第1章 基本数据类型，输入和输出

In [1]:
```
#include <iostream>
using namespace std;
```

第1节 输入和输出

In [2]:
```
int a = 5;
int b;
char c;
```

In [3]:
```
scanf("%d,%c",&b,&c);
```

In [4]:
```
printf("a=%d,b=%d,c=%d",a,b,c)
```

```
a=5,b=0,c=0
```

Out[4]: 11

由于notebook的原因，这里无法正常输入，因此这里单独解释一下，你需要按照双引号里面的顺序
依次进行输入，比如上面需要依次输入b的值，逗号，c的值，如果不按照里面的格式输入，b和c无法正确获得。
然后printf输出三个值后，还有一个返回值，由于返回值不重要，这里不提了。如果我结尾添加一个分号，这个返回值就不会输出

格式控制:
%nd按大于等于n的宽度输出
%.nf按n位小数输出

In [5]:
```
printf("hello%3dworld%.3f",a,float(a));
```

```
hello  5world5.000
```

使用cin和cout输入输出，注意cin和cout是两个实例，与<<和>>运算符使用，返回的类型还是cin的类型，
到底是什么类型，后面还会再说。另外，这里使用cin居然能进行输入，比scanf好用。

In [6]:
```
float f;
```

In [7]:
```
cin >> f;
```

In [8]:
```
cout << b << endl << c;
```

```
0
```

In [9]:
```
cout << "hello world";
```

```
hello world
```

In [10]:
```
ostream& hahaha = cout;
hahaha << "hello world";
```

```
hello world
```

关于cout的格式控制，也很简单，就是在<<后面添加set宽度或者set精度参数，就能返回一个设置好格式的cout对象，
然后继续使用<<输出你想输出的东西,这里就拿设计输出精度来举例说明

In [11]:
```
hahaha << setprecision(3) << f;
```

```
3.14
```

In [12]:
```
hahaha.operator<<(f);
```

```
3.14
```

这里就基本输入输出进行说明，到了后面的对象以及IO（真正讲解输入和输出的章节）的时候，这两个对象还会详细讲解。
你在这里可能懵逼，怎么hahaha还能当cout来用，以后你就明白了。这里的<<和>>也就是普通的operator罢了，后面还能
重载换成别的,而且，它也只是一个普通的函数罢了，上面我就通函数调用的方式使用了它，不要觉得神奇，这是类与对象最基础的概念。

In [13]:
```
int c;
int count=0;
while((c=cin.get())!=EOF & count<=5){
    count+=1;
    cout << c << endl<< endl;
}
```

```
97

65

10

98

98

98
```

cin.get()从输入流中一个一个取字符，第一次我输入了2个字符+Enter回车（换行符，ascii码是10），然后cin.get()依次取出3个字符，并分别输出，这里执行了3次循环；然后我输入bbbbb回车，输出了3次，也就是执行了3次循环，说明只成功读了3次字符，后面就不读了。
这是因为循环条件对于count的取值范围是0，1，2，3，4，5，区间就是这6个数字，只能执行6次读取。

另外EOF的ascii码是FF，也就是-1，这里c用int表示，是因为这样输入的所有字符保存到c里面后都一定是个正数，不会是-1，那么这样如何
结束循环呢？注意cin.get()的返回值有两种情况，一种是读取到字符，返回字符，一种是从输入流读不到东西了，这时候返回-1，
并不是说输入流里面结尾有个EOF这么一个标志存在，没有这个东西，包括文件结尾，没有这个标志，仅仅就是这个函数读不到东西时返回-1；
此外，由于这里的交互式环境没法中断输入，所以我加了count这么一个变量控制循环，在一般c++编译环境中，按ctrl+z后再按enter键就能
结束输入流。

