# Cryptography in C and C++

MICHAEL WELSCHENBACH
Translated by DAVID KRAMER

Apress™

Translator and Compositor: David Kramer

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Assistant Publisher: Grace Wong

Copy Manager: Nicole LeClerc

Production Manager: Kari Brooks-Copony

Proofreader: Anne Friedman

TEX Support: Fred Bartlett and Arthur Ogawa

Manufacturing Manager: Tom Debolski

Cover Designer: Kurt Krames

Library of Congress Cataloging-in-Publication Data
Welschenbach, Michael.

[Kryptographie in C und C++. English]
Cryptography in C and C++ / Michael Welschenbach ; translated by David Kramer.–
2nd American ed., rev. and enl.
p. cm.
The first American edition is a translation of the second German edition, which has been revised and expanded from the first German edition.
Includes bibliographical references and index.
ISBN 1-59059-502-5
1. Computer security. 2. Cryptography. 3. C (Computer program language) 4. C++ (Computer program language) I. Title.

QA76.9.A25W4313 2005
005.8–dc22 2005002553



Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1



Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de. For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.



The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

*To my family, as always*

# Contents

# List of Figures

# List of Tables

# Foreword

CRYPTOGRAPHY IS AN ANCIENT ART, well over two thousand years old. The need to keep certain information secret has always existed, and attempts to preserve secrets have therefore existed as well. But it is only in the last thirty years that cryptography has developed into a science that has offered us needed security in our daily lives. Whether we are talking about automated teller machines, cellular telephones, Internet commerce, or computerized ignition locks on automobiles, there is cryptography hidden within. And what is more, none of these applications would work without cryptography!

The history of cryptography over the past thirty years is a unique success story. The most important event was surely the discovery of public key cryptography in the mid 1970s. It was truly a revolution: We know today that things are possible that previously we hadn't even dared to think about. Diffie and Hellman were the first to formulate publicly the vision that secure communication must be able to take place spontaneously. Earlier, it was the case that sender and receiver had first to engage in secret communication to establish a common key. Diffie and Hellman asked, with the naivety of youth, whether one could communicate secretly without sharing a common secret. Their idea was that one could encrypt information without a secret key, that is, one that no one else could know. This idea signaled the birth of public key cryptography. That this vision was more than just wild surmise was shown a few years later with the advent of the RSA algorithm.

Modern cryptography has been made possible through the extraordinarily fruitful collaboration between mathematics and computer science. Mathematics provided the basis for the creation and analysis of algorithms. Without mathematics, and number theory in particular, public key cryptography would be impossible. Mathematics provides the results on the basis of which the algorithms operate.

If the cryptographic algorithms are to be realized, then one needs procedures that enable computation with large integers: The algorithms must not function only in theory; they must perform to real-world specifications. That is the task of computer science.

This book distinguishes itself from all other books on the subject in that it makes clear this relationship between mathematics and computing. I know of no book on cryptography that presents the mathematical basis so thoroughly while providing such extensive practical applications, and all of this in an eminently readable style.

What we have here is a master writing about his subject. He knows the theory, and he presents it clearly. He knows the applications, and he presents a host of procedures for realizing them. He knows much, but he doesn't write like a know-it-all. He presents his arguments clearly, so that the reader obtains a clear understanding. In short, this is a remarkable book.

So best wishes to the author! And above all, best wishes to you, the reader!

Albrecht Beutelspacher

# About the Author

MICHAEL WELSCHENBACH CURRENTLY WORKS FOR SRC Security Research & Consulting GmbH in Bonn, Germany. He graduated with a master's degree in mathematics from the University of Cologne and has gained extensive experience in cryptological research over the years. Currently, his favorite programming languages are C and C++. When not working, he enjoys spending time with his wife and two sons, programming, reading, music, photography, and digital imaging.

