1.进制

古代的人为了记录数量，有的在石头上留下划痕，有的在绳子上打结。

例如，有 8 个东西，就记录 8 次：

`||||||||`

这种方法很简单：**有几个，就记几次。**

但如果要表示 100、1000，甚至更大的数量，就需要重复大量的计数行为。

于是，人们开始创造新的符号，让一个符号能够表示更多的数量。

例如，罗马数字中：

- `Ⅰ` 表示 1；
- `Ⅴ` 表示 5；
- `X` 表示 10；
- `C` 表示 100。

这样，539 可以写成：

`CCCCCXXXVIIII`

相比划痕 539 次，已经方便了很多。

但是仔细观察就会发现，**重复的问题仍然存在**：

`CCCCC` 是 5 个 `C`，`XXX` 是 3 个 `X`，`IIII` 是 4 个 `I`。

所以 `CCCCCXXXVIIII` 本质上是在说：

`5 个 C + 3 个 X + 1 个 V + 4 个 I`

既然，我们有更大的计数单位，我们可以直接记录：

`V 个 C+III 个 X+I 个 V+IIII 个 I`

这种方法稍显复杂：**符号既告诉我们 “有几个”，又告诉我们 “几个什么”。**

但如果要表示 100000000000，甚至更大的数量，就需要创建出大量的符号。

于是，人们创造了**进制**，**用符号告诉我们 “有几个”，用位置告诉我们 “几个什么”。**

1.1 十进制

十进制，规定从右向左的位置依次表示：1、10、100、...。每向左移动一位，所表示的单位就变成原来的 10 倍。

这样，每 10 个都可以变为后一位置上的 1，我们只需要 `0～9` 这 10 个数字，再利用它们所在的**位置**，就能够表示任意大的整数。

例如，`539 = 5 * 100 + 3 * 10 + 9`

1.2 二进制

那么，每向左一位一定要变成 10 倍吗？

当然不是。

如果规定每向左一位变成原来的 **2 倍**，各个位置表示的单位就变成：1、2、4、...。

这时，每个位置只需要表示 “0 个” 或 “1 个”，因此只需要 `0` 和 `1` 两个数字，再利用它们所在的**位置**，就能够表示任意大的整数。

这就是**二进制**。

所以，进制最重要的思想并不是记住某个公式，而是：

> **用数字表示数量，用位置表示单位。**

不同的进制，只是规定了**向左移动一位，单位变成原来的多少倍**。

1.3 十六进制

十六进制，规定从右向左的位置依次表示：1、16、256、...。每向左移动一位，所表示的单位就变成原来的 16 倍。

这样，每 16 个都可以变为更大位置上的 1。但 `0～9` 这 10 个数字还不够，所以增加了 A、B、C、D、E、F 这几个字母分别表示 `10～15`。

例如，`5AB = 5 * 256 + 10 * 16 + 11`。

1.4 十进制转其他进制

以二进制为例，假设数量 x 一开始都在个位。

由于，逢 2 进 1，则要往前进 $\lfloor x/2 \rfloor$，剩下 $x\%2$。

不妨令 $x = \lfloor x/2 \rfloor$，第二位继续往前进 $\lfloor x/2 \rfloor$，剩下 $x\%2$

以此类推，直到 x 为 0。

```C++
int x;
cin >> x;
while(x){
```

```
    cout << x % 2;
    x /= 2;
}
```

1.5 小数

人类最初以 “一” 作为基本计数单位，后来发现存在比一更小的量，且可以无限细分。为了表示这些量，我们在整数后加上小数点 “.”，引入小数部分。在十进制中，小数点后每一位的权值依次为 0.1，0.01，…，仍然遵循“逢十进一”的规则。进制中，小数点后的数字仍然满足逢十进一，小数点后一位的权为零点一、第二位的权是零点零一，…。

类似的二进制下的小数部分同样也用 '.'（点）与整数部分隔开。小数点后一位的代表零点五、第二位代表零点二五，…。例如，$(0.011)_2$ = 一 × 零点二五 + 一 × 零点一二五 = 零点三七五 = $(0.375)_{10}$

