程序设计基础与实验

关于浮点数

刘新国

浙江大学计算机学院
浙江大学 CAD&CG 国家重点实验室

November 10, 2021

内容提要

数据类型

浮点数表示格式

位运算

解析浮点数

运算符和优先级

内容提要

内容提要

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）
- 然后将小数部分 0.625 转化为二进制：（乘以 2 取整数）

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）
- 然后将小数部分 0.625 转化为二进制：（乘以 2 取整数）
  - $0.625 * 2 = 1.25$, 将整数部分 1 添加到小数后面，得到 110.1

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）
- 然后将小数部分 0.625 转化为二进制：（乘以 2 取整数）
  - $0.625 * 2 = 1.25$, 将整数部分 1 添加到小数后面，得到 110.1
  - $0.25 * 2 = 0.5$, 将整数部分 0 添加到小数后面，得到 110.10

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）
- 然后将小数部分 0.625 转化为二进制：（乘以 2 取整数）
  - $0.625 * 2 = 1.25$, 将整数部分 1 添加到小数后面，得到 110.1
  - $0.25 * 2 = 0.5$, 将整数部分 0 添加到小数后面，得到 110.10
  - $0.5 * 2 = 1.0$, 将整数部分 1 添加到小数后面，得到 110.101

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）
- 然后将小数部分 0.625 转化为二进制：（乘以 2 取整数）
  - $0.625 * 2 = 1.25$, 将整数部分 1 添加到小数后面，得到 110.1
  - $0.25 * 2 = 0.5$, 将整数部分 0 添加到小数后面，得到 110.10
  - $0.5 * 2 = 1.0$, 将整数部分 1 添加到小数后面，得到 110.101
  - 小数部分为 0，结束。

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）
- 然后将小数部分 0.625 转化为二进制：（乘以 2 取整数）
  - $0.625 * 2 = 1.25$, 将整数部分 1 添加到小数后面，得到 110.1
  - $0.25 * 2 = 0.5$, 将整数部分 0 添加到小数后面，得到 110.10
  - $0.5 * 2 = 1.0$, 将整数部分 1 添加到小数后面，得到 110.101
  - 小数部分为 0，结束。

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）
- 然后将小数部分 0.625 转化为二进制：（乘以 2 取整数）
  - $0.625 * 2 = 1.25$, 将整数部分 1 添加到小数后面，得到 110.1
  - $0.25 * 2 = 0.5$, 将整数部分 0 添加到小数后面，得到 110.10
  - $0.5 * 2 = 1.0$, 将整数部分 1 添加到小数后面，得到 110.101
  - 小数部分为 0，结束。

将 0.7 转化成二进制

- $0.7 * 2 = 1.4$, 将整数部分 1 添加到小数后面，得到 0.1

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）
- 然后将小数部分 0.625 转化为二进制：（乘以 2 取整数）
  - $0.625 * 2 = 1.25$, 将整数部分 1 添加到小数后面，得到 110.1
  - $0.25 * 2 = 0.5$, 将整数部分 0 添加到小数后面，得到 110.10
  - $0.5 * 2 = 1.0$, 将整数部分 1 添加到小数后面，得到 110.101
  - 小数部分为 0，结束。

将 0.7 转化成二进制

- $0.7 * 2 = 1.4$, 将整数部分 1 添加到小数后面，得到 0.1
- $0.4 * 2 = 0.8$, 将整数部分 0 添加到小数后面，得到 0.10

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）
- 然后将小数部分 0.625 转化为二进制：（乘以 2 取整数）
  - $0.625 * 2 = 1.25$, 将整数部分 1 添加到小数后面，得到 110.1
  - $0.25 * 2 = 0.5$, 将整数部分 0 添加到小数后面，得到 110.10
  - $0.5 * 2 = 1.0$, 将整数部分 1 添加到小数后面，得到 110.101
  - 小数部分为 0，结束。

将 0.7 转化成二进制

- $0.7 * 2 = 1.4$, 将整数部分 1 添加到小数后面，得到 0.1
- $0.4 * 2 = 0.8$, 将整数部分 0 添加到小数后面，得到 0.10
- $0.8 * 2 = 1.6$, 将整数部分 1 添加到小数后面，得到 0.101

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）
- 然后将小数部分 0.625 转化为二进制：（乘以 2 取整数）
  - $0.625 * 2 = 1.25$, 将整数部分 1 添加到小数后面，得到 110.1
  - $0.25 * 2 = 0.5$, 将整数部分 0 添加到小数后面，得到 110.10
  - $0.5 * 2 = 1.0$, 将整数部分 1 添加到小数后面，得到 110.101
  - 小数部分为 0，结束。

将 0.7 转化成二进制

- $0.7 * 2 = 1.4$, 将整数部分 1 添加到小数后面，得到 0.1
- $0.4 * 2 = 0.8$, 将整数部分 0 添加到小数后面，得到 0.10
- $0.8 * 2 = 1.6$, 将整数部分 1 添加到小数后面，得到 0.101
- $0.6 * 2 = 1.2$, 将整数部分 1 添加到小数后面，得到 0.1011

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）
- 然后将小数部分 0.625 转化为二进制：（乘以 2 取整数）
  - $0.625 * 2 = 1.25$, 将整数部分 1 添加到小数后面，得到 110.1
  - $0.25 * 2 = 0.5$, 将整数部分 0 添加到小数后面，得到 110.10
  - $0.5 * 2 = 1.0$, 将整数部分 1 添加到小数后面，得到 110.101
  - 小数部分为 0，结束。

将 0.7 转化成二进制

- $0.7 * 2 = 1.4$, 将整数部分 1 添加到小数后面，得到 0.1
- $0.4 * 2 = 0.8$, 将整数部分 0 添加到小数后面，得到 0.10
- $0.8 * 2 = 1.6$, 将整数部分 1 添加到小数后面，得到 0.101
- $0.6 * 2 = 1.2$, 将整数部分 1 添加到小数后面，得到 0.1011
- $0.2 * 2 = 0.4$, 将整数部分 0 添加到小数后面，得到 0.10110

二进制的小数

将 6.625 转化成二进制

- 首先将整数部分转化成二进制：$6 = 110$ （除以 2 取余数）
- 然后将小数部分 0.625 转化为二进制：（乘以 2 取整数）
  - $0.625 * 2 = 1.25$, 将整数部分 1 添加到小数后面，得到 110.1
  - $0.25 * 2 = 0.5$, 将整数部分 0 添加到小数后面，得到 110.10
  - $0.5 * 2 = 1.0$, 将整数部分 1 添加到小数后面，得到 110.101
  - 小数部分为 0，结束。

将 0.7 转化成二进制

- $0.7 * 2 = 1.4$, 将整数部分 1 添加到小数后面，得到 0.1
- $0.4 * 2 = 0.8$, 将整数部分 0 添加到小数后面，得到 0.10
- $0.8 * 2 = 1.6$, 将整数部分 1 添加到小数后面，得到 0.101
- $0.6 * 2 = 1.2$, 将整数部分 1 添加到小数后面，得到 0.1011
- $0.2 * 2 = 0.4$, 将整数部分 0 添加到小数后面，得到 0.10110
- 一直进行下去，直到小数为 0，或者达到指定的位数。

实数类型的表示

S	E	M
符号	阶码	尾数
1	8/11	23/52

实数类型的表示

S	E	M
符号	阶码	尾数
1	8/11	23/52

IEEE 浮点数表示：$S \times 2^E \times 1.M$ 或者 $S \times 2^E \times 0.M$

- 符号：1 个 bit，0–正数，1–负数

实数类型的表示

S	E	M

符号	阶码	尾数
1	8/11	23/52

IEEE 浮点数表示：$S \times 2^E \times 1.M$ 或者 $S \times 2^E \times 0.M$

- 符号：1 个 bit，0–正数，1–负数
- 阶码：浮点数的指数，采用移码表示法（无符号的指数）

实数类型的表示

S	E	M
符号	阶码	尾数
1	8/11	23/52

IEEE 浮点数表示：$S \times 2^E \times 1.M$ 或者 $S \times 2^E \times 0.M$

- 符号：1 个 bit，0–正数，1–负数
- 阶码：浮点数的指数，采用移码表示法（无符号的指数）
  - float 浮点数：8 bits，指数 = 阶码 - 127

实数类型的表示

S	E	M
符号	阶码	尾数
1	8/11	23/52

IEEE 浮点数表示：$S \times 2^E \times 1.M$ 或者 $S \times 2^E \times 0.M$

- 符号：1 个 bit，0–正数，1–负数
- 阶码：浮点数的指数，采用移码表示法（无符号的指数）
  - float 浮点数：8 bits，指数 = 阶码 - 127
    $E_{min} = -127, E_{max} = 128$

实数类型的表示

S	E	M
符号	阶码	尾数
1	8/11	23/52

IEEE 浮点数表示：$S \times 2^E \times 1.M$ 或者 $S \times 2^E \times 0.M$

- 符号：1 个 bit，0–正数，1–负数
- 阶码：浮点数的指数，采用移码表示法（无符号的指数）
  - float 浮点数：8 bits，指数 = 阶码 - 127
    $E_{min} = -127, E_{max} = 128$
  - double 浮点数：11 bits，指数 = 阶码 - 1023

实数类型的表示

S	E	M
符号	阶码	尾数
1	8/11	23/52

IEEE 浮点数表示：$S \times 2^E \times 1.M$ 或者 $S \times 2^E \times 0.M$

- 符号：1 个 bit，0–正数，1–负数
- 阶码：浮点数的指数，采用移码表示法（无符号的指数）
  - float 浮点数：8 bits，指数 = 阶码 - 127
    $E_{min} = -127, E_{max} = 128$
  - double 浮点数：11 bits，指数 = 阶码 - 1023
    $E_{min} = -1023, E_{max} = 1024$