# About the Translator

DAVID KRAMER EARNED HIS PH.D. in mathematics at the University of Maryland, and his M.A. in music at Smith College. For many years he worked in higher education, first as a professor of mathematics and computer science, and later as a director of academic computing. Since 1995 he has worked as an independent editor and translator. He has edited hundreds of books in mathematics and the sciences and has translated a number of books in a variety of fields, including *The Definitive Guide to Excel VBA* and *The Definitive Guide to MySQL*, both by Michael Kofler; and *Enterprise JavaBeans 2.1*, by Stefan Denninger and Ingo Peters; all published by Apress. Other translations include *Luck, Logic, and White Lies*, by Jörg Bewersdorff; *The Game's Afoot! Game Theory in Myth and Paradox*, by Alexander Mehlmann; the children's musical *Red Riding! Red Riding!* by Ernst Ekker with music by Sergei Dreznin; *In Quest of Tomorrow's Medicines*, by Jürgen Drews; and the novel *To Err Is Divine*, by Ágota Bozai.

# Preface to the Second American Edition

> When I have to wrestle with figures, I feel I'd like to stuff myself into a hole in the ground, so I can't see anything. If I raise my eyes and see the sea, or a tree, or a woman—even if she's an old 'un—damme if all the sums and figures don't go to blazes. They grow wings and I have to chase 'em.
>
> —Nikos Kazanzakis, *Zorba the Greek*

THE SECOND AMERICAN EDITION OF this book has again been revised and enlarged. The chapter on random number generators has been completely rewritten, and the section on primality testing was substantially revised. The new results of Agrawal, Kayal, and Saxena on primality tests, whose discovery in 2002 that "PRIMES is in P" caused a sensation, are covered. The chapter on Rijndael/AES has been relocated for a better presentation, and it is pointed out that the standardization of Rijndael as the Advanced Encryption Standard has meanwhile been made official by the U.S. National Institute of Standards and Technology (NIST).

Unlike previous editions of the book, the second American edition does not contain a CD-ROM with the source code for the programs presented. Instead, the source code is available for download at `www.apress.com` in the Downloads section.

I wish to thank the publishers and translators who have meanwhile made this book available in Chinese, Korean, Polish, and Russian and through their careful reading have contributed to the quality of this edition.

I again thank David Kramer for his engaging and painstaking English translation, and Gary Cornell, of Apress, for his willingness to bring out the second American edition.

Finally, I wish to thank Springer Science publishers, and in particular once again Hermann Engesser, Dorothea Glausinger, and Ulrike Sricker, for their pleasant collaboration.

# Preface to the First American Edition

> Mathematics is a misunderstood and even maligned discipline. It's not the brute computations they drilled into us in grade school. It's not the science of reckoning. Mathematicians do not spend their time thinking up cleverer ways of multiplying, faster methods of adding, better schemes for extracting cube roots.
>
> —Paul Hoffman, *The Man Who Loved Only Numbers*

THE FIRST AMERICAN EDITION IS A TRANSLATION OF the second German edition, which has been revised and expanded from the first German edition in a number of ways. Additional examples of cryptographic algorithms have been added, such as the procedures of Rabin and El Gamal, and in the realization of the RSA procedure the hash function RIPEMD-160 and formatting according to PKCS #1 have been adopted. There is also a discussion of possible sources of error that could lead to a weakening of the procedure. The text has been expanded or clarified at a number of points, and errors have been corrected. Additionally, certain didactic strategies have been strengthened, with the result that some of the programs in the source code differ in certain details from those presented in the book. Not all technical details are of equal importance, and the desire for fast and efficient code is not always compatible with attractive and easy-to-read programs.

And speaking of efficiency, in Appendix D running times are compared to those for certain functions in the GNU Multiprecision Library. In this comparison the FLINT/C exponentiation routine did not do at all badly. As a further extension, Appendix F provides references to some arithmetic and number-theoretic packages.

The software has been expanded by several functions and in places has been significantly overhauled, and in the process a number of errors and points of imprecision were removed. Additional test functions were developed and existing test functions expanded. A security mode was implemented, whereby security-critical variables in the individual functions are deleted by being overwritten. All C and C++ functions are now clearly cited and annotated in the appendices.

Since current compilers represent varying stages of development of the C++ standard, the C++ modules of the FLINT/C package have been set up in such a way that both traditional C++ header files of the form `xxxxx.h` and the new

ANSI header files can be used. For the same reason the use of the operator `new()` has been checked, as always, as to whether the null pointer is returned. This type of error handling does not make use of the ANSI standard *exceptions*, but it nonetheless functions with current compilers, while the method that conforms to the standard, by which `new()` generates an error via `throw()`, is not universally available.

