

# Programming Concepts

Rex Jaeschke

Programming Concepts

© 1995, 2000–2001, 2009 Rex Jaeschke.

Edition: 2.0

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means whatsoever, except in the case of brief quotations embodied in critical reviews and articles.

The information in this book is subject to change without notice and should not be construed as a commitment by the author or the publisher. Although every precaution has been taken in the preparation of this book, the author and the publisher assume no responsibility for errors or omissions.

Java is a trademark of Sun Microsystems.

**The training materials associated with this book are available for license. Interested parties should contact the author at the address below.**

Please address comments, corrections, and questions to the author:

Rex Jaeschke  
2051 Swans Neck Way  
Reston, VA 20191-4023  
+1 (703) 860-0091  
[www.RexJaeschke.com](http://www.RexJaeschke.com)  
[rex@RexJaeschke.com](mailto:rex@RexJaeschke.com)

<b>1. Number Systems .....</b>	<b>1</b>
1.1 The Decimal Number System .....	1
1.2 The Binary Number System .....	2
1.3 Binary Coded Decimal Numbers .....	5
1.4 The Octal Number System .....	6
1.5 The Hexadecimal Number System .....	8
1.6 Programming Calculators .....	10
<b>2. Bit Manipulation .....</b>	<b>11</b>
2.1 The Bitwise AND Operator .....	11
2.2 The Bitwise Inclusive OR Operator .....	15
2.3 The Bitwise NOT Operator .....	17
2.4 The Bitwise Exclusive OR Operator .....	19
2.5 The Bitwise SHIFT Operators .....	20
2.6 Programming Calculators .....	21
<b>3. Data Representation .....</b>	<b>23</b>
3.1 Basic Machine Objects .....	23
3.2 Object Alignment .....	24
3.3 Characters .....	25
3.4 Strings .....	27
3.5 Integers .....	30
3.5.1 Unsigned Integers .....	30
3.5.2 Signed Integers .....	30
3.5.3 Signed-Magnitude Representation .....	30
3.5.4 Characters as Integers .....	35
3.6 Floating-Point Numbers .....	35
3.6.1 Representation .....	35
3.6.2 Range and Precision .....	36
3.6.3 Some Examples .....	36
3.6.4 Pseudo-Numbers .....	37
3.7 Addresses and Pointers .....	38
3.8 Byte/Word Ordering .....	38
3.9 Logical Versus Physical Object Size .....	40
3.10 Formatted Versus Unformatted Representation .....	40



# 1. Number Systems

We all know how to count. How do we count? Well, we just “know” how; there probably are some rules but we've long forgotten them.

For quite a few centuries now, the predominant number system has been one based on the number 10; hence the term *decimal*, from the Latin *decima*, meaning “one tenth”.

While an understanding of computer-based arithmetic is not necessary for most programming tasks, without such knowledge we cannot exploit certain kinds of hardware or programming languages, or understand some of the principles on which modern computing is based.

Let's begin by looking more formally at the decimal number system, the one we use every day, for things like prices, street addresses, and telephone numbers.

## 1.1 The Decimal Number System

Consider the number 12,458. We all “know” that number represents the value twelve thousand four hundred and fifty eight. How do we know that? Well for starters, the decimal number system has 10 distinct symbols, the digits 0–9. Decimal numbers are written such that the value of a digit is based on its position within that number. For example, the digit 5 can represent five, fifty, or five hundred. (This is different to some other number systems such as those developed by the Romans and Japanese.) Let's break down the decimal number 12,458 into its parts, as follows:

$1 \times 10,000$	=	$1 \times 10^4$	=	10,000
$2 \times 1,000$	=	$2 \times 10^3$	=	2,000
$4 \times 100$	=	$4 \times 10^2$	=	400
$5 \times 10$	=	$5 \times 10^1$	=	50
$8 \times 1$	=	$8 \times 10^0$	=	8
				12,458

The digits, starting from the right-most one going left, are referred to as the *ones* digit, the *tens* digit, the *hundreds* digit, the *thousands* digit, the *ten thousands* digit, and so on, for obvious reasons.

