APL

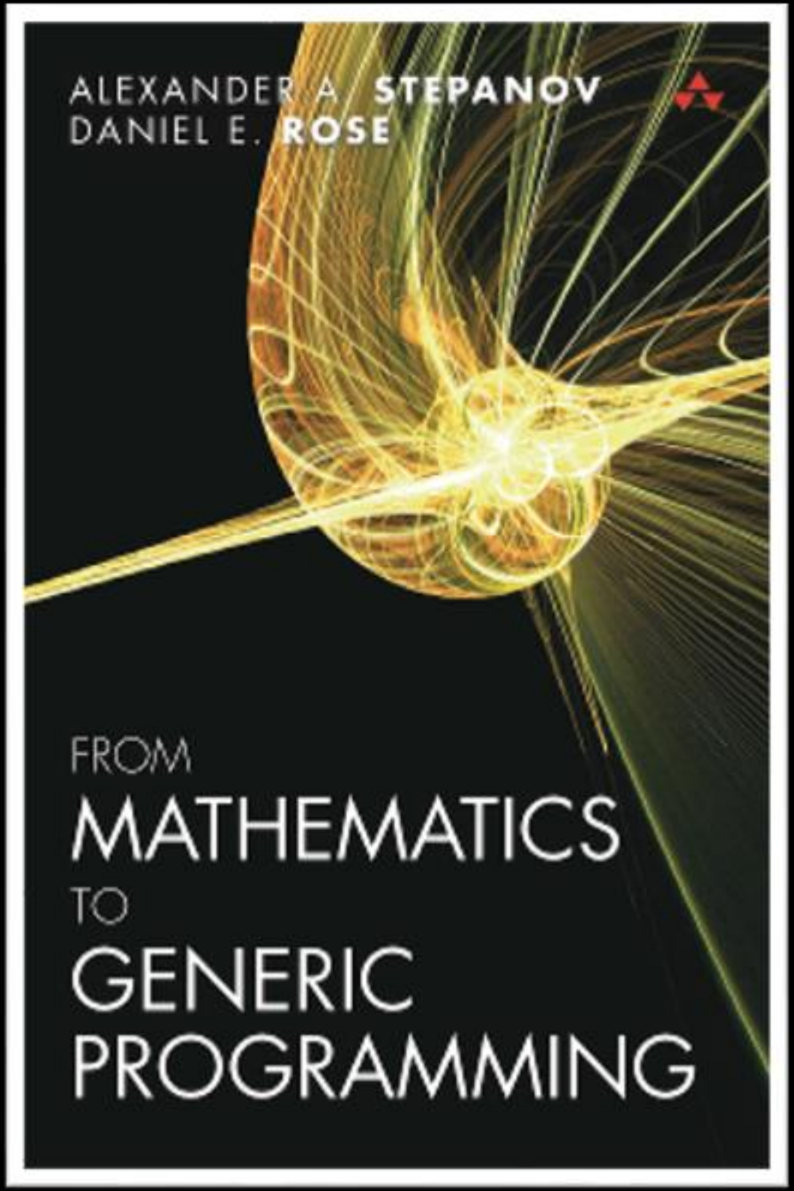
C++

Meetup

COC Berlin Code of Conduct

DISCORD

Discord Link: https://discord.gg/nxwbTHd
Github Repo: https://github.com/codereport/FM2GP-2025
code_report: Twitter | BlueSky | Mastodon
CoC: https://berlincodeofconduct.org/