Although the focus of this book is the fundamentals of asymmetric cryptography, the recent nomination of *Rijndael* by the American National Institute of Standards and Technology (NIST) to be the advanced encryption standard (AES) encouraged me to include a final chapter (Chapter 11) with an extensive description of this algorithm. I am indebted to Gary Cornell, at Apress, for bringing up the subject and convincing me that this would be a worthwhile complement to the topics of this book. I would like to thank Vincent Rijmen, Antoon Bosselaers, Paulo Barreto, and Brian Gladman for their kind permission to include the source code for their Rijndael implementations in the source code that accompanies this book.

I wish to thank all the readers of the first edition, particularly those who called errors to my attention, made comments, or suggested improvements. All their communications were most welcome. As always, the author assumes all responsibility for errors that may yet remain in the text or the software, as well as for any new errors that may have crept in.

I offer my heartfelt thanks to Gary Cornell, at Apress, and again to Hermann Engesser, Dorothea Glaunsinger, and Ulrike Stricker, at Springer-Verlag, for their unstinting commitment and friendly collaboration.

I am deeply grateful to my translator, David Kramer, who has contributed with distinguished expertise and indefatigable dedication many valuable hints, which have been incorporated into the German edition of this book as well.

## Warning

Before making use of the programs contained in this book please refer to the manuals and technical introductions for the relevant software and computers. Neither the author nor the publisher accepts any responsibility for losses due to improper execution of the instructions and programs contained in this book or due to errors in the text or in the programs that despite careful checking may remain. The programs in the downloadable source code are protected by copyright and may not be reproduced without permission of the publisher.

## Disclaimer

In this book frequent use is made of the term “leading zeros.” The use of this term is in no way to be construed as alluding to any person or persons, in public or private life, living or dead, and any such correspondence is entirely coincidental.

# Preface to the First German Edition

> Mathematics is the queen of the sciences, and number theory is the queen of mathematics. Frequently, she deigns to assist astronomy and other of the natural sciences, but primacy is due her under all circumstances.
>
> —Carl Friedrich Gauss

WHY DO WE NEED A book on cryptography whose principal focus is the arithmetic of whole numbers—the integers—and its application to computer programming? Is this not a rather insignificant subject in comparison to the important problems with which computer science generally involves itself? So long as one confines oneself to the range of numbers that can be represented by the standard numerical types of a programming language, arithmetic is a rather simple affair, and the familiar arithmetic operations make their traditional appearances in programs accompanied by the familiar symbols $+, -, /, *$.

But if one requires results whose length far exceeds what can be expressed in 16 or 32 bits, then the situation begins to get interesting. Even the basic arithmetic operations are no longer available for such numbers, and one gets nowhere without first investing considerable effort in solving problems that never even seemed like problems before. Anyone who investigates problems in number theory, whether professionally or as a hobby, in particular the topic of contemporary cryptography, is familiar with such issues: The techniques of doing arithmetic that we learned in school now demand renewed attention, and we find ourselves sometimes dealing with incredibly involved processes.

The reader who wishes to develop programs in these areas and is not inclined to reinvent the wheel will find included with this book a suite of functions that will serve as an extension of C and C++ for calculating with large integers. We are not talking about "toy" examples that say, "this is how it works in principle," but a complete collection of functions and methods that satisfy the professional requirements of stability, performance, and a sound theoretical basis.

Making the connection between theory and practice is the goal of this book, that is, to close the gap between the theoretical literature and practical programming problems. In the chapters ahead we shall develop step by step the fundamental calculational principles for large natural numbers, arithmetic in finite rings and fields, and the more complex functions of elementary number theory, and we shall elucidate the many and various possibilities for applying

these principles to modern cryptography. The mathematical fundamentals will be explained to the extent necessary for understanding the programs that are presented here, and for those interested in pursuing these matters further there are extensive references to the literature. The functions that we develop will then be brought together and extensively tested, resulting in a useful and comprehensive programming interface.