实数类型的表示

S	E	M
符号	阶码	尾数
1	8/11	23/52

IEEE 浮点数表示：$S \times 2^E \times 1.M$ 或者 $S \times 2^E \times 0.M$

- 符号：1 个 bit，0–正数，1–负数
- 阶码：浮点数的指数，采用移码表示法（无符号的指数）
  - float 浮点数：8 bits，指数 = 阶码 - 127
    $E_{min} = -127, E_{max} = 128$
  - double 浮点数：11 bits，指数 = 阶码 - 1023
    $E_{min} = -1023, E_{max} = 1024$
- 尾数：二进制的小数

实数类型的表示

S	E	M
符号	阶码	尾数
1	8/11	23/52

IEEE 浮点数表示：$S \times 2^E \times 1.M$ 或者 $S \times 2^E \times 0.M$

- 符号：1 个 bit，0–正数，1–负数
- 阶码：浮点数的指数，采用移码表示法（无符号的指数）
  - float 浮点数：8 bits，指数 = 阶码 - 127
    $E_{min} = -127, E_{max} = 128$
  - double 浮点数：11 bits，指数 = 阶码 - 1023
    $E_{min} = -1023, E_{max} = 1024$
- 尾数：二进制的小数
  - 尾数 = 1.M，当指数段不全为 0 （格式化值）

实数类型的表示

S	E	M
符号	阶码	尾数
1	8/11	23/52

IEEE 浮点数表示：$S \times 2^E \times 1.M$ 或者 $S \times 2^E \times 0.M$

- 符号：1 个 bit，0–正数，1–负数
- 阶码：浮点数的指数，采用移码表示法（无符号的指数）
  - float 浮点数：8 bits，指数 = 阶码 - 127
    $E_{min} = -127, E_{max} = 128$
  - double 浮点数：11 bits，指数 = 阶码 - 1023
    $E_{min} = -1023, E_{max} = 1024$
- 尾数：二进制的小数
  - 尾数 = 1.M，当指数段不全为 0 （格式化值）
  - 尾数 = 0.M，当指数段全为 0 （非格式化值）

特殊的浮点数

指数段	尾数段	数值	注解
有 0 有 1	–	$1.M \times 2^E$	一般情况
全 0	全 0	0	符号位除外，所有 bit 为 0
全 0	不全为 0	$0.M \times 2^{E_{min}}$	非常接近 0 的小数
全 1	全 0	∞	无穷大
全 1	不全为 0	NaN	(Not any Number) 非数值

详细可参考 IEEE_754 浮点数标准

- float 具有 4 个字节（32 个 bit）

特殊的浮点数

指数段	尾数段	数值	注解
有 0 有 1	–	$1.M \times 2^E$	一般情况
全 0	全 0	0	符号位除外，所有 bit 为 0
全 0	不全为 0	$0.M \times 2^{E_{min}}$	非常接近 0 的小数
全 1	全 0	∞	无穷大
全 1	不全为 0	NaN	(Not any Number) 非数值

详细可参考 IEEE_754 浮点数标准

- float 具有 4 个字节（32 个 bit）
- double 具有 8 个字节（64 个 bit）

特殊的浮点数

指数段	尾数段	数值	注解
有 0 有 1	–	$1.M \times 2^E$	一般情况
全 0	全 0	0	符号位除外，所有 bit 为 0
全 0	不全为 0	$0.M \times 2^{E_{min}}$	非常接近 0 的小数
全 1	全 0	∞	无穷大
全 1	不全为 0	NaN	(Not any Number) 非数值

详细可参考 IEEE_754 浮点数标准

- float 具有 4 个字节（32 个 bit）
- double 具有 8 个字节（64 个 bit）
- double 具有更高的精度和更大的表示范围

实数类型常量

float

- float 只具有 7~8 个有效数字

double

实数类型常量

float

- float 只具有 7~8 个有效数字
- float 的表示范围约：$\pm(10^{-38} \sim 10^{38})$

double

实数类型常量

float

- float 只具有 7~8 个有效数字
- float 的表示范围约：$\pm(10^{-38} \sim 10^{38})$

double

- double 具有 15~16 个有效数字

实数类型常量

float

- float 只具有 7~8 个有效数字
- float 的表示范围约：$\pm(10^{-38} \sim 10^{38})$

double

- double 具有 15~16 个有效数字
- double 的表示范围约：$\pm(10^{-308} \sim 10^{308})$

数值精度和取值范围

数值精度和取值范围是两个不同的概念

- float x = 1234567.89;

数值精度和取值范围

数值精度和取值范围是两个不同的概念

- float x = 1234567.89;
  - 虽在取值范围内，但无法精确表达。

数值精度和取值范围

数值精度和取值范围是两个不同的概念

- float x = 1234567.89;
  - 虽在取值范围内，但无法精确表达。
  - 因为 float 只有 7 个有效数字

数值精度和取值范围

数值精度和取值范围是两个不同的概念

- float x = 1234567.89;
  - 虽在取值范围内，但无法精确表达。
  - 因为 float 只有 7 个有效数字
- float y = 1.2e55;

数值精度和取值范围

数值精度和取值范围是两个不同的概念

- float x = 1234567.89;
  - 虽在取值范围内，但无法精确表达。
  - 因为 float 只有 7 个有效数字
- float y = 1.2e55;
  - y 的精度要求不高，但超出取值范围。

数值精度和取值范围

数值精度和取值范围是两个不同的概念

- float x = 1234567.89;
  - 虽在取值范围内，但无法精确表达。
  - 因为 float 只有 7 个有效数字
- float y = 1.2e55;
  - y 的精度要求不高，但超出取值范围。
  - 因为 float 最大数约为 10^{38}

数值精度和取值范围

数值精度和取值范围是两个不同的概念

- float x = 1234567.89;
  - 虽在取值范围内，但无法精确表达。
  - 因为 float 只有 7 个有效数字
- float y = 1.2e55;
  - y 的精度要求不高，但超出取值范围。
  - 因为 float 最大数约为 10^{38}
- **并非所有实数都能在计算机中精确表示**

数值精度和取值范围

数值精度和取值范围是两个不同的概念

- float x = 1234567.89;
  - 虽在取值范围内，但无法精确表达。
  - 因为 float 只有 7 个有效数字
- float y = 1.2e55;
  - y 的精度要求不高，但超出取值范围。
  - 因为 float 最大数约为 10^{38}
- 并非所有实数都能在计算机中精确表示
  - 最多表示 2^{32} 个 float

数值精度和取值范围

数值精度和取值范围是两个不同的概念

- float x = 1234567.89;
  - 虽在取值范围内，但无法精确表达。
  - 因为 float 只有 7 个有效数字
- float y = 1.2e55;
  - y 的精度要求不高，但超出取值范围。
  - 因为 float 最大数约为 10^{38}
- 并非所有实数都能在计算机中精确表示
  - 最多表示 2^{32} 个 float
  - 最多表示 2^{64} 个 double

实数常量的表示

- 普通表示：符号 + 整数部分 + 小数点 + 小数部分

实数常量的表示

- 普通表示：符号 + 整数部分 + 小数点 + 小数部分
  - 例如：-12345.678

实数常量的表示

- 普通表示：符号 + 整数部分 + 小数点 + 小数部分
  - 例如：-12345.678
- 科学计数法表示：符号 + 尾数 e/E 指数

实数常量的表示

- 普通表示：符号 + 整数部分 + 小数点 + 小数部分
  - 例如：-12345.678
- 科学计数法表示：符号 + 尾数 e/E 指数
  - 例如：$-1.2345678E5$

实数常量的表示

- 普通表示：符号 + 整数部分 + 小数点 + 小数部分
  - 例如：-12345.678
- 科学计数法表示：符号 + 尾数 e/E 指数
  - 例如：$-1.2345678E5$
  - 一般表示数值很大，或者很小的数。

实数常量的表示

- 普通表示：符号 + 整数部分 + 小数点 + 小数部分
  - 例如：-12345.678
- 科学计数法表示：符号 + 尾数 e/E 指数
  - 例如：$-1.2345678E5$
  - 一般表示数值很大，或者很小的数。
    - 例如普朗克常量 6.026×10^{-27} 可表示为：6.026E-27，或者 60.26E-28

实数常量的表示

- 普通表示：符号 + 整数部分 + 小数点 + 小数部分
  - 例如：-12345.678
- 科学计数法表示：符号 + 尾数 e/E 指数
  - 例如：$-1.2345678E5$
  - 一般表示数值很大，或者很小的数。
    - 例如普朗克常量 6.026×10^{-27} 可表示为：6.026E-27，或者 60.26E-28
- 实型常量的类型都是 double

实数常量的表示

- 普通表示：符号 + 整数部分 + 小数点 + 小数部分
  - 例如：-12345.678
- 科学计数法表示：符号 + 尾数 e/E 指数
  - 例如：$-1.2345678E5$
  - 一般表示数值很大，或者很小的数。
    - 例如普朗克常量 6.026×10^{-27} 可表示为：6.026E-27，或者 60.26E-28
- 实型常量的类型都是 double
- 如需表示 float 类型常量，可用 f 作为后缀。

