

C/C++的数据

Wang Houfeng

wanghf@pku.edu.cn

EECS, PKU

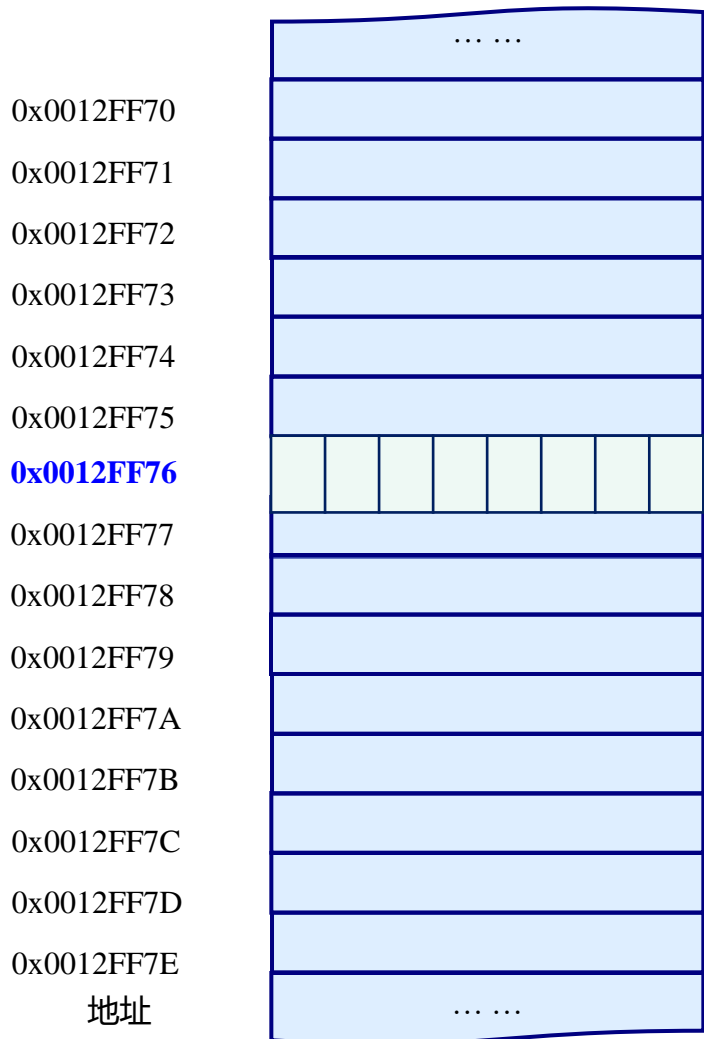
内容

➤ C/C++的基本数据类型

- 数据的输入输出简介

计算机中的数

存储器



所有信息（数值与符号）都以二进制形式表示

所有的数据都放在存储器中

存储器由若干单元构成

每个单元表示一个字节（8位）

很多数据需要占据多个字节（单元）

每个单元有一个编号，称为地址（位置）

编写程序时，可以为数据命一个名字，以避免记忆物理地址带来的困难

变量

程序中的名字称为标识符

数在计算机中的表示

1字节由8个2
进制位表示

- 所有数据在计算机中都以2进制形式表示；
- 计算机表示数通常以**字节（8位）**为最小单元；
- **数值**表示：最高位为符号位（0表示正，1为负）
 - 数值表示均以**补码**形式，补码的形成方式如下（以整数为例）：
 - ① 将10进制**绝对值**转换为2进制（如何转换？）；
 - ② 其绝对值的2进制表示形式称为原码（高位为0）；
 - ③ 正数的补码与原码形式完全相同；
 - ④ 负数的原码只需将最高位的符号为改为1即可；
 - ⑤ 负数的补码：在正数补码基础上各位先取反，然后末位加1

原码、补码、反码

- **原码**：最高位为符号位，其余各位为数值本身的绝对值
- **反码**：
 - 正数：反码与原码相同
 - 负数：符号位为1，其余位对原码取反
- **补码**：
 - 正数：原码、反码、补码相同
 - 负数：最高位为1，其余位为原码取反，再对末位数加1