## Programming Concepts

Consider the following example in which we add two numbers together:

$$\begin{array}{r} 546 \\ +478 \\ \hline 1,024 \end{array}$$

When two digits are added, the result may be larger than one digit, in which case, we *carry* 1 to the next column to the left. For example,  $6 + 8 = 14$ , so we carry 1 and put down 4. The next column to the left now becomes  $4 + 7 + 1 = 12$ , the 1 is carried and the 2 is put down. And so the process is repeated. With subtraction, when we subtract one digit from another we may have to *borrow* 1 instead of carrying it. For example:

$$\begin{array}{r} 1,024 \\ -478 \\ \hline 546 \end{array}$$

Whether we realize it or not, we need to know 400 rules to do basic addition, subtraction, multiplication, and division of two decimal digits. We simply have to remember them! And even though we can take a few shortcuts ( $3 + 4$  is equivalent to  $4 + 3$ , for example) there are still a lot of rules.

If we were to build a machine to perform these basic arithmetic operations, it would have to know all these rules. It would also need a way to represent each of the 10 digits. Mechanically, this is possible; an old-style automobile mileage odometer is one example. However, representing 10 different states makes the machine more complicated. This is particularly so if the machine is based on magnetic, electrical, or electronic properties, as are all modern computers.

The decimal number system is based on the number 10. The numbers we normally write then are base-10 numbers. We can write such numbers with the base explicitly shown. For example, the number 123 can be written formally with a decimal base using the notation  $123_{10}$ , where the base is shown as a subscript. In ordinary usage, the subscript is omitted because the writer usually means “base-10 number system” and most readers know only that number system. However, other number systems are in common use in computing, as we shall see in the sections that follow.

## 1.2 The Binary Number System

The rules we have seen for writing base-10 numbers can be applied to numbers of any arbitrary base. In fact, it's really the other way around; the base-10 system is a specific instance of the general number system idea.

To have a number system with base  $B$ , we simply need to pick  $B$  unique symbols, assign distinct values to them, and specify their value order. Therefore, to have a base-2 (or binary) number system, we need two symbols; 0 and 1 are used where 0 is less than 1. The following table shows some numbers expressed in both base 10 and base 2:

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
100	1100100
1,000	1111101000
10,000	10011100010000
100,000	11000011010100000
1,000,000	11110100001001000000

The larger the number system base (or *radix*), the bigger the number that can be expressed with a given number of digit positions. Conversely, the smaller the radix, the more digit positions are required to express a given number. Since 2 is the smallest possible radix, it produces the most unwieldy numbers, as shown in the table above.

Numbers written in binary can get verbose, but they have one very important property: The number of rules needed to perform arithmetic operations on them is very small compared to the base-10 system. For example, in addition, we have  $0 + 0$ ,  $0 + 1$ ,  $1 + 0$ , and  $1 + 1$ , with only the last one producing a carry.

## 2. Bit Manipulation

We've learned about number systems, but we haven't yet discussed the practical applications of this knowledge. That will be the subject of this chapter.

For the purpose of this discussion, we will work with a computer that can process 8 bits at a time. Our machine will have three 8-bit registers called R1, R2, and R3. A *register* is a device containing one electronic component for each bit. That component can be set to the zero state or the one state. Therefore, our registers are capable of storing numbers in the range  $0-11111111_2$  (i.e.,  $0-255_{10}$ ). We can think of R1, R2, and R3 as being three 8-bit integer variables.

In this chapter, we will perform operations on the bits in one or two of the registers, storing the result in the third register.

### 2.1 The Bitwise AND Operator

Consider the problem of testing to see if a number is odd or even. Some languages achieve this via a modulus or remainder operator, which often involves a division operation. If we were to look at a set of numbers written in binary, we would notice the following useful property: All of the odd numbers have their least-significant bit set (i.e., it has the value 1) and all of the even numbers have their least-significant bit clear (i.e., it has the value 0). Therefore, to test if a number is even, we need only to see if the least-significant bit is clear.