实数常量的表示

- 普通表示：符号 + 整数部分 + 小数点 + 小数部分
  - 例如：-12345.678
- 科学计数法表示：符号 + 尾数 e/E 指数
  - 例如：$-1.2345678E5$
  - 一般表示数值很大，或者很小的数。
    - 例如普朗克常量 6.026×10^{-27} 可表示为：6.026E-27，或者 60.26E-28
- 实型常量的类型都是 double
- 如需表示 float 类型常量，可用 f 作为后缀。
  - 例如：3.14f, 5f

内容提要

位运算

位运算是定义在整数类型数据上的一种运算，包括：

位逻辑运算 – 在数据的每一位 bit 上独立进行运算

位运算

位运算是定义在整数类型数据上的一种运算，包括：

位逻辑运算 – 在数据的每一位 bit 上独立进行运算

- 按位反 $\sim$

位运算

位运算是定义在整数类型数据上的一种运算，包括：

位逻辑运算 – 在数据的每一位 bit 上独立进行运算

- 按位反 $\sim$
- 按位与 $\&$

位运算

位运算是定义在整数类型数据上的一种运算，包括：

位逻辑运算 – 在数据的每一位 bit 上独立进行运算

- 按位反 $\sim$
- 按位与 &
- 按位或 |

位运算

位运算是定义在整数类型数据上的一种运算，包括：

位逻辑运算 – 在数据的每一位 bit 上独立进行运算

- 按位反 ~
- 按位与 &
- 按位或 |
- 按位异或 ^

位运算

位运算是定义在整数类型数据上的一种运算，包括：

位逻辑运算 – 在数据的每一位 bit 上独立进行运算

- 按位反 $\sim$
- 按位与 &
- 按位或 |
- 按位异或 ^

移位运算 – 在数据的所有的 bit 位依次进行移位

位运算

位运算是定义在整数类型数据上的一种运算，包括：

位逻辑运算 – 在数据的每一位 bit 上独立进行运算

- 按位反 $\sim$
- 按位与 &
- 按位或 |
- 按位异或 ^

移位运算 – 在数据的所有的 bit 位依次进行移位

- 左移位 $<<$

位运算

位运算是定义在整数类型数据上的一种运算，包括：

位逻辑运算 – 在数据的每一位 bit 上独立进行运算

- 按位反 $\sim$
- 按位与 &
- 按位或 |
- 按位异或 ^

移位运算 – 在数据的所有的 bit 位依次进行移位位

- 左移位 $<<$
- 右移位 $>>$

位逻辑运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$\mathsf{a} = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1$$
$$\mathsf{b} = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1$$

- 按位反: $\sim \mathsf{a} = 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0$

位逻辑运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$\mathtt{a} = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1$$
$$\mathtt{b} = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1$$

- 按位反: $\sim \mathtt{a} = 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0$
- 按位与: $\mathtt{a}\ \&\ \mathtt{b} = 0\ 0\ 0\ 0\ 1\ 0\ 0\ 1$

位逻辑运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$\begin{aligned} a &= 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1 \\ b &= 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1 \end{aligned}$$

- 按位反: $\sim a = 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0$
- 按位与: $a\ \&\ b = 0\ 0\ 0\ 0\ 1\ 0\ 0\ 1$
- 按位或: $a\ |\ b = 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1$

位逻辑运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$\begin{aligned} a &= 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1 \\ b &= 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1 \end{aligned}$$

- 按位反: $\sim a = 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0$
- 按位与: $a\ \&\ b = 0\ 0\ 0\ 0\ 1\ 0\ 0\ 1$
- 按位或: $a\ |\ b = 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1$
- 按位异或: $a\ \hat{}\ b = 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0$

位逻辑运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$\begin{aligned} a &= 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1 \\ b &= 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1 \end{aligned}$$

- 按位反: $\sim a = 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0$
- 按位与: $a\ \&\ b = 0\ 0\ 0\ 0\ 1\ 0\ 0\ 1$
- 按位或: $a\ |\ b = 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1$
- 按位异或: $a\ \hat{}\ b = 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0$
  - 互异为真

异或运算（^）

^ 可以理解为抛弃进位的加法。例如:

$1 \oplus 1 = 0$;

$0 \oplus 0 = 0$;

$1 \oplus 0 = 1$;

$0 \oplus 1 = 1$;

$a \oplus b =$ 末位 $(a + b)$

异或运算（^）

^ 可以理解为抛弃进位的加法。例如:

$1 \oplus 1 = 0$;

$0 \oplus 0 = 0$;

$1 \oplus 0 = 1$;

$0 \oplus 1 = 1$;

a $\oplus$ b = 末位 (a + b)

异或运算 ^ 具有可结合性，证明：

- a ^ b ^ c = 末位 (a + b) ^ c

异或运算（^）

^ 可以理解为抛弃进位的加法。例如:

$1 \oplus 1 = 0;$

$0 \oplus 0 = 0;$

$1 \oplus 0 = 1;$

$0 \oplus 1 = 1;$

$a \oplus b =$ 末位 $(a + b)$

异或运算 ^ 具有可结合性，证明：

- $a \hat{} b \hat{} c =$ 末位 $(a + b) \hat{} c =$ 末位 $(a + b + c)$

异或运算（^）

^ 可以理解为抛弃进位的加法。例如:

$1 \oplus 1 = 0$;

$0 \oplus 0 = 0$;

$1 \oplus 0 = 1$;

$0 \oplus 1 = 1$;

$a \oplus b =$ 末位 $(a + b)$

异或运算 ^ 具有可结合性，证明：

- a ^ b ^ c = 末位 (a + b) ^ c = 末位 (a + b + c)
- a ^ (b ^ c) = a ^ 末位 (b + c)

异或运算（^）

^ 可以理解为抛弃进位的加法。例如:

$1 \oplus 1 = 0$;

$0 \oplus 0 = 0$;

$1 \oplus 0 = 1$;

$0 \oplus 1 = 1$;

$a \oplus b =$ 末位 $(a + b)$

异或运算 ^ 具有可结合性，证明：

- a ^ b ^ c = 末位 (a + b) ^ c = 末位 (a + b + c)
- a ^ (b ^ c) = a ^ 末位 (b + c) = 末位 (a + b + c)

异或运算（^）

^ 可以理解为抛弃进位的加法。例如:

$1 \oplus 1 = 0$;

$0 \oplus 0 = 0$;

$1 \oplus 0 = 1$;

$0 \oplus 1 = 1$;

$a \oplus b =$ 末位 $(a + b)$

异或运算 ^ 具有可结合性，证明：

- a ^ b ^ c = 末位 (a + b) ^ c = 末位 (a + b + c)
- a ^ (b ^ c) = a ^ 末位 (b + c) = 末位 (a + b + c)

  因此 a ^ b ^ c = a ^ (b ^ c)

异或运算（^）

^ 可以理解为抛弃进位的加法。例如:

$1 \oplus 1 = 0$;

$0 \oplus 0 = 0$;

$1 \oplus 0 = 1$;

$0 \oplus 1 = 1$;

$a \oplus b =$ 末位 $(a + b)$

异或运算 ^ 具有可结合性，证明：

- a ^ b ^ c = 末位 (a + b) ^ c = 末位 (a + b + c)
- a ^ (b ^ c) = a ^ 末位 (b + c) = 末位 (a + b + c)

  因此 a ^ b ^ c = a ^ (b ^ c)

异或运算（^）

^ 可以理解为抛弃进位的加法。例如:

$1 \oplus 1 = 0$;

$0 \oplus 0 = 0$;

$1 \oplus 0 = 1$;

$0 \oplus 1 = 1$;

$a \oplus b =$ 末位 $(a + b)$

异或运算 ^ 具有可结合性，证明：

- a ^ b ^ c = 末位 (a + b) ^ c = 末位 (a + b + c)
- a ^ (b ^ c) = a ^ 末位 (b + c) = 末位 (a + b + c)

  因此 a ^ b ^ c = a ^ (b ^ c)

异或运算 ^ 具有可交换性：

a ^ b = b ^ a

位运算的性质

- 按位与、或、异或都是可交换的

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a

位运算的性质

- 按位与、或、异或都是可交换的
  - $a \,\&\, b = b \,\&\, a$
- 按位与、或、异或都是可结合的

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ~0 = 0xFFFFFFFF

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ∼0 = 0xFFFFFFFF
- a & 0 = 0;

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ~ 0 = 0xFFFFFFFF
- a & 0 = 0;　　a & ~ 0 = a;

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ~ 0 = 0xFFFFFFFF
- a & 0 = 0; a & ~ 0 = a;
- a | 0 = a;

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ~ 0 = 0xFFFFFFFF
- a & 0 = 0;　　a & ~ 0 = a;
- a | 0 = a;　　a | ~ 0 = ~ 0;

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ∼0 = 0xFFFFFFFF
- a & 0 = 0;　　a & ∼0 = a;
- a | 0 = a;　　a | ∼0 = ∼0;
- a ^ 0 = a;

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ∼0 = 0xFFFFFFFF
- a & 0 = 0;　　a & ∼0 = a;
- a | 0 = a;　　a | ∼0 = ∼0;
- a ^ 0 = a;　　a ^ ∼0 = ∼a;

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ∼0 = 0xFFFFFFFF
- a & 0 = 0;　　a & ∼0 = a;
- a | 0 = a;　　a | ∼0 = ∼0;
- a ^ 0 = a;　　a ^ ∼0 = ∼a;
- a & a = a;

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ∼0 = 0xFFFFFFFF
- a & 0 = 0;　　a & ∼0 = a;
- a | 0 = a;　　a | ∼0 = ∼0;
- a ^ 0 = a;　　a ^ ∼0 = ∼a;
- a & a = a;　　a & ∼a = 0;

