程序设计实习

C++ 面向对象程序设计

张勤健
zqj@pku.edu.cn

北京大学信息科学技术学院

2024 年 2 月 21 日

大纲

qsort 库函数[1]

```c
/* qsort example */
#include <stdio.h>  /* printf */
#include <stdlib.h> /* qsort */

int values[] = {40, 10, 100, 90, 20, 25};
int compare(const void *a, const void *b) {
  return (*(int *)a - *(int *)b);
}

int main() {
  int i;
  qsort(values, 6, sizeof(int), compare);
  for (i = 0; i < 6; i++)
    printf("%d ", values[i]);
  return 0;
}
```

[1] http://www.cplusplus.com/reference/cstdlib/qsort/

qsort 库函数

C 语言快速排序库函数：

```
1  void qsort(void *base, size_t num, size_t size, int (*compar)(const void *, const void *));
```

可以对任意类型的数组进行排序

qsort 库函数

C 语言快速排序库函数：

```
1  void qsort(void *base, size_t num, size_t size, int (*compar)(const void *, const void *));
```

可以对任意类型的数组进行排序

- base: 数组起始地址

qsort 库函数

C 语言快速排序库函数：

```
1  void qsort(void *base, size_t num, size_t size, int (*compar)(const void *, const void *));
```

可以对任意类型的数组进行排序

- base: 数组起始地址
- num: 数组元素的个数

qsort 库函数

C 语言快速排序库函数：

```
1  void qsort(void *base, size_t num, size_t size, int (*compar)(const void *, const void *));
```

可以对任意类型的数组进行排序

- base: 数组起始地址
- num: 数组元素的个数
- size: 每个元素的大小（由此可以算出每个元素的地址）

qsort 库函数

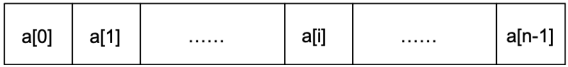
C 语言快速排序库函数：

```
1  void qsort(void *base, size_t num, size_t size, int (*compar)(const void *, const void *));
```

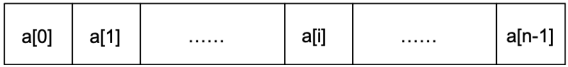
可以对任意类型的数组进行排序

- base: 数组起始地址
- num: 数组元素的个数
- size: 每个元素的大小（由此可以算出每个元素的地址）
- compar: 元素谁在前谁在后的规则

函数指针和 qsort 库函数

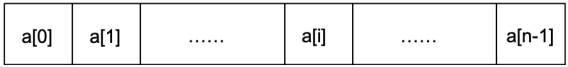


函数指针和 qsort 库函数



对数组排序，需要知道：

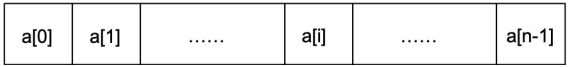
函数指针和 qsort 库函数



对数组排序，需要知道：

- 数组起始地址

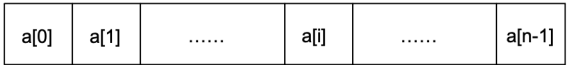
函数指针和 qsort 库函数



对数组排序，需要知道：

- 数组起始地址
- 数组元素的个数

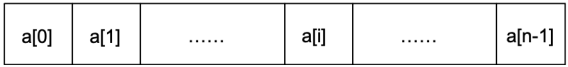
函数指针和 qsort 库函数



对数组排序，需要知道：

- 数组起始地址
- 数组元素的个数
- 每个元素的大小（由此可以算出每个元素的地址）

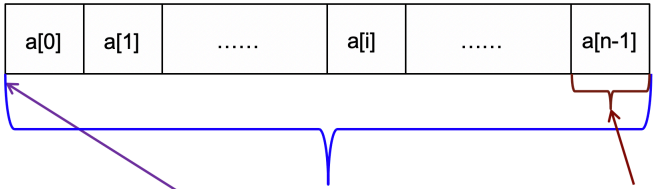
函数指针和 qsort 库函数



对数组排序，需要知道：

- 数组起始地址
- 数组元素的个数
- 每个元素的大小（由此可以算出每个元素的地址）
- 元素谁在前谁在后的规则

函数指针和 qsort 库函数




```
void qsort(void *base, int nelem, unsigned int width,
    int ( * pfCompare)( const void *, const void *));
```

base: 待排序数组的起始地址，
nelem: 待排序数组的元素个数，
width: 待排序数组的每个元素的大小（以字节为单位）
pfCompare :比较函数的地址