Computers have instructions that can test any combination of bits in a register. The instruction used for testing bits is known as the bitwise AND; it is used as follows:

```
R3 = R1 AND R2
```

Here, the bit patterns stored in registers R1 and R2 are ANDed together and the result is stored in R3. The bit pattern of the result is such that a bit in R3 is set *only* if both corresponding bits in R1 and R2 are also set; otherwise it is clear. This is shown by the following *truth table*, where B1, B2, and B3 are corresponding bits in registers R1, R2, and R3, respectively:

Table 2–1: Bitwise AND Truth Table

B1	B2	B3
0	0	0
0	1	0
1	0	0
1	1	1

## Programming Concepts

Such a table is called a truth table since by making 0 represent false and 1 represent true, the third column shows the combined truth of the first two. For example, only if B1 is true *and* B2 is true will B3 be true.

The table shows what happens for each bit in a bit pattern. By applying this process to each bit, R3 will contain the resulting bit pattern. Let's use this information to see if the number 8 is even or odd:

$$R1 = 8 = 00001000$$

$$R2 = 1 = 00000001$$

---

$$R3 = R1 \text{ AND } R2 = 0 = 00000000$$

R1 contains 8, the number of interest. The value stored in R2 is very important; the bits set in it must correspond to the bits of R1 in which we are interested. Since we are only interested in the least-significant bit, we set R2 to 1. The bit pattern stored in R2 is often referred to as a *bitmask* or simply a *mask*; it's as if we place a mask over R1 and only those bits behind the holes in the mask show through. The result stored in R3 is zero (or false) and this indicates the value in R1, in this case 8, is even. Let's try it for the number 9:

$$R1 = 9 = 00001001$$

$$R2 = 1 = 00000001$$

---

$$R3 = R1 \text{ AND } R2 = 1 = 00000001$$

Now the result stored in R3 is nonzero (or true), which indicates the value in R1, in this case 9, is odd. A common way of expressing such operations in a program is shown in the following pseudo-code:

```
IF R1 AND 1 IS TRUE THEN
    R1 IS ODD
ELSE
    R1 IS EVEN
```

From a set theory point of view, the AND operator is used to produce the intersection of two sets.

Another interesting application of bit testing is seeing if a number is a multiple of some power of 2. Let's see if the number 36 is a multiple of 4:

$$R1 = 36 = 00100100$$

$$R2 = 4 = 00000100$$

$$R3 = R1 \text{ AND } R2 = 4 = 00000100$$

The result stored in R3 is nonzero (or true) which indicates the value in R1 is a multiple of the value stored in R2.

Let's look at another use for manipulating bits. Consider the case in which we need to store a person's sex and marital status. There are two possible codes for sex: male (0) and female (1); and four possible codes for marital status: single (0), married (1), divorced (2), and widowed (3). Most often, programmers would store these attributes in separate fields of that person's record. Let's make the (reasonable) assumption that each of these variables takes up 8 bits of storage. However, we can represent the two sex codes in one bit and the four marital codes in two bits. Therefore, we need only three bits to store this information instead of 16, a considerable saving.

The bit pattern 00000101 can be used to represent a divorced female. Here the least-significant bit represents the sex code, with a value of 1 indicating female. The adjacent two bits represent marital status with the value 10 indicating divorced. Based on this knowledge, we can easily understand the following table:

Decimal Value	Binary Value	Marital Status	Sex Code
0	00000000	single	male
1	00000001	single	female
2	00000010	married	male
3	00000011	married	female
4	00000100	divorced	male
5	00000101	divorced	female
6	00000110	widowed	male
7	00000111	widowed	female

If a person's attribute field contains the bit pattern 00000010 how would we test to see if that person is a married female? We would AND that pattern with the appropriate mask. If the result is the same as the mask used, the pattern contains the property indicated by the mask. For example:

R1 = 2    00000010

R2 = 3    00000011

R3 = R1 AND R2 = 2    00000010

Since R3 does not contain the same value of the mask stored in R2, we deduce that this person is not a married female. (In fact, the person is a married male.) The fact that the result is nonzero (true) is irrelevant.