Beginning with the representation of large numbers, in the following chapters we shall first deal with the fundamentals of computation. For addition, subtraction, multiplication, and division of large numbers we shall create powerful basic functions. Building on these, we shall explain modular arithmetic in residue classes and implement the relevant operations in library functions. A separate chapter is devoted to the time-intensive process of exponentiation, where we develop and program various specialized algorithms for a number of applications in modular arithmetic.

After extensive preparation, which includes input and output of large numbers and their conversion into various bases, we study algorithms of elementary number theory using the basic arithmetic functions, and we then develop programs, beginning with the calculation of the greatest common divisor of large numbers. We shall then move on to such problems as calculating the Legendre and Jacobi symbols, and inverses and square roots in finite rings, and we shall also become familiar with the Chinese remainder theorem and its applications.

In connection with this we shall go into some detail about the principles of identifying large prime numbers, and we shall program a powerful multistage primality test.

A further chapter is devoted to the generation of large random numbers, in which a cryptographically useful bit generator is developed and tested with respect to its statistical properties.

To end the first part we shall concern ourselves with testing arithmetic and other functions. To do this we shall derive special test methods from the mathematical rules of arithmetic, and we shall consider the implementation of efficient external tools.

The subject of the second part is the step-by-step construction of the C++ class LINT (Large INTegers), in the course of which we shall embed the C functions of the first part into the syntax and semantics of the object-oriented programming language C++. We shall put particular weight on formatted input and output of LINT objects with flexible stream functions and manipulators, as well as error handling with exceptions. The elegance with which algorithms can be formulated in C++ is particularly impressive when the boundaries between standard types and large numbers as LINT objects begin to dissolve, resulting in the syntactic closeness to the implemented algorithms and in great clarity and transparency.

Finally, we shall demonstrate the application of the methods we have developed by implementing an extensive RSA cryptosystem for encryption and the creation of digital signatures. In the process we shall explain the theory of the RSA procedure and its operation as the most prominent representative of asymmetric cryptosystems, and in a self-contained example we shall develop an extensible kernel for applications of this ultramodern cryptographic process according to the object-oriented principles of the programming language C++.

We shall round all of this off with a glimpse of further possible extensions of the software library. As a small highlight at the end we shall present four functions in 80x86 assembly language for multiplication and division, which will improve the performance of our software. Appendix D contains a table of typical calculation times with and without the assembler supplement.

All readers of this book are heartily invited to join me on this path, or perhaps—depending on individual interest—to focus on particular sections or chapters and try out the functions presented there. The author hopes that it will not be taken amiss that he refers to his readers, together with himself, as "we." He hopes thereby to encourage them to take an active role in this journey through a cutting-edge area of mathematics and computer science, to figure things out for themselves and take from this book what is of greatest benefit. As for the software, let the reader not be lacking in ambition to extend the scope or speed of one or more functions through new implementations.

I wish to thank Springer-Verlag and particularly Hermann Engesser, Dorothea Glaunsinger, and Ulrike Stricker for their interest in the publication of this book and for their friendly and active collaboration. The manuscript was reviewed by Jörn Garbers, Josef von Helden, Brigitte Nebelung, Johannes Ueberberg, and Helga Welschenbach. I offer them my heartfelt thanks for their critical suggestions and improvements, and above all for their care and patience. If despite all of our efforts some errors remain in the text or in the software, the author alone bears the responsibility. I am extremely grateful to my friends and colleagues Robert Hammelrath, Franz-Peter Heider, Detlef Kraus, and Brigitte Nebelung for their insights into the connections between mathematics and computer science over many years of collaboration that have meant a great deal to me.

Part I

# Arithmetic and Number Theory in C

How necessary arithmetic and the entire art of mathematics are can be easily measured, in that nothing can be created that is not connected with precise number and measurement, and no independent art can exist without its measures and proportions.

—Adam Ries: *Book of Calculation*, 1574

Typographical rules for manipulating numerals are actually arithmetical rules for operating on numbers.

—D. R. Hofstadter: *Gödel, Escher, Bach: An Eternal Golden Braid*

The human brain would no longer be burdened with anything that needed to be calculated! Gifted people would again be able to think instead of scribbling numbers.