2.二进制编码

二进制就像家里的电灯开关，只有 “开” 和 “关” 两种状态，简单明了，不容易出错。而十进制就像风扇的档位旋钮，要让它稳稳地停在 1 到 10 档的每一档上，机械结构会变得非常复杂。为此，计算机在设计时采用的是二进制，也就是说保存数据时，都会转换成二进制进行保存。

那么具体如何用二进制表示各种数值呢？则需要了解进制编码，毕竟单纯的二进制也无法表示负数。

2.1 原码

原码（original code）是一种二进制编码。它规定最高位为符号位，其中 0 表示正数，1 表示负数，其余位的数字表示二进制数本身。

例如，当二进制数位只有 4 位时，可以表示的原码如图 2.1.1 所示:

	正数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

	负数
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

图 2.1.1

原码虽能够简单的表示数值，但它在运算上具有缺陷：$(+2)+(-2)=(0010)_{原}+(1010)_{原}=(1100)_{原}=-4$。也就是说，它的符号位无法参与运算。因此，紧接着人类发明了 "反码"。

2.2 反码

反码（ones' complement)

- 正数：与原码相同（最高位为 0，后面是数值位）
- 负数：符号位仍然为 1，但其他位与原码相反。

下图 2.2.1 展示 4 位二进制反码的情况：

图 2.2.1

在 4 位反码中，任何一个数字 a 与其相反数 $-a$ 相加，结果为 $(1111)_{反}$，即 $(-0)_{10}$。

如果将 a 或 $-a$ 逐渐的减小 1，第一次减小，结果变为 $(1110)_{反}$，即 $(-1)_{10}$；第二次减小，结果变为 $(1101)_2$，即 $(-2)_{10}$，以此类推，可以发现所有结果计算正确。

如果将 a 逐渐的增大 1，第一次增大，结果变为 $(1000)_{反}$，即 $(0)_2$；第二次增大，结果变为 $(1001)_{反}$，即 $(1)_2$，以此类推，可以发现所有结果总是会少 1。**这个问题归根结底，是因为 0 有两种表述方式。所以，接着产生了 "补码"。**

2.3 补码

补码（two's complement）

- 正数：与反码相同。
- 负数：在反码的基础上加 1。

下图 2.3.1 展示 4 位二进制补码的情况：

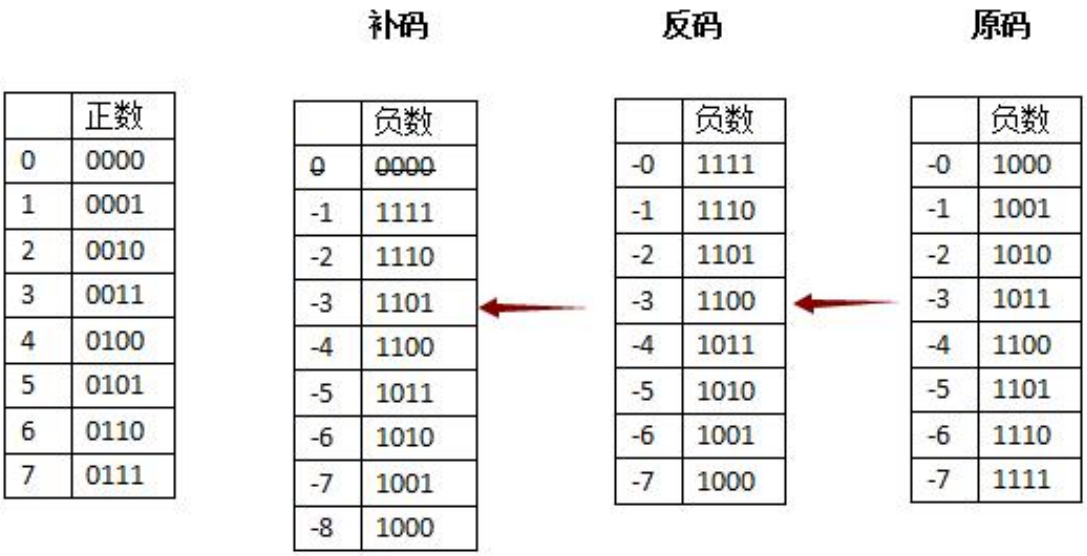

图 2.3.1

因为，在 4 位反码中，任何一个数字 a 与其相反数 $-a$ 相加，结果为 $(1111)_{反}$。而补码对负数 + 1，结果则为 $(0000)_{补}$，即 $(0)_{10}$。