## 3. Data Representation

For the most part it is not necessary to know how objects (such as variables) of a particular type are mapped on a given host system. However, such information is useful in helping understand why certain errors occur and, in some cases, is necessary to solve certain kinds of problems. For example, to communicate successfully between routines written in different languages, we need to know how objects of various types are mapped by each language.

### 3.1 Basic Machine Objects

The concept of *type* is a logical one imposed by a language at a level higher than machine code. However, once the compiler has done its job, definitions of variables having some type, and statements involving expressions of various types, have been turned into accesses to objects at the machine level. At this level, only a few physical “types” exist. Those most often available are bit, byte, and word. We already discussed “bit” in §1.2.

A *byte* is a group of adjacent bits treated as a unit by a computer and its peripherals. While the size of a byte can vary from one system to another, on most modern systems a byte has 8 bits.

When memory is organized in bytes, each byte has a unique identifier called its *address*. A computer that has its memory organized in this manner is said to have a *byte architecture* and to be *byte addressable*. Examples of byte-architecture machines include the PDP-11, VAX-11, and Alpha from Digital Equipment Corporation (Digital), Intel's 80x86, Motorola's 680x0, and IBM's S/360 and S/370.

A term heard now and again is *nibble*, which is half a byte (pun intended). And since bytes are most often 8 bits, nibbles are 4 bits. 4-bit nibbles correspond to the groups of 4 bits used when writing BCD values as well as when writing binary values in hexadecimal.

A *word* is a group of adjacent bits treated as a unit by a computer and its peripherals. The size of a word can vary from one system to another. For example, there are machines with word sizes of 8, 12, 16, 24, 32, 36, 48, 60, and 64 bits. Some machines (such as Intel's 80x86 and Digital's Alpha) can even operate at one of several different word sizes. The word size of a machine typically corresponds to the number of bits in each of its general-purpose registers.

When memory is organized by words instead of bytes, each word has a unique address. A computer that has its memory organized in such a manner is said to have a *word architecture* and to be *word addressable*. Examples of word-architecture machines include Digital's PDP-10 (36-bit word), CDC's 6000 and Cyber series (60-bit word), and Cray Research's supercomputers (64-bit word).

On a word-addressable machine, multiple characters are typically packed into a single word. While these characters can be referred to as bytes, they are not themselves addressable—on these machines, the address of a byte is the address of its parent word plus some offset into that word.

Few machines allow individual bits to be addressed. Machines that do are said to be *bit addressable*.

## Programming Concepts

Although raw memory consists of bits, bytes, and words, many machine instructions either impose or expect there to exist some type structure built on top of these primitives. For example, a “compare two double-precision floating-point values” instruction involves comparing two sets of bytes or words based on their arithmetic values, not just on raw bit patterns. Similarly, an “increment double-precision floating-point value by 1” instruction causes 1 to be added to a multiple-byte or multiple-word object having some arithmetic structure.

In the sections that follow, we will examine various ways in which these higher-level types are mapped onto the basic types of bit, byte, and word. Note, however, that the discussion on mapping is not exhaustive, as each new hardware or language designer may provide new ways of representing types.

A number of other terms are used to describe whole words, parts of words, or multiple words. Examples are *halfword*, *longword*, *doubleword*, *quadword*, and *octaword*. However, these terms are used differently (if at all) by different vendors so they will not be discussed further here.

**Exercise 3-1:** Find out if your machine is byte- or word-addressable. What is the machine's word size? Can it operate in different word-size modes?