函数指针和 qsort 库函数

```
1  void qsort(void *base, int nelem, unsigned int width, int (*pfCompare)(const void *, const void *));
```

pfCompare: 函数指针，它指向一个“比较函数”。
该比较函数应为以下形式：

```
int 函数名 (const void *elem1, const void *elem2);
```

比较函数是程序员自己编写的

函数指针和 qsort 库函数

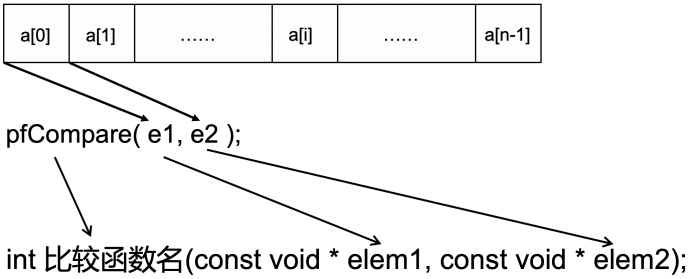
排序就是一个不断比较并交换位置的过程。

qsort 函数在执行期间，会通过 pfCompare 指针调用“比较函数”，调用时将要比较的两个元素的地址传给“比较函数”，然后根据“比较函数”返回值判断两个元素哪个更应该排在前面。

函数指针和 qsort 库函数

排序就是一个不断比较并交换位置的过程。

qsort 函数在执行期间，会通过 pfCompare 指针调用“比较函数”，调用时将要比较的两个元素的地址传给“比较函数”，然后根据“比较函数”返回值判断两个元素哪个更应该排在前面。



比较函数编写规则：

1) 如果 * elem1应该排在 * elem2前面，则函数返回值是负整数
2) 如果 * elem1和* elem2哪个排在前面都行，那么函数返回0
3) 如果 * elem1应该排在 * elem2后面，则函数返回值是正整数

函数指针和 qsort 库函数

实例：

下面的程序，功能是调用 qsort 库函数，将一个 unsigned int 数组按照个位数从小到大进行排序。比如 8、23、15 三个数，按个位数从小到大排序，就应该是 23、15、8

函数指针和 qsort 库函数

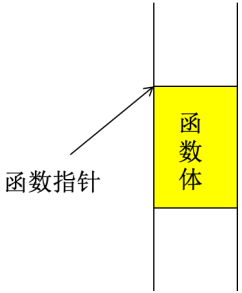
实例：

下面的程序，功能是调用 qsort 库函数，将一个 unsigned int 数组按照个位数从小到大进行排序。比如 8、23、15 三个数，按个位数从小到大排序，就应该是 23、15、8

```c
#include <stdio.h>
#include <stdlib.h>
int MyCompare(const void *elem1, const void *elem2) {
    unsigned int *p1, *p2;
    p1 = (unsigned int *)elem1; // "* elem1" 非法
    p2 = (unsigned int *)elem2; // "* elem2" 非法
    return (*p1 % 10) - (*p2 % 10);
}
#define NUM 5
int main() {
    unsigned int an[NUM] = {8, 123, 11, 10, 4};
    qsort(an, NUM, sizeof(unsigned int), MyCompare);
    for (int i = 0; i < NUM; i++)
        printf("%u ", an[i]);
    return 0;
}
```