—Sten Nadolny: *The Discovery of Slowness*, trans. Ralph Freedman

CHAPTER 1

# Introduction

God created the integers. All the rest is the work of man.

—Leopold Kronecker

If you look at zero you see nothing; but look through it and you will see the world.

—Robert Kaplan, *The Nothing That Is: A Natural History of Zero*

TO BE INVOLVED WITH MODERN cryptography is to dive willy-nilly into number theory, that is, the study of the natural numbers, one of the most beautiful areas of mathematics. However, we have no intention of becoming deep-sea divers who raise sunken treasure from the mathematical ocean floor, which in any case is unnecessary for cryptographic applications. Our goals are much more modest. On the other hand, there is no limit to the depth of involvement of number theory with cryptography, and many significant mathematicians have made important contributions to this area.

The roots of number theory reach back to antiquity. The Pythagoreans—the Greek mathematician and philosopher Pythagoras and his school—were already deeply involved in the sixth century B.C.E. with relations among the integers, and they achieved significant mathematical results, for example the famed Pythagorean theorem, which is a part of every school child's education. With religious zeal they took the position that all numbers should be commensurate with the natural numbers, and they found themselves on the horns of a serious dilemma when they discovered the existence of "irrational" numbers such as $\sqrt{2}$, which cannot be expressed as the quotient of two integers. This discovery threw the world view of the Pythagoreans into disarray, to the extent that they sought to suppress knowledge of the irrational numbers, a futile form of behavior oft repeated throughout human history.

Two of the oldest number-theoretic algorithms, which have been passed down to us from the Greek mathematicians Euclid (third century B.C.E.) and Eratosthenes (276–195 B.C.E.), are closely related to the most contemporary encryption algorithms that we use every day to secure communication across the Internet. The "Euclidean algorithm" and the "sieve of Eratosthenes" are both quite up-to-date for our work, and we shall discuss their theory and application in Sections 10.1 and 10.5 of this book.

Among the most important founders of modern number theory are to be counted Pierre de Fermat (1601–1665), Leonhard Euler (1707–1783), Adrien Marie Legendre (1752–1833), Carl Friedrich Gauss (1777–1855), and Ernst Eduard Kummer (1810–1893). Their work forms the basis for the modern development of this area of mathematics and in particular the interesting application areas such as cryptography, with its asymmetric procedures for encryption and the generation of digital signatures (cf. Chapter 17). We could mention many more names of important contributors to this field, who continue to this day to be involved in often dramatic developments in number theory, and to those interested in a thrilling account of the history of number theory and its protagonists, I heartily recommend the book *Fermat's Last Theorem*, by Simon Singh.

Considering that already as children we learned counting as something to be taken for granted and that we were readily convinced of such facts as that two plus two equals four, we must turn to surprisingly abstract thought constructs to derive the theoretical justification for such assertions. For example, set theory allows us to derive the existence and arithmetic of the natural numbers from (almost) nothing. This "almost nothing" is the empty (or null) set $\varnothing := \{\ \}$, that is, the set that has no elements. If we consider the empty set to correspond to the number $0$, then we are able to construct additional sets as follows. The successor $0^+$ of $0$ is associated with the set $0^+ := \{\, 0 \,\} = \{\, \varnothing \,\}$, which contains a single element, namely the null set. We give the successor of $0$ the name $1$, and for this set as well we can determine a successor, namely $1^+ := \{\, \varnothing, \{\, \varnothing \,\} \,\}$. The successor of $1$, which contains $0$ and $1$ as its elements, is given the name $2$. The sets thus constructed, which we have rashly given the names $0$, $1$, and $2$, we identify—not surprisingly—with the well-known natural numbers $0$, $1$, and $2$.

This principle of construction, which to every number $x$ associates a successor $x^+ := x \cup \{\, x \,\}$ by adjoining $x$ to the previous set, can be continued to produce additional numbers. Each number thus constructed, with the exception of $0$, is itself a set whose elements constitute its *predecessors*. Only $0$ has no predecessor. To ensure that this process continues ad infinitum, set theory formulates a special rule, called the *axiom of infinity:* There exists a set that contains $0$ as well as the successor of every element that it contains.