例：用一个字节表示 **-46** 的补码（注意计算步骤）：

绝对值的原码： 0 0 1 0 1 1 1 0

实际原码： 1 0 1 0 1 1 1 0

反码： 1 1 0 1 0 0 0 1

补码： 1 1 0 1 0 0 1 0

于是： $[-46]_{\text{补}} = [1101\ 0010]_2 = [D2]_{16}$

原码：+0 和 -0 二种形式表示

-127 ~ +127（8位）

反码：+0 和 -0 表示不同

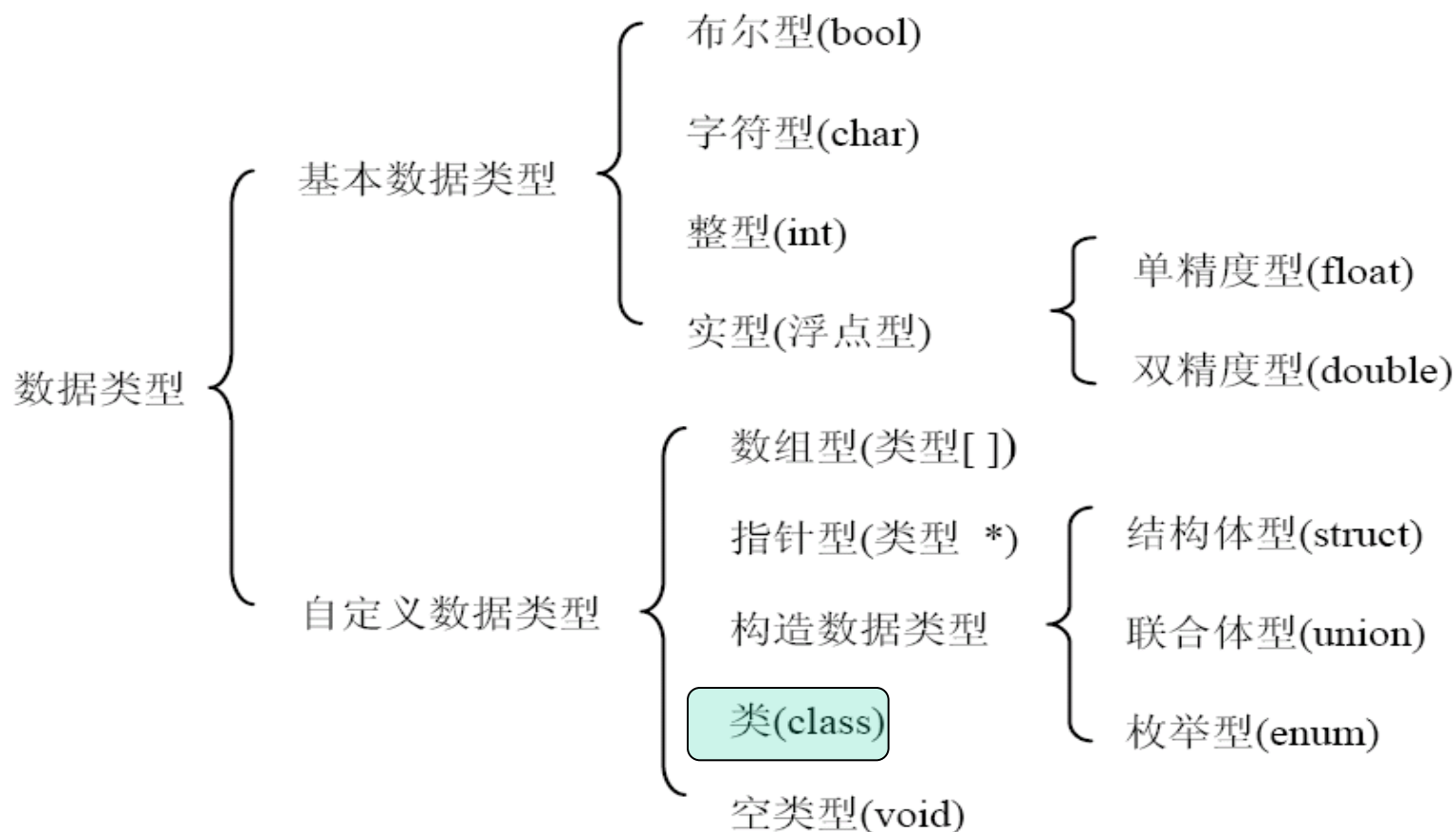
-127 ~ +127（8位）

补码：+0 和 -0 表示相同

-128 ~ +127（8位）

C/C++中数的类型

程序中的所有数都有类型，C/C++的类型如下：



4种基本类型数据的长度与范围

数据类型标识符	字节数	数值范围
bool	1	false, true
char	1	-128~127
signed char	1	-128~127
unsigned char	1	0~255
short [int]	2	-32 768~32 767
signed short [int]	2	-32 768~32 767
unsigned short [int]	2	0~65 535
int	4	-2 147 483 648~2 147 483 647
signed int	4	-2 147 483 648~2 147 483 647
unsigned int	4	0~4 294 967 295
long [int]	4	-2 147 483 648~2 147 483 647
signed long [int]	4	-2 147 483 648~2 147 483 647
unsigned long [int]	4	0~4 294 967 295
float	4	3.4e-38~3.4e38
double	8	1.7e-308~1.7e308
long double	8	1.7e-308~1.7e308

- C/C++标准没有具体规定以上各类数据所占内存字节数，只要求long型数据长度不短于int型，short型不长于int型。

整型



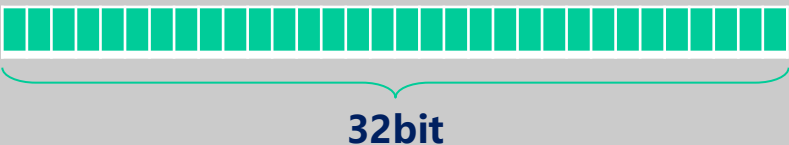
	整型
基本型	int
短整型	short short int
长整型	long long int

```
int a = 0;
```


VC中的整型长度

类型	长度	范围
short [int]	2	-32 768~32 767
signed short [int]	2	-32 768~32 767
unsigned short [int]	2	0~65 535
int	4	-2 147 483 648~2 147 483 647
signed int	4	-2 147 483 648~2 147 483 647
unsigned int	4	0~4 294 967 295
long [int]	4	-2 147 483 648~2 147 483 647
signed long [int]	4	-2 147 483 648~2 147 483 647
unsigned long [int]	4	0~4 294 967 295

整型长度图示

	整型	内存空间
基本型	int	 32bit
短整型	short short int	 16bit
长整型	long long int	 32bit

- VC: short (2 bytes) ; int/long (4 bytes)

标准C/C++ 没有具体规定以上各类数据所占内存字节数，
只要求long型数据长度不短于int型，short型不长于int型。

如何知道系统中某类型的长度

- 使用sizeof(): 计算某种类型所占的字节数

```
#include <iostream>
using namespace std;
int main()
{
    cout << "sizeof(short int)=" << sizeof(short int) << endl;
    cout << "sizeof(int)=" << sizeof(int) << endl;
    cout << "sizeof(long int)=" << sizeof(long int) << endl;
    return 0;
}
```