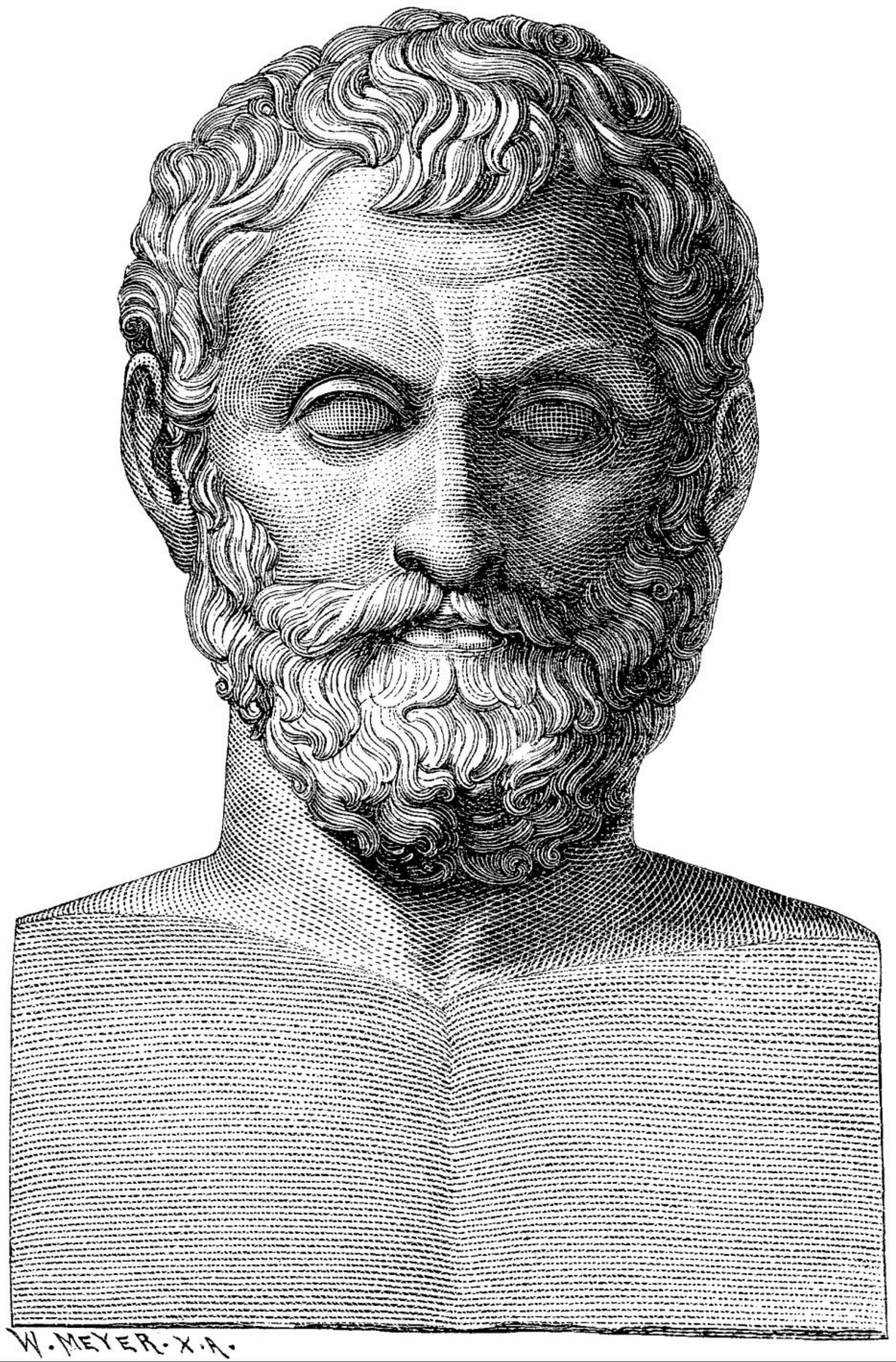
From Mathematics to Generic Programming

Chapter 2/9

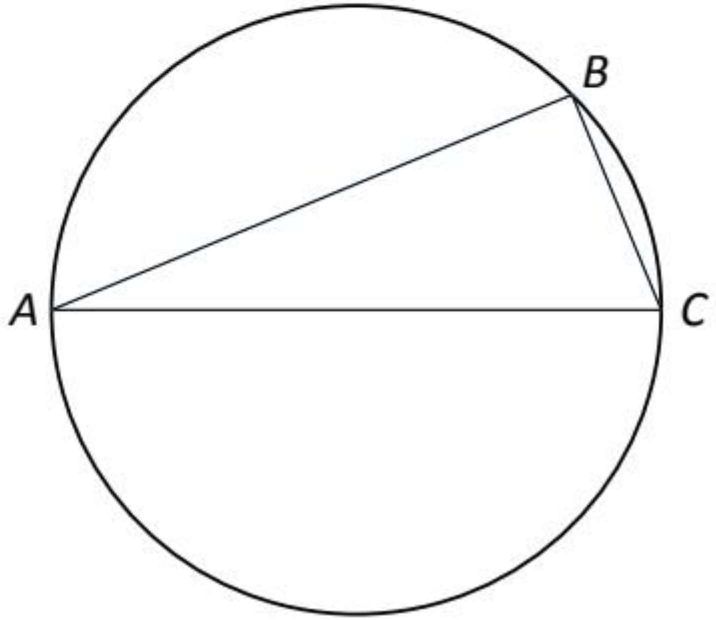
1. What This Book Is About
2. The First Algorithm
3. Ancient Greek Number Theory
4. Euclid's Algorithm
5. The Emergence of Modern Number Theory
6. Abstraction in Mathematics
7. Deriving a Generic Algorithm
8. More Algebraic Structures
9. Organizing Mathematical Knowledge
10. Fundamental Programming Concepts
11. Permutation Algorithms
12. Extensions of GCD
13. A Real-World Application