From this postulated existence of (at least) one so-called *successor set*, which, beginning with $0$, contains all successors, set theory derives the existence of a minimal successor set $\mathbb{N}$, which is itself a subset of every successor set. This minimal and thus uniquely determined successor set $\mathbb{N}$ is called the set of *natural numbers*, in which we expressly include zero as an element.[1]

[1] It was not decisive for this choice that according to standard DIN 5473 zero belongs to the natural numbers. From the point of view of computer science, however, it is practical to begin counting at zero instead of 1, which is indicative of the important role played by zero as the neutral element for addition (additive identity).

The natural numbers can be characterized by means of the axioms of Giuseppe Peano (1858–1932), which coincide with our intuitive understanding of the natural numbers:

(I) The successors of two unequal natural numbers are unequal: From $n \neq m$ it follows that $n^+ \neq m^+$, for all $n, m \in \mathbb{N}$.

(II) Every natural number, with the exception of $0$, has a predecessor: $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$.

(III) The principle of *complete induction:* If $S \subset \mathbb{N}$, $0 \in S$, and $n \in S$ always imply $n^+ \in S$, then $S = \mathbb{N}$.

The principle of complete induction makes it possible to derive the arithmetic operations with natural numbers in which we are interested. The fundamental operations of addition and multiplication can be defined recursively as follows. We begin with **addition**:

For every natural number $n \in \mathbb{N}$ there exists a function $s_n$ from $\mathbb{N}$ to $\mathbb{N}$ such that

(i) $s_n(0) = n$,

(ii) $s_n\left(x^+\right) = (s_n(x))^+$ for all natural numbers $x \in \mathbb{N}$.

The value of the function $s_n(x)$ is called the *sum* $n + x$ of $n$ and $x$.

The existence of such functions $s_n$ for all natural numbers $n \in \mathbb{N}$ must, however, be proved, since the infinitude of natural numbers does not a priori justify such an assumption. The existence proof goes back to the principle of complete induction, corresponding to Peano's third axiom above (see [Halm], Chapters 11–13). For **multiplication** one proceeds analogously:

For every natural number $n \in \mathbb{N}$ there exists a function $p_n$ from $\mathbb{N}$ to $\mathbb{N}$ such that

(i) $p_n(0) = 0$,

(ii) $p_n\left(x^+\right) = p_n(x) + n$ for all natural numbers $x \in \mathbb{N}$.

The value of the function $p_n(x)$ is called the *product* $n \cdot x$ of $n$ and $x$.

As expected, multiplication is defined in terms of addition. For the arithmetic operations thus defined one can prove, through repeated application of complete induction on $x$ in accordance with Axiom III, such well-known arithmetic laws as associativity, commutativity, and distributivity (cf. [Halm], Chapter 13). Although we usually use these laws without further ado, we shall help ourselves to them as much as we please in testing our FLINT functions (see Chapters 13 and 18).

In a similar way we obtain a definition of **exponentiation**, which we give here in view of the importance of this operation in what follows.

For every natural number $n \in \mathbb{N}$ there exists a function $e_n$ from $\mathbb{N}$ to $\mathbb{N}$ such that

(i) $e_n(0) = 1$,

(ii) $e_n\left(x^+\right) = e_n(x) \cdot n$ for every natural number $x \in \mathbb{N}$.

The value of the function $e_n(x)$ is called the $x$th *power* $n^x$ of $n$. With complete induction we can prove the *power law*

$$n^x n^y = n^{x+y}, \quad n^x \cdot m^x = (n \cdot m)^x, \quad (n^x)^y = n^{xy},$$

to which we shall return in Chapter 6.

In addition to the calculational operations, the set $\mathbb{N}$ of natural numbers has defined on it an order relation "$<$" that makes it possible to compare two elements $n, m \in \mathbb{N}$. Although this fact is worthy of our great attention from a set-theoretic point of view, here we shall content ourselves with noting that the order relation has precisely those properties that we know about and use in our everyday lives.