基本概念

程序运行期间，每个函数都会占用一段连续的内存空间。而函数名就是该函数所占内存区域的起始地址 (也称“入口地址”)。我们可以将函数的入口地址赋给一个指针变量，使该指针变量指向该函数。然后通过指针变量就可以调用这个函数。这种指向函数的指针变量称为“函数指针”。



定义形式

```
类型名 (* 指针变量名)(参数类型 1, 参数类型 2, ...);
```

定义形式

```
类型名 (* 指针变量名)(参数类型 1, 参数类型 2, ...);
```

例如：

```
1 int (*pf)(int, char);
```

定义形式

```
类型名 (* 指针变量名)(参数类型 1, 参数类型 2, ...);
```

例如：

```
1 int (*pf)(int, char);
```

表示 pf 是一个函数指针，它所指向的函数，返回值类型应是 int，该函数应有两个参数，第一个是 int 类型，第二个是 char 类型。

使用方法

可以用一个原型匹配的函数的名字给一个函数指针赋值。
要通过函数指针调用它所指向的函数，写法为：

```
函数指针名（实参表）;
```

使用方法

可以用一个原型匹配的函数的名字给一个函数指针赋值。
要通过函数指针调用它所指向的函数，写法为：

```
函数指针名 (实参表);
```

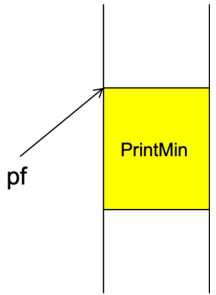
```c
#include <stdio.h>
void printMin(int a, int b) {
  if (a < b)
    printf("%d", a);
  else
    printf("%d", b);
}
int main() {
  void (*pf)(int, int);
  int x = 4, y = 5;
  pf = printMin;
  pf(x, y);
  return 0;
}
```

使用方法

可以用一个原型匹配的函数的名字给一个函数指针赋值。
要通过函数指针调用它所指向的函数，写法为：

```
函数指针名 (实参表);
```

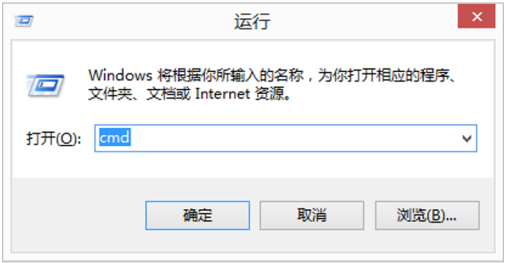
```c
#include <stdio.h>
void printMin(int a, int b) {
  if (a < b)
    printf("%d", a);
  else
    printf("%d", b);
}
int main() {
  void (*pf)(int, int);
  int x = 4, y = 5;
  pf = printMin;
  pf(x, y);
  return 0;
}
```