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ∼0 = 0xFFFFFFFF
- a & 0 = 0;　　a & ∼0 = a;
- a | 0 = a;　　a | ∼0 = ∼0;
- a ^ 0 = a;　　a ^ ∼0 = ∼a;
- a & a = a;　　a & ∼a = 0;
- a | a = a;

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ∼0 = 0xFFFFFFFF
- a & 0 = 0;　　a & ∼0 = a;
- a | 0 = a;　　a | ∼0 = ∼0;
- a ^ 0 = a;　　a ^ ∼0 = ∼a;
- a & a = a;　　a & ∼a = 0;
- a | a = a;　　a | ∼a = ∼0;

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ∼0 = 0xFFFFFFFF
- a & 0 = 0;　　a & ∼0 = a;
- a | 0 = a;　　a | ∼0 = ∼0;
- a ^ 0 = a;　　a ^ ∼0 = ∼a;
- a & a = a;　　a & ∼a = 0;
- a | a = a;　　a | ∼a = ∼0;
- a ^ a = 0;

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ~0 = 0xFFFFFFFF
- a & 0 = 0;　　a & ~0 = a;
- a | 0 = a;　　a | ~0 = ~0;
- a ^ 0 = a;　　a ^ ~0 = ~a;
- a & a = a;　　a & ~a = 0;
- a | a = a;　　a | ~a = ~0;
- a ^ a = 0;　　a ^ ~a = ~0;

位运算的性质

- 按位与、或、异或都是可交换的
  - a & b = b & a
- 按位与、或、异或都是可结合的
  - a & b & c = a & (b & c)
- ~0 = 0xFFFFFFFF
- a & 0 = 0;　　a & ~0 = a;
- a | 0 = a;　　a | ~0 = ~0;
- a ^ 0 = a;　　a ^ ~0 = ~a;
- a & a = a;　　a & ~a = 0;
- a | a = a;　　a | ~a = ~0;
- a ^ a = 0;　　a ^ ~a = ~0;
- a ^ b ^ b = a;

移位运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$\mathtt{a} = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1$$
$$\mathtt{b} = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1$$

- 左移位　　$\mathtt{a} << 1 = 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0$

移位运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$\mathtt{a} = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1$$
$$\mathtt{b} = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1$$

- 左移位　　$\mathtt{a} << 1 = 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0$
  - 右侧补 0

移位运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$\mathtt{a} = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1$$
$$\mathtt{b} = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1$$

- 左移位 $\mathtt{a} << 1 = 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0$
  - 右侧补 0
- 右移位 $\mathtt{a} >> 1 = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1$
  $\mathtt{b} >> 2 = 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1$

移位运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$\mathrm{a} = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1$$
$$\mathrm{b} = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1$$

- 左移位　　$\mathrm{a} << 1 = 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0$
  - 右侧补 0
- 右移位　　$\mathrm{a} >> 1 = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1$
  $\mathrm{b} >> 2 = 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1$
  - 对于无符号的数，左侧补 0

移位运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$\mathtt{a} = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1$$
$$\mathtt{b} = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1$$

- 左移位 $\mathtt{a} << 1 = 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0$
  - 右侧补 0
- 右移位 $\mathtt{a} >> 1 = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1$
  $\mathtt{b} >> 2 = 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1$
  - 对于无符号的数，左侧补 0
  - 对于有符号的数

移位运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$a = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1$$
$$b = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1$$

- 左移位　　$a << 1 = 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0$
  - 右侧补 0
- 右移位　　$a >> 1 = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1$
  $b >> 2 = 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1$
  - 对于无符号的数，左侧补 0
  - 对于有符号的数
    - 左侧既可以填充 0，

移位运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$a = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1$$
$$b = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1$$

- 左移位　　$a << 1 = 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0$
  - 右侧补 0
- 右移位　　$a >> 1 = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1$
  $b >> 2 = 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1$
  - 对于无符号的数，左侧补 0
  - 对于有符号的数
    - 左侧既可以填充 0，
    - 也可以复制符号位上的值。

移位运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$a = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1$$
$$b = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1$$

- 左移位　　$a << 1 = 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0$
  - 右侧补 0
- 右移位　　$a >> 1 = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1$
  $b >> 2 = 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1$
  - 对于无符号的数，左侧补 0
  - 对于有符号的数
    - 左侧既可以填充 0，
    - 也可以复制符号位上的值。
    - C 语言没有规定，取决于编译器的实现。(大部分编译器复制符号位上的值)

移位运算

考虑两个 8 位整数 (16 位和 32 位整数也一样):

$$a = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1$$
$$b = 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1$$

- 左移位　　$a << 1 = 0\ 0\ 0\ 1\ 0\ 1\ 1\ 0$
  - 右侧补 0
- 右移位　　$a >> 1 = 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1$
  $b >> 2 = 0\ 0\ 0\ 1\ 1\ 0\ 1\ 1$
  - 对于无符号的数，左侧补 0
  - 对于有符号的数
    - 左侧既可以填充 0，
    - 也可以复制符号位上的值。
    - C 语言没有规定，取决于编译器的实现。(大部分编译器复制符号位上的值)
    - 应该避免对有符号数进行右移运算

移位运算的性质

对于无符号的整数，移位与 2 的关系

移位运算的性质

对于无符号的整数，移位与 2 的关系

- 左移 1 位 $\Longleftrightarrow \times 2$

移位运算的性质

对于无符号的整数，移位与 2 的关系

- 左移 1 位 $\Longleftrightarrow \times 2$
- 右移 1 位 $\Longleftrightarrow \div 2$ （在不发生溢出的时候）

位运算与赋值运算复合

- `&=` 例如：a &= b $\Longleftrightarrow$ a = a & b
- `|=` 例如：a |= b $\Longleftrightarrow$ a = a | b
- `^=` 例如：a ^= b $\Longleftrightarrow$ a = a ^ b
- `>>=` 例如：a >>= n $\Longleftrightarrow$ a = a >> n
- `<<=` 例如：a <<= n $\Longleftrightarrow$ a = a << n

位运算与赋值运算复合

- &= 例如：a &= b $\Longleftrightarrow$ a = a & b
- |= 例如：a |= b $\Longleftrightarrow$ a = a | b
- ^= 例如：a ^= b $\Longleftrightarrow$ a = a ^ b
- >>= 例如：a >>= n $\Longleftrightarrow$ a = a >> n
- <<= 例如：a <<= n $\Longleftrightarrow$ a = a << n

这些复合赋值运算符和其它的赋值运算符具有相同的优先级和结合性

异或运算应用 – 无临时变量的交换

```
int a = 1, b = 2;
```

标准的交换步骤

```
int temp;

temp = a;
a = b;
b = temp;
```

异或运算应用 – 无临时变量的交换

```
int a = 1, b = 2;
```

标准的交换步骤

```
int temp;

temp = a;
a = b;
b = temp;
```

需要定义和使用一个临时变量 temp

异或运算应用 – 无临时变量的交换

```
int a = 1, b = 2;
```

标准的交换步骤

```
int temp;

temp = a;
a = b;
b = temp;
```

需要定义和使用一个临时变量 temp

使用异或运算进行交换

```
a = a ^ b;
b = b ^ a;
a = a ^ b
```

异或运算应用 – 无临时变量的交换

```
int a = 1, b = 2;
```

标准的交换步骤

```
int temp;

temp = a;
a = b;
b = temp;
```

需要定义和使用一个临时变量 temp

使用异或运算进行交换

```
a = a ^ b;
b = b ^ a;
a = a ^ b
```

或

```
a ^= b; b ^= a; a ^= b;
```

异或运算应用 – 无临时变量的交换

```
int a = 1, b = 2;
```

标准的交换步骤

```
int temp;

temp = a;
a = b;
b = temp;
```

需要定义和使用一个临时变量 temp

使用异或运算进行交换

```
a = a ^ b;
b = b ^ a;
a = a ^ b
```

或

```
a ^= b; b ^= a; a ^= b;
```

或

```
a ^= b ^= a ^= b
```

异或运算应用 – 无临时变量的交换

```
int a = 1, b = 2;
```

标准的交换步骤

```
int temp;

temp = a;
a = b;
b = temp;
```

需要定义和使用一个临时变量 temp

使用异或运算进行交换

```
a = a ^ b;
b = b ^ a;
a = a ^ b
```

```
a ^ b ^ b = a
a ^ b ^ a = b
```

或

```
a ^= b; b ^= a; a ^= b;
```

或

```
a ^= b ^= a ^= b
```

异或运算应用 – 确定重复数

假设有 N+1 个变量，存放了整数 $1, 2, 3, \ldots, N$，其中有且只有一个整数重复了 2 次。如何快速确定哪一个整数重复了？

可行的方法：

- 逐对比较，检查没有相同的。运算量：$N \times (N-1)/2$

异或运算应用 – 确定重复数

假设有 N+1 个变量，存放了整数 $1, 2, 3, \ldots, N$，其中有且只有一个整数重复了 2 次。如何快速确定哪一个整数重复了？

可行的方法：

- 逐对比较，检查没有相同的。运算量：$N \times (N-1)/2$
- 计算所有的变量的和，与 1+2+...+N 比较，得到重复的数。运算量 N+1 次加减法

异或运算应用 – 确定重复数

假设有 N+1 个变量，存放了整数 $1, 2, 3, \ldots, N$，其中有且只有一个整数重复了 2 次。如何快速确定哪一个整数重复了？

可行的方法：

- 逐对比较，检查没有相同的。运算量： $N \times (N-1)/2$
- 计算所有的变量的和，与 1+2+...+N 比较，得到重复的数。运算量 N+1 次加减法
  - 加法求和可能会溢出（当 N 非常大的时候）