上面的cin.get()可以用scanf("%c",&c)代替，并且速度更快，适用于大量数据输入输出。

In [14]:
```
c='h'
```

Out[14]:
```
104
```

这里这个赋值表达式也是有返回值的，这就是上面为什么能用(c=cin.get())!=EOF这个语句的原因，python里面赋值表达式是没有返回值的。

第2节 数据的表示

这里先定义一个16进制圆盘，用来表示一个4bit的整数，圆盘均匀分为16份，分别是0~F，其中F为1111.
这个圆盘可用于表示0~15的无符号整数，也可以表示-8~7的有符号整数，通过这个圆盘，你可以学习补码相关知识。
由于现在的计算机都是按字节编码，最小的单位就是1个字节，所以一个数据类型最少都是1个字节，也就是8位，没有半字节的数据类型。其次，对于32位机器来说，寻址范围就是1024x1024x1024x4个最小单位，也就是这么多字节，
也就是4GB空间，上面的32位是指地址线可表示32位数，然后就能寻找到这么多的单位，也就是这么多个字节。

由于半个字节这里不方便研究，这里主要研究1个字节和2个字节的整数的表示，分别是char和short类型，下面举几个例子

In [15]:
```
char a = 'a';
printf("%c %d",a,a);
```

```
a 97
```

字母a背后的16进制表示就是61，对应10进制就是97，对应2进制就是0110 0001（用8421表示法迅速表示出6，然后表示出1）
后面各种数据类型，第一一定要反应出它的十六进制和二进制表示，然后才能继续分析。
对于char这种字符型数据，用1个字节表示，范围就是-128~127，对于0~127点字符，可以查看ASCII编码表，
这里先记住A=41(65),a=61(97),其他的记不住的,你直接print打出来就行了,也不用每次翻ascii码表.
另外,从10进制到16进制和二进制,一定要多练,这个是基本功

In [16]:
```
printf("%d",sizeof(a));
printf("\n");
printf("%lu",sizeof(char));
```

```
input_line_27:2:14: warning: format specifies type 'int' but the argument has type 'unsigned long' [-Wformat]
 printf("%d",sizeof(a));
         ~~  ^~~~~~~~~
         %lu
1
1
```

这里输出变量a和char类型的字节大小，对于第一句，警告sizeof返回的是unsigned long类型，但是我用%d也就是int有符号整型进行输出，
然后提示我用%lu，也就是unsigned long的输出格式控制；由于这里两种整数的正数表示完全一样，所以无所谓，输出结果一定是正确的。
这种警告虽然看上去在这里不好看，影响排版，但是会无时无刻提醒你，哪里可能有问题，然后你要思考为什么出现这种问题，是否有影响。
另外，这里long和int是不一样的，分别是8个字节和4个字节，对于有的编译器，两个都是4个字节，一般用不上long，忽略。

In [17]:
```
printf("%lu",sizeof(int));
printf("\n");
printf("%lu",sizeof(long));
```

```
4
8
```

然后下面看一下2个字节的无符号整数和带符号整数，首先2个字节的无符号整数范围是0~65535，带符号整数范围-32768~32767.

In [18]:
```
unsigned short b = 65535 ;
unsigned short c = b+2;
printf("%d\n",b+2);          // b+2中的2会按照4个字节表示，求和结果也是4个字节，因此65537可以正确算出来
printf("%d",c);              // c定义为无符号short类型，因此b+2还是无符号short类型，65535+1=0，因此加2等于1
```

```
65537
1
```

Out[18]: 1

In [19]:
```
unsigned short b = 0xffff ;
unsigned short c = b+2;
printf("%d\n",b+2);
printf("%d",c);
```

```
65537
1
```

In [20]:
```
short b = 32767 ;
short c = b+1;
printf("%d\n",b+1);          // 这里2还是用4个字节表示，求和结果4个字节，没有溢出
printf("%d\n",c);              // 这里2个字节，结果溢出了
printf("%c",b-32670);
```

```
32768
-32768
a
```