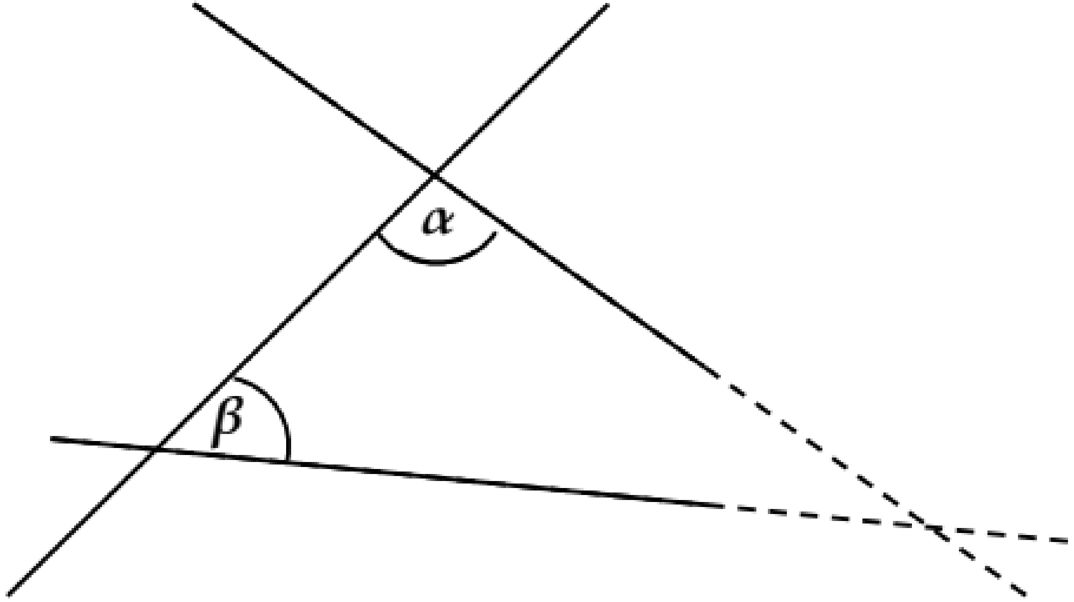
1. What This Book Is About
2. The First Algorithm
3. Ancient Greek Number Theory
4. Euclid’s Algorithm
5. The Emergence of Modern Number Theory
6. Abstraction in Mathematics
7. Deriving a Generic Algorithm
8. More Algebraic Structures
9. Organizing Mathematical Knowledge
10. Fundamental Programming Concepts
11. Permutation Algorithms
12. Extensions of GCD
13. A Real-World Application

1. What This Book Is About
2. The First Algorithm
3. Ancient Greek Number Theory
4. Euclid’s Algorithm
5. The Emergence of Modern Number Theory
6. Abstraction in Mathematics
7. Deriving a Generic Algorithm
8. More Algebraic Structures
9. Organizing Mathematical Knowledge
10. Fundamental Programming Concepts
11. Permutation Algorithms
12. Extensions of GCD
13. A Real-World Application


W.MEYER·X.A.

