#### 南京大学 计算机科学与技术系

Department of Computer Science & Technology, NJU

# Chapter 4

# 程序中数据的描述

刘奇志

#### ● 程序中数据的描述

- → 数据类型简介
- → 基本类型 (简单数据的描述)
  - 字符型
  - 整型
  - 浮点型
  - 逻辑型
  - 枚举类型
- → 基本类型的选用
  - sizeof()操作符
- → 基本类型的转换
  - 伪随机数的生成
- ◆ 复杂数据的描述方法简介
- → 类型名的自定义

# 信息 (information)

- 现实世界中的对象及其属性在人们头脑中反映为各种不同的信息
  - → 数字
  - → 文字
  - → 声音
  - → 图形
  - **\*** · · ·
- ◆ 计算机中,这些信息一般是用一系列0和1来表示的(简单),它们对应着电器设备的两个稳定状态: 开关的开/关、电压的高/低、电流的有/无。

- 对于基于0和1表示的信息,常用的计量单位
  - →位(bit,由一个0或1构成)
    - 计算机中最小的信息单位
  - → 字节 (byte, 由8个2进制位构成)
    - 存储空间的基本计量单位
    - 千字节 (kilobyte, 简称KB, 由1024个字节构成)
    - 兆字节 (megabyte, 简称MB, 由1024个千字节构成)
    - 吉字节 (gigabyte, 简称GB, 由1024个兆字节构成)
    - 太字节 (terabyte, 简称TB, 由1024个吉字节构成)
    - 更大的计量单位还有PB (petabyte), EB (Exabyte), ZB (zettabyte), 以及YB (yottabyte)。
    - 上述计量单位可以用来衡量内存和外存等的容量,比如,内存的容量可以为512MB、1GB、2GB等,硬盘的容量可以为40GB、80GB、160GB等

# 数据

◆ 在程序中,各种形式的信息表现为数据,它们是程序的处理对象和结果, 是程序的重要组成部分。

 常量
 文件
 数据库

●程序设计语言通常将数据划分成不同的类型,并分别用专门的单词/符号来描述。

# 数据类型 (data type)

- 一种数据类型可以看成由两个集合构成:
  - → 值集: 可以取哪些值(值域,包括这些值的结构)
  - → 操作集: 值集中的值可参与哪些操作
    - 如:整型的值集是由整数所构成的集合,它的操作集包括:加、减、乘、除等运算。
  - → 程序中的数据可取的值除了受到数据类型的约束之外,还会受计算机的存储空间和 存储方式的限制,比如,整型的值集只是数学里整数集合的一个子集。

- 数据为什么要分成不同的类型
  - → 便于合理分配内存,产生高效代码。
    - -true (1)  $\rightarrow$ 00 00 00 01 ?  $\rightarrow$  01 ?
  - → 便于运算,便于数据的处理。
    - 求余数→int ? double ?
  - ◆便于自动进行类型一致性检查,保护数据,提高程序的可靠性。
    - -int home price = 89.5 ? 895000

## 数据类型机制

- 按对数据类型的处理方式,程序设计语言可分为:
  - 静态类型语言与动态类型语言
    - √静态类型语言:要在运行前的程序中指定每个数据的类型
    - √动态类型语言:程序运行中才确定数据的类型
  - 强类型语言与弱类型语言
    - √强类型语言:自动严格检查类型
    - √弱类型语言:不作或很少作类型检查
  - □C是静态的弱类型语言, C++是静态的强类型语言

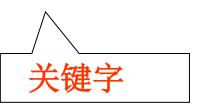
# C语言数据类型

- 基本类型 (basic types, fundamental types)
  - → 由系统预先定义好的数据类型,常常又称为标准类型或内置类型(built-in types)。 对应能由计算机的机器指令直接操作的数据。
- 派生类型 (derived types, compound types)
  - → 由程序员定义,又称为构造类型