整型分为有符号和无符号

	有符号	无符号
基本型	int signed int	unsigned int
短整型	short short int signed short signed short int	unsigned short unsigned short int
长整型	long long int signed long signed long int	unsigned long unsigned long int

signed 可缺省

unsigned 不可缺省

有符号 vs. 无符号

unsigned int i = 123;



signed int i = -123; 原码 \longleftrightarrow 取反+1 \longleftrightarrow 补码

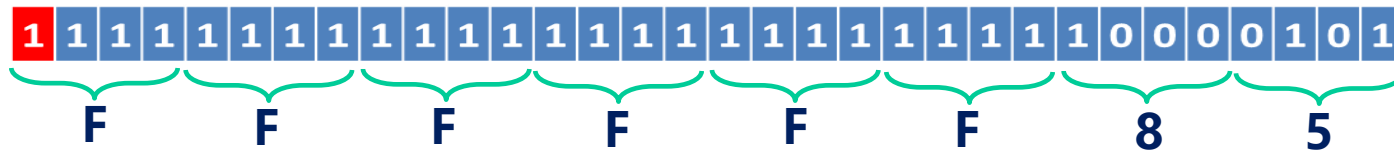


计算机内的数值均用补码表示（正数的补码等于原码值）

十六进制输出

signed int i = -123;

2进制-16进制：4位对1位



```
#include <iostream>
using namespace std;
int main()
```

```
{
```

```
    int a = -123;
```

```
    cout << hex << a << endl;
```

```
    return 0;
```

```
}
```

fffffff85

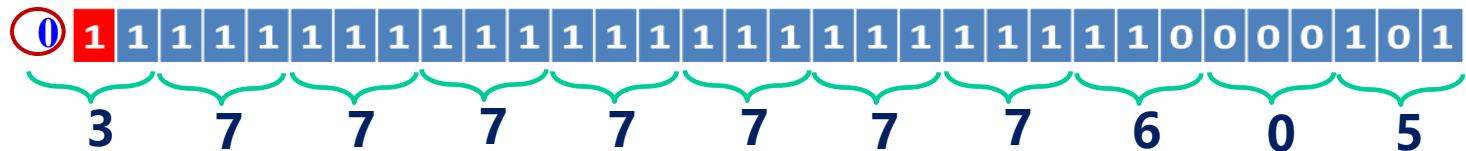
← 16进制

16进制中的16个符号：0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

八进制输出

signed int i = -123;

2进制-8进制：3位对1位



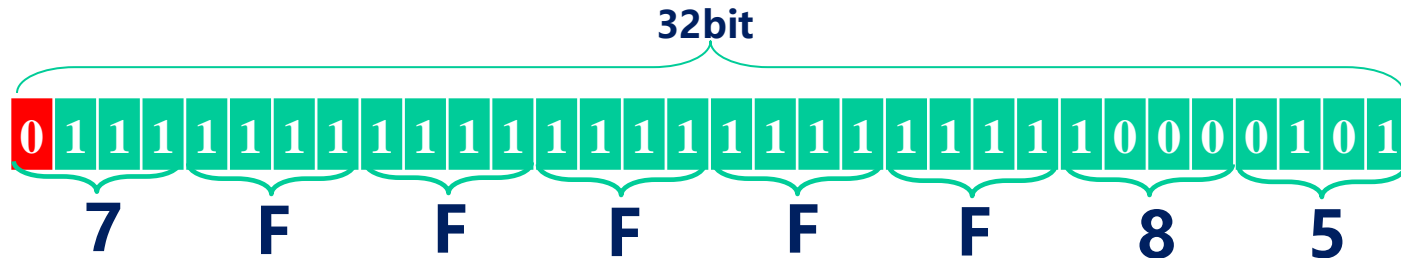
```
#include <iostream>
using namespace std;
int main()
{
    int a = -123;
    cout << oct << a << endl;
    return 0;
}
```

37777777605

← 8进制

8进制中的8个符号：0,1,2,3,4,5,6,7

十六进制表示



```
#include <iostream>
using namespace std;
int main()
{
    int a = 0x7FFFFFFF85;
    cout << dec << a << endl;
    cout << oct << a << endl;
    return 0;
}
```

2147483525
17777777605

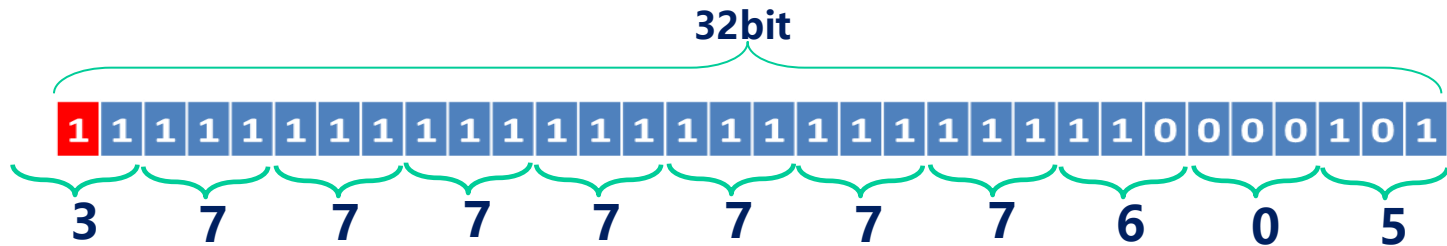
int a = 0x7FFFFFFF85;

cout << dec << a << endl; ← 10进制

cout << oct << a << endl; ← 8进制

return 0;