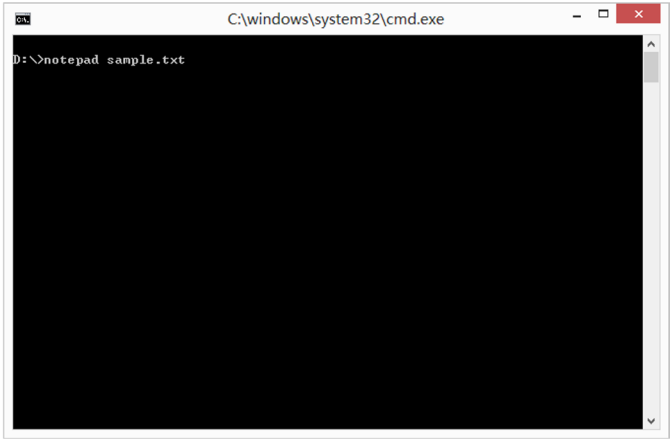
输出结果:
4

命令行方式运行程序

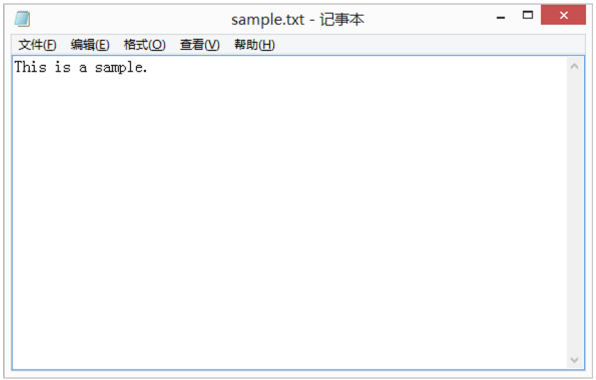
Windows + R 键：



命令行方式运行程序



命令行方式运行程序



命令行参数

将用户在 CMD 窗口输入可执行文件名的方式启动程序时，跟在可执行文件名后面的那些字符串，称为“命令行参数”。命令行参数可以有多个，以空格分隔。比如，在 CMD 窗口敲：

```
notepad sample.txt
```

命令行参数

将用户在 CMD 窗口输入可执行文件名的方式启动程序时，跟在可执行文件名后面的那些字符串，称为“命令行参数”。命令行参数可以有多个，以空格分隔。比如，在 CMD 窗口敲：

```
notepad sample.txt
```

其中：“notepad”，“sample.txt”就是命令行参数
如何在程序中获得命令行参数呢?

命令行参数

```
1  int main(int argc, char *argv[]) {
2    //code
3  }
```

命令行参数

```
1  int main(int argc, char *argv[]) {
2    //code
3  }
```

argc: 代表启动程序时，命令行参数的个数。C/C++ 语言规定，可执行程序程序本身的文件名，也算一个命令行参数，因此，argc 的值至少是 1。

命令行参数

```
1  int main(int argc, char *argv[]) {
2    //code
3  }
```

argc: 代表启动程序时，命令行参数的个数。C/C++ 语言规定，可执行程序程序本身的文件名，也算一个命令行参数，因此，argc 的值至少是 1。
argv: 指针数组，其中的每个元素都是一个 char* 类型的指针，该指针指向一个字符串，这个字符串里就存放着命令行参数。

命令行参数

```
1  int main(int argc, char *argv[]) {
2    //code
3  }
```

argc: 代表启动程序时，命令行参数的个数。C/C++ 语言规定，可执行程序程序本身的文件名，也算一个命令行参数，因此，argc 的值至少是 1。
argv: 指针数组，其中的每个元素都是一个 char* 类型的指针，该指针指向一个字符串，这个字符串里就存放着命令行参数。
例如，argv[0] 指向的字符串就是第一个命令行参数，即可执行程序的执行路径名，argv[1] 指向第二个命令行参数，argv[2] 指向第三个命令行参数……。

命令行参数

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  for (int i = 0; i < argc; ++i)
    printf("%s\n", argv[i]);
  return 0;
}
```

命令行参数

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  for (int i = 0; i < argc; ++i)
    printf("%s\n", argv[i]);
  return 0;
}
```

将上面的程序编译成 sample.exe，然后在控制台窗口敲:

```
sample.exe para1 para2 s.txt 5 "hello world"
```

命令行参数

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  for (int i = 0; i < argc; ++i)
    printf("%s\n", argv[i]);
  return 0;
}
```

将上面的程序编译成 sample.exe，然后在控制台窗口敲:

```
sample.exe para1 para2 s.txt 5 "hello world"
```

输出结果就是:

```
sample.exe
para1
para2
s.txt
5
hello world
```

命令行重定向

重定向输出 (>)

```
test.exe > out.txt
```

将标准输出重定向到 out.txt 文件中

重定向输入 (<)

