**2 Representing and Manipulating Information**

✨整数的计算机运算满足真正整数运算的许多性质，比如结合律、交换律。

浮点数的计算机运算由于表示的精度有限，浮点运算是不可结合的

比如：在大多数机器上，C表达式(3.14+1e20)-1e20求得的值会是0.0，而3.14+(1e20-1e20)求得的值会是3.14。

造成整数和浮点数具有不同的数学属性的原因是它们处理数字表示有限性的方式不同——整数虽然表示范围小但表示精确，浮点数虽然表示范围大但表示近似

# 英语

|  |  |
| --- | --- |
| 1. conceptual概念性的 2. subsequent随后的 3. associate A with B将A和B关联 4. neither两者都不 5. verbose冗长的 6. vagaries变化无常 7. portable便捷的 8. vice versa反之亦然 9. circumvent规避 10. peculiar奇特的 | 1. monolithic整体式的 2. partition 划分(v)、部分(n) 3. notation表示法 4. convenient方便的 5. tedious繁琐的 6. strive to努力 7. virtually几乎 8. inspect检查 9. formulate制定 |

# 2.1 *Information Storage*

大多数计算机使用 8 位（字节）作为最小的可寻址内存单元

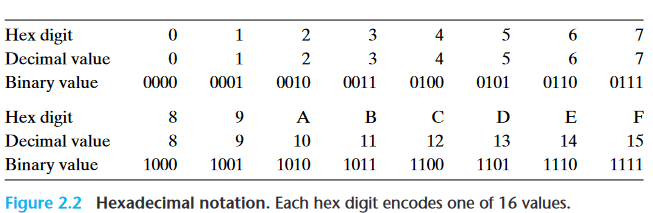
机器语言程序将内存视为一个非常大的字节数组，称为虚拟内存（逻辑上，实际上是用DRAM芯片实现的）。虚拟内存中的每个字节都唯一由一个数值标识，这个数值称为地址；所有可能的地址的集合称为虚拟地址空间

**|** C语言中指针的作用：

指针是 C 的核心功能。它们提供了引用数据结构（包括数组）元素的机制。就像变量一样，指针有两个方面：它的值和它的类型。值指示某个对象的位置，而它的类型指示该位置存储什么类型的对象（例如整数或浮点数）——起始位置开始占多少字节。

## 2.1.1 *Hexadecimal Notation*

因为二进制和十进制描述位信息的缺点注释1，引入了十六进制表示法来描述位信息。下图是二进制、十进制、十六进制相关联的转换表



在 C语言 中，以 0x 或 0X 开头的数字常量被解释为十六进制，十六进制中的“A→F”既可以写成大写也可以写成小写。

使用机器级程序的一项常见任务是手动在位模式的十进制、二进制和十六进制表示形式之间进行转换

二进制与十六进制的相互转换：从二进制最低位起往高位发展，每4位二进制转换为一位十六进制（不足4位的二进制补前导零）。从十六进制最低位起往高位发展，每位十六进制可以转换为4位二进制

二进制与十进制之间的相互转换：以二进制小数点为分界线，小数点左边依次对应位值乘以2的对应位次幂(0、1、2、3……)结果相加；小数点右边依次对应位值乘以2的对应位次幂(-1、-2、-3……)结果相加。以十进制小数点为分界线，小数点左侧整数部分除2倒取余，小数点右侧乘2正取整。

✨当x是2的n次幂，n=4i+j，则x的十六进制为(j=0→1,j=1→2,j=2→4,j=3→8)然后加i个0

十进制与十六进制的相互转换：用二进制做媒介或者除16倒取余、乘16正取整。

## 2.1.2 Data Sizes

每台计算机都有一个字大小，指示指针数据的标称大小。指针数据的标称大小是指一个指针变量在内存中所占用的字节数，不与指针变量的类型有关，指针变量的类型只决定指针指向的数据占内存多少字节。

而由于虚拟地址正是以这样一个字编码的，所以字长决定的最重要的系统参数是虚拟地址空间的最大大小——假设字长w位，那么虚拟地址空间的范围是，程序最多可以访问个字节

大多数64位机器也可以运行为32位机器编译的程序，这是一种向后兼容。因此，举例来说，当程序prog.c用gcc -m32 prog.c伪指令编译后，该prog程序既可以在32位机器也可在64位机器上运行；而若使用gcc -m64 prog.c伪指令编译后，该prog程序只能在64位机器上运行——**因此区分32位程序和64位程序主要是看该程序是如何编译的而不是其运行在什么机器类型上**