注意，上面执行b+1或者b+2时，都是按4个字节的二进制相加的，然后赋值给short类型变量c时，截断，这就是背后的计算过程；
总之,同一个二进制可以用于编码不同的数据类型,这里主要讲了无符号整型(原码),有符号整型(补码),字符型,还没讲浮点数的表示

在加减乘除模运算中，两个数的运算结果的精度取决于这两个数中精度较高的那个，就是原来最多有多少个二进制格子，算完后还是多少个，下面用除法举例说明

In [21]:
```
float a ;
a = 10/3 ;  // 这里10和3都是整数，算出来也只会是整数3，然后再存到float变量a当中
a
```

Out[21]: 3.00000f

In [22]:
```
float b;
int c;
b = float(10)/3;  // 右边算出来是3.33，保存到float变量b中
c = float(10)/3;  // 右边算出来是3.33，保存到整型变量c中
cout << b << endl << c;
```

```
3.33
3
```

注意，整型变量和浮点型变量转换时，不是简单的截断二进制或者补0转换过去，而是有个复杂的转换，由于还没详细讲浮点数的表示，这里暂时不需要了解二进制如何转换的，对于10进制的转换如下： 3.33转换为整数，直接舍弃小数部分，3转换为浮点数，小数部分为0.

__关于自增运算符和自减运算符__
由于两者一样，这里以自增运算符为例说明,a++,和++a都会将a的值加1，但是a++表达式本身的返回值是a的老值，++a表达式返回的值是a的新值

In [23]:
```
int a = 0;
cout << (a++) << endl << a;
```

```
input_line_36:3:11: warning: unsequenced modification and access to 'a' [-Wunsequenced]
cout << (a++) << endl << a;
          ^              ~
```

```
0
1
```

In [24]:
```
int a = 0;
cout << (++a) << endl << a;
```

```
input_line_37:3:10: warning: unsequenced modification and access to 'a' [-Wunsequenced]
cout << (++a) << endl << a;
         ^              ~
```

```
1
1
```

关系运算符
计算结果是bool类型数据，只有ture和false两种，并且bool类型是用1个字节保存的，
true就是int的1，false就是int的0

In [25]:
```
bool e = (8>7);
cout << e << endl << sizeof(e);
```

```
1
1
```

In [26]:
```
bool e = (8<7);
cout << e << endl << sizeof(e);
```

```
0
1
```

然后随便举几个有符号数，无符号数比较的例子，顺带熟悉一下数据的基本表示

第一个例子，-1居然大于0，这是因为其中一个是无符号数，就按无符号数进行比较，
-1变成无符号数就是65535，第二个数0变成无符号数就是0，因此65535>0

In [27]:
```
unsigned short a = -1;
short b = 0;
cout << (a>b);
```

```
1
```

第二个例子，32768居然小于32767，这是因为short(32768u)=-32768,是个负数，比较的
两个数short(a)和b都是有符号数，就按有符号数比较

In [28]:
```
unsigned short a = 32768;
short b = 32767;
cout << (short(a)<b);
```

```
1
```

还有一些更加奇奇怪怪的结果，和编译器的处理规则有关，只有考试才会遇到，编程的时候尽量别搞这么繁琐。
类似cout << a+++b; 这么刁钻的表达式，到底是(a++)+b还是a+(++b)，可以告诉你是第一种，但是不需要
搞懂这种表达式编译器是怎么处理的，没什么用。

逻辑运算符
三种：与&&或||非!
返回值类型就是bool类型，只有true和false

强制类型转换
上面提到了两种情况，一种是赋值语句，一种是类似函数调用的形式，比如int(),float()，这两种都属于强制类型转换，就按正常的思维转换即可，容易和数据扩展与截断混淆，数据的扩展和截断一般发生在将不同精度的数据赋值或者一起运算时发生的，比如short赋值给int，
会自动扩展，高位补0，int赋值给short会自动截断，这种扩展和截断带来的数值变化可能非常离谱，变成差距很大的完全不同的另一个数，
而强制类型转换不是简单进行扩展和截断，而是考虑了原数值的大小，转换后一般和原来的数值相等或者差距不大。