# 基本类型

#### 🍳 包括

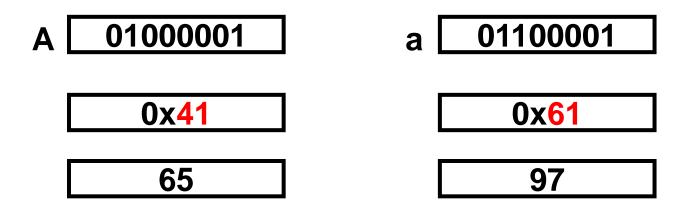
- → 字符型char
- **→**整型···int
- → 浮点型
  - 单精度float
  - 双精度double
  - 长双精度long double
- → 逻辑型bool



C89没有逻辑类型, C99提供了\_Bool逻辑类型 C11提供了bool逻辑类型

# 字符型

- 用于描述文字符号类的信息
- 在计算机中实际存放的是字符对应的机器数,即每个字符对应一种0和1的组合方式。



- C标准规定普通字符型数据在计算机中占用1个字节空间,即8个2进制位空间。
- 根据字符型数据在计算机中占用空间的大小,可以推算出其取值范围。

#### ● 值域:

- → 对应的十六进制数为00~7F、80、81~FF,
- → 对应的十进制数为0~127、128、129~255,
- → 对应的256种字符一般为ASCII码表中规定的字符。

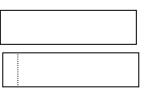
#### • 操作集:

- → 算术操作
- → 关系和逻辑操作
- → 位操作
- → 赋值操作
- → 条件操作
- **→** ······
- → C语言允许字符型数据参与以上操作,实际上是其对应的ASCII码在参与操作

# 字符型变量

| ● 定义字符型变量时用char,  | ,可以加类型修饰符 |
|-------------------|-----------|
| ♥ 正义子行型受重的用Cliar, | ,り以加尖型修饰行 |

→ char



由具体系统决定被看成 有符号整数或无符号整数

→ signed char

- 对应的十进制数为: 0~127、

0~127,

-0,

-1~-127 (原码)

-128、 -127~-1 (补码)

→ unsigned char

- 对应的十进制数为: 0~127、

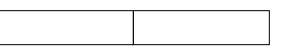
128

129~255

被看成无符号整数

被看成有符号整数

→ wchar\_t (宽字符)



标准未提及

## 字符型常量

- 普通字符: 由两个单引号(') 括起来的一个字符
  - → 如: 'A', '5', '+', '\$', '', ...
- 转义符 (escape sequence):由两个单引号(')括起来的一个特殊字符序列,其中的字符序列以\开头,后面是一个特殊字符或八进制ASCII码或十六进制ASCII码

#### → 特殊转义符

- '\n'(换行符), '\a'(响铃符), '\t'(制表符), '\b'(退格符), '\'(反斜杠), '\'(单引号), '\''(双引号), ...

#### → ASCII码转义符

一般用于只有数字小键盘的场合

- 八进制: '\ddd', 如: '\101' (表示'A', 101是A对应的八进制ASCII码)
- 十六进制: '\xdd'(或'\Xdd'),如: '\x41'(表示'A',41是A对应的十六进制ASCII码

一般用于键盘只能输入数字和少数英文字母的场合

● 例4.1 用格式符%c将各种类型的数据显示为字符。

```
#include <stdio.h>
int main()
{
   printf("ASCII 65 in decimal represents the character: %c \n", 'A');
   printf("ASCII 65 in decimal represents the character: %c \n", 65);
   return 0;
}
```

ASCII 65 in decimal represents the character: A ASCII 65 in decimal represents the character: A

## 自学

```
printf("%c", 'A');
                     65
printf("%d", 65);
char ch = 'A';
printf("%c", ch);
char ch = 65;
printf("%c", ch);
int ch = 'A';
printf("%d", ch);
                     65
```

cout << ...

◎ 例4.2 字符型变量值的输入及其参与关系和算术操作。

```
int main( )
{ char ch;
 do
    printf("Input Y or N (y or n) : ");
                                             ch = getchar();
     scanf("%c", &ch);
     if (ch >= 'A' \&\& ch <= 'Z') ch += 32;
     //ch = (ch >= 'A' && ch <= 'Z') ? ch + 'a' - 'A' : ch
     printf("%c", ch);
  } while(ch != 'y' && ch != 'n');
 if(ch == 'y')
 else
```