八进制表示



```
#include <iostream>
using namespace std;
int main()
```

```
{
```

```
    int a = 037777777605;
```

```
    cout << dec << a << endl;
```

```
    cout << hex << a << endl;
```

```
    return 0;
```

```
}
```

```
-123
fffffff85
```

整型最大最小值？

unsigned int i = Max;



signed int k = Max;



```
#include <iostream>
using namespace std;
int main()
{
    unsigned int a = 0xFFFFFFFF;
    signed int b = 0x7FFFFFFF;
    cout << dec << a << ', ' << b << endl;
    return 0;
}
```

结果：

4294967295, 2147483647

若最大有符号数的高位再置1

signed int a;



signed int b;



32bit

2147483647

32bit

32位补码表示

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    signed int a = 0x7FFFFFFF;
```

```
    signed int b = 0xFFFFFFFF;
```

```
    cout << dec << a << ',' << b << endl;
```

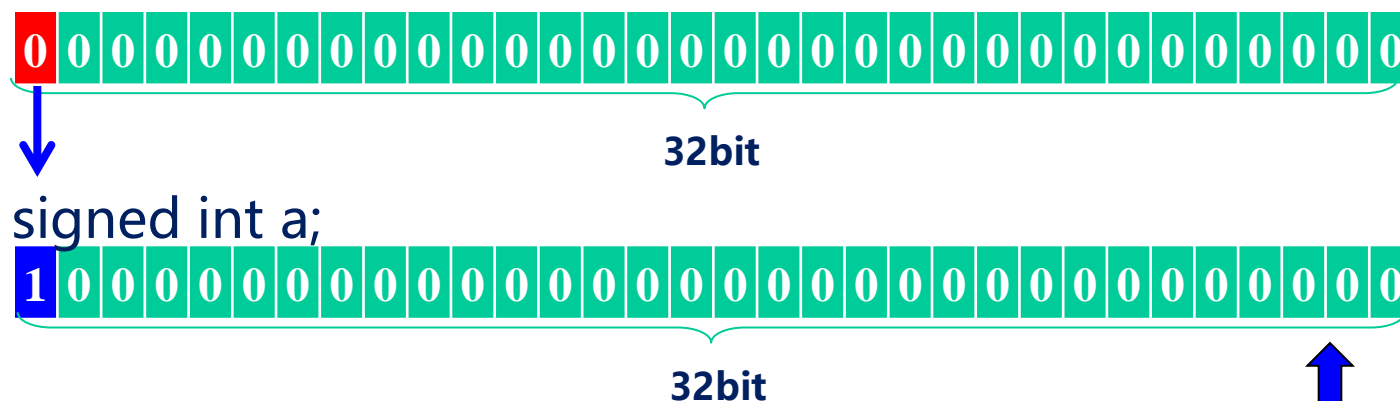
```
    return 0;
```

```
}
```

结果:

2147483647, -1

再看两个特别的数



```
#include <iostream>
using namespace std;
int main()
{
    signed int a = 0x80000000;
    cout << dec << a << endl;
    return 0;
}
```

32位最小值 (补码)

结果:

-2147483648

int	4	-2 147 483 648 ~ 2 147 483 647
signed int	4	-2 147 483 648 ~ 2 147 483 647
unsigned int	4	0 ~ 4 294 967 295

整数的特殊进制表示汇总

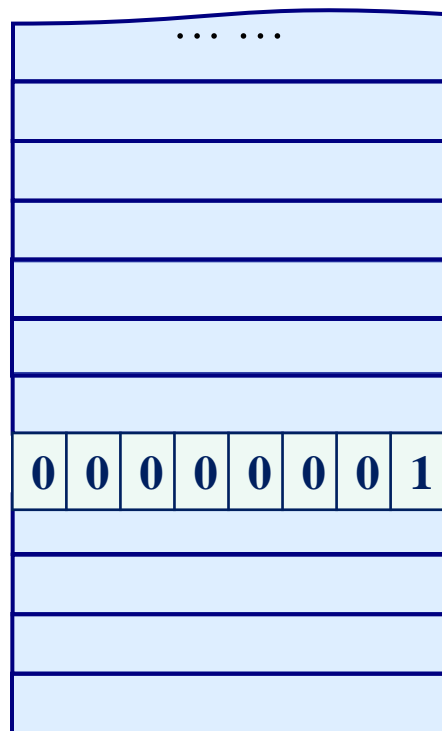
- 8进制：以零 **0** 开始的数字串，如 **0**75
- 16 进制：以 0x 开始的数字字母（A,B,C,D,E,F）串，如 **0x**9A4
- 10 进制：没有特殊标记，如 186

整数的溢出

- 试图大于可表示的最大正整数（字长限制）时，出现**上溢出**；此时，正数变负数；
- 试图小于可表示的最小负整数时，出现**下溢出**，负数变正数。
- **注意**：当发生溢出时，计算机并不报错，但实际所得到的结果已经不是期望的结果了，**应特别小心(怎么办?)**

布尔型

布尔类型(bool)是C++新增的类型，只能取true和false两个值，分别对应整数1和0



- 用于存储“真”和“假”的变量
 - 占一个字节
 - 值只能为 1 或 0
 - 1 代表 True
 - 0 代表 False
- 赋给布尔型变量的值
 - 可以赋任何值给它，但
 - 赋 0 存 0，表示False
 - 赋非零存1，表示True

布尔型举例