计算机和编译器支持多种不同方式编码的数字格式，如不同长度的整数和浮点数。下面是C语言中支持的各种数据类型以及其字节长度：

|  |  |
| --- | --- |
|  | 有些数据类型的确切字节数取决于该程序是如何被编译的——32位or64位  注意int32\_t、int64\_t  补充：C语言对类型声明的关键字顺序和省略可选int来说允许存在多种形式，比如  unsigned long、unsigned long int、long unsigned、long unsigned int均表示一个意思 |

✨使用固定大小的整数类型是程序员准确控制数据表示的最佳途径——为了避免由于依赖典型大小和不同编译器设置带来的数据类型大小的变化无常，ISO C99使用了一种大小固定，不随编译器和机器设置变化的数据类型，其中就有int32\_t,int64\_t

**大多数数据类型都采用有符号值进行编码**，除非以关键字 unsigned 为前缀或使用固定大小数据类型的特定无符号声明注释2。但是**char是一个例外**，虽然大多数编译器和机器将char对待为有符号数据，但是标准C并不保证一定会这么对待。一般来说如上图中的标准C声明所示，**将char声明为signed char确保其按照一字节有符号数编码**——幸运的是一般程序对char有无符号并不敏感

**|** **C语言标准对不同数据类型的数字范围设置了下界，但是却没有上界**

## 2.1.3 *Addressing and Byte Ordering*

对于跨越多个字节的程序对象，我们必须建立两个约定：**对象的地址是什么，以及我们如何对内存中的字节进行排序。**

1. 多字节对象的地址是什么？

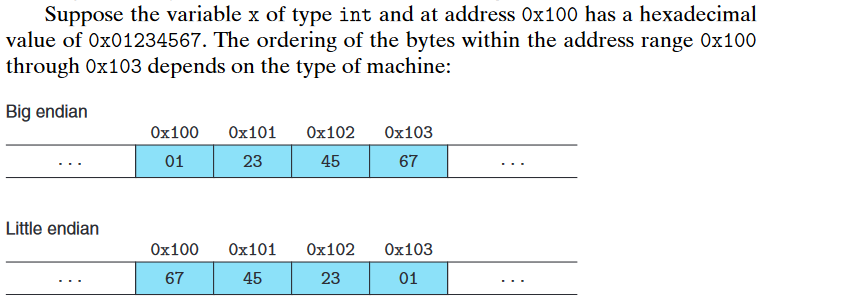
实际上，在所有机器中，多字节对象都存储为连续的字节序列，**对象的地址由所使用字节的最小地址给出。**

**|** *For example, suppose a variable x of type int has address 0x100; that is, the value of the address expression &x is 0x100. Then (assuming data type int has a 32-bit representation) the 4 bytes of x would be stored in memory locations 0x100, 0x101, 0x102, and 0x103.*

1. 内存中字节数据是怎么放置的——大小端

大端法：数据的高位放置在内存的低位，数据的低位放置在内存的高位

小端法：数据的高位放置在内存的高位，数据的低位放置在内存的低位



**|** **大多数Intel兼容机均使用小端模式，大多数IBM、Oracle使用的是大端模式**

✨目前许多比较新的微处理器是双端法(bi-endian)，可以把它们配置成大端或者小端的机器运行。但是实际情况是，一旦选择了操作系统那么字节的顺序也就固定了下来——比如ARM是采用双端法，但是其上最常见的两种操作系统Android、IOS均只支持小端

大多数情况不需要考虑字节顺序均可以得到相同的结果，但是有时候字节顺序会成为问题：

* + 1. 不同机器之间通过网络传递二进制信息

常见的问题是：大端发送的数据在小端接收、小端发送的数据在大端接收

正确的做法是：发送端发送的数据先将内部的数据顺序转换为网络标准，接收端再将网络标准转换为内部的数据顺序

* + 1. 查看表示整数数据的字节序列

当检查机器级程序代码时，可能会涉及到查看表示整数数据的字节序列，如下所示：



在4004d3地址处的“01 05 43 0b 20 00 低到高”指令，经过反汇编工具得到该机器指令表示的意义是“x86指令 add %eax,0x200b43(%rip)”，指令的含义是采用相对寄存器寻址方式，将内存地址(%rip+0x200b43)上的值与%eax寄存器值相加写回%eax，而这里如果采用小端，那么是%rip+0x00200b43，如果采用大端那么是%rip+0x430b2000