异或运算应用 – 确定重复数

假设有 N+1 个变量，存放了整数 $1, 2, 3, \ldots, N$，其中有且只有一个整数重复了 2 次。如何快速确定哪一个整数重复了？

可行的方法：

- 逐对比较，检查没有相同的。运算量：$N \times (N-1)/2$
- 计算所有的变量的和，与 1+2+...+N 比较，得到重复的数。运算量 N+1 次加减法
  - 加法求和可能会溢出（当 N 非常大的时候）
- 把加法换作：异或

异或运算应用 – 确定重复数

假设有 N+1 个变量，存放了整数 $1, 2, 3, \ldots, N$，其中有且只有一个整数重复了 2 次。如何快速确定哪一个整数重复了？

可行的方法：

- 逐对比较，检查没有相同的。运算量：$N \times (N-1)/2$
- 计算所有的变量的和，与 1+2+...+N 比较，得到重复的数。运算量 N+1 次加减法
  - 加法求和可能会溢出（当 N 非常大的时候）
- 把加法换作：异或

异或运算应用 – 确定重复数

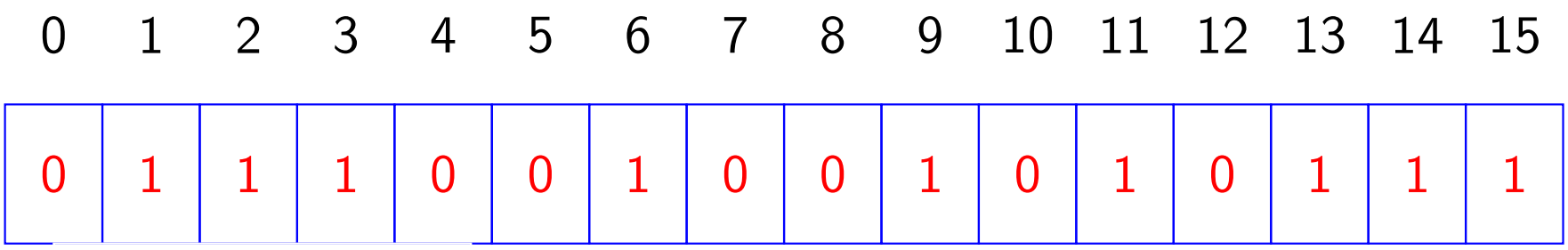
假设有 N+1 个变量，存放了整数 $1, 2, 3, \ldots, N$，其中有且只有一个整数重复了 2 次。如何快速确定哪一个整数重复了？

可行的方法：

- 逐对比较，检查没有相同的。运算量：$N \times (N-1)/2$
- 计算所有的变量的和，与 1+2+...+N 比较，得到重复的数。运算量 N+1 次加减法
  - 加法求和可能会溢出（当 N 非常大的时候）
- 把加法换作：异或

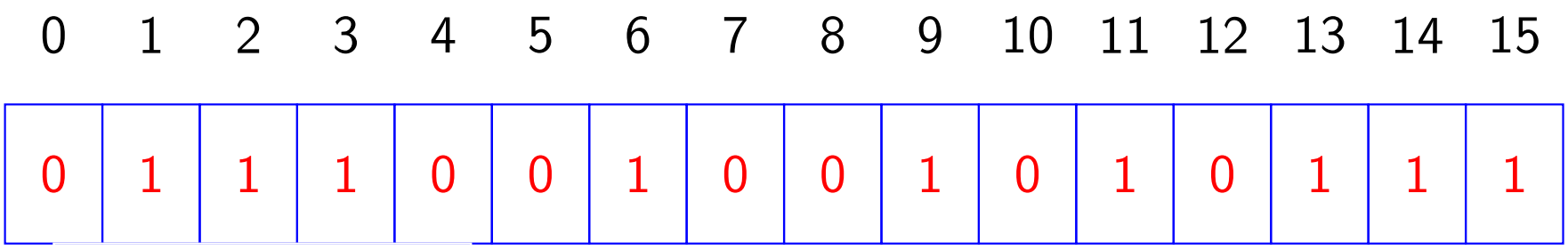
```
a0^a1^a2^...^aN ^ 1^2^...^N
```

位运算应用 – 输出整数的 bit 值



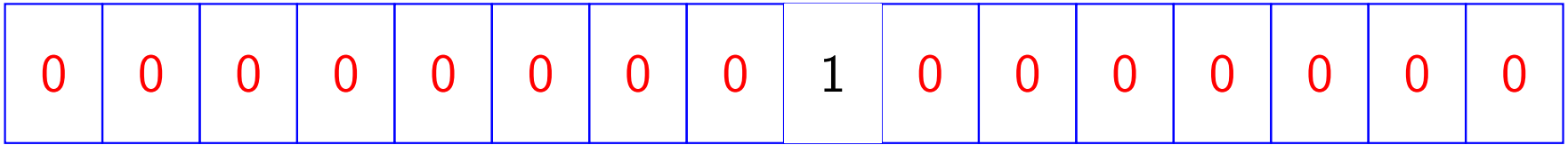
如何获取第 k 位的值？

位运算应用 – 输出整数的 bit 值



如何获取第 k 位的值？

产生一个掩码：$1 << k$，（只有第 k 位为 1）



位运算应用 – 输出整数的 bit 值

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	1	0	0	1	0	0	1	0	1	0	1	1	1

如何获取第 k 位的值？

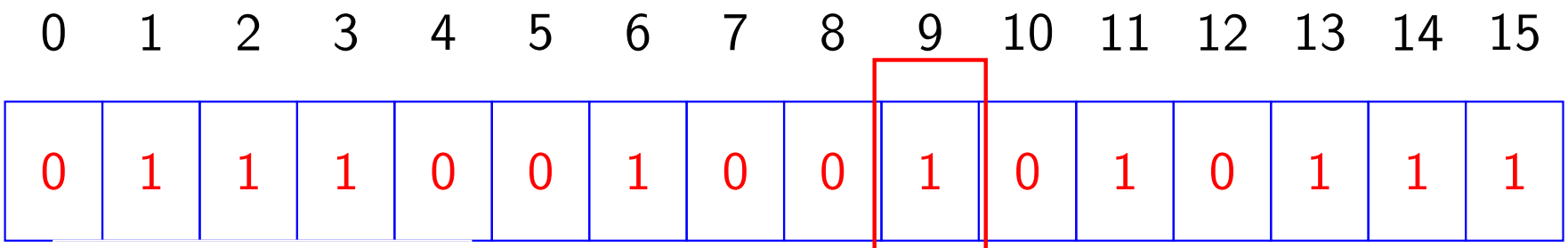
产生一个掩码：$1 << k$，（只有第 k 位为 1）

0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

执行按位与运算：$n \,\&\, (1 << k)$

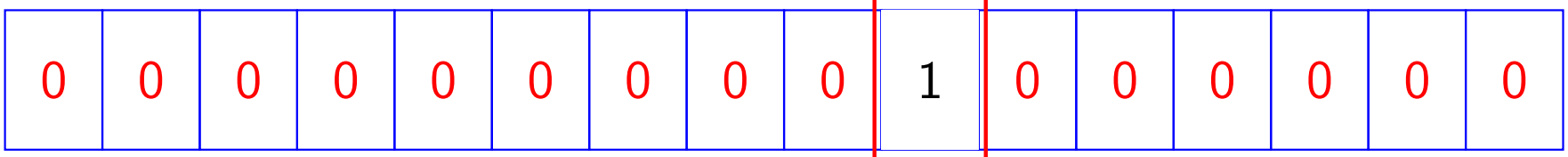
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