Theorem 9.1 (Thales' Theorem): *For any triangle ABC formed by connecting the two ends of a circle's diameter (AC) with any other point B on the circle,* $\angle ABC = 90°$.





$$\alpha + \beta < 180°$$

```cpp
auto multiply0(int n, int a) -> int {
  if (n == 1) return a;
  return multiply0(n - 1, a) + a;
}
```

```cpp
auto odd(int n) -> bool { return n & 0x1; }
auto half(int n) -> int { return n >> 1; }

// the Egyptian multiplication algorithm in C++:
int multiply1(int n, int a) {
  if (n == 1) return a;
  int result = multiply1(half(n), a + a);
  if (odd(n)) result = result + a;
  return result;
}
```

```cpp
auto odd(int n) -> bool { return n & 0x1; }
auto half(int n) -> int { return n >> 1; }


// multiply-accumulate function:
int mult_acc0(int r, int n, int a) {
  if (n == 1) return r + a;
  if (odd(n)) {
    return mult_acc0(r + a, half(n), a + a);
  } else {
    return mult_acc0(r, half(n), a + a);
  }
}
```

```cpp
auto odd(int n) -> bool { return n & 0x1; }
auto half(int n) -> int { return n >> 1; }

int mult_acc1(int r, int n, int a) {
  if (n == 1) return r + a;
  if (odd(n)) r = r + a;
  return mult_acc1(r, half(n), a + a);
}
```

```cpp
auto odd(int n) -> bool { return n & 0x1; }
auto half(int n) -> int { return n >> 1; }

// So we can reduce the number of times we have to compare
// w/ 1 by a factor of 2 simply by checking for oddness 1st:
int mult_acc2(int r, int n, int a) {
  if (odd(n)) {
    r = r + a;
    if (n == 1) return r;
  }
  return mult_acc2(r, half(n), a + a);
}
```

```cpp
auto odd(int n) -> bool { return n & 0x1; }
auto half(int n) -> int { return n >> 1; }

// getting back tail recursion
int mult_acc3(int r, int n, int a) {
  if (odd(n)) {
    r = r + a;
    if (n == 1) return r;
  }
  n = half(n);
  a = a + a;
  return mult_acc3(r, n, a);
}
```

```cpp
auto odd(int n) -> bool { return n & 0x1; }
auto half(int n) -> int { return n >> 1; }

// change to iterative
int mult_acc4(int r, int n, int a) {
  while (true) {
    if (odd(n)) {
      r = r + a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}
```

```cpp
auto odd(int n) -> bool { return n & 0x1; }
auto half(int n) -> int { return n >> 1; }

int mult_acc4(int r, int n, int a) {
  while (true) {
    if (odd(n)) {
      r = r + a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}

int multiply2(int n, int a) {
  if (n == 1) return a;
  return mult_acc4(a, n - 1, a);
}
```

```cpp
auto odd(int n) -> bool { return n & 0x1; }
auto half(int n) -> int { return n >> 1; }

int mult_acc4(int r, int n, int a) {
  while (true) {
    if (odd(n)) {
      r = r + a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}
```

```cpp
// This is pretty good, except when n is a power of 2. The first thing we do is
// subtract 1, which means that mult_acc4 will be called with a number whose bi-
// nary representation is all 1s, the worst case for our algorithm. So we’ll
// avoid this by doing some of the work in advance when n is even, halving it
// (and doubling a) until n becomes odd:
int multiply3(int n, int a) {
  while (!odd(n)) {
     a = a + a;
    n = half(n);
  }
  if (n == 1) return a;
  return mult_acc4(a, n - 1, a);
}
```

```cpp
auto odd(int n) -> bool { return n & 0x1; }
auto half(int n) -> int { return n >> 1; }

int mult_acc4(int r, int n, int a) {
  while (true) {
    if (odd(n)) {
      r = r + a;
      if (n == 1) return r;
    }
    n = half(n);
    a = a + a;
  }
}


// But now we notice that we're making mult_acc4 do one unnecessary test for
// odd(n), because we're calling it with an even number. So we'll do one halving
// and doubling on the arguments before we call it, giving us our final version:
int multiply4(int n, int a) {
  while (!odd(n)) {
    a = a + a;
    n = half(n);
  }
  if (n == 1) return a;
  // even(n - 1) ==> n - 1 ≠ 1
  return mult_acc4(a, half(n - 1), a + a);
}
```

discussion

APL

Meetup