#### ♥ 例4.3 数字字符与整数的区别示例。

```
实际应用中,数字字符更多是用来描述字符
                         串的一分子,比如,"以3结尾的学号",
int main( )
                         而不是用来参加数值运算。
 int i = 3;
 char ch = '3';
 printf("%d %d \n", 10 * i, 10 * ch);
 return 0;
                30
                                               510
      3
                                   '3'
   00000011
                                00110011
```

◆ 例4.4 宽字符操作简单示例\*\*。 #include <locale.h> int main( ) setlocale(LC\_ALL,""); //设置为本地区域字符库 wchar t wch = 25105; wprintf(L"%c \n", wch); wch = getwchar(); wprintf(L"%c \n", wch); return 0;

# 整型

- 用于描述整数
- 包括:
  - → 基本整型 (int)
  - → 短整型 (short int, int可以省略)
  - → 长整型 (long int, int可以省略)
  - → 加长整型 (VC提供了\_int64类型)

C99提供了long long int, int可以省略

# 整型数据占用空间

- 标准大致规定了几种整型数据在计算机中占用空间的大小:
  - → int型数据,以32位机为例,占4个字节
  - → short int  $\leq$  int  $\leq$  long int
    - 一般地,短整型数据占用的空间为int型的一半,长整型数据占用的空间为int型的两倍。有些开发环境中的int型数据占用的空间与长整型数据或短整型数据的相等。

| short |  |  |
|-------|--|--|
| int   |  |  |
| long  |  |  |

根据整型数据在计算机中占用空间的大小,可以推算值域。

- ♥ 值域(以32位机为例,int型数据的取值范围):
  - → 用二进制表示:

- → 对应的十六进制数为 00000000~7FFFFFFF、 8000000、 8000001~FFFFFFF
- → 对应的十进制数为

   0~2147483647、
   -0、
   -1~-2147483647、
   按原码理解2进制数的值

0~2147483647 -2147483648 -2147483647~-1 按补码理解2进制数的值

→ 具体的值域可以查看文件limits.h

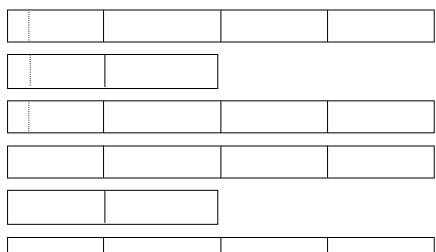
## • 操作集:

- → 算术操作
- → 关系和逻辑操作
- → 位操作
- → 赋值操作
- → 条件操作

**→** ··

# 整型变量

- ◆ 在定义上述整型变量时,可以在类型关键字int前加signed或unsigned修饰,int有时可以省略)
  - → (signed) int
  - → (signed) short (int)
  - → (signed) long (int)
  - → unsigned (int)
  - → unsigned short (int)
  - → unsigned long (int)
  - **→** ...
- unsigned型的无符号数只能表示非负整数,不过其所表示的最大正整数比相应的 signed型所表示的最大正整数约大一倍(以32位机为例, unsigned int型数据的取值 范围)
  - → 0~2147483647、
  - → 2147483648、
  - 2147483649~4294967295



## 整型常量 (整数)

- 默认情况下,整数一般按int型看待,超过int型范围的正整数按unsigned int型或其他能够表示更大范围正数的类型看待,超过int型范围的负数按long int或其他能够表示更大范围负数的类型看待。
- 不管数据大小,可以在数字后加L(l) 表示long int型整数,加LL(ll) 表示long long int型整数,加U(u) 表示unsigned int型整数,加UL(ul/LU/lu) 表示unsigned long int型整数,加ULL(ull/LLU/llu) 表示unsigned long long int型整数。
- 给整型变量赋值时,最好用同类型的整型常量,比如,
  - $\rightarrow$  int i = 0;
  - $\rightarrow$  long int sum = 0L;
  - $\rightarrow$  unsigned sum = 4294967295u;

- C程序中,整数可以用十进制、八进制或十六进制形式来书写:
  - → 十进制: 无前后缀。由0~9数字组成,第一个数字不能是0(整数0除外),如: 59, 128,-72;
  - → 八进制:数字<sup>0</sup>为前缀,无后缀。由0~7数字组成,如:073,0200,-0110;
  - → 十六进制: Ox或OX为前缀, 无后缀。由O~9数字和字母A~F(或a~f)组成, 如: Ox3B, Ox8O, -Ox48.
- 注意: C语言中没有2进制整数(最新版标准开始有)