## 3.2 Object Alignment

Some machines require, or can benefit from, *object alignment*. That is, they require, or prefer, to have objects of certain types aligned on particular address boundaries. Prior to the late 1970s, most machines required objects of pretty much every type except individual characters and character strings, to have an address that was a multiple of at least two, and for some types, a multiple of four or even eight. (Examples include Digital's PDP-11 and IBM's S/360.)

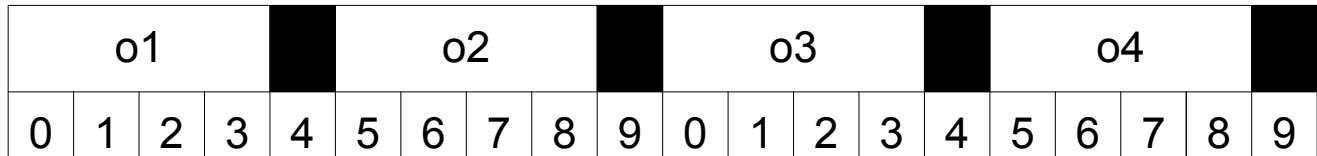
Throughout the 1980s, new mainstream CPUs were more flexible, for the most part allowing any object to be aligned anywhere in memory. However, optimizing compilers were careful to align objects where possible, since doing so improved memory access times. (Examples include Digital's VAX-11, Intel's 80x86, Motorola's 680x0, and IBM's 370 series.)

Late in the 1980s, a new breed of machines became popular. These were based on RISC architectures and inherent in their design is the requirement that certain object alignment requirements must be met. (Examples include Sun's SPARC, MIPS' Rx000 series, IBM's RS/6000, and the PowerPC.)

For the most part, a high-level language programmer need not care about object alignment. However, some knowledge of this subject can be helpful, particularly in debugging *memory access violations*. A memory access violation occurs when a program attempts to access a location in memory for which it has no privilege or which is not part of the current program. It can also result from an attempt to access an unaligned object on a machine that requires alignment.

In the following picture, o1, o2, o3, and o4 are 4-byte objects located at the relative addresses shown. If they each require (or can benefit from) 4-byte alignment then only o1 is aligned on an address that is a multiple of 4.

Figure 1: Alignment Considerations for 4-byte Objects



**Exercise 3-2\*:** Does your machine have any object alignment requirements? If so, what are they? If not, can it benefit from certain object alignment anyway and do your compilers give you the option of specifying alignment?

### 3.3 Characters

As stated earlier, on most systems, a byte contains 8 bits, allowing for 256 distinct values. Since the character sets used by Western European-based cultures have less than 256 characters, each of their characters can be represented in a single byte. As a result, the term *character* has become synonymous with byte. However, to represent Chinese, Japanese, and Korean characters, for example, at least two bytes are required.

The representation of large character sets is outside the scope of this article and will not be discussed further here. Suffice it to say that a new coding system is taking over from ASCII and EBCDIC. It is called *Unicode* and it uses a 16-bit character, allowing for 65,536 different characters.<sup>1</sup> As such, it accommodates the world's commercially and academically significant writing systems. Java was the first mainstream programming language to require Unicode. C# and the other languages that support Microsoft's .NET also support it.

When it comes right down to it, inside a computer, everything is represented as a bit pattern. And since bit patterns consist of binary digits, everything is represented as a number. However, for the most part, programmers tend not to think of characters as numbers, and they get quite confused when they hear about arithmetic being performed on characters.

Every character in a character set has a unique internal integer value assigned to it. For example, in the ASCII character set, the letter 'A' has value 65, the digit '0' has value 48, and the punctuation mark '!' has value 33.

Consider the internal representation of the characters '@', 'A', and 'B', in the U.S. variant of the ASCII character set. (The bits are numbered 0–7 starting from the right-most bit, which is the least-significant.)

Table 3–1: Some ASCII Characters

Character	Decimal	Binary
@	64	01000000
A	65	01000001
B	66	01000010

<sup>1</sup> In Version 3.1, Unicode was extended to allow up to a million characters.