```
test.exe < in.txt
```

将 in.txt 文件中的内容重定向到标准输入中

位运算基本概念

位运算：用于对整数类型（`int`,`char`,`long` 等）变量中的某一位 (bit)，或者若干位进行操作。

位运算基本概念

位运算：用于对整数类型（`int`,`char`,`long` 等）变量中的某一位 (bit)，或者若干位进行操作。
比如:

1. 判断某一位是否为 1
2. 只改变其中某一位，而保持其他位都不变。

位运算基本概念

位运算：用于对整数类型（`int`,`char`,`long` 等）变量中的某一位 (bit)，或者若干位进行操作。
比如:

1. 判断某一位是否为 1
2. 只改变其中某一位，而保持其他位都不变。

C/C++ 语言提供了六种位运算符来进行位运算操作：

- `&` 按位与 (双目)

位运算基本概念

位运算：用于对整数类型（`int`,`char`,`long` 等）变量中的某一位 (bit)，或者若干位进行操作。
比如:

1. 判断某一位是否为 1
2. 只改变其中某一位，而保持其他位都不变。

C/C++ 语言提供了六种位运算符来进行位运算操作：

- `&` 按位与 (双目)
- `^` 按位异或 (双目)

位运算基本概念

位运算：用于对整数类型（`int`,`char`,`long` 等）变量中的某一位 (bit)，或者若干位进行操作。
比如:

1. 判断某一位是否为 1
2. 只改变其中某一位，而保持其他位都不变。

C/C++ 语言提供了六种位运算符来进行位运算操作：

- `&` 按位与 (双目)
- `^` 按位异或 (双目)
- `|` 按位或 (双目)

位运算基本概念

位运算：用于对整数类型（`int`,`char`,`long` 等）变量中的某一位 (bit)，或者若干位进行操作。
比如:

1. 判断某一位是否为 1
2. 只改变其中某一位，而保持其他位都不变。

C/C++ 语言提供了六种位运算符来进行位运算操作：

- `&` 按位与 (双目)
- `^` 按位异或 (双目)
- `|` 按位或 (双目)
- `~` 按位非 (取反)(单目)

位运算基本概念

位运算：用于对整数类型（`int`,`char`,`long` 等）变量中的某一位 (bit)，或者若干位进行操作。
比如:

1. 判断某一位是否为 1
2. 只改变其中某一位，而保持其他位都不变。

C/C++ 语言提供了六种位运算符来进行位运算操作：

- `&` 按位与 (双目)
- `^` 按位异或 (双目)
- `|` 按位或 (双目)
- `~` 按位非 (取反)(单目)
- `>>` 左移 (双目)

位运算基本概念

位运算：用于对整数类型（`int`,`char`,`long` 等）变量中的某一位 (bit)，或者若干位进行操作。
比如:

1. 判断某一位是否为 1
2. 只改变其中某一位，而保持其他位都不变。

C/C++ 语言提供了六种位运算符来进行位运算操作：

- `&` 按位与 (双目)
- `^` 按位异或 (双目)
- `|` 按位或 (双目)
- `~` 按位非 (取反)(单目)
- `>>` 左移 (双目)
- `<<` 右移 (双目)

按位与“&”

将参与运算的两操作数各对应的二进制位进行与操作，只有对应的两个二进位均为 1 时，结果的对应二进制位才为 1，否则为 0。

按位与“&”

将参与运算的两操作数各对应的二进制位进行与操作，只有对应的两个二进位均为 1 时，结果的对应二进制位才为 1，否则为 0。
例如：表达式“21 & 18”的计算结果是 16(即二进制数 10000)，因为：

21 用二进制表示就是：
0000 0000 0000 0000 0000 0000 0001 0101
18 用二进制表示就是:
0000 0000 0000 0000 0000 0000 0001 0010
二者按位与所得结果是：
0000 0000 0000 0000 0000 0000 0001 0000

按位与“&”

通常用来将某变量中的某些位清 0 且同时保留其他位不变。也可以用来获取某变量中的某一位。

按位与“&”

通常用来将某变量中的某些位清 0 且同时保留其他位不变。也可以用来获取某变量中的某一位。

例如，如果需要将`int`型变量 n 的低 8 位全置成 0，而其余位不变，则可以执行：

```
1  n = n & 0xffffff00;
```

按位与“&”

通常用来将某变量中的某些位清 0 且同时保留其他位不变。也可以用来获取某变量中的某一位。

例如，如果需要将`int`型变量 n 的低 8 位全置成 0，而其余位不变，则可以执行：

