassic Park

Bombing of World Trade Center

Bill Clinton sworn in

Video Games: Doom and MYST

1993

Movie: Jurassic Park

Bombing of World Trade Center

Bill Clinton sworn in

Video Games: Doom and MYST

1993

Alex resumes work on Generic Programming

Andrew Koenig suggests writing a standard library proposal

1994

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

October 31, 1995

1983

programming pearls

By Jon Bentley

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 far more valuable than a black box that gobbles programs and flashes "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. But before we get to the subject itself, we must keep it in perspective. Coding skill is just one small part of writing correct programs. The majority of the task is the subject of the three previous columns: problem definition, algorithm design, 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 programmer has to write subtle 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 $N \geq 0$ and that $X[1] \leq X[2] \leq \cdots \leq X[N]$. 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 the integer P (for position); when P is zero T is not in $X[1..N]$, otherwise $1 \leq P \leq N$ and $T = X[P]$.

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.

I've given this problem as an in-class assignment in courses at Bell Labs and IBM. The professional programmers had one hour (sometimes more) 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 (and I wasn't always convinced of the correctness of the code in which no bugs were found).

I found this amazing: only about 10 percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult. In the history in Section 6.2.1 of his *Sorting and Searching*, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.

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 *MustBe(range)* 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.

```
initialize range to designate X[1..N]
loop
    { invariant: MustBe(range) }
    if range is empty,
        return that T is nowhere in the
        array
    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
  endloop
```

The crucial part of this program is the *loop invariant*, which is enclosed in {}'s. 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.

We'll now refine the program, making sure that all our actions respect the invariant. The first issue we must face is the representation of *range*: we'll use two indices L and U (for "lower" and "upper") to represent the range $L..U$. (There are other possible representations for a range, such as its begin-

programming pearls

By Jon Bentley

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 far more valuable than a black box that gobbles programs and flashes "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. But before we get to the subject itself, we must keep it in perspective. Coding skill is just one small part of writing correct programs. The majority of the task is the subject of the three previous columns: problem definition, algorithm design, 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 programmer has to write subtle 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 $N \geq 0$ and that $X[1] \leq X[2] \leq \cdots \leq X[N]$. 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 the integer P (for position); when P is zero T is not in $X[1..N]$, otherwise $1 \leq P \leq N$ and $T = X[P]$.

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.

I've given this problem as an in-class assignment in courses at Bell Labs and IBM. The professional programmers had one hour (sometimes more) 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 (and I wasn't always convinced of the correctness of the code in which no bugs were found).

I found this amazing: only about 10 percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult. In the history in Section 6.2.1 of his *Sorting and Searching*, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.

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 *MustBe*(*range*) 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.

```
initialize range to designate X[1..N]
loop
    { invariant: MustBe(range) }
    if range is empty,
        return that T is nowhere in the
        array
    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
  endloop
```

The crucial part of this program is the *loop invariant*, which is enclosed in {}'s. 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.

We'll now refine the program, making sure that all our actions respect the invariant. The first issue we must face is the representation of *range*: we'll use two indices L and U (for "lower" and "upper") to represent the range $L..U$. (There are other possible representations for a range, such as its begin-

"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

"I want to hire the other ten percent."
– Mark Hamburg, Photoshop Lead

Jon Bentley's Solution (translated to C++)

```
int binary_search(int x[], int n, int v) {
    int l = 0;
    int u = n - 1;

    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++)

```
int binary_search(int x[], int n, int v) {
    int l = 0;
    int u = n - 1;

    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++)

```
int binary_search(int x[], int n, int v) {
    int l = 0;
    int u = n - 1;

    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++)

```
int binary_search(int x[], int n, int v) {
    int l = 0;
    int u = n - 1;

    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++)

```
int binary_search(int x[], int n, int v) {
    int l = 0;
    int u = n - 1;

    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;
}
```

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;
}
```

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;
}
```

1998

INTERNATIONAL STANDARD

ISO/IEC 14882

First edition
1998-09-01

Programming languages — C++

Langages de programmation — C++

Processed and adopted by ASC X3 and approved by ANSI as an American National Standard.