```
#include <iostream>
void main()
{
    bool b1=true,b2=false;
    cout<<"b1=true 时, b1="<<b1<<endl;
    cout<<"b2=false 时, b2="<<b2<<endl;
    int i;
    cout<<"请输入一个正整数i: ";
    cin>>i;
    b1=i>3;
    cout<<"b1=i>3 时, b1="<<b1<<endl;
    b2=(bool)-100; //类型强制转换 非0
    cout<<"b2=-100 时, b2="<<b2<<endl;
}
```

b1=true 时, b1=**1**
b2=false 时, b2=**0**
请输入一个正整数i: 4
b1=i>3 时, b1=1
b2=-100 时, b2=1

字符类型 (Char)

- 用于表示字符(如A, 6, #)的数据类型
- 字符型表示的长度与字符的编码关系密切，采用**8位的 ASCII**
- char 类型所表示的是字符的编码——一个整数（8位）。从这种意义上讲，char也表示8位的整数。取值范围为 -128 —— 127，但如果定义为无符号型，则取值范围是 0 —— 255

unsigned char 和 signed char(**等价于char**)

字符型常量的表示

- 由单引号直接表示: 'a', 'A', '6', '#' 等
- 由 ASCII 值表示, 尤其是表示某些起控制作用的非打印字符, 如退格, 换行, 响铃等, 如 8 表示“退格键”。
- 通过换码序列 (转义字符) 表示, 下表最左列

常用的转义字符		
字符形式	含义	ASCII编码
\n	换行, 将当前位置移到下一行开头	10
\t	水平制表, 跳到下一个tab位置	9
\b	退格, 将当前位置移到前一列	8
\r	回车, 将当前位置移到本行开头	13
\'	单撇号字符	39
\"	双撇号字符	34

更详细的转义表示

转义字符	描 述
\a	响铃(audible bell)
\b	退格(backspace)
\f	换页(formfeed)
\n	换行(newline)
\r	回车(carriage return)
\t	水平制表(horizontal tab)
\v	垂直制表(vertical tab)
\\	反斜线(backslash)
\'	单引号(single quote)
\"	双引号(double quote)
\DDD	八进制数 DDD 对应的字符(octal number)
\xHH	十六进制数 HH 对应的字符(hexadecimal number)

举例

- 退格符 bs(backspace) 的不同表示:

`char bs=8` // ASCII 表示

`char bs='\b'` // 转义表示

`char bs='\010'` // 8进制

`char bs='\X8'` // 16 进制, 以 \x 引到

字符型数据的特点

- 一个字符占一个字节
 - 字符要**转化成数值存储**（**对应的ASCII码值**）
 - 一个字符变量只能存放**一个**字符
- 字符数据的存储形式
 - 内存中字符变量对应**ASCII码**
 - 字符可以作为整数值使用（**对应的ASCII码值**）

字符类型

- C语言中允许将字符类型数据看作整数类型
 - 字符类型数据既可以以字符形式输出，也可以以整数形式输出，整数值为其对应的 ASCII 码值。
 - 字符数据可以进行算术运算