%C

● 例4.5 用格式符%d将各种类型的数据显示为十进制整数。 #include <stdio.h> int main( ) printf("ASCII of the character is: %d \n", 'A'); printf("ASCII is: %d \n", 65); printf("ASCII in decimal is: %d \n", 0x41); printf("ASCII of the character is: %d \n", '7'); printf("ASCII of the character is: %d \n", '\a'); ASCII of the character is: 65 return 0; ASCII is: 65 ASCII in decimal is 65 ASCII of the character is: 55 ASCII of the character is: 7

♥ 例4.6 整型数据范围溢出示例。

```
int main()
{
  int a = 2147483647, b = 1;
  printf("%d \n", a + b);
  return 0;
}
```

## -2147483648

- 如果将输出格式符%d改成%u (即按unsigned int型数据输出),则结果为 2147483648。
  - → printf("%u \n", a+b);//cout << (unsigned)(a + b) << endl;</pre>

# 浮点型

- 用于描述浮点数。
- 分类
  - → 实浮点型 (real floating types)
    - 单精度浮点型 (float)
    - 双精度浮点型 (double)
    - 长双精度浮点型 (long double)
  - → 复型 (complex types,含有实部和虚部两个元素)
    - 单精度复型 (float \_Complex)
    - 双精度复型 (double \_Complex)
    - 长双精度复型 (long double \_Complex) 。
    - 旧标准未规定复型

## 浮点型数据占用空间

- 标准规定:
  - → float ≤ double ≤ long double
  - → 在计算机中以二进制规格化形式存储,即尾数和指数分别占用不同的空间
- 根据所占用空间可以推算出它们的取值范围和精度。

| • | 以32位机为例, | 一般地, |
|---|----------|------|
|   |          |      |

→ 单精度数据占32位空间,其中尾数部分占24位,含1位符号位,指数部分占8位,含1位指数的符号位 float

→ 双精度数据占64位空间,其中尾数部分占53位,含1位符号位,指数部分占11位,含1位指数的符号位

音1位指数的符号位 double

→ 长双精度数据占80位空间(有的系统占128位空间),其中尾数部分占64位,含1 位符号位,指数部分占16位,含1位指数的符号位

long double

#### ● 值域与精度(以32位机为例):

- → 単精度
  - 值域大约是: -3.4×10<sup>38</sup> ~ 3.4×10<sup>38</sup>
  - 能够表示的最小正数大约是1.175×10<sup>-38</sup>
  - 分辨率大约是1.192×10<sup>-7</sup>,即有6位数字有效

#### → 双精度

- 值域大约是: -1.8×10<sup>308</sup> ~ 1.8×10<sup>308</sup>
- 能够表示的最小正数大约是2.225×10<sup>-308</sup>
- 分辨率大约是2.22×10<sup>-16</sup>,即有15位数字有效
- → 长双精度(以80位为例)
  - 值域大约是: -1.2×10<sup>4932</sup> ~ 1.2×10<sup>4932</sup>
  - 能够表示的最小正数大约是3.362×10-4932
  - 分辨率大约是1.08×10<sup>-19</sup>,即有18位数字有效
- 具体值域和精度可以查看文件float.h

#### • 操作集:

- → 算术操作(求余数运算除外)
- → 关系和逻辑操作
- → 赋值操作
- → 条件操作

**→** ··

# 实浮点型变量

● 定义实浮点型变量时,在变量名前加类型关键字float、double或long double即可。

## 实浮点型常量 (实数)

- C程序中,实数有两种表示法
  - → 小数表示法:由(符号)、整数部分、小数点(.)和小数几个部分构成,如: 314.16,-0.00911
  - → 科学表示法: 由(符号)、规格化小数(在小数点前只有一位非0整数的小数)、字母E(或e)、(符号)和整数几个部分组成,如: 3.1416E2(即3.1416×10²), -9.11e-3(即-9.11×10⁻³)