如果将 a 逐渐的减小 1，第一次减小，结果变为 $(1111)_{补}$，即 $(-1)_{10}$；第二次减小，结果变为 $(1110)_{补}$，即 $(-2)_{10}$，以此类推，可以发现所有值运算正确。

如果将 a 逐渐的增大 1，第一次增大，结果变为 $(0001)_{补}$，即 $(1)_{10}$；第二次增大，结果变为 $(0010)_{补}$，即 $(2)_{10}$，以此类推，可以发现所有值运算正确。

至此，我们不仅解决了正负相加的问题，也解决了 +0 和 -0 同时存在的问题。需要注意的是，C++ 采用补码来对二进制位进行解释，已没有原码、反码的概念，这个概念更多是方便人们理解。

真题

1. 下列二进制表示的十进制数值分别是（ ）

[10000011]原=（ ）
[10000011]补=（ ）

A. -125，-3
B. -3, -125
C. -3，-3
D. -125，-125

2. 关于计算机中的编码，下列说法中错误的是（）

A. 对于无符号数，原码就是真值

B. 正数的反码是其本身
C. 负数的反码和补码是不一样的
D. 负数的反码，在其原码的基础上，各个位取反

3. 在 8 位二进制原码表示中，八进制数 -5 的二进制形式是什么（ ）

A. 10000101
B. 11111010
C. 11111011
D. 00000101

4. 整数-5 的 16 位补码表示是( )。

A. 1005

B. 1006

C. FFFA

D. FFFB

5. 如果 16 位短整数 -2 的二进制是"FFFE"，则短整数 -4 的十六进制是( )。

A. FF04

B. FFFA

C. FFFC

D. FFFH

6. 8 位二进制原码能表示的最小整数是：（ ）

A.-127

B.-128

C.-255

D.-256

7. 反码表示中，零的表示形式有：

A. 1 种

B. 2 种

C. 8 种

D. 16 种

8. 若 X 的 8 位补码为 0000 1010，则 X/2 的补码是（ ）。

A. 0000 0101

B. 1000 0101

C. 0000 0101 或 1000 0101

D. 算术右移后结果取决于符号位

9. 若 X 的 8 位补码为 1000 1010，则 X/2 的补码是（ ）。

A. 1000 0101

B. 1100 0101

C. 1100 0101 或 0000 0101

D. 算术右移后结果取决于符号位

3.内存

程序运行时的数据都存储在内存中。

3.1 计算单位

内存中计算位置大小的最小单位是**比特（bit）**，每一比特可以存下一位 0/1。这正是因为，计算机是用二进制形式表示数据的。较大一点的单位是**字节（Byte）**，1 B = 8 b。

比特和字节还可以和 K（千）、M（兆）、G（吉）、T（太）组成更大的计量单位。这就好比使用 g（克）来称重，也可以用 Kg（千克）来称重似的。唯一不同的是，在平常 K 一般表示 1000，在计算机中 K 表示为 1024，M = 1024 K，G = 1024 M，T = 1024 G。组合起来例如，1 KB = 1024 B、1 MB = 8192 Kb 等。

基本的数据类型 `char` 占用 1 B，`int`、`float` 占用 4 B，`long long`、`double` 占用 8 B。其中，`char`、`int`、`long long` 等整数类型都采用二进制补码的方式进行翻译。通过补码的定义很轻松的算出可存储的数据范围。例如，`int` 类型，占用 4 个字节，即 32 位。32 位补码最小范围则是最高位为 1，其他位为 0 的情况（只有最高位的权为负数，其他位都为正数）。那么最小范围则是 -2^{31}。最大范围则是，最高位为 0，其他位为 1 的情况。那么最大范围则是 $2^0+2^1+2^2+\ldots+2^{30}=2^{31}-1$。

表 3.1.1 给出常用数据类型的范围：

C/C++ 数据类型	最小值	最大值
char	-128	127
int	-2147483648 约为 -2e9	2147483647 约为 2e9
long long	-9223372036854775808 约为 -9e18	9223372036854775807 约为 9e18