```
#include <stdio.h>
void main()
{
    char ch1,ch2;
    ch1='a'; ch2='b';
    printf("ch1=%c,ch2=%c\n",ch1,ch2);
    printf("ch1=%d,ch2=%d\n",ch1,ch2);
}
```

程序运行结果:

ch1=a,ch2=b

ch1=97,ch2=98

浮点型

浮点型 = 实型

浮点型	长度	字节数	有效位	范围
float	32bit	4	7位	$-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$
double	64bit	8	16位	$-1.7 \times 10^{308} \sim 1.7 \times 10^{308}$
long double	64bit	8	16位	$-1.7 \times 10^{308} \sim 1.7 \times 10^{308}$

长双精度的具体长度与机器相关

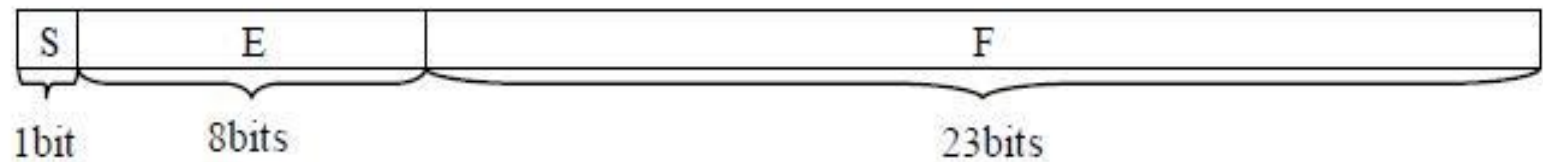
几个观察的结果

- float型可以表示的十进制范围是 $-3.402823466e^{38} \sim 3.402823466e^{38}$ ，而作为同为4个字节的整型数却只能表示-2147483648~2147483647的范围，使用同样的内存空间，浮点数能比定点数表示大得多的范围
- 计算机中浮点数是指小数点可以移动的一种表示
- 为何不使用浮点数来代替定点数呢？
 - 浮点数实现起来比较复杂，有些处理器还**专门配置**了硬件浮点运算单元用于浮点运算
 - 浮点数的**精确度有限**

浮点数的存储

- 浮点型的存储方式和其他类型有很大不同，不同编译器之间也存在差别，但大部分都遵从IEEE标准
- 浮点数由三个部分表示
 - 符号
 - 指数偏差：不同于指数本身
 - 小数

以4字节（32位）的float为例



- 符号位**S**（sign bit）、指数偏差**E**（exponent bias）和小数部分**F**（fraction），三部分都是二进制码表示
- 符号位**S**：占1位，**0**代表浮点数为正，**1**代表负
- 一般形式为： $R = (S) * (1 + F) * 2^e$
 - **R**：实数，**S**：正负符号，**F**：小数部分，**E**：指数偏差，不同于指数 **e**）。
 - 例： $0.5 = 1 * 2^{-1}$

小数部分有1位隐含的整数位1

指数与指数偏差

- 指数偏差E：8位，大小 $0 \sim 255$
- 实际指数e的计算： $e = E - 127$ (**$E = e + 127$**)
 - 指数位决定浮点型数值的范围。浮点数内部表示中的指数位并非真正指数值，真正指数值按上述式子转换
 - 偏差值是为了校正指数值而设定的，每种类型的偏差值是固定的，不同类型的偏差值不同（如下表）
 - 用长度为8个比特的无符号整数来表示所有的指数取值，这使得两个浮点数的指数大小的比较更为容易

GCC中各浮点型的储存分段

数据类型	符号位	指数位	基数位	偏差值
float	1	8	23	127
double	1	11	52	1023
long double	1	15	64	16383


注：不同编译系统偏差值可能有差异

小数部分F

- 小数部分F：占23位，实际上是将浮点数转换成二进制码，再按科学记数法保留小数部分
 - 小数位位数确定其精确度
 - long double型的小数位即为其科学计数法的基数位的二进制表示形式
 - float型和double型的基数位与long double型略有不同：float型和double型**确保1位整数位为1**，但在小数部分（F）不显示整数位1，只包含整数位1后面的纯小数部分，**这样，可以得到多一位的存储空间保留小数**

所以： $R = (S) * (1 + F) * 2^e$

以float 为例

- 十进制数：3.75的浮点表示
- 转化为二进制码为11.11
- **保留1为整数位为1**，该二进制码按科学计数法表达为 $1.111e1$ （小数点向左移动1位）
 - 即符号位为： $s=0$ （正数）
 - 指数部分： $e=1$ ， $E=e+127=128$
 - 小数部分： F记录值为 $111000...000$


再看例子（1）

- 将十进制 0.5 转化成 32 位二进制浮点数（float）
- 0.5 的二进制码为 0.1，科学技术法为 1.0×2^{-1} ，即小数点向右移 1 位， $e=-1$
 - 即符号位为： $s=0$ （正数）
 - 指数部分： $e=-1$ ， $E=e+127=126$ （二进制： 01111110）
 - 小数部分： 1.0 把整数部分的 1 去掉后，剩下小数部分 F 为 23 个 0
 - 完整的数据表示为：

0 01111110 000000000000000000000000

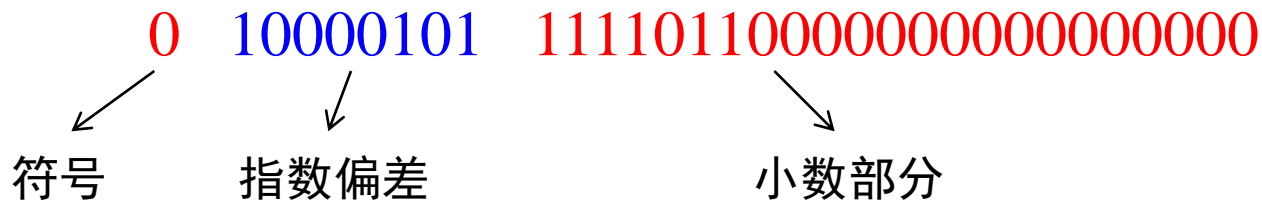
符号 指数偏差 小数部分

再看例子（2）

- 将十进制 125.5 转化成 32 位二进制浮点数
- 125.5的二进制码为1111101.1，按科学技术法写为 1.1111011×2^6 ，小数点向左移6位，则 $e=6$ ，
 - 即符号位为： $s=0$ （正数）
 - 指数部分： $e=6$ ， $E=e+127=133$ (二进制： 10000101)
 - 小数部分： 1.1111011把整数部分的1去掉后，剩下小数部分F为 1111011后面补0，直至小数部分为 23 位
 - 完整的数据表示为：

0 10000101 111101100000000000000000

符号 指数偏差 小数部分



C/C++中实数的有效数据

```
main( )
```

```
{
```

```
    float  x1, y1;
```

```
    double x2, y2;
```

```
    x1 = 111111.111;  y1 = 222222.222;
```

```
    printf(" x1 = %f \n", x1);
```

```
    printf(" y1 = %f \n", y1);
```

```
    printf(" x1+y1 = %f\n\n", x1 + y1);
```

```
    x2 = 111111.111;  y2 = 222222.222;
```

```
    printf(" x2 = %f \n", x2);
```

```
    printf(" y2 = %f \n", y2);
```

```
    printf(" x2+y2 = %f\n", x2 + y2);
```

```
}
```

float 表示: 7位有效位

double表示: 16 位有效位

- **float** 表示为何是 7位有效位? 因为小数的2进制位是(23+1)位, 于是, $2^{24}=16777216$, 其中, $10^7 < 16777216 < 10^8$

运行结果:

x1 = 111111.109375

y1 = 222222.218750

x1 + y1 = 333333.328125

x2 = 111111.111000

y2 = 222222.222000

x2 + y2 = 333333.333000

精确度问题

```
#include <iostream>
using namespace std;
int main( )
```

```
{
    float a, b;
    double c;
    a = 12345.6789;
    c = a;
    cout << "a = " << a << endl;
    cout << "c = " << c << endl;
    cout.setf(ios::fixed | ios::showpoint); //用定点格式显示浮点数
    cout.precision(5); //小数点后显示5位小数位
    cout << "a = " << a << endl;
    cout << "c = " << c << endl;
    c = 12345.6789;
    cout << "c = " << c << endl;
    return 0;
}
```