```
1  n = n & 0xffffff00;
```

也可以写成：

```
1  n &= 0xffffff00;
```

按位与“&”

通常用来将某变量中的某些位清 0 且同时保留其他位不变。也可以用来获取某变量中的某一位。

例如，如果需要将`int`型变量 n 的低 8 位全置成 0，而其余位不变，则可以执行：

```
1  n = n & 0xffffff00;
```

也可以写成：

```
1  n &= 0xffffff00;
```

如果 n 是 short 类型的，则只需执行：

```
1  n &= 0xff00;
```

按位与“&”

如何判断一个`int`型变量 n 二进制表示中的第 7 位（从右往左）是否是 1？

按位与“&”

如何判断一个`int`型变量 n 二进制表示中的第 7 位（从右往左）是否是 1？
只需看表达式`n & 0x40` 的值是否等于 $0x40$ 即可。
$0x40$: 0100 0000

按位或“|”

将参与运算的两操作数各对应的二进制位进行或操作，只有对应的两个二进位都为 0 时，结果的对应二进制位才是 0，否则为 1。

按位或“|”

将参与运算的两操作数各对应的二进制位进行或操作，只有对应的两个二进位都为 0 时，结果的对应二进制位才是 0，否则为 1。
例如：表达式“`21 | 18`”的计算结果是 23，因为：
21:
0000 0000 0000 0000 0000 0000 0001 0101
18:
0000 0000 0000 0000 0000 0000 0001 0010
`21 | 18`:
0000 0000 0000 0000 0000 0000 0001 0111

按位或“|”

按位或运算通常用来将某变量中的某些位置 1 且保留其他位不变。

按位或“|”

按位或运算通常用来将某变量中的某些位置 1 且保留其他位不变。
例如，如果需要将 int 型变量 n 的低 8 位全置成 1，而其余位不变，则可以执行：

```
1  n |= 0xff;
```

0xff: 1111 1111

按位异或“^”

将参与运算的两操作数各对应的二进制位进行异或操作，即只有对应的两个二进位不相同时，结果的对应二进制位才是 1，否则为 0。

按位异或“^”

将参与运算的两操作数各对应的二进制位进行异或操作，即只有对应的两个二进位不相同时，结果的对应二进制位才是 1，否则为 0。
例如：表达式“21 ^ 18 ”的值是 7(即二进制数 111)。:
21：
0000 0000 0000 0000 0000 0000 0001 0101
18：
0000 0000 0000 0000 0000 0000 0001 0010
21 ^ 18:
0000 0000 0000 0000 0000 0000 0000 0111

按位异或“^”

按位异或运算通常用来将某变量中的某些位取反，且保留其他位不变。

按位异或“^”

按位异或运算通常用来将某变量中的某些位取反，且保留其他位不变。
例如，如果需要将int型变量 n 的低 8 位取反，而其余位不变，则可以执行：

```
1  n ^= 0xff;
```

0xff: 1111 1111

按位异或“^”

异或运算的特点是：

如果 `a ^ b = c`，那么就有 `c ^ a = b`以及`c ^ b = a`
此规律可以用来进行最简单的加密和解密。

按位异或“^”

另外异或运算还能实现不通过临时变量，就能交换两个变量的值：

```
1  int a = 5, b = 7;
2  a = a ^ b;
3  b = b ^ a;
4  a = a ^ b;
```

即实现 a,b 值交换。

按位非“~”

按位非运算符“~”是单目运算符。其功能是将操作数中的二进制位 0 变成 1，1 变成 0。

按位非“~”

按位非运算符“~”是单目运算符。其功能是将操作数中的二进制位 0 变成 1，1 变成 0。
例如，表达式“~21”的值是整型数 $0xffffffea$：
21：
0000 0000 0000 0000 0000 0000 0001 0101
~21:
1111 1111 1111 1111 1111 1111 1110 1010

左移运算符“<<”