- C程序中的实数可以用十进制或十六进制形式来书写:
  - → 十进制实数:没有前后缀
  - → 十六进制实数:以Ox或OX为前缀,没有后缀
    - 科学表示法的十六进制实数由规格化十六进制小数、字母P(或p)、符号和十六进制整数四部分组成,比如0x1.fP-3(即 $0x1.f\times16^{-3}$ )。
- 默认情况下,实数一般按double型看待。可以在数字后加F(f)表示float型实数,加L(l)表示long double型实数。

● 例4.7 实浮点型数据的输入/输出。

```
#include<stdio.h>
int main( )
{ float x, y = 12.3456779F;
  scanf("%f", &x);
 printf("%f \n", x);
 printf("%.11f \n", y);
 printf("%e \n", 3.14159265);//cout << scientific << 3.14159 << endl;</pre>
 printf("%e \n", 0x1.fP-3);
  return 0;
```

● %e是按科学计数法显示结果,默认情况下结果占13格,其中,小数点前的整数部分与小数点本身各占1格,小数部分占6格,然后是字母e与正(负)号各占1格,指数部分占3格。

```
● 例4.8 实浮点型数据的精度问题。
#include <stdio.h>
int main( )
 float x = 0.1f;
 float y = 0.2f;
 float z = x + y;
 if(z == 0.3)
    printf("They are equal.\n");
 else
    printf("They are not equal! The value of z is %.10f", z);
 return 0;
```

```
◎ 例4.8 实浮点型数据的精度问题。
#include <stdio.h>
int main( )
 float x = 0.1f;
 float y = 0.2f;
 float z = x + y;
 if(z == 0.3)
    printf("They are equal.\n");
 else
    printf("They are not equal! The value of z is %.10f", z);
 return 0;
  //输出 "They are not equal! The value of z is 0.300000119"
```

```
 \begin{array}{lll} (0.1)_{10} &= (0.0001100110011...)_2 \\ &= (0.01100110011001100110011)_2 \times 2^{-2} \\ (0.2)_{10} &= (0.0011001100110...)_2 \\ &= (0.11001100110011001100110)_2 \times 2^{-2} \\ (0.1+0.2)_{10} &= (1.0011001100110011001)_2 \times 2^{-2} \\ (0.3)_{10} &= (0.01100110011...)_2 \\ &= (1.10011001100110011001101)_2 \times 2^{-2} \\ \end{array}
```

#### 为什么?

1: 二进制

2: 浮点数的存储格式

● 对浮点数进行关系操作时,往往得不到正确的结果,应避免对两个浮点数进行"=="和"!="操作

```
→ x == y 可写成: fabs(x-y) < 1e-6
```

$$\rightarrow z == 0.3$$
 可写成: fabs(z-0.3) < 1e-6

```
float a, b, c;
a = 1.345f;
b = 1.123f;
c = a + b;
if(c == 2.468)
    printf("They are equal.\n");
else
    printf("They are not equal! The value of c is %.10f, or %f", c, c);
```

They are not equal! The value of c is 2.4679999352 or 2.468000

可以使用常数FLT\_EPSILON(单精度浮点型,1.192092896e-7F)或DBL\_EPSILON(双精度浮点型,2.2204460492503131e-16),需要包含这些常数定义的float.h,这些常数被当成最小的正数。

```
0.0001
#define EPSILON
                               Define your own tolerance
float a, b, c;
a = 1.345f;
b = 1.123f;
c = a + b;
if(((2.468-EPSILON)<c) && (c<(2.468+EPSILON)))
     printf("They are equal.\n");
else
     printf("They are not equal! The value of c is %.10f, or %f", C, C);
```

They are equal.

● 例4.9 求级数1 + x + x²/2! + x³/3! + ··· + x<sup>n</sup>/n! 的和.