3.2 无符号数

在 C++ 中，整数类型又可以分为 **有符号数（signed）** 和 **无符号数（unsigned）**，它们的主要区别在于能否表示负数。所有的类型默认都是有符号数。例如，`int` 实际上是 `signed int`；`long long` 实际上是 `signed long long` 都是有符号数，可以表示负数、0、正数。

当配合 `unsigned` 创建整型变量，例如 `unsigned int`、`unsigned long long` 等，代表这是一个无符号数，它只能表示非负数。对于它的二进制位计算机直接用二进制本身进行翻译。例如，$(11111111)_2 = 256$。

表 3.2.1 给出常用数据类型的范围：

C/C++ 数据类型	最小值	最大值
unsigned char	0	255
unsigned int	0	4294967295
unsigned long long	0	18446744073709551615

例题

1. int a = 1234，b = -1234567;int c = a + b; cout << (unsigned int) c; 请问输出的是什么？

A.-1233333 B. 4293733963 C.12354564 D.42723616

2. 一个 32 位整型变量占用（）个字节。

A. 32

B. 128

C. 4

D. 8

3. 32 位 int 类型的存储范围是（）？

A. -2147483647 ~ +2147483647

B. -2147483647 ~ +2147483648

C. -2147483648 ~ +2147483647

D. -2147483648 ~ +2147483648

4. 记 1KB 为 1024 字节（byte），1MB 为 1024KB，那么 1MB 是多少二进制位

（bit）？（）

A. 1000000

B. 1048576

C. 8000000

D. 8388608

5. 移动公司推出套餐 10Mb/s 的套餐，请问一秒能传输大概多少字节（ ）

A. 10000000

B. 1000000

C. 80000000

D. 8000000

4.位运算

位运算，是 C++ 中用于对整数在二进制编码形式下的运算。

4.1 左移运算

左移运算符号为 `<<`。

使用方式：`操作数 1 << 操作数 2`。

作用：计算出**操作数 1** 在二进制编码下整体向左移动**操作数 2** 位（末尾补 0） 的结果。

在数据保证移位后不会溢出时，每左移一位相当于将操作数 1 代表的数值乘以 2。

例如：`5 << 2 = 0101 << 2 = 010100 = 20`；`-10 << 2 = 10110 << 2 = 1011000 = -40`。

4.2 右移运算

右移运算符号为 `>>`。

使用方式： `操作数 1 >> 操作数 2`,

作用：计算出**操作数 2** 在二进制编码下整体向右移动**操作数 2** 位（开头补同样的数字） 的结果。

每右移一位相当于将操作数 1 代表的数值除以 2 并向下取整。

例如：`5 >> 2 = 0101 >> 2 = 0001 = 1`；`-10 >> 2 = 10110 >> 2 = 11101 = -3`。

4.3 按位与运算

按位与运算符号为 `&`。

使用方式：`操作数 1 & 操作数 2`。

作用：计算出两个操作数在二进制编码下**逐位**作与运算后的结果。

例如 `-7 & 3 = 1001 & 0011 = 0001 = 1` 。其计算过程如图 4.3.1 所示：

```
  1001
& 0011
------
  0001
```

图 4.3.1

运算律：

1. 满足交换律。`a & b = b & a`。
2. 满足结合律。`(a & b) & c = a & (b & c)`。

4.4 按位或运算

按位或运算符号为 `|`。

使用方式：`操作数 1 | 操作数 2`。

作用：计算出两个操作数在二进制编码下**逐位**作或运算后的结果。

例如 `-6 | 3 = 1010 | 0011 = 1011 = -5` 。其计算过程如图 4.4.1 所示：

```
  1010
| 0011
------
  1011
```

图 4.4.1

运算律：

1. 满足交换律。`a | b = b | a`。
2. 满足结合律。`(a | b) | c = a | (b | c)`。

4.5 按位非运算

按位非运算符号为 `~`。

使用方式：`~操作数`。

作用：计算出操作数在二进制编码下逐位作非运算后的结果。

例如 `~-1 = ~1111 = 0000 = 0` 、`~0 = ~0000 = 1111 = -1`、`~7 = ~0111 = 1000 = -8`。

4.6 按位异或运算

按位异或运算符号为 `^`。

使用方式：`操作数 1 ^ 操作数 2`。

作用：计算出两个操作数在二进制编码下逐位作异或运算后的结果。异或的规则为：相同为 0，不同为 1。

例如 `-6 ^ 3 = 1010 ^ 0011 = 1001 = -7`。其计算过程如 图 4.6.1 所示：

```
  1010
^ 0011
------
  1001
```