* + 1. 程序规避正常系统的类型

在C语言中，这种操作可以通过强制类型转换cast和联合union来实现

**使用这种方法可以判断当前的机器是小端还是大端**

#include "stdio.h"  
union {  
 char c;  
 int b  
}example;  
int main(){  
 int a=0x0128;//0x0 0 0 8  
 printf("%x\n",\*((char \*)&a));  
 example.b=0x1234;  
 printf("%x",example.c);  
}

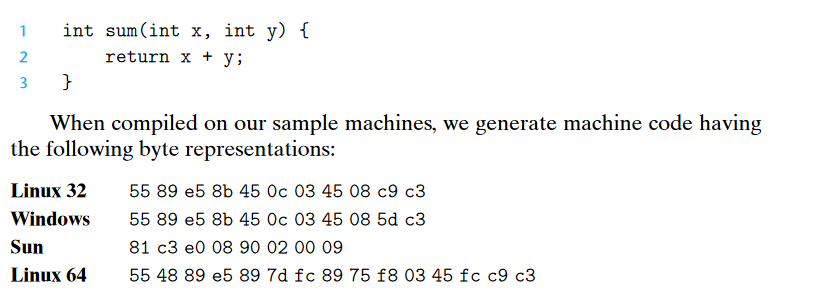
C

## 2.1.4 *Representing Strings*

**C 中的字符串由以空字符'\0'（ASCII值为 0）结尾的字符数组编码。**也正因为采用ASCII编码的字符只占一个字节，不需要考虑字节顺序。所以任何只要使用ASCII作为其字符编码的系统对于同样一个采用ASCII编码的文本文件都可以得到同样的结果——T*ext data are more platform independent than binary data.*

## 2.1.5 *Representing Code*

以下面的sum C子程序为例，可以看到指令编码是不同的

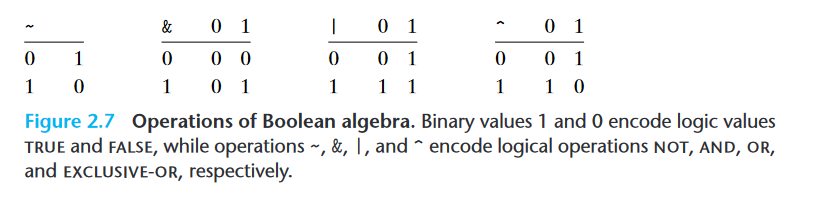


即使是同样的程序，运行在不同的机器类型/操作系统上也使用不同的且不兼容的指令和编码方式——二进制代码是不兼容的

✨计算机系统的一个基本概念是，**从机器的角度来看，程序只是一个字节序列**。机器没有关于原始源程序的信息，除了一些为了帮助调试而维护的辅助表之外。

## 2.1.6 *Introduction to Boolean Algebra*

**|** *Since binary values are at the core of how computers encode, store, and manipulate information, a rich body of mathematical knowledge has evolved around the study of the values 0 and 1*

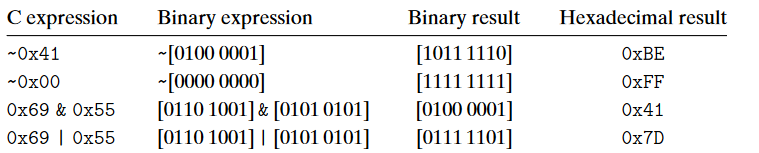


也可以将上述的按位运算拓展到位向量的运算，原理仍是向量对应位按位运算

位向量的一个很有用的应用是表示有限集合。我们可以用位向量 编码任何子集此时|运算为两集合的并、&运算为两集合的交

## 2.1.7 *Bit-Level Operations in C*

之前提到的"|、&、~、^"均可用到C语言的任何整数类型上，一般是将整数值转换为二进制来执行对应的布尔运算，最后再将二进制结果转变为十六进制，如下所示：



使用异或运算，我们可以实现一种更具有空间优势（不需要第三个中间变量来暂存值）的交换两个变量值的技术：

void swap(int \*x,int \*y){  
 \*y=\*x^\*y;  
 \*x=\*x^\*y;  
 \*y=\*x^\*y;  
}

C

其执行过程如下表所示（a^a=0,0^a=a)：

|  |  |  |
| --- | --- | --- |
| Step | \*x | \*y |
| Initial | a | b |
| Step1 | a | a^b |
| Step2 | b | a^b |
| Step3 | b | b |