#### 分析:

0、初始化:

定义一个变量sum用于存储和,初始值为1;

用户输入两个值,分别放在变量x和n中;

1、构数:

计算某一项 xi/i! 时隐含着循环(循环i次);

将某一项的值保存在变量item中;

2、累加:

依次将每一项item的值加到sum中去;

其中的"依次..."隐含着循环(循环n次)。

b/a

循环变量为j

j从1到i

循环变量为i

i从1到n

```
int main()
{ double x, sum = 1, item, b;
 int n, a, i, j;
 scanf("%lf%d", &x, &n);
 for (i=1; i \le n; i++)
  \{ a = 1, b = 1; \}
     for (j=1; j \le i; j++)
     { b *= x; // 计算x<sup>j</sup>
         a *= j; // 计算j!
                 // 计算x<sup>i</sup>/i!
     item = b/a;
                 // x<sup>i</sup>/i!加到sum中
     sum += item;
 printf("sum = %f \n", sum);
```

算法1

```
int main()
{ double x, sum = 1, item, b;
                                  , a = 1, b =
 int n, a, i, j;
 scanf("%lf%d", &x, &n);
                                  b *= x; // 计算x<sup>j</sup>
 for (i=1; i \le n; i++)
                                  a *= i; // 计算i!
  \{ a = 1, b = 1; 
     for (j=1; j \le i; j++)
                     // 计算x<sup>j</sup>
         b *= x;
          a *= j; // 计算j!
                     // 计算x<sup>i</sup>/i!
     item = b/a;
                  // x<sup>i</sup>/i!加到sum中
     sum += item;
 printf("sum = %f \n", sum);
```

```
int main()
{ double x, sum = 1, item, b = 1;
 int n, a = 1, i, j;
 scanf("%lf%d", &x, &n);
 for (i=1; i <= n; i++)
 { b *= x; // 计算x<sup>i</sup>
    a *= i; // 计算i!
    sum += item; // x<sup>i</sup>/i!加到sum中
 printf("sum = %f \n", sum);
```

算法2

### 利用item<sub>i</sub> = item<sub>i-1</sub>\*x/i进一步减少计算量

```
int main()
int n, a = 1, i, j;
 scanf("%lf%d", &x, &n);
 for (i=1; i <= n; i++)
 { b *= x; // 计算x<sup>i</sup>
                          item *= x/i;
   a *= i; // 计算i!
   sum += item; // x<sup>i</sup>/i!加到sum中
 printf("sum = %f \n", sum);
```

### 自学

```
int main()
{ double x, sum = 1, item = 1;
 int n, i;
 scanf("%lf%d", &x, &n);
 for (i=1; i<=n; i++)
    sum += item; // x<sup>i</sup>/i!加到sum中
 printf("sum = %f \n", sum);
 • • •
```

算法3

### 自学

- 算法3除了减少了计算量外; 当 x<sup>i</sup>/i! 不太大,而 x<sup>i</sup> 或 i! 很大以至 于超出计算机所能表示的数值范围时,算法1和算法2就不能得出正确的结果,由于算法3不直接计算 x<sup>i</sup> 或 i!,而是计算 x<sup>i</sup>/i!,因此算法3可行性更好。
- 算法3会带来精度损失,因为 x<sup>i</sup>/i! 是基于 x<sup>i-1</sup>/(i-1)! 的计算结果的,而 x<sup>i-1</sup>/(i-1)! 的计算结果有精度损失,并且这样的精度损失还会不断叠加。

## 逻辑型 (布尔型)

- 用来表示真假是非这样的逻辑概念。
- 逻辑型数据在计算机中占1个字节空间,实际存放的是0和1,逻辑型往往 也被归入整型。

stdbool.h

● 新标准规定逻辑型的类型关键字是\_Bool 或 bool。

### ● 逻辑常量

- → false (表示条件不成立)
- → true (表示条件成立)

- C89标准没有规定逻辑型
- 从C99标准开始规定了逻辑型,类型关键字是\_Bool,可用来定义逻辑型变量,并在头文件"stdbool.h"中定义宏名bool代替\_Bool,同时定义宏名false(代替整数0)和true(代替整数1)作为逻辑型常量

### ● 值域:

- → true、false
- → 它们一般是逻辑操作的操作数,以及关系操作或逻辑操作的结果

### • 操作集:

- → 逻辑操作
- → 关系操作
- → 赋值操作
- → 条件操作
- → 算术操作
- → 位操作
- **→** · · ·
- → 实际上是其对应的整数0和1在参与操作