图 2.1.6.1

运算律：

1. 满足交换律。`a ^ b = b ^ a`。
2. 满足结合律。`(a ^ b) ^ c = a ^ (b ^ c)`。

真题练习

1. 二进制数 11 1011 1001 0111 和 01 0110 1110 1011 进行按位与运算的结果是（）。

A. 01 0010 1000 1011

B. 01 0010 1001 0011

C. 01 0010 1000 0001

D. 01 0010 1000 0011

2. 二进制数 1010 | 1100 的结果是（）

A. 1000

B. 1110

C. 1010

D. 1100

3. 以下哪个位运算可以交换两个变量的值（无需临时变量）( )

A. a = a ^ b; b = a ^ b; a = a ^ b;

B. a = a & b; b = a | b; a = a & b;

C. a = a | b; b = a ^ b; a = a ^ b;

D. a = ~a; b = ~b; a = ~a;

4. 以下代码的说法正确的是什么( )

```
#include <iostream>
using namespace std;
int main() {
int a = 0b1101;
int b = 0b1011;
cout << (a ^ b);
return 0;
}
```

A. 进行的是整体异或运算

B. 进行的是按位同或运算

C. 进行的是按位与运算

D. 进行的是按位异或运算

5. 补码 1111 1101 进行运算 1111 1101>>1 以后得到的结果是 （）

A. 1111 1100

B.-2

C. 1111 1101

D. 1111 1010

6. a|10（a 与 10 都是 10 进制，且二进制表示最高位为 1）运算的结果是（ ）。

A. 使 a 的二进制表示从右往左的第二位为 1

B. 使 a 的二进制表示从右往左的第一位为 0

C. 使 a 的二进制表示从右往左第二位为 0

D. 使 a 的二进制表示最高位为 0

4.7 基本操作

`x & 0`

`x & 1`

`x | 1`

`a ^ 0`

`a ^ 100 ^ 100`

T1

```
Plain Text
#include <iostream>
using namespace std;
int main()
{
    signed char x,y;
    cin >> x >> y;

    unsigned int z = (x << 16) + (y >> 8);

    cout << z;

    return 0;
}
```

假设输入的 x,y 均不会超过 char 范围，完成下面的判断题和单选题：

判断题

1. 第 8 行，因为 `x` 是字符类型，只能存储 8 位。所以 `x << 16` 会溢出，程序出现未定义行为 (　)

【考点分析】出现隐式转换，自动将 char 升级为 int，8 位变为 32 位，不会产生溢出。

2. 题目结果与 `y` 的输入无关。

【考点分析】因为字符的 ASCII 码值一定是个 0 或整数，所以右移会往最高位补 0，最后结果一定为 0。

3. 题目不可能输出 0 和负数。

【考点分析】unsigne int 能表示 0

4. 若把第 8 行，`unsigned int` 改为 `unsigned short` 程序不会溢出。

【考点分析】short 一共 16 位，会溢出。

5. 当 `x` 输入 127 的时候，程序输出的值最大。

【考点分析】因为 x、y 是 char，输入的时候，x 只会读取 1，y 只会读取 2。

T2

```
Plain Text
#include <iostream>
using namespace std;
int main()
{
    signed int x,y;
    cin >> x >> y;

    unsigned long long  z = (x << 18ll) + (y >> 31);

    cout << z;

    return 0;
}
```

假设输入的 x, y 均不会超过 signed int 范围，完成下面的判断题和单选题：

判断题

1. 因为 x 是 int 类型，只能存储 32 位，所以 x << 18ll 有几率溢出 (　)