表示论与编码
上面整数的表示有2进制，10进制，16进制等，然后$2^n, 10^n, 16^n$好像向量空间里面的基向量一样，进制表示好像坐标一样，
整数的集合看成一个抽象的代数，就是整数环，在整数环的结构下研究整数的表示，会涉及到更深的东西，这就是代数，表示论，编码理论，
通讯与密码学，代数方程，代数班的同学专门研究这个的，并且有更多的应用，我们这里从计算机基本数据类型角度来看，只需要简单掌握
基本数据类型（char，short，int等）的2进制，10进制，16进制的表示和转换即可，这个在后面计算机组成原理
和汇编中很重要，作为基本功，即使现在不熟悉，也要勤加练习。

浮点数的表示
由于这部分看了忘，忘了再看，而且本人又不是计算机专业的，经常用不上，所以肯定记不住的，这部分直接放弃，每次重新回顾太浪费时间了，
下面直接用例子讲清楚浮点数到底怎么表示的，不需要记住，今天看明白后，明天忘掉即可。

首先float是4字节表示的，也就是32位， 比如 0 10000011 10000000000000000000000 我靠，看得头皮发麻，这么长一串，还不用16进制表示，
我把这么一串32位二进制分成了3个部分，第一位表示符号，0代表正数，1代表负数，然后8位表示阶码，这里用E直接代替这8位二进制(无符号数)，然后后面23位表示
尾数(也是无符号数)，用M表示，那么这个数的大小规定为 +1.M x 2^{E+bias} = +1.M x 2^{E-127}，
比如对于这里，+1.M = +1.10000000000000000000000B = 1.5 ,然后2^{E+bias} = $2^{10000011B-127}$ = $2^{131-127}$=2^4=16，最终结果就是1.5x16=24;

注意，上面阶码变成无符号整数后，需要减去一个偏置常数bias，E+bias叫做这个阶码的反码，不用理会，你就当没反码这个概念，就按照前面的步骤理解就行了。

In [29]:
```
float a = 0b01000001110000000000000000000000;
printf("%f",a);
```

```
1103101952.000000
```

上面和预想的不一样的原因是，编译器会将二进制数0bxxxxxxxx视为整型变量，算出整数值，然后赋值给浮点类型变量a。
如果想将二进制原样写入内存，需要按照下面的方式直接进行逐位复制，将整数的二进制原模原样复制给浮点数的内存单元中。

In [30]:
```
#include <string.h>

float a;
unsigned int hexValue = 0x41c00000;  // 十六进制值

// 将十六进制值的字节内容复制到浮点数变量的内存中
memcpy(&a, &hexValue, sizeof(float));

printf("%f\n", a);  // 打印结果
```

```
24.000000
```

Out[30]: 10

ok，结果完全和预期的一样，然后再举一个例子：
1 01111101 11000000000000000000000
1表示负数
阶码01111101表示64+32+16+8+4+1=125，再减去偏置127等于-2
尾数11000000000000000000000就是1.11000000000000000000000B=1+0.5+0.25=1.75
因此这个数就是-1.75x2^{-2}=-0.4375

In [31]:
```
float b;
unsigned int binValue = 0b10111110111000000000000000000000;  // 二进制数值

// 将十六进制值的字节内容复制到浮点数变量的内存中
memcpy(&b, &binValue, sizeof(float));

printf("%f\n", b);  // 打印结果
```

```
-0.437500
```