Now that we have begun with establishing the empty set as the sole fundamental building block of the natural numbers, we now proceed to consider the materials with which we shall be concerned in what follows. Although number theory generally considers the natural numbers and the integers as given and goes on to consider their properties without excessive beating about the bush, it is nonetheless of interest to us to have at least once taken a glance at a process of "mathematical cell division," a process that produces not only the natural numbers, but also the arithmetic operations and rules with which we shall be deeply involved from here on.

## About the Software

The software described in this book constitutes in its entirety a package, a so-called function library, to which frequent reference will be made. This library has been given the name FLINT/C, which is an acronym for "functions for large integers in number theory and cryptography."

The FLINT/C library contains, among other items, the modules shown in Tables 1-1 through 1-5, which can be found as source code at `www.apress.com`.

*Table 1-1. Arithmetic and number theory in C in directory flint/src*

| | |
|---|---|
| `flint.h` | header file for using functions from `flint.c` |
| `flint.c` | arithmetic and number-theoretic functions in C |
| `kmul.{h,c}` | functions for Karatsuba multiplication and squaring |
| `ripemd.{h,c}` | implementation of the hash function RIPEMD-160 |
| `sha{1,256}.{h,c}` | implementations of the hash functions SHA-1, SHA-256 |
| `entropy.c` | generation of entropy as start value for pseudorandom sequences |
| `random.{h,c}` | generation of pseudorandom numbers |
| `aes.{h,c}` | implementation of the Advanced Encryption Standard |

*Table 1-2. Arithmetic modules in 80x86 assembler (see Chapter 19) in directory flint/src/asm*

| | |
|---|---|
| `mult.{s,asm}` | multiplication, replaces the C function `mult()` in `flint.c` |
| `umul.{s,asm}` | multiplication, replaces the C function `umul()` |
| `sqr.{s,asm}` | squaring, replaces the C function `sqr()` |
| `div.{s,asm}` | division, replaces the C function `div_l()` |

*Table 1-3. Tests (see Section 13.2 and Chapter 18) in directories flint/test and flint/test/testvals*

| | |
|---|---|
| `testxxx.c[pp]` | test programs in C and C++ |
| `xxx.txt` | test vectors for AES |

*Table 1-4. Libraries in 80x86 assembler (see Chapter 19) in directories flint/lib and flint/lib/dll*

| | |
|---|---|
| `flinta.lib` | library with assembler functions in OMF (object module format) |
| `flintavc.lib` | library with assembler functions in COFF (common object file format) |
| `flinta.a` | archive with assembler functions for emx/gcc under OS/2 |
| `libflint.a` | archive with assembler functions for use under LINUX |
| `flint.dll` | DLL with the FLINT/C functions for use with MS VC/C++ |
| `flint.lib` | link library for `flint.dll` |

*Table 1-5. RSA implementation (see Chapter 17) in directory flint/rsa*

| | |
|---|---|
| `rsakey.h` | header file for the RSA classes |
| `rsakey.cpp` | implementation of the RSA classes `RSAkey` and `RSApub` |
| `rsademo.cpp` | example application of the RSA classes and their functions |

A list of the individual components of the FLINT/C software can be found in the file `readme.doc` is the source code. The software has been tested with the indicated development tools on the following platforms:

- GNU gcc under Linux, SunOS 4.1, and Sun Solaris
- GNU/EMX gcc under OS/2 Warp, DOS, and Windows (9x, NT)
- Borland BCC32 under Windows (9x, NT, 2000, XP)
- lcc-win32 under Windows (9x, NT, 2000, XP)
- Cygnus cygwin B20 under Windows (9x, NT, 2000, XP)
- IBM VisualAge under OS/2 Warp and Windows (9x, NT, 2000, XP)
- Microsoft C under DOS, OS/2 Warp, and Windows (9x, NT)
- Microsoft Visual C/C++ under Windows (9x, NT, 2000, XP)
- Watcom C/C++ under DOS, OS/2 Warp, and Windows (3.1, 9x, NT, XP)
- OpenWatcom C/C++ under Windows (2000, XP)

The assembler programs can be translated with Microsoft MASM,[2] with Watcom WASM, or with the GNU assembler GAS. They are contained in the downloadable source code in translated form as libraries in OMF (object module

[2] Call: `ml /Cx /c /Gd` ⟨`filename`⟩.

format) and COFF (common object file format), respectively, as well as in the form of a LINUX archive, and are used instead of the corresponding C functions when in translating C programs the macro `FLINT_ASM` is defined and the assembler object modules from the libraries, respectively archives, are linked.