【考点分析】**错误。**18ll 代表长整型的 18，经过隐式转换，运算结果变为长整型可以存储 64 位，所以不会溢出。

2. y 无论输入什么，y >> 31 的结果不变 （　）

【考点分析】**错误。**

y 是 signed int，y >> 31 是算术右移（高位补符号位）。

如果 y >= 0（最高位为 0），y >> 31 结果为 0。

如果 y < 0（最高位为 1），y >> 31 结果为 -1（即 0xFFFFFFFF）。

因此，y >> 31 的结果取决于 y 的符号位，不是固定的。

3. 当 x 为 -1 时，z 达到最大，与 y 的值毫无联系 （　　）

【考点分析】**错误。**当 x = -1 时，(x << 18ll) 的结果为 $111110000\underbrace{}_{46\text{个}}\underbrace{}_{18\text{个}}$。当 y >> 31 的结果为 -1 时，二进制为 $11111\underbrace{}_{64\text{个}}$，结果就为 $1111101111\underbrace{}_{45\text{个}}\underbrace{}_{18\text{个}}$。当 y >> 31 的结果为 0 时，二进制为 $000\underbrace{}_{64\text{个}}$，结果就为 $111110000\underbrace{}_{46\text{个}}\underbrace{}_{18\text{个}}$。比较可得，当 y 为输入的是 0 或正数的时候达到最大。

4. 若把 y 的数据类型更改成 signed char（保证输入的是 ASCII 中的字符），那么

`z` 的值与 `y` 无关 （　）

【考点分析】**正确。**因为 ASCII 值只能是 0 或正数。

5.地址

在计算机中，内存自动的会给每个字节大小的位置分配一个地址，地址是一串十六进制的数字。也就是说除了标识符可以找到相应的盒子，通过地址也能找到相应的盒子。这就和我们的家庭小区一样，通过小区名可以找到小区，通过小区的街道也可以找到小区。

而由于，一个基本的数据类型可能会占用多个字节（例如，`int`、`float` 占用 4 B），一个盒子则会占用多个地址，C++ 会用最小的地址来代表这个位置，称为**首地址**。

一个变量的首地址可以通过取地址运算符获得，符号为 `&`。

使用方式：`&num`。

作用：将 `num` 代表的内存区域的首地址作为运算结果。

例如，

```
C++
代码块
// 程序 1.1.1
#include<bits/stdc++.h>
using namespace std;
int main(){
    int x = 5;

    cout << x << " " << &x;

    return 0;
}
```

该程序在作者电脑的结果为：

```
C++
5
0x6ffe4c
```

5.1 解地址运算符

通过解地址运算符，可以使用某个地址上的内存区域。

解地址运算符，符号为 `*`。

使用方式：`*num`，`num` 必须是一个地址。

运算结果：将相应的内存区域作为运算结果。

例如，

C++
代码块
```
// 程序 2.1
#include<bits/stdc++.h>
using namespace std;
int main(){
    int x = 5;
    cout << x << endl;
    *(&x) = 10;
    cout << x;

    return 0;
}
```

程序输出：

```
C++
5
10
```

5.2 指针

为了区分地址和整数，C++ 设定地址为指针数据类型，只有指针数据类型的变量才能存储地址。我们只需要定义指针的地址类型、标识符，就可以得到指针变量。例如，`int *a`，它的含义是定义一个标识符为 `a`，存储的是一个整型类型的地址。

大家可能会好奇，为什么指针不是这样定义：`pointer name`。因为，不同的数据类型占用的位置大小不同，仅通过首地址无法确定内存区域的范围。

5.3 指针传递

在函数调用时，值传递是不会影响到实参本身的。但通过指针传递，就可以在被调函数中通过地址找到相应的变量，就会影响到实参本身。

```
C++
```

代码块

```
//程序 3.1.1
#include<bits/stdc++.h>
using namespace std;

void swap(int *x,int *y){
    int t = *x;
    *x = *y;
    *y = t;
    return;
}

int main(){
    int a = 4,b = 7;
    Function(&a,&b)
    cout << a << " " << b;

    return 0;
}
```

该程序输出结果为：

```
C++
7 4
```

5.4 引用传递

使用指针传递，在语义上并不直观。例如在 `swap()` 函数中，我们的本意是交换两个变量的值，却不得不传递它们的地址——这种写法偏离了实际意图，容易让人产生误解。