位级操作的一个常见用途是实现掩码操作，其中掩码是指示字内选定的一组位的位模式。例如，掩码0xFF是选中一个字的最低8位。考虑到可移植性，一般使用~0来生成位全1的字掩码，而不是使用32位的0xffffffff

## 2.1.8 *Logical Operations in C*

C 还提供了一组逻辑运算符 ||、&& 和 !，分别对应于逻辑的 or、and 和 not 操作。逻辑运算的操作和位操作截然不同：1. 逻辑操作对待任何非0数据为True，对待0为False。下面是一些逻辑表达式的例子：

|  |  |
| --- | --- |
|  | 仅在运算数据为0或1时，位运算和逻辑运算结果等价   1. 逻辑运算的&&当第一个数据为0时，就不再计算第二个数据；||当第一个数据为1时，就不再计算第二个数据 |

✨x==y等价于!(x^y)

## 2.1.9 *Shift Operations in C*

C 还提供了一组移位操作，用于将位模式左移和右移

### 2.1.9.1 Left shift

对于位表示形式为的操作数x，x<<k表示即x的二进制序列左移k位，低位补0，丢失k位高位有效值

移位操作是从左往右结合的，所以x<<j<<k等价于(x<<j)<<k

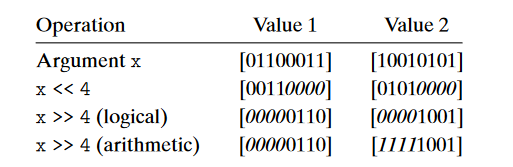
### 2.1.9.2 Right shift

右移操作较左移有微妙的差别，可以分为两种右移操作：

逻辑右移：同左移的处理不足位机制，x>>k表示将x的二进制序列右移k位，不足的高位补0，丢失k位低位

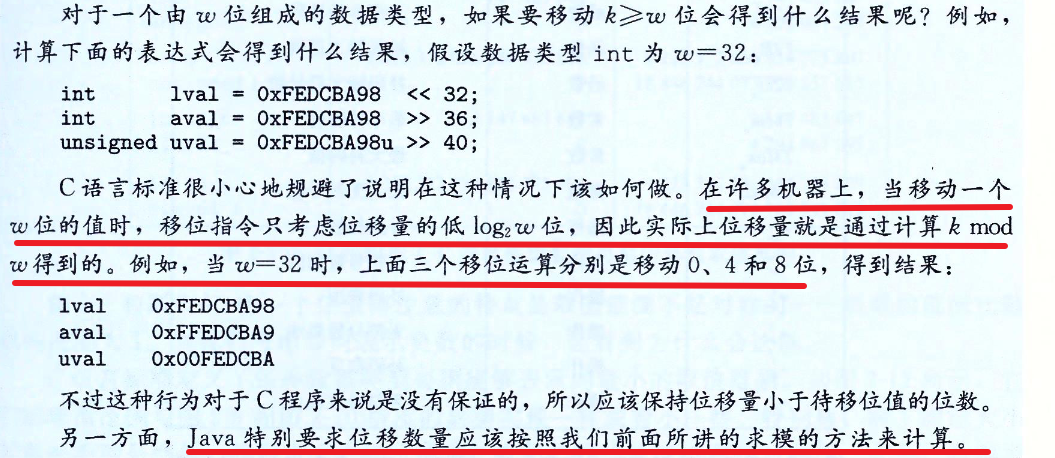
算术右移：不足的高位根据原来操作数的符号决定补0还是1

移位示例如下所示：



✨C 标准没有精确定义哪种类型的右移应与有符号数一起使用——可以使用算术移位或逻辑移位注释3。然而，在实践中，几乎所有编译器/机器组合都对**有符号数据使用算术右移**，并且许多程序员认为情况就是如此。另一方面，对于**无符号数据，右移必须是逻辑的。**

✨若k很大时移动k位，C语言很小心的规避了这种情况，实际上的位移量是k%w(w为数据宽度)。但是这种移位法C语言并不保证，因此应该保证位移量小于数据位宽。但Java是特别要求按照k%w移位的



# 2.2 *Integer Representations*

[注释1] 二进制表示法过于冗长，而使用十进制表示法时，与位模式之间的转换非常繁琐。

[注释2] 如uint32\_t,uint64\_t

[注释3] 与 C 不同，Java 对如何执行右移有精确的定义。表达式 x>>k 将 x 算术移位 k 个位置，而 x >>> k 对其进行逻辑移位。