```
1  a << b
```

表示将 a 各二进位全部左移 b 位后得到的值。左移时，高位丢弃，低位补 0。a 的值不因运算而改变。

左移运算符“<<”

```
1  a << b
```

表示将 a 各二进位全部左移 b 位后得到的值。左移时，高位丢弃，低位补 0。a 的值不因运算而改变。

例如:

```
1  9 << 4
```

9 的二进制形式：

0000 0000 0000 0000 0000 0000 0000 1001

因此，表达式9 << 4的值，就是将上面的二进制数左移 4 位，得：

0000 0000 0000 0000 0000 0000 1001 0000

即为十进制的 144。

左移运算符“<<”

实际上，左移 1 位，就等于是乘以 2，左移 n 位，就等于是乘以 2^n。而左移操作比乘法操作快得多。

右移运算符“>>”

```
1  a >> b
```

表示将 a 各二进位全部右移 b 位后得到的值。右移时，移出最右边的位就被丢弃。a 的值不因运算而改变。

右移运算符“>>”

```
1  a >> b
```

表示将 a 各二进位全部右移 b 位后得到的值。右移时，移出最右边的位就被丢弃。a 的值不因运算而改变。

对于有符号数，如`long`,`int`,`short`,`char`类型变量，在右移时，符号位（即最高位）将一起移动，并且大多数 C/C++ 编译器规定，如果原符号位为 1，则右移时高位就补充 1，原符号位为 0，则右移时高位就补充 0。

右移运算符“>>”

```
1  a >> b
```

表示将 a 各二进位全部右移 b 位后得到的值。右移时，移出最右边的位就被丢弃。a 的值不因运算而改变。

对于有符号数，如`long`,`int`,`short`,`char`类型变量，在右移时，符号位（即最高位）将一起移动，并且大多数 C/C++ 编译器规定，如果原符号位为 1，则右移时高位就补充 1，原符号位为 0，则右移时高位就补充 0。

实际上，右移 n 位，就相当于左操作数除以 2^n，并且将结果往小里取整。

```
-25 >> 4 = -2
-2 >> 4 = -1
18 >> 4 = 1
```

右移运算符“>> ”

```c
#include <stdio.h>
int main() {
  int n1 = 15;
  short n2 = -15;
  unsigned short n3 = 0xffe0;
  char c = 15;
  n1 = n1 >> 2;
  n2 >>= 3;
  n3 >>= 4;
  c >>= 3;
  printf("n1=%d,n2=%x,n3=%x,c=%x", n1, n2, n3, c);
}
```

右移运算符“>> ”

```c
#include <stdio.h>
int main() {
  int n1 = 15;
  short n2 = -15;
  unsigned short n3 = 0xffe0;
  char c = 15;
  n1 = n1 >> 2;
  n2 >>= 3;
  n3 >>= 4;
  c >>= 3;
  printf("n1=%d,n2=%x,n3=%x,c=%x", n1, n2, n3, c);
}
```

n1: 0000 0000 0000 0000 0000 0000 0000 1111

n1 >>= 2: 变成 3; 0000 0000 0000 0000 0000 0000 0000 0011

右移运算符“>> ”

```c
#include <stdio.h>
int main() {
  int n1 = 15;
  short n2 = -15;
  unsigned short n3 = 0xffe0;
  char c = 15;
  n1 = n1 >> 2;
  n2 >>= 3;
  n3 >>= 4;
  c >>= 3;
  printf("n1=%d,n2=%x,n3=%x,c=%x", n1, n2, n3, c);
}
```

n1: 0000 0000 0000 0000 0000 0000 0000 1111

n1 >>= 2: 变成 3; 0000 0000 0000 0000 0000 0000 0000 0011

n2: 1111 1111 1111 0001

n2 >>= 3: 变成 0xfffffffe, 即-2; 1111 1111 1111 1110

右移运算符“>> ”