相比之下，使用引用传递（如 `swap(a, b)`）则直接明了：参数 `x` 和 `y` 作为变量 `a` 和 `b` 的别名，直指同一个盒子。虽然引用的本质仍然依赖于指针机制，但它从语法层面隐藏了底层技术细节，让代码更贴近我们的思维方式。

引用变量的定义格式也很简洁：`int &x = y;` 表示 `x` 是 `y` 的引用（别名），它们共用同一块内存空间。

5.5 指针运算

指针可以与整数进行加减运算。它会让指针以 地址类型所占的字节大小” 为步长增大或减小。例如，

```c++
代码块
// 程序 13.5.1
#include<bits/stdc++.h>
using namespace std;
int main(){
    int arr[5] = {10, 20, 30, 40, 50};
    int *p = &arr[0];      // 指向 arr[0]

    cout << p + 0 << " " << p + 1 << " " << p + 2 << endl;

    long long brr[5] = {10, 20, 30, 40, 50};
    long long *p2 = &brr[0];      // 指向 arr[0]

    cout << p2 + 0 << " " << p2 + 1 << " " << p2 + 2 << endl;
    return 0;
}
```

最后程序输出为：

```c++
0x6ffe10 0x6ffe14 0x6ffe18
0x6ffde0 0x6ffde8 0x6ffdf0
```

指向 `int` 的指针，每次加一都是加 4。指向 long long 的指针，每次加一都是加 8。

5.6 数组与指针

数组可以通过 `[]`+ 编号选出元素，根本原因是元素的地址是连续的，使得任意一个元素的首地址，都可以通过第一个元素的首地址推导出来。例如，`int a[10];`。`a[0]` 是第一个元素，其首地址是 `&a[0]`。`a[1]` 的首地址就可以表示为 `&a[0] + 1`，`a[2]` 的首地址就可以表示为 `&a[0] + 2`，以此类推。所以，数组与指针是密切联系的。

数组名在使用时，会隐式转换为指向其第一个存储单元的指针常量。也就是说，如果 `a` 是一个数组，那么 `a` 和 `&a[0]` 是等价的；`a + i`、`i + a` 和 `&a[i]` 是等价的。更进一步说，`*(a + i)` 、`*(i + a)`、 `a[i]` 和 `i[a]` 是等价的。

所以，`[]` 元素选择符和 `%` 模运算很像，都是一个复合的运算过程。编号选择符的本质就是加法运算符和解地址运算符的复合操作。例如，

```C++
代码块
// 程序 13.6.1
```

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    int a[100] = {1,2,3,4,5,6};

    for(int i = 0; i < 5; i++){
        cout << a + i << " " << &a[i] << " " << *(a + i) << endl;
        cout << " " << *(i + a) << " " << a[i] << " " << i[a];
    }

    return 0;
}
```

该程序在作者电脑输出的结果为：

```
C++
0x6ffc90 0x6ffc90 1 1 1 1
0x6ffc94 0x6ffc94 2 2 2 2
0x6ffc98 0x6ffc98 3 3 3 3
0x6ffc9c 0x6ffc9c 4 4 4 4
0x6ffca0 0x6ffca0 5 5 5 5
```

5.7 函数和数组

利用数组的存储单元地址连续的特性，我们只需要在函数中传递数组的第一个存储单元的首地址，即可在函数中使用数组。例如，

```
c++
代码块
//程序 13.7.1
#include<bits/stdc++.h>
using namespace std;

void Function(int *a,int (*b)[20],int x){
    for(int i = 0; i < x; i++){
        a[i] = i;
    }
    for(int i = 0; i < x; i++){
        b[i][i] = 20;
    }
}

int x[5],y[5][20];
```

```
int main(){

    Function(x,y,5);
    for(int i = 0; i <= 4; i++){
        cout << x[i] << " " << y[i][i] <<< endl;
    }
    return 0;
}
```

该程序输出结果为:

```
C++
0 20
1 20
2 20
3 20
4 20
```

实际上，在函数的形参中，以下三种声明完全等价:

```
c++
void Function(int a[10]);    // 看似接收数组，实际是指针 10 会被编译器忽略。
void Function(int a[]);      // 更明确的指针形式
void Function(int* a);       // 最本质的形式
```