位运算应用 – 输出整数的 bit 值

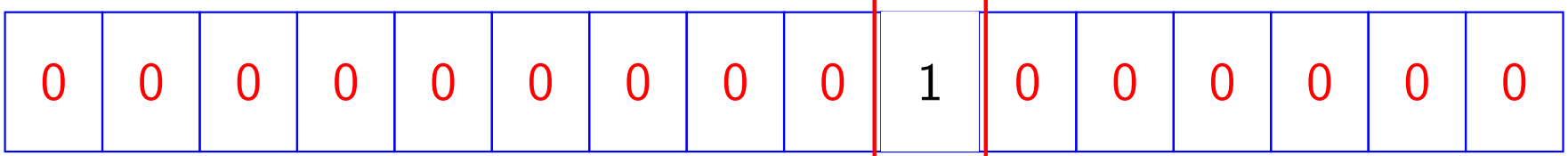


如何获取第 k 位的值？

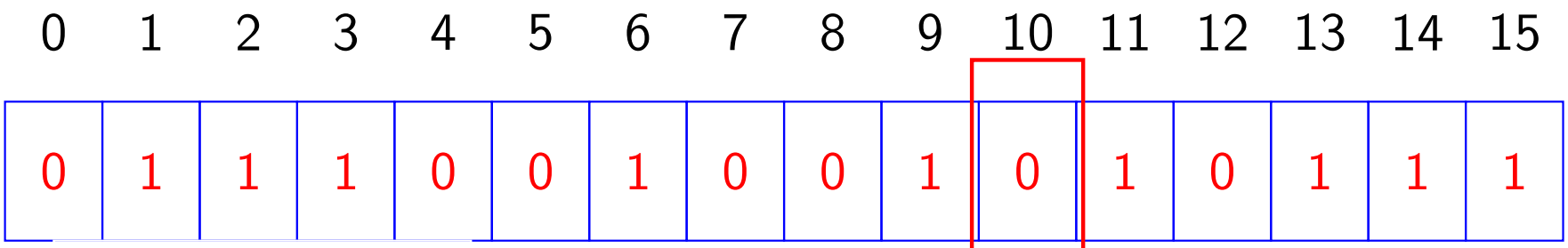
产生一个掩码：$1 << k$，（只有第 k 位为 1）



执行按位与运算：$n \,\&\, (1 << k)$

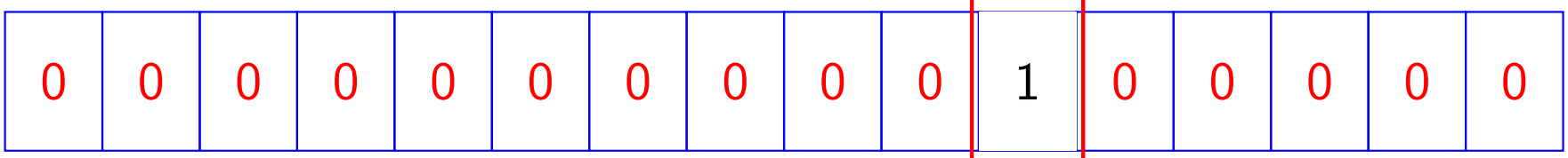


位运算应用 – 输出整数的 bit 值

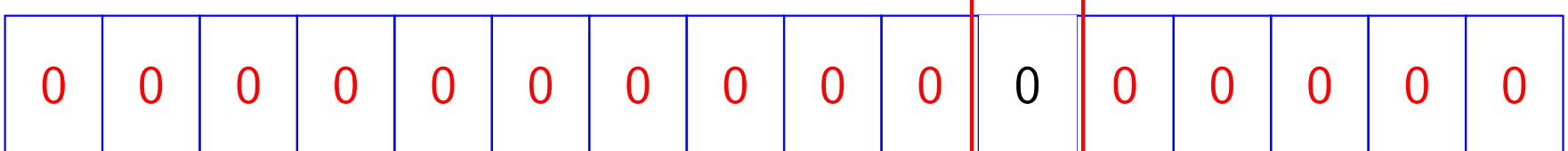


如何获取第 k 位的值？

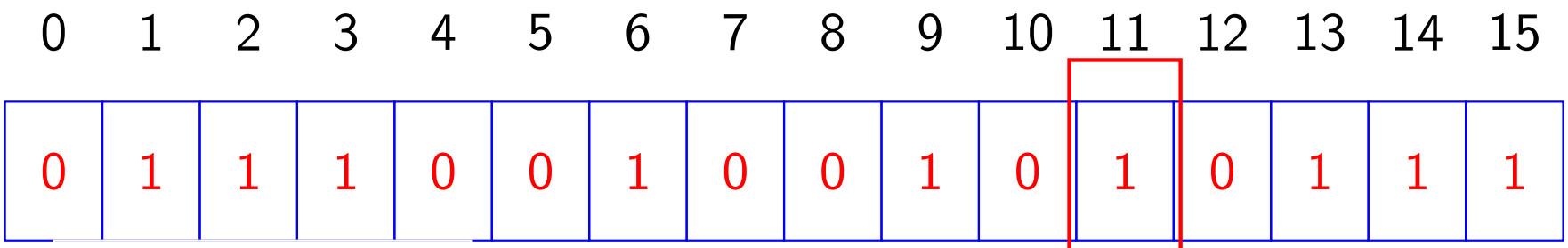
产生一个掩码：$1 << k$，（只有第 k 位为 1）



执行按位与运算：$n \,\&\, (1 << k)$

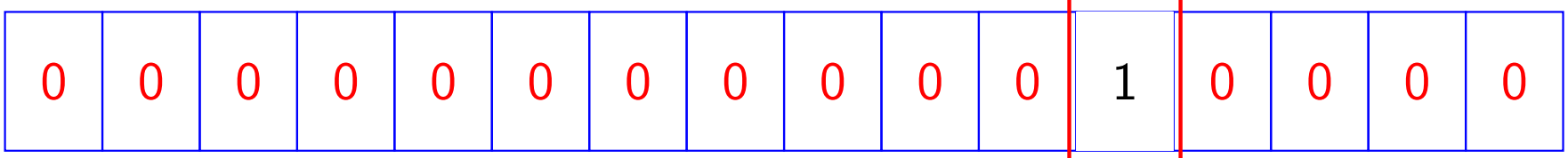


位运算应用 – 输出整数的 bit 值

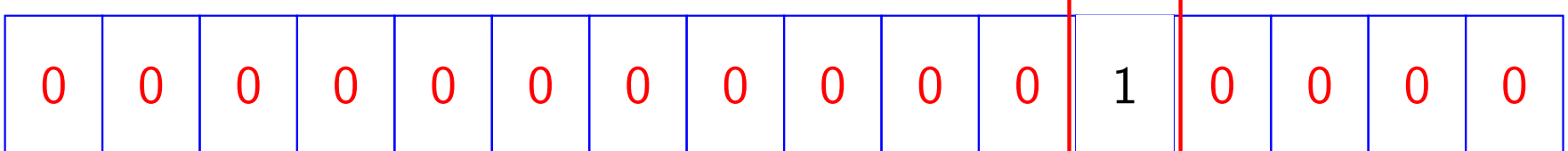


如何获取第 k 位的值？

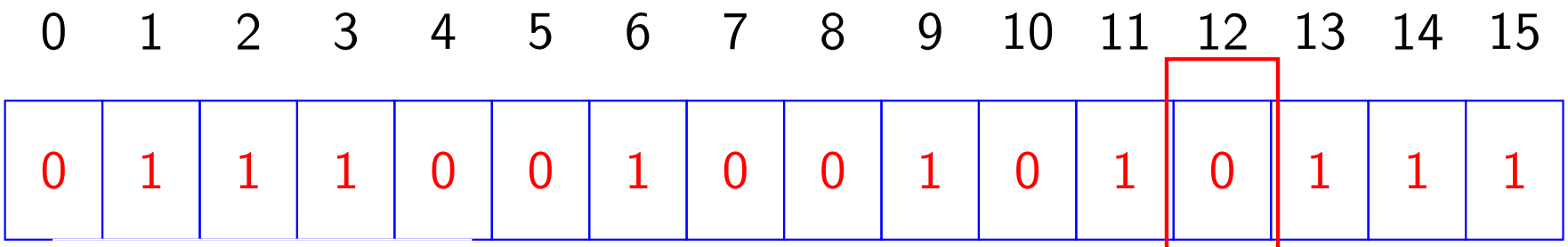
产生一个掩码：$1 << k$，（只有第 k 位为 1）



执行按位与运算：$n \,\&\, (1 << k)$

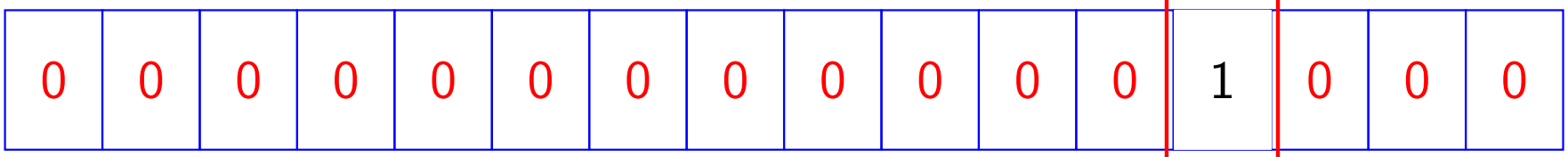


位运算应用 – 输出整数的 bit 值

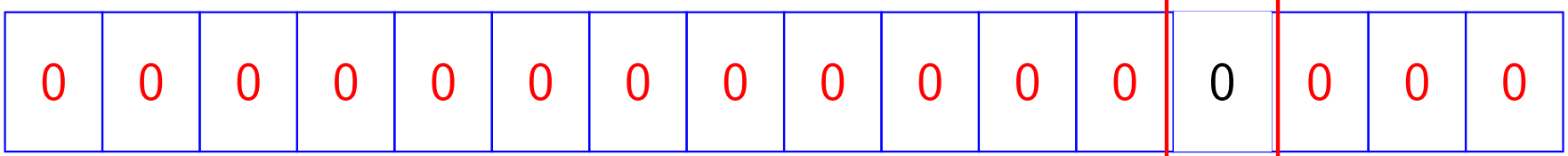


如何获取第 k 位的值？

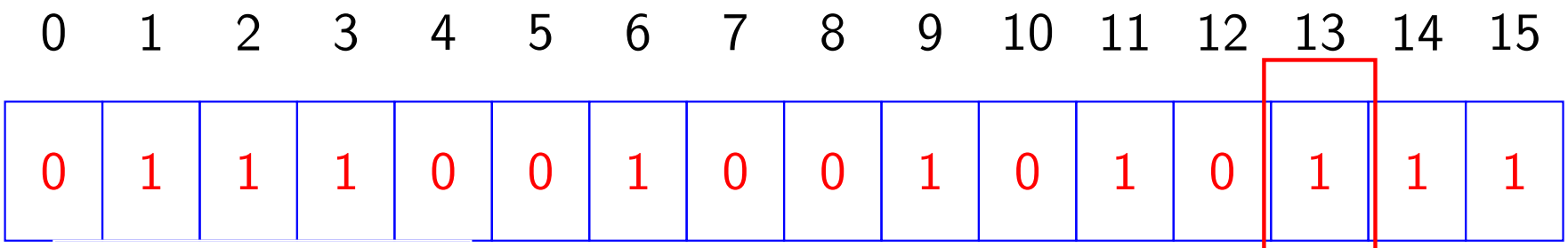
产生一个掩码：$1 << k$，（只有第 k 位为 1）



执行按位与运算：$n \,\&\, (1 << k)$

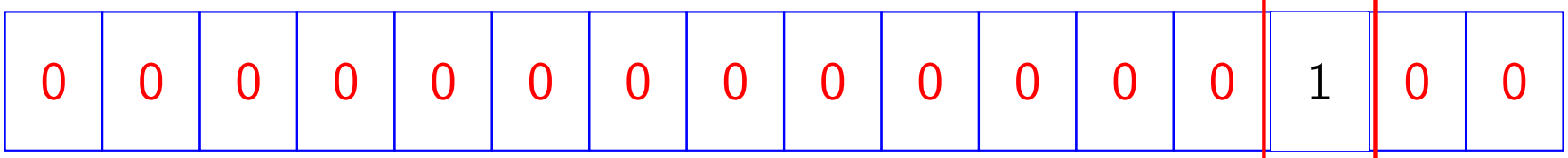


位运算应用 – 输出整数的 bit 值

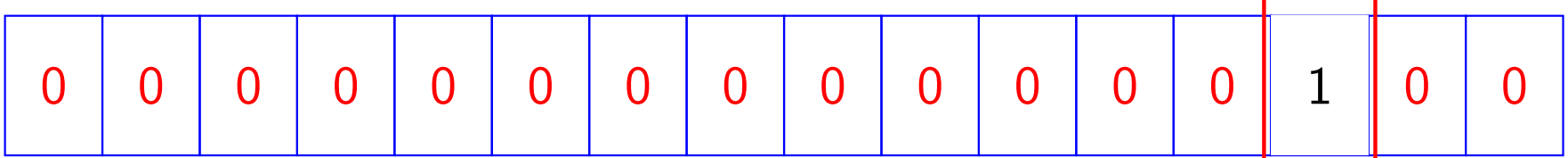


如何获取第 k 位的值？

产生一个掩码：$1 << k$，（只有第 k 位为 1）



执行按位与运算：$n \,\&\, (1 << k)$



位运算应用 – 输出整数的 bit 值

```
void print_bits ( int n, int high, int low )
{
    int k;
    for( k = high; k>=low; k-- ) {
        int mask = 1<<k; //生成掩码
        if( mask & n )    //掩码对应的bit值
            putchar('1');
        else
            putchar('0');
    }
}
```

- 应用移位运算，为第 k 位比特位制造一个掩码:
  $mask = 1 << k$
- 将掩码 mask 和整数进行逻辑与，获得 bit 值: mask & n
  mask & n 非零，代表该比特为 1

位运算应用 – 输出整数的 bit 值

```
int main()
{
    int n; float x;
    printf("输入一个整数和实数:");
    scanf("%d %f", &n, &x);
    printf("整数%9d = ", n); print_bits(n, 31, 0);
                putchar('\n');
    printf("实数%9f = ", x); print_bits(*(int*)&x,
        31, 0); putchar('\n');
    return 0;
}
```

位运算应用 – 输出整数的 bit 值

```
int main()
{
    int n; float x;
    printf("输入一个整数和实数:");
    scanf("%d %f", &n, &x);
    printf("整数%9d = ", n); print_bits(n, 31, 0);
                  putchar('\n');
    printf("实数%9f = ", x); print_bits(*(int*)&x,
        31, 0); putchar('\n');
    return 0;
}
```

```
输入一个整数和实数:234 13.567
整数       234 = 00000000000000000000000011101010
实数 13.567000 = 01000001010110010001001101111
```

位运算应用 – 输出整数的 bit 值

```
int main()
{
    int n; float x;
    printf("输入一个整数和实数:");
    scanf("%d %f", &n, &x);
    printf("整数%9d = ", n); print_bits(n, 31, 0);