```
a = 12345.7
c = 12345.7
a = 12345.67871
c = 12345.67871
c = 12345.67890
请按任意键继续. . .
```

内容

➤ C/C++的基本数据类型

➤ 数据的输入输出再介绍

C语言的格式化输出

- 格式控制包括格式说明和普通字符：
 - 格式说明由“%”和格式字符组成，用于将输出数据转换成指定类型的数据和格式，如“%d”；
 - 普通字符，想要原样输出的字符。
- 输出列表：要输出的数据序列，可以是表达式。

例：假定 $a=3, b=4$ 则：

```
printf("%d * %d = %d", a, b, a*b);
```

格式说明

普通字符

3 * 4 = 12

输出结果：

修饰符	功 能
m	输出数据域宽,数据长度<m,左补空格;否则按实际输出
.n	对实数,指定小数点后位数(四舍五入) 对字符串,指定实际输出位数
-	输出数据在长度内左对齐 (缺省右对齐)
+	指定在有符号数的正数前显示正号(+)
0	输出数值时指定左边不使用的空位置自动填0
#	在八进制和十六进制数前显示前导0, 0x
l	在d, o, x, u前, 指定输出精度为long型 在e, f, g前, 指定输出精度为double型

几点说明:

- 附加的修饰符，用于限定输出的长度，具体情况如下：
 - 具体的整数（m）位于格式符前，表示输出的宽度；
 - 若格式控制为“%m.nf”，则表示输出为浮点数，共m位，其中，小数位为n位，当数值长度小于m时，左边补空，右靠齐。若形式为：“%-m.nf”，则右边补空，左靠齐
 - 字符串输出的几种形式：
 - %ms: 输出占 m 列，如s 实际大于 m列，则实际输出，否则，左边补空； %-ms 是右边补空；
 - %m.ns: 取 s的左边 n 个符号输出，不够m列，左边补空； %-m.ns，含义同上，右边补空。若 n>m，m自动取n值。

```
例 int a=1234;  
    float f=123.456;  
    char ch='a';  
    printf("%8d,%2d\n",a,a);  
    printf("%f,%8f,%8.1f,%.2f,%.2e\n",f,f,f,f,f);  
    printf("%3c\n",ch);
```

请大家写出运行结果

运行 1234,1234

结果: 123.456000,123.4560, 123.5,123.46,1.23e+02
 a

C++中的输入输出格式控制

操 作 符	描 述	备 注
dec	按十进制输出	常量控制符 在 <code>iostream.h</code> 中
hex	按十六进制输出	
oct	按八进制输出	
endl	插入换行符，并刷新流	
ends	插入空字符	
setbase(n)	设置整数基数为 <code>n</code>	函数控制符 在 <code>iomanip.h</code> 中
setfill(c)	设置填充字符 <code>c</code>	
setprecision(n)	设置实数精度为 <code>n</code>	
setw(n)	设置字段宽度为 <code>n</code>	
setiosflags(ios::fixed)	设置浮点数以固定的小数位数显示	
setiosflags(ios::scientific)	设置浮点数以科学记数法显示	
setiosflags(ios::left)	输出数据左对齐	
setiosflags(ios::right)	输出数据右对齐	
setiosflags(ios::skipws)	忽略前导的空格	
setiosflags(ios::uppercase)	十六进制数大写输出	
setiosflags(ios::lowercase)	十六进制数小写输出	
setiosflags(ios::showpos)	输出正数时给出“+”号	
resetioflags()	终止已设置的输出格式状态	

比较C语言的格式
化输入输出

C++中的缺省输出格式

不同类型数据的输出有其**缺省**的(默认的)输出格式:

♥ 输出整型数: 十进制、域宽为0、右对齐、空格填充。

若数的实际宽度超过域宽，
则按实际长度输出。

♥ 输出实型数: 精度6位、浮点输出、
域宽为0、右对齐、空格填充。

若整数部分超过7 位或有效数字在小数点后第4 位之后，自动转为科学计数法格式输出。

♥ 输出字符或字符串: 域宽为0、右对齐、空格填充。

宽度控制的例子

```
#include <iostream.h>
#include <iomanip.h>
void main( )
{
    double d1=12.3456789, d2=123456.789, d3=0.0000123456;
    cout<<d1<<','<<d2<<','<<d3<<endl;
    cout<<setw(10)<<d1<<','<<setw(10)<<d2<<','
        <<setw(10)<<d3<<endl;
    char s[10]="abcd", c='k';
    cout<<s<<','<<setw(4)<<c<<endl;
}
```

输出结果:

12.3457, 123457, 1.23456e-005

□□□12.3457, □□□□123457, 1.23456e-005

abcd, □□□k

注意: □表示空格

setw(n): 设置紧接着输出数据的宽度为 n (只影响后面一个数)

只能输出**6位**有效位

如**n**小于输出数据的实际宽度, 则按实际宽度显示

```
#include<iostream>
#include<iomanip>
using namespace std;
int main() {
```

进制控制例子