- 3 > 4 的结果为false
- !(20>10) 的结果为false
- 当m为11时,!(m<10) 的结果为true
- 当m为3, n为4时, (m>1) && (n<20) 的结果为true
- ◆ 在m为0, n为4, 或m为3, n为21, 以及m为0, n为21时, (m>1) && (n<20) 的结果均为false
- 当m为0, n为21时, (m>1) || (n<20) 的结果为false
- ◆ 在m为0, n为4, 或m为3, n为21, 或m为3, n为4时, (m>1) | (n<20) 的结果均为true

● 例4.10 逻辑型数据不可以直接输入输出。

```
int main( )
 bool b = true;
 if(b)
    printf("true \n");
 else
    printf("false \n");
 return 0;
  //间接输出
```

```
cout << boolalpha << (2>1) ;
```

# 枚举类型的好处

- 可以约束该操作数不在所列举的值之外取值。
  - → 比如,对于星期这样的数据,如果用int型来描述,将会面临"1到底表示什么意思?"、"星期日用0还是7表示?"等问题,如果用0~6表示一个星期的每一天,则对于表示星期的int型变量d,不易防止"d = 10;"、"d = d\*2;"等逻辑错误。
  - → 当一个操作数可取的值只是有限的几个整数时,可以将其定义成某种枚举类型的变量,而不是定义成int型变量。
  - ★ 枚举类型通常与结构类型或联合类型结合起来使用。

## 枚举类型

- 是一种程序员用关键字enum构造出来的数据类型,程序员构造这种类型时,要一枚一枚列举出该类型变量所有可能的取值。根据构造的枚举类型再定义具体的枚举变量。
- 比如,
   enum Color {RED, YELLOW, BLUE};
   Color c1, c2, c3; 标识符的一种, 习惯用大写字母开头后面是小写字母的英文单词

Color是构造的枚举类型名,花括号里列出了Color类型变量可以取的值,它们又叫枚举符或枚举常量(标识符的一种,习惯用大写字母的英文单词表示)。c1、c2和c3是三个类型为Color的枚举变量,这三个变量的取值都只能是RED、YELLOW或BLUE。

- 程序执行到枚举类型的构造时,内存数据区不开辟空间存储各个枚举值
  - → 执行到变量的定义,内存数据区会开辟空间存储变量的值
- 同一作用域里,不能有相同的枚举值

```
int main()
{ enum Color3{RED, YELLOW, BLUE};
enum Color7{RED, ORANGE, YELLOW, GREEN, CYAN, BLUE, PURPLE}; ×
```

### 枚举类型变量的定义(四种形式都可以)

● <枚举类型名> <枚举类型变量名>;

```
enum Color {RED, YELLOW, BLUE};
Color c1, c2, c3;
```

♥ enum <枚举类型名> <枚举类型变量名>;

```
enum Color {RED, YELLOW, BLUE};
.....
enum Color c1, c2, c3; //加enum, 提醒Color是枚举类型
```

• enum <枚举类型名> {<枚举值表>} <枚举类型变量名>;

```
enum Color {RED, YELLOW, BLUE} c1, c2, c3;
```

● enum {<枚举值表>} <枚举类型变量名>;

```
enum {RED, YELLOW, BLUE} c1, c2, c3;
```

- 枚举变量所占空间大小与int型变量的相等
- 在计算机中实际存放的是枚举符对应的整数,默认情况下,花括号里第一个枚举符对应0,后面依次加1
- 也可以人为指定(不是赋值,因为构造类型时不在内存开辟空间)所对应的整数。
  - → 比如, enum Color {RED=1, YELLOW, BLUE}; 则YELLOW对应2, BLUE对应3。
- 如果人为指定不当,则虽然程序编译不出错,但运行结果可能会出错。
  - → 比如, enum Color {RED=2, YELLOW=1, BLUE};
    则BLUE对应2,这样, RED和BLUE对应相同的整数,会给后面的程序带来意想不到的错误。

#### ♥ 常见的枚举类型还有:

- → enum Weekday {SUN, MON, TUE, WED, THU, FRI, SAT};
- → enum Month {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};

### ● 逻辑型可看成C语言系统构造的一个枚举类型:

→ enum bool { false, true };