Out[31]: 10

ok,结果也完全一致，虽然上面看上去好像把浮点数的表示讲完了，但是完全用上面的规则表示实数轴还存在一些问题，具体什么问题我们就不细说了，你可以自己探索，我们这里直接把完整的IEEE 754浮点数表示规则说一遍：
当阶码为0000 0000 时，这个数的大小规定为 ±0.M；
当阶码为0000 0001 ~ 1111 1110 时，这个数的大小规定为 ±1.M x 2^{E+bias} = +1.M x 2^{E-127}；
当阶码为1111 1111 且尾数全是0时，这个数大小规定为正负无穷大；
当阶码为1111 1111 且尾数不全为0时，这个编码规定不是一个数字，用NaN表示。

合理性分析，首先让尾数变成0，然后把8位阶码能表示的所有数在数轴上标出来，也就是：
负无穷，$-2^{128}, -2^{127}, \ldots, -2^{-126}, 0, 2^{-126}, 2^{-125}, \ldots, 2^{0}, 2^{1}, \ldots 2^{127}, 2^{128}$,正无穷

然后我们固定一个阶码进行分析，当符号位固定是0，阶码加偏置后是 -126 时，然后我们的23位尾数可以随便变化，
那么这个数x的范围就是1.M x 2^{-126} = 2^{-126} + 2^{-126} x 0.M;
也就是从2^{-126}出发，加上所有可能的2^{-126} x 0.M, 加上的这个数小于1，
于是可以推出上面的这个数x的范围就是2^{-126} 到2^{-125} 中间的一堆数，并且有2^{23}个，因为M是23位。

通过上面第一步，我们知道阶码可以将实数轴上从负无穷到正无穷中间的256个特殊的数表示出来，
然后通过第二步分析，我们发现这么多小区间，每个小区间里面一大堆数又能通过改变尾数进行表示，
于是乎，实数轴被我表示出来了，虽然中间不是均匀等分，也漏了很多数，但至少表示出了很多数了，精度尚可接受

In [32]:
```
float c;
unsigned int binValue = 0b01111111100000000000000000000000;  // 二进制数值

// 将十六进制值的字节内容复制到浮点数变量的内存中
memcpy(&c, &binValue, sizeof(float));

printf("%f\n", c);  // 打印结果
```

```
inf
```

Out[32]: 4

In [33]:
```
float d;
unsigned int binValue = 0b01111111100000000000000000001111;  // 二进制数值

// 将十六进制值的字节内容复制到浮点数变量的内存中
memcpy(&d, &binValue, sizeof(float));
```

```
printf("%f\n", d);  // 打印结果
```

```
nan
```

Out[33]: 4

ok,浮点数就到这里了，不管记不记得住，都不用管了，用得少自然忘得快，反正我是记不住。

__关于一些度量单位__
首先度量数据长度的单位有：
比特，位（bit），通常用小写b表示；
字节（Byte），通常用大写B表示；
字（Word），一般32位机器规定1字=16位，不同机器规定不一样；

然后度量机器数据通路长度的单位有：
字长，比如32位机器，64位机器，这里32位就是字长

容量度量单位：
1GB=1024MB=1024x1024KB=1024x1024x1024B
通信带宽：
1Gbps=1000Mbps=1000x1000Kbps=1000x1000x1000bps= 10^9 bit/s

__关于大端和小端存储__
首先我们的物理空间下面表示低地址，上面表示高地址
举例：一个int型变量0x12345678，如果小端存储，78放在最下面，如果是大端存储，12放在最下面
无论大端还是小端，数据地址起点都是从低地址开始，上面的int类型按小端存储，起点开始第一个字节就是78，
如果按大端存储，起点开始的第一个字节就是12，而union类型数据共享同一个存储空间，可以用来判断机器的存储方式。

In [34]:
```
union NUM{
    int a;
    char b;
};
```

In [38]:
```
NUM num;
num.a=0x12345678;
printf("%x\n",num.a);
printf("%x",num.b);
```

```
12345678
78
```

char和int共享同一个存储空间，因此char的第一个字节就是int的第一个字节，这里第一个字节存的是78，因此这是小端存储

In [ ]:
```

```