Date of ANSI Approval: 7/27/98

Published by American National Standards Institute,
11 West 42nd Street, New York, New York 10036

Printed in the United States of America

Reference number
ISO/IEC 14882:1998(E)

INTERNATIONAL STANDARD

ISO/IEC 14882

First edition
1998-09-01

Programming languages — C++

Langages de programmation — C++

Processed and adopted by ASC X3 and approved by ANSI as an American National Standard.

Date of ANSI Approval: 7/27/98

Published by American National Standards Institute,
11 West 42nd Street, New York, New York 10036

Printed in the United States of America

Reference number
ISO/IEC 14882:1998(E)

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.

M. Jazayeri, R. Loos, D. Musser (Eds.): Generic Programming '98, LNCS 1766, pp. 69–79, 2000.

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.

M. Jazayeri, R. Loos, D. Musser (Eds.): Generic Programming '98, LNCS 1766, pp. 69–79, 2000.

Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@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.

Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@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.

"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."

2002

Adobe

Adobe Software Technology Lab

Adobe Software Technology Lab

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.

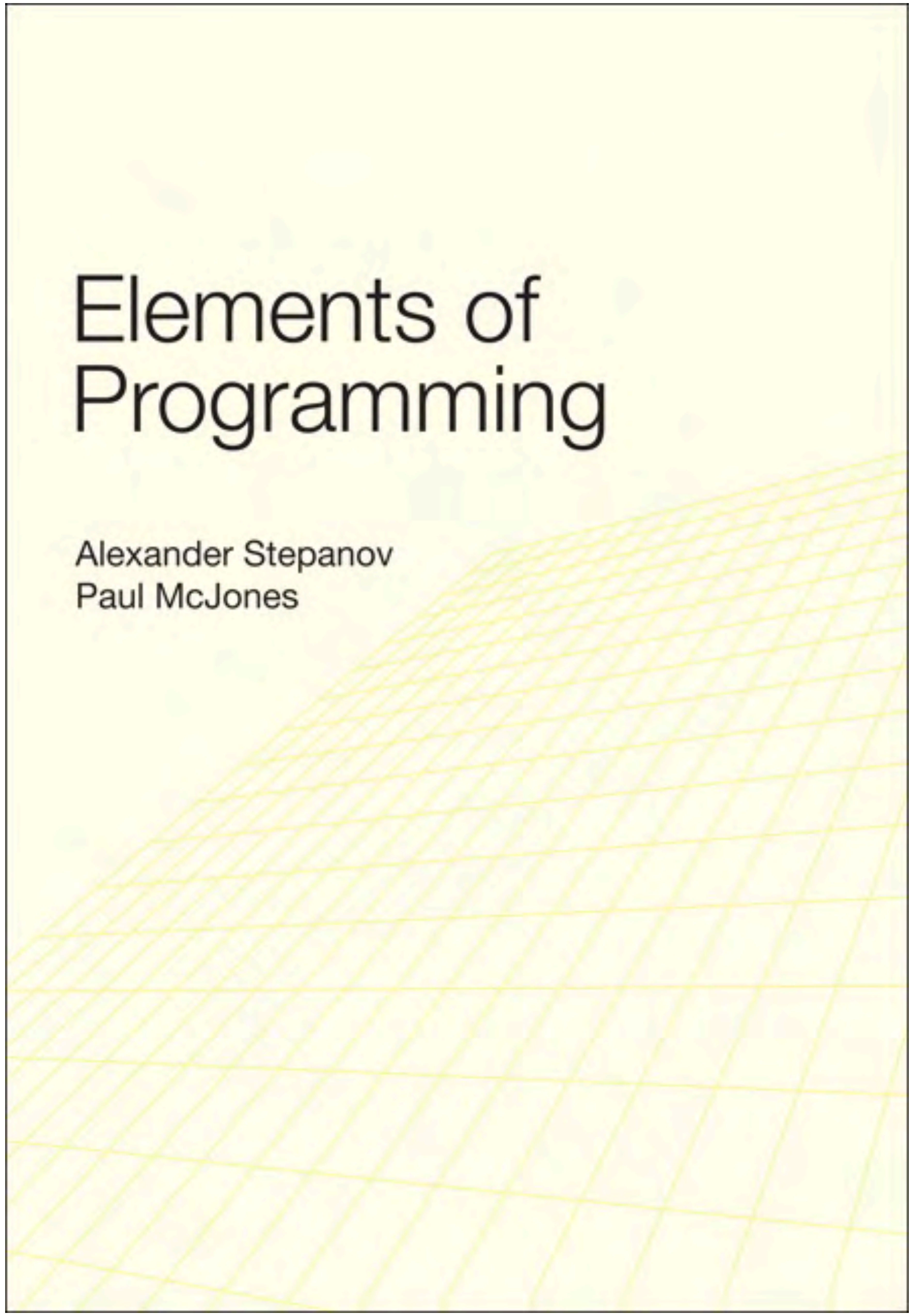
NOTES ON THE FOUNDATIONS OF PROGRAMMING

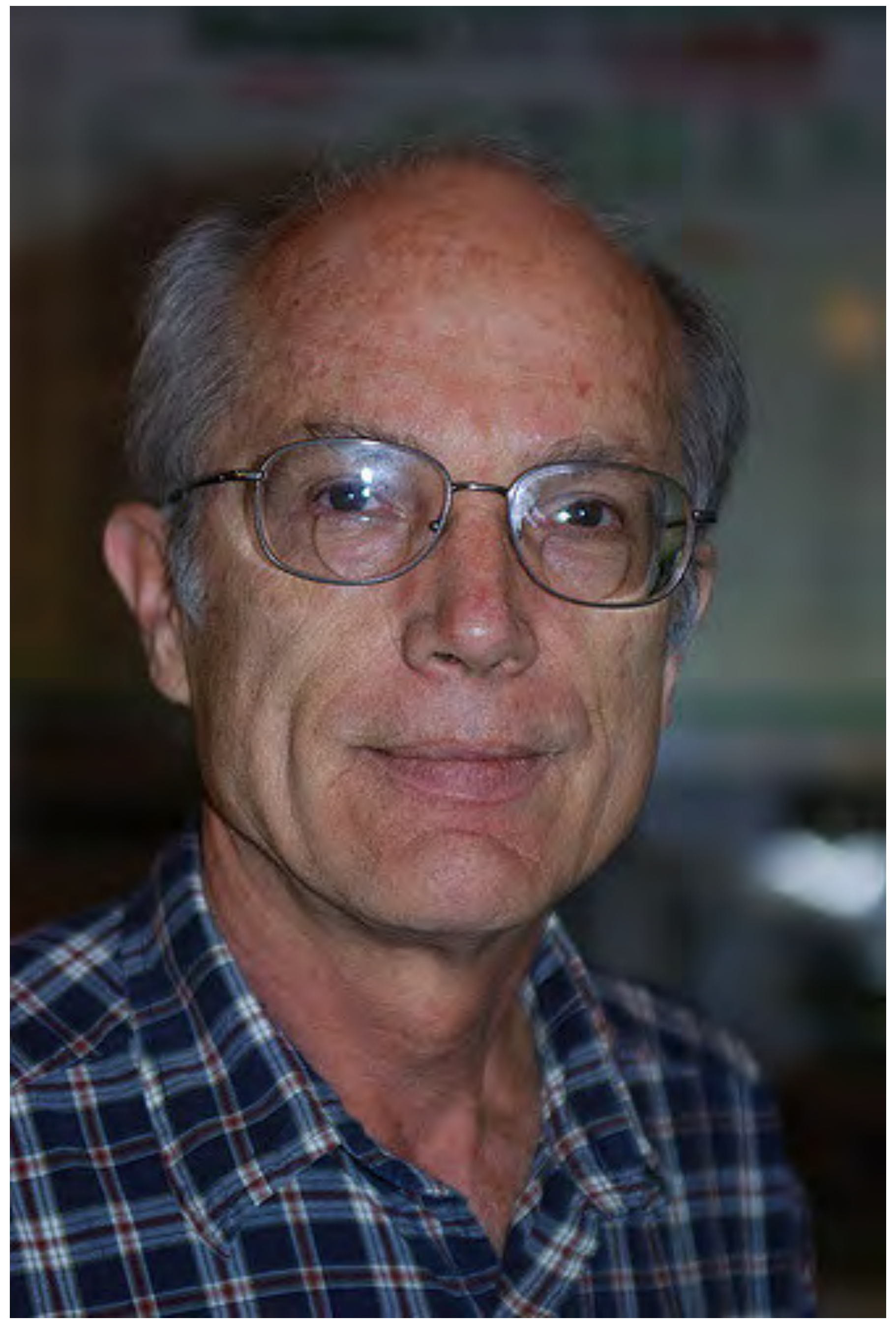
ALEX STEPANOV AND MAT MARCUS

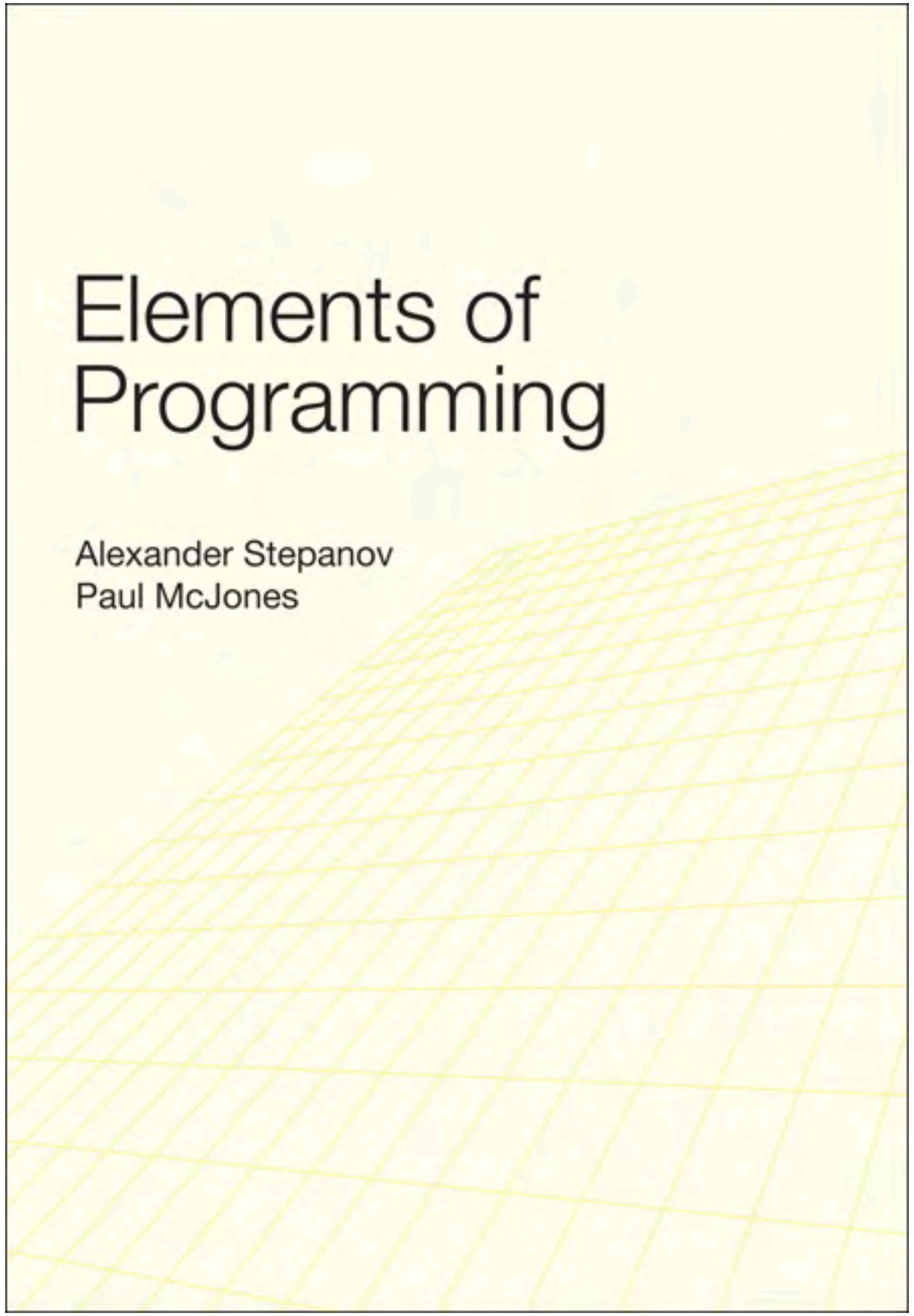
Disclaimer: Please do not redistribute. Instead, requests for a current draft should go to Mat Marcus. These notes are a work in progress and do not constitute a book. In particular, most of the current effort is directed towards writing up new material. As a consequence little time remains for structuring, 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.