                putchar('\n');
    printf("实数%9f = ", x); print_bits(*(int*)&x,
        31, 0); putchar('\n');
    return 0;
}
```

```
输入一个整数和实数:234 13.567
整数       234 = 00000000000000000000000011101010
实数 13.567000 = 01000001010110010001001001101111
```

这里利用了指针和类型转换: &x 获取了float x 的指针，然后将其转换为整数指针 (int*)，在通过整数指针，将它的 bits 转化为整数，保存到变量 n 中。只有整数才可以进行位运算

内容提要

位运算应用 – 查看浮点数 float 的符号位，阶码，尾数

符号	阶码	尾数
1 bit	8 bit	23 bit

位运算应用 – 查看浮点数 float 的符号位，阶码，尾数

符号	阶码	尾数
1 bit	8 bit	23 bit

获取符号

```
int get_float_sign ( float x )
{
    int n = *(int*)&x; //复制到int变量中
    int mask = 1<<31;  //生成符号位的掩码
    return mask & n ? -1 : 1;
}
```

位运算应用 – 查看浮点数 float 的符号位，阶码，尾数

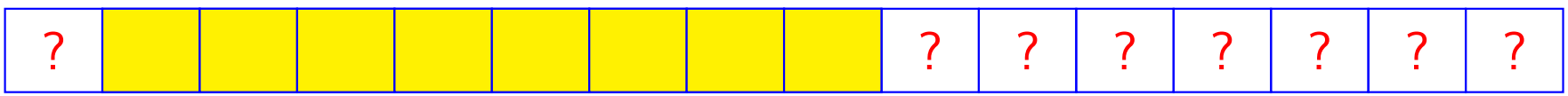
获取阶码

```
int get_float_exp ( float x )
{
    int n = *(int*)&x; //复制到int变量中
    n = (n >> 23) & 0xFF;
    n = n - 127;
    return n;
}
```

位运算应用 – 查看浮点数 float 的符号位，阶码，尾数

获取阶码

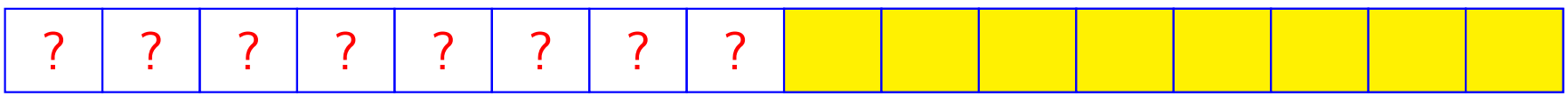
```
int get_float_exp ( float x )
{
    int n = *(int*)&x; //复制到int变量中
    n = (n >> 23) & 0xFF;
    n = n - 127;
    return n;
}
```



位运算应用 – 查看浮点数 float 的符号位，阶码，尾数

获取阶码

```
int get_float_exp ( float x )
{
    int n = *(int*)&x; //复制到int变量中
    n = (n >> 23) & 0xFF;
    n = n - 127;
    return n;
}
```

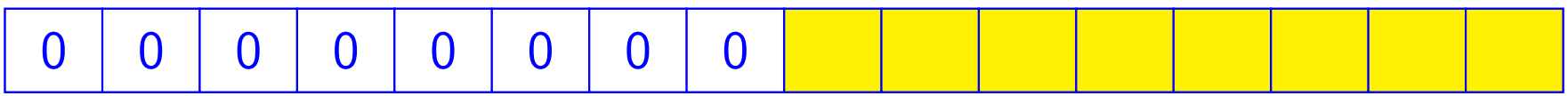


- 向右移位（23 位）

位运算应用 – 查看浮点数 float 的符号位，阶码，尾数

获取阶码

```
int get_float_exp ( float x )
{
    int n = *(int*)&x; //复制到int变量中
    n = (n >> 23) & 0xFF;
    n = n - 127;
    return n;
}
```



- 向右移位（23 位）
- 保留低 8 位，其余清零

位运算应用 – 查看浮点数 float 的符号位，阶码，尾数

获取尾数

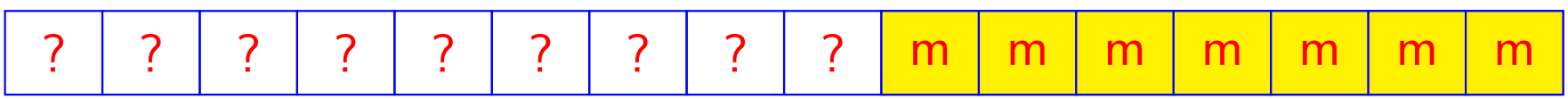
```
float get_float_mantissa ( float x )
{
    int n = *(int*)&x; //复制到int变量中
    int e = n & (0XFF<<23); // 阶码段
    n &= ~(0X1FF<<23); // 符号位和阶码段清零
    if( e ) {//不全为0
        n |= 0x7F<<23; // 阶码段置为0X7F (0x7F
            =01111111, 代表指数为0)
    }
    return *(float*)&n;
}
```

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

位运算应用 – 查看浮点数 float 的符号位，阶码，尾数

获取尾数

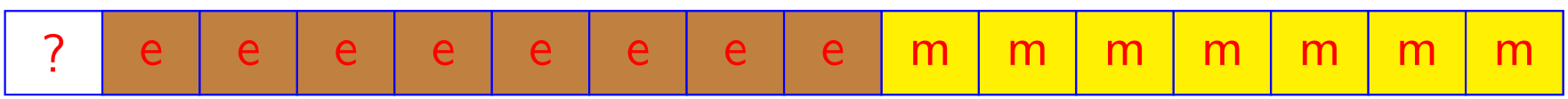
```
float get_float_mantissa ( float x )
{
    int n = *(int*)&x; //复制到int变量中
    int e = n & (0XFF<<23); // 阶码段
    n &= ~(0X1FF<<23); // 符号位和阶码段清零
    if( e ) {//不全为0
        n |= 0x7F<<23; // 阶码段置为0X7F (0x7F
            =01111111, 代表指数为0)
    }
    return *(float*)&n;
}
```