A typical compiler call, here for the GNU compiler `gcc`, looks something like the following (with the paths to the source directories suppressed):

```
gcc -O2 -o rsademo rsademo.cpp rsakey.cpp flintpp.cpp
   randompp.cpp flint.c aes.c ripemd.c sha256.c entropy.c
   random.c -lstdc++
```

The C++ header files following the ANSI standard are used when in compilation the macro `FLINTPP_ANSI` is defined; otherwise, the traditional header files `xxxxx.h` are used.

Depending on the computer platform, there may be deviations with regard to the compiler switches; but to achieve maximum performance the options for speed optimization should always be turned on. Because of the demands on the *stack*, in many environments and applications it will have to be adjusted.[3] Regarding the necessary stack size for particular applications, one should note the suggestion about the exponentiation functions in Chapter 6 and in the overview on page 117. The stack requirements can be lessened by using the exponentiation function with dynamic stack allocation as well as by the implementation of dynamic registers (see Chapter 9).

The C functions and constants have been provided with the macros

| | |
|---|---|
| __FLINT_API | Qualifier for C functions |
| __FLINT_API_A | Qualifier for assembler functions |
| __FLINT_API_DATA | Qualifier for constants |

as in

```
extern int __FLINT_API add_l(CLINT, CLINT, CLINT);
extern USHORT __FLINT_API_DATA smallprimes[];
```

or, respectively, in the use of the assembler functions

```
extern int __FLINT_API_A div_l (CLINT, CLINT, CLINT, CLINT);
```

These macros are generally defined as empty comments `/**/`. With their aid, using the appropriate definitions, compiler- and linker-specific instructions to functions and data can be made. If the assembler modules are used and not

[3] With modern computers with virtual memory, except in the case of DOS, one usually does not have to worry about this point, in particular with Unix or Linux systems.

the GNU compiler gcc, the macro __FLINT_API_A is defined by __cdecl, and some compilers understand this as an instruction that the assembler functions corresponding to the C name and calling conventions are to be called.

For modules that import FLINT/C functions and constants from a dynamic link library (DLL) under Microsoft Visual C/C++, in translation the macros -D__FLINT_API=__cdecl and -D__FLINT_API_DATA= __declspec (dllimport) must be defined. This has already been taken into account in flint.h, and it suffices in this case to define the macro FLINT_USEDLL for compilation. For other development environments analogous definitions should be employed.

The small amount of work involved in initializing a FLINT/C DLL is taken care of by the function FLINTInit_l(), which provides initial values for the random number generator[4] and generates a set of dynamic registers (see Chapter 9). The complementary function FLINTExit_l() deallocates the dynamic registers. Sensibly enough, the initialization is not handed over to every individual process that uses the DLL, but is executed once at the start of the DLL. As a rule, a function with creator-specific signature and calling convention should be used, which is executed automatically when the DLL is loaded by the run-time system. This function can take over the FLINT/C initialization and use the two functions mentioned above. All of this should be considered when a DLL is created.

Some effort was made to make the software usable in security-critical applications. To this end, in *security mode* local variables in functions, in particular CLINT and LINT objects, are deleted after use by being overwritten with zeros. For the C functions this is accomplished with the help of the macro PURGEVARS_L() and the associated function purgevars_l(). For the C++ functions the destructor ∼LINT() is similarly equipped. The assembler functions overwrite their working memory. The deletion of variables that are passed as arguments to functions is the responsibility of the calling functions.

If the deletion of variables, which requires a certain additional expenditure of time, is to be omitted, then in compilation the macro FLINT_UNSECURE must be defined. At run time the function char* verstr_l() gives information about the modes set at compile time, in which additionally to the version label X.x, the letters “a” for assembler support and “s” for security mode are output in a character string if these modes have been turned on.

[4] The initial values are made up of 32-bit numbers taken from the system clock. For applications in which security is critical it is advisable to use suitable random values from a sufficiently large interval as initial values.