1

2009


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 [1977, 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 {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 | ∈ . 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:

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
                   (structure | 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.

Adobe

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
                   (structure | 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.

Adobe

This concept describes a homogeneous functional procedure:

$$
\begin{aligned}
&\mathit{HomogeneousFunction}(\mathsf{F}) \triangleq \\
&\quad \mathit{FunctionalProcedure}(\mathsf{F}) \\
&\wedge\ \mathsf{Arity}(\mathsf{F}) > 0 \\
&\wedge\ (\forall \mathsf{i}, \mathsf{j} \in \mathbb{N})(\mathsf{i}, \mathsf{j} < \mathsf{Arity}(\mathsf{F})) \Rightarrow (\mathsf{InputType}(\mathsf{F}, \mathsf{i}) = \mathsf{InputType}(\mathsf{F}, \mathsf{j})) \\
&\wedge\ \mathsf{Domain} : \mathit{HomogeneousFunction} \to \mathit{Regular} \\
&\qquad \mathsf{F} \mapsto \mathsf{InputType}(\mathsf{F}, 0)
\end{aligned}
$$

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.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

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 parametrization mechanism. However, the power of templates 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 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 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.

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—Polymorphism; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

General Terms Design, Languages

Keywords Generic programming, constrained generics, parametric polymorphism, C++ templates, C++0x, concepts

1. Introduction

The C++ language [25, 62] supports parametrized 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 [3,44], Generative Programming [11], and Template Metaprogramming [1, 66]. 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 emulation 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 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, 20, 32, 54, 55, 65], 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 *ad hoc* 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 [61, 63]: 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 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:

```
list<int> lst;
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 code (rather than by a bug in the STL implementation) is the following innocuous line of output:

```
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

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

Adobe

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

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 parametrization mechanism. However, the power of templates 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 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 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.

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—Polymorphism; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

General Terms Design, Languages

Keywords Generic programming, constrained generics, parametric polymorphism, C++ templates, C++0x, concepts

1. Introduction

The C++ language [25, 62] supports parametrized 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 [3,44], Generative Programming [11], and Template Metaprogramming [1, 66]. 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 emulation 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 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, 20, 32, 54, 55, 65], 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 *ad hoc* 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 [61, 63]: 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 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:

```
list<int> lst;
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 code (rather than by a bug in the STL implementation) is the following innocuous line of output:

```
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

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

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

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 parametrization mechanism. However, the power of templates 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 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 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.