位运算应用 – 查看浮点数 float 的符号位，阶码，尾数

获取尾数

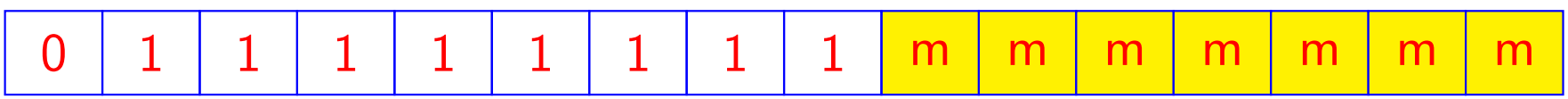
```
float get_float_mantissa ( float x )
{
    int n = *(int*)&x; //复制到int变量中
    int e = n & (0XFF<<23); // 阶码段
    n &= ~(0X1FF<<23); // 符号位和阶码段清零
    if( e ) {//不全为0
        n |= 0x7F<<23; // 阶码段置为0X7F (0x7F
            =01111111, 代表指数为0)
    }
    return *(float*)&n;
}
```



位运算应用 – 查看浮点数 float 的符号位，阶码，尾数

获取尾数

```
float get_float_mantissa ( float x )
{
    int n = *(int*)&x; //复制到int变量中
    int e = n & (0XFF<<23); // 阶码段
    n &= ~(0X1FF<<23); // 符号位和阶码段清零
    if( e ) {//不全为0
        n |= 0x7F<<23; // 阶码段置为0X7F (0x7F
            =01111111, 代表指数为0)
    }
    return *(float*)&n;
}
```



位运算应用 – 查看浮点数 float 的符号位，阶码，尾数

查看浮点数

```
void print_float(float x)
{
    print_bits( *(int*)&x, 31, 31 );
    printf("<%c> ", get_float_sign(x)>0?'+':'-');
    print_bits(*(int*)&x, 30, 23 ); print_space(1);
    printf("<%-4d> ", get_float_exp(x));
    print_bits(*(int*)&x, 22, 0 );
    printf(" <%f>\n", get_float_mantissa(x));
}
```

```
输入一个浮点数:12.3456
符号 阶码段    <指数> 尾数段                          <尾数>
0<+> 10000010 <3    > 10001011000011110010100 <1.543200>

输入一个浮点数:1.5432
符号 阶码段    <指数> 尾数段                          <尾数>
0<+> 01111111 <0    > 10001011000011110010100 <1.543200>
```

内容提要

长度运算符 sizeof

一个计算常量、变量、以及数据类型的长度的运算符。长度以字节为单位

- 单目运算符

长度运算符 sizeof

一个计算常量、变量、以及数据类型的长度的运算符。长度以字节为单位

- 单目运算符
- 用 sizeof 表示

长度运算符 sizeof

一个计算常量、变量、以及数据类型的长度的运算符。长度以字节为单位

- 单目运算符
- 用 sizeof 表示
- 运算对象可以是

长度运算符 sizeof

一个计算常量、变量、以及数据类型的长度的运算符。长度以字节为单位

- 单目运算符
- 用 sizeof 表示
- 运算对象可以是
  - 常量，例如 sizeof(5)

长度运算符 sizeof

一个计算常量、变量、以及数据类型的长度的运算符。长度以字节为单位

- 单目运算符
- 用 sizeof 表示
- 运算对象可以是
  - 常量，例如 sizeof(5)
  - 变量，例如 sizeof(x)

长度运算符 sizeof

一个计算常量、变量、以及数据类型的长度的运算符。长度以字节为单位

- 单目运算符
- 用 sizeof 表示
- 运算对象可以是
  - 常量，例如 sizeof(5)
  - 变量，例如 sizeof(x)
  - 数据类型名，例如 sizeof(float)

长度运算符 sizeof

一个计算常量、变量、以及数据类型的长度的运算符。长度以字节为单位

- 单目运算符
- 用 sizeof 表示
- 运算对象可以是
  - 常量，例如 sizeof(5)
  - 变量，例如 sizeof(x)
  - 数据类型名，例如 sizeof(float)

假设定义了：int a; 那么：

长度运算符 sizeof

一个计算常量、变量、以及数据类型的长度的运算符。长度以字节为单位

- 单目运算符
- 用 sizeof 表示
- 运算对象可以是
  - 常量，例如 sizeof(5)
  - 变量，例如 sizeof(x)
  - 数据类型名，例如 sizeof(float)

假设定义了：int a; 那么：

- 表达式 sizeof(a) 的值为变量 a 的长度，等于 4

长度运算符 sizeof

一个计算常量、变量、以及数据类型的长度的运算符。长度以字节为单位

- 单目运算符
- 用 sizeof 表示
- 运算对象可以是
  - 常量，例如 sizeof(5)
  - 变量，例如 sizeof(x)
  - 数据类型名，例如 sizeof(float)

假设定义了：int a; 那么：

- 表达式 sizeof(a) 的值为变量 a 的长度，等于 4
- 表达式 sizeof(int) 表示整型 int 的长度，等于 4

长度运算符 sizeof

一个计算常量、变量、以及数据类型的长度的运算符。长度以字节为单位

- 单目运算符
- 用 sizeof 表示
- 运算对象可以是
  - 常量，例如 sizeof(5)
  - 变量，例如 sizeof(x)
  - 数据类型名，例如 sizeof(float)

假设定义了：int a; 那么：

- 表达式 sizeof(a) 的值为变量 a 的长度，等于 4
- 表达式 sizeof(int) 表示整型 int 的长度，等于 4
- 表达式 sizeof(double) 表示 doulbe 类型数据的长度，等于 8

运算符优先级

优先级	名称	运算符	功能和特点
1. 初等类	括号	()	可改变优先级顺序
	下标	[]	数组下标
	指针	$->$	结构指针引用成员
	指针	.	结构体引用成员

运算符优先级

优先级	名称	运算符	功能和特点
1. 初等类	括号	()	可改变优先级顺序
	下标	[]	数组下标
	指针	$->$	结构指针引用成员
	指针	.	结构体引用成员
2. 单目类 右 $\rightarrow$ 左结合	逻辑非	!	逻辑取反
	按位反	$\sim$	按位取反
	正号	$+$	指定符号为正
	负号	$-$	取相反值
	类型转换	(类型名)	强制转换类型
	地址	&	去变量地址
	自增	$++$	自增
	自减	$--$	自减
	长度	sizeof	取长度/字节数

运算符优先级

优先级	名称	运算符	功能和特点
3. 算术乘除	乘号	*	算数运算
	除号	/	
	余数	%	
4. 算术加减	加号	+	
	减号	−	
5. 移位类	左移	<<	向左移位
	右移	>>	向右移位
6. 关系比较	大于	>	结果为逻辑值
	小于	<	
	不小于	>=	
	不大于	<=	
7. 相等比较	等于	==	
	不等	!=	

运算符优先级

优先级	名称	运算符	功能和特点
8	按位与	&	位逻辑运算
9	按位异或	^	
10	按位或	|	
11	逻辑与	&&	逻辑运算
12	逻辑或	||	
13	条件运算	? :	右 → 左结合
14	赋值	= += -= *= /= %= &= ^= |= >>= <<=	右 → 左结合
15	逗号运算符	,	构造逗号表达式

字符处理程序解析

读入一行字符，统计以空格分隔的单词个数。

```
int cnt = 0, word = 0;
char ch;
printf("输入一行字符: ");
while( (ch = getchar()) != '\n' ) {
    if( ch==' ' ) //单词分隔符
        word = 0; //清除单词标志
    else if( word==0 ) {//开始新的单词
        word = 1; //设置单词标志
        cnt++;    //增加单词个数
    }
}
printf("单词个数=%d\n", cnt);
```

字符处理程序解析

读入一行字符，统计以空格分隔的单词个数。

```
int cnt = 0, word = 0;
char ch;
printf("输入一行字符: ");
while( (ch = getchar()) != '\n' ) {
    if( ch==' ' ) //单词分隔符
        word = 0; //清除单词标志
    else if( word==0 ) {//开始新的单词
        word = 1; //设置单词标志
        cnt++;    //增加单词个数
    }
}
printf("单词个数=%d\n", cnt);
```

字符处理程序解析

读入一行字符，统计以空格分隔的单词个数。

```
int cnt = 0, word = 0;
char ch;
printf("输入一行字符: ");
while( (ch = getchar()) != '\n' ) {
    if( ch==' ' ) //单词分隔符
        word = 0; //清除单词标志
    else if( word==0 ) {//开始新的单词
        word = 1; //设置单词标志
        cnt++;    //增加单词个数
    }
}
printf("单词个数=%d\n", cnt);
```

字符处理程序解析

```
while( (ch = getchar()) != '\n' )
```

能否替换为下面的代码？

```
while( ch = getchar() != '\n' )
```

为什么？

总结

- 小数的二进制

总结

- 小数的二进制
- 浮点数表示

总结

- 小数的二进制
- 浮点数表示
- 位运算、位逻辑、移位运算

总结

- 小数的二进制
- 浮点数表示
- 位运算、位逻辑、移位运算
- 应用位运算，解析浮点数

今天到此为止