```
    int a, b;
    cout<<"以八进制格式输入整数: \na=0";
    cin >> oct >> a;
    cout << "b=0";
    cin >> b; //仍然保持八进制设置
    cout<<"以十六进制显示整数 a=0x" << hex << a << ", b=0x" << b << endl;
    cout<<"以十进制显示整数  a=" << dec << a << ", b=" << b << endl;
    cout<<"以八进制显示整数  a=0" << oct << a << ", b=0" << b << endl;
    cout<<resetiosflags(ios::oct|ios::showbase);
    cout<<setiosflags(ios::hex|ios::showbase);
    cout<<"以十六进制显示整数 a="<<a<<",b="<<b<<endl;
    cout<<resetiosflags(ios::hex|ios::showbase);
    cout<<setiosflags(ios::dec|ios::showbase);
    cout<<"以十进制显示整数  a="<<a<<",b="<<b<<endl;
    cout<<resetiosflags(ios::dec|ios::showbase);
    cout<<setiosflags(ios::oct|ios::showbase);
    cout<<"以八进制显示整数  a="<<a<<",b="<<b<<endl;
    return 0;
}
```

```
以八进制格式输入整数:
a=0 20
b=0 10
以十六进制显示整数 a=0x10, b=0x8
以十进制显示整数  a=16, b=8
以八进制显示整数  a=020, b=010
以十六进制显示整数 a=0x10, b=0x8
以十进制显示整数  a=16, b=8
以八进制显示整数  a=020, b=010
Press any key to continue_
```

取消8进制输出形式

setprecision控制浮点、定点等

- 浮点数的有效位控制：
 - `setprecision(n)` 控制n位有效位数
- 定点数的小数位控制：
 - `setprecision(n)` 控制小数点后n位数
 - 需与`setiosflags(ios::fixed)`合用
- 指数形式输出的小数位数控制
 - `setprecision(n)` 控制小数位数为 n
 - 需与`setiosflags(ios::scientific)`合用

setprecision: 控制浮点位数

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double number1 = 132.364, number2 = 26.91;
    double quotient = number1 / number2;
    cout << quotient << endl;
    cout << setprecision(5) << quotient << endl;
    cout << setprecision(4) << quotient << endl;
    cout << setprecision(3) << quotient << endl;
    cout << setprecision(2) << quotient << endl;
    cout << setprecision(1) << quotient << endl;
    return 0;
}
```

输出结果:

4.91877
4.9188
4.919
4.92
4.9
5

共输出5为数字（不算小数点）

共输出4为数字（不算小数点）

Setprecision(n)与setiosflags(ios::fixed)合用

```
#include <iostream>
```

```
#include <iomanip>
```

```
#include <cmath>
```

```
using namespace std;
```

```
int main() {
```

```
    double s=20.7843000;
```

```
    cout << s << endl;
```

```
    cout << setiosflags( ios::fixed );
```

```
    cout << "setprecision( 1 )" << setprecision( 1 ) << s << endl;
```

```
    cout << "setprecision( 2 )" << setprecision( 2 ) << s << endl;
```

```
    cout << "setprecision( 3 )" << setprecision( 3 ) << s << endl;
```

```
    cout << "setprecision( 4 )" << setprecision( 4 ) << s << endl;
```

```
    cout << "setprecision( 5 )" << setprecision( 5 ) << s << endl;
```

```
    cout << "setprecision( 6 )" << setprecision( 6 ) << s << endl;
```

```
    cout << "setprecision( 7 )" << setprecision( 7 ) << s << endl;
```

```
    cout << "setprecision( 8 )" << setprecision( 8 ) << s << endl;
```

```
    cout << setprecision(6);
```

```
    return 0; }
```

与定点合用：控制小数点后面的位数

输出结果：

20.7843

setprecision(1)20.8

setprecision(2)20.78

setprecision(3)20.784

setprecision(4)20.7843

setprecision(5)20.78430

setprecision(6)20.784300

setprecision(7)20.7843000

setprecision(8)20.78430000

//重新设置成原默认设置

配合setiosflags(ios::fixed)显示小数点

```
#include<iostream>
#include<iomanip>
#include<cmath>
using namespace std;
int main() {
    double s=20.7843000;
    cout << s << endl;
    cout << setiosflags( ios::fixed);
    cout << "setprecision( 0 )" << setprecision( 0 ) << s << endl;
    cout << "setprecision( 1 )" << setprecision( 1 ) << s << endl;
    cout << "setprecision( 2 )" << setprecision( 2 ) << s << endl;
    cout << "setprecision( 3 )" << setprecision( 3 ) << s << endl;
    cout << setiosflags( ios::fixed|ios::showpoint );
    cout << "setprecision( 0 )" << setprecision( 0 ) << s << endl;
    cout << "setprecision( 1 )" << setprecision( 1 ) << s << endl;
    cout << "setprecision( 2 )" << setprecision( 2 ) << s << endl;
    cout << "setprecision( 3 )" << setprecision( 3 ) << s << endl;
    return 0;
}
```

输出结果:

20.7843

setprecision(0)21

setprecision(1)20.8

setprecision(2)20.78

setprecision(3)20.784

setprecision(0)21.

setprecision(1)20.8

setprecision(2)20.78

setprecision(3)20.784

必显示小数点

再看setprecision(后面一直继承)

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{ double pi=3.141592653589793;
  double feet=5280;
  double number=123.456789
  cout<< setprecision(5)<< " \n pi= " <<pi<< " \n feet= " <<feet
    << " \n number= " <<number<<endl;
  cout<<setiosflags(ios::fixed)<< " \n pi= " <<pi
    << " \n feet= " <<feet << " \n number= " <<number<<endl;
  cout<<setiosflags(ios::scientific) << " \n pi= " <<pi
    << " \n feet= " <<feet << " \n number= " <<number<<endl;
  return 0;
}
```

输出结果:

pi=3.1416

feet=5280

number=123.46

pi=3.14159

feet=5280.00000

number=123.45678

pi=3.14159e+000

feet=5.28000e+003

number=1.23457e+002