```c
#include <stdio.h>
int main() {
  int n1 = 15;
  short n2 = -15;
  unsigned short n3 = 0xffe0;
  char c = 15;
  n1 = n1 >> 2;
  n2 >>= 3;
  n3 >>= 4;
  c >>= 3;
  printf("n1=%d,n2=%x,n3=%x,c=%x", n1, n2, n3, c);
}
```

n1: 0000 0000 0000 0000 0000 0000 0000 1111

n1 >>= 2: 变成 3; 0000 0000 0000 0000 0000 0000 0000 0011

n2: 1111 1111 1111 0001

n2 >>= 3: 变成 0xfffffffe, 即-2; 1111 1111 1111 1110

n3: 1111 1111 1110 0000

n3 >>= 4: 变成 0xffe; 0000 1111 1111 1110

右移运算符“>> ”

```c
#include <stdio.h>
int main() {
  int n1 = 15;
  short n2 = -15;
  unsigned short n3 = 0xffe0;
  char c = 15;
  n1 = n1 >> 2;
  n2 >>= 3;
  n3 >>= 4;
  c >>= 3;
  printf("n1=%d,n2=%x,n3=%x,c=%x", n1, n2, n3, c);
}
```

n1: 0000 0000 0000 0000 0000 0000 0000 1111

n1 >>= 2: 变成 3; 0000 0000 0000 0000 0000 0000 0000 0011

n2: 1111 1111 1111 0001

n2 >>= 3: 变成 0xfffffffe, 即-2; 1111 1111 1111 1110

n3: 1111 1111 1110 0000

n3 >>= 4: 变成 0xffe; 0000 1111 1111 1110

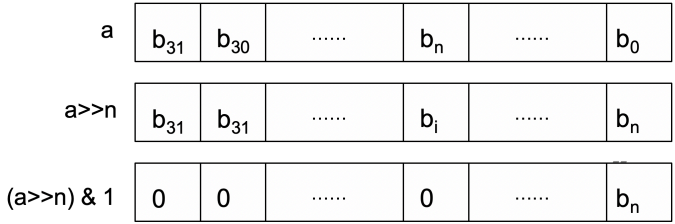
c: 0000 1111

思考题

有两个 int 型的变量 a 和 $n(0 \leq n \leq 31)$，要求写一个表达式，使该表达式的值和 a 的第 n 位相同。

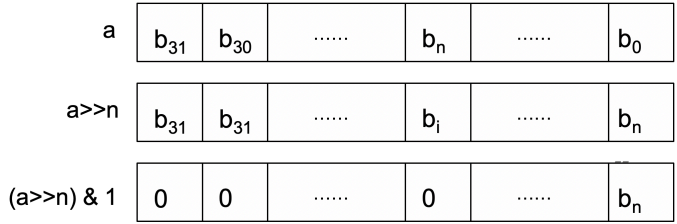
思考题

有两个 int 型的变量 a 和 $n(0 \leq n \leq 31)$，要求写一个表达式，使该表达式的值和 a 的第 n 位相同。



思考题

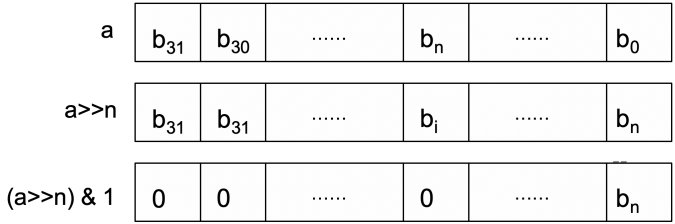
有两个 int 型的变量 a 和 $n(0 \leq n \leq 31)$，要求写一个表达式，使该表达式的值和 a 的第 n 位相同。



答案：`(a >> n) & 1`

思考题

有两个 int 型的变量 a 和 $n(0 \leq n \leq 31)$，要求写一个表达式，使该表达式的值和 a 的第 n 位相同。



答案：`(a >> n) & 1`

思考：另一解法：`(a & (1 << n)) >> n`?