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—Polymorphism; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

General Terms Design, Languages

Keywords Generic programming, constrained generics, parametric polymorphism, C++ templates, C++0x, concepts

1. Introduction

The C++ language [25, 62] supports parametrized 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 [3,44], Generative Programming [11], and Template Metaprogramming [1, 66]. 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 emulation 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 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, 20, 32, 54, 55, 65], 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 *ad hoc* 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 [61, 63]: 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 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:

```
list<int> lst;
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 code (rather than by a bug in the STL implementation) is the following innocuous line of output:

```
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

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-348-4/06/0010...$5.00.

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

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 parametrization mechanism. However, the power of templates 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 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 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.

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—Polymorphism; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

General Terms Design, Languages

Keywords Generic programming, constrained generics, parametric polymorphism, C++ templates, C++0x, concepts

1. Introduction

The C++ language [25, 62] supports parametrized 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 [3,44], Generative Programming [11], and Template Metaprogramming [1, 66]. 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 emulation 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 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, 20, 32, 54, 55, 65], 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 *ad hoc* 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 [61, 63]: 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 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:

```
list<int> lst;
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 code (rather than by a bug in the STL implementation) is the following innocuous line of output:

```
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

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

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

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 parametrization mechanism. However, the power of templates 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 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 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.

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—Polymorphism; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

General Terms Design, Languages

Keywords Generic programming, constrained generics, parametric polymorphism, C++ templates, C++0x, concepts

1. Introduction

The C++ language [25, 62] supports parametrized 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 [3,44], Generative Programming [11], and Template Metaprogramming [1, 66]. 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 emulation 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 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, 20, 32, 54, 55, 65], 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 *ad hoc* 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 [61, 63]: 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 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:

```
list<int> lst;
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 code (rather than by a bug in the STL implementation) is the following innocuous line of output:

```
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

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

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

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 parametrization mechanism. However, the power of templates 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 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 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.

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—Polymorphism; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

General Terms Design, Languages

Keywords Generic programming, constrained generics, parametric polymorphism, C++ templates, C++0x, concepts

1. Introduction

The C++ language [25, 62] supports parametrized 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 [3,44], Generative Programming [11], and Template Metaprogramming [1, 66]. 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 emulation 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 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, 20, 32, 54, 55, 65], 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 *ad hoc* 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 [61, 63]: 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 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:

```
list<int> lst;
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 code (rather than by a bug in the STL implementation) is the following innocuous line of output:

```
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

2011

[This Slide Intentionally Left Blank]

2012

Document number: N3351=12-0041
Date: 2012-01-13
Working group: Evolution
Reply to: Bjarne Stroustrup <bs@cs.tamu.edu>
Andrew Sutton <asutton@cs.tamu.edu>

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
Sean Parent, Adobe Systems Incorporated (Aug. 1-3)
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

[†]Participated in editing of this report.

Document number: N3351=12-0041
Date: 2012-01-13
Working group: Evolution
Reply to: Bjarne Stroustrup <bs@cs.tamu.edu>
Andrew Sutton <asutton@cs.tamu.edu>

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
Sean Parent, Adobe Systems Incorporated (Aug. 1-3)
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

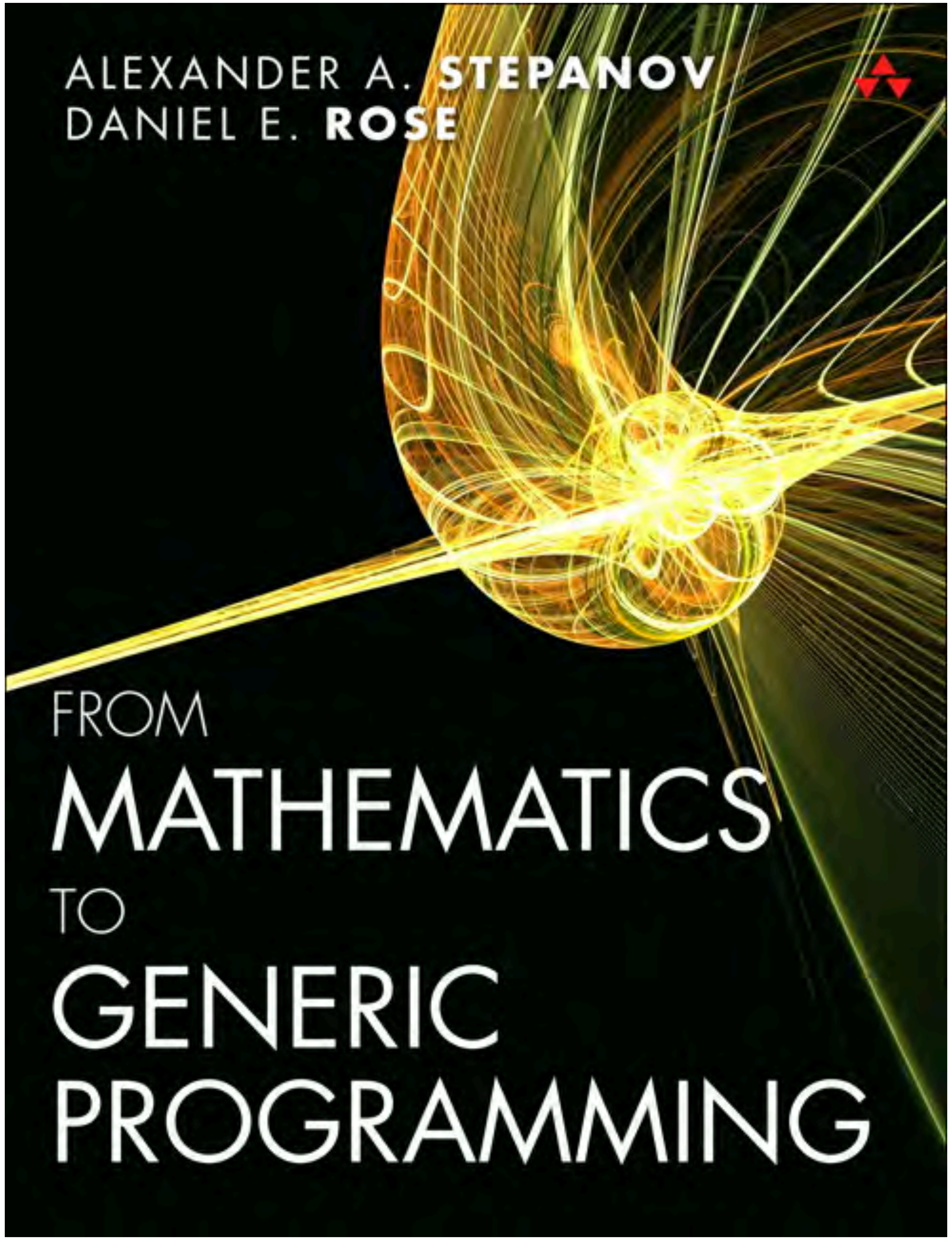
Abstract

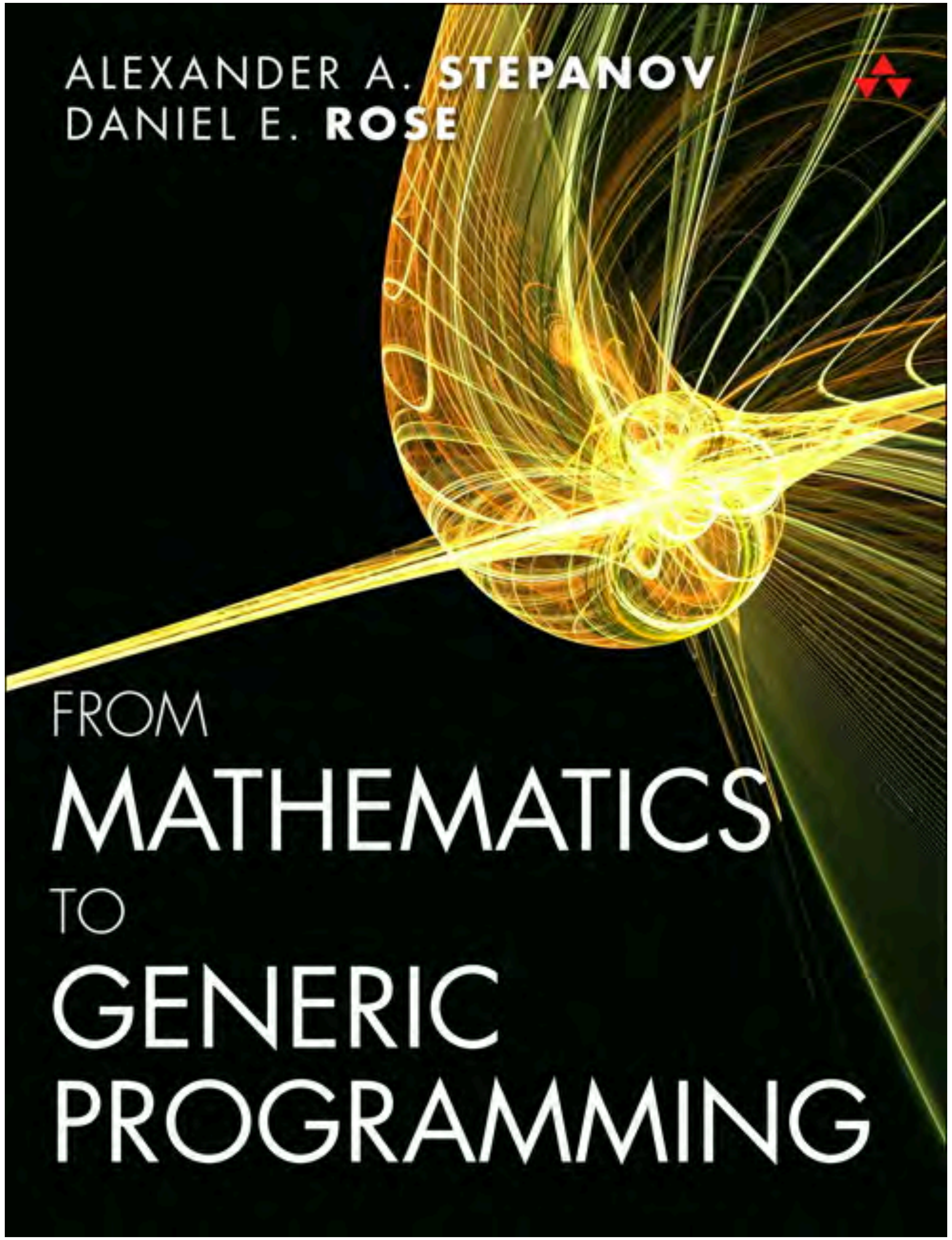
This report presents a concept design for the algorithms part of the STL and outlines the design of the supporting language mechanism. Both are radical simplifications of what was proposed in the C++0x draft. In particular, this design consists of only 41 concepts (including supporting concepts), does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

Contents

†Participated in editing of this report.

2015


ALEXANDER A. STEPANOV
DANIEL E. ROSE
FROM
MATHEMATICS
TO
GENERIC
PROGRAMMING


ALEXANDER A. STEPANOV
DANIEL E. ROSE
FROM
MATHEMATICS
TO
GENERIC
PROGRAMMING

2016

2020

17 Templates [temp]

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-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

[*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*]

2 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*.

3 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:*

```
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>
    constexpr pauli<T> sigma1 = { { 0, 1 }, { 1, 0 } };
  template<class T>
    constexpr pauli<T> sigma2 = { { 0, -1i }, { 1i, 0 } };
```


Adobe

17 Templates [temp]

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

```
  constexpr pauli<T> sigma1 = { { 0, 1 }, { 1, 0 } };
template<class T>
  constexpr pauli<T> sigma2 = { { 0, -1i }, { 1i, 0 } };
```

“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

Adobe

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

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