### 🍨 值域:

- → 实际上是若干个有名字的整型常量的集合
- ★ 枚举类型往往也被归入整型。

### • 操作集:

- → 算术操作
- → 关系和逻辑操作
- → 位操作
- → 赋值操作
- → 条件操作
- **→** · · ·
- → 实际上是其对应的整数在参与操作

◆ 只能把相同枚举类型的数据赋给枚举变量。

```
Weekday d1, d2;
d1 = SUN;
d2 = d1;

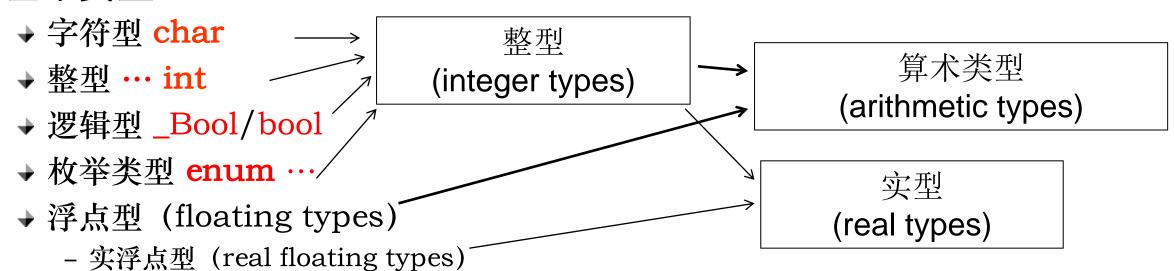
d1 = 1 ×
d1 = RED ×
```

enum Weekday {SUN, MON, TUE, WED, THU, FRI, SAT};

♥ 例4.11 枚举类型数据不可以直接输入输出。

```
int main( )
 enum Weekday {SUN, MON, TUE, WED, THU, FRI, SAT};
 Weekday d1 = SUN, d2 = SAT;
 if(d1 < d2)
    printf("Sunday is the first day of a week. \n");
 else
    printf("Which day is the first day of a week? \n");
 return 0;
```

#### 🍳 基本类型



- ✓ 单精度 float
- ✓ 双精度 double
- ✓ 长双精度 long double
- 复型

# 基本类型的转换

- 程序执行过程中,要求参加双目操作的两个操作数类型相同
- 当不同类型的操作数进行这类操作时,会进行类型转换,即操作数的类型会转换成另一种数据类型
- ◆ 这里的数据类型通常指的是基本类型,不是基本类型的两个不同类型的操作数往往不能 参加这类操作
- 类型转换方式有两种:
  - → 隐式类型转换: 由系统自动按一定规则进行的转换
  - → 显式类型转换: 由程序员在程序代码中标明,强制系统进行的转换
- 不管哪一种方式,类型转换都是"<mark>临时"的,即在类型转换过程中,操作数本身的类型并没有被转换,只是参加当前操作的数值被临时"看作"另一种类型的数值而已。</mark>

♥ 例4.12 基本类型的转换示例。

```
int main( )
 int r = 10;
                                //隐式类型转换
 float c = 2 * 3.14 * r;
 double s = 3.14 * (double)r * (double)r; //显式类型转换
 double v = 4.0 / 3 * 3.14 * r * r * r; // 隐式类型转换
 printf("%f %f %f"), c, s, v);
 return 0;
```

- 对于含有多个操作符的表达式,其类型转换过程是逐步进行的,而不是一次性将所有操作数转换成同种类型的数据再分别参加操作。
  - → 比如, "double v = 4/3\*3.14\*r\*r\*r;", v的结果为1\*3.14\*r\*r\*r→
    1.0\*3.14\*r\*r\*r → 3.14\*r\*r\*r →...
  - → 思考: 如果写成 "double v = 3.14\*r\*r\*r\*4/3;" 呢?

## 隐式类型转换规则

- 对于赋值操作,右操作数的类型转换为左边变量定义的类型;
- 对于逻辑操作、条件操作第一个表达式中的操作数、关系表达式的结果,不是bool型的数据,非0转换为true,0转换为false;
  - → 比如, 在a为0时, !(a)为true。

```
int i, sum=0;
scanf("%d", &i);
if(i)
    sum += i;
printf("%d \n", sum);
```

```
while(1)
```

高 long double
double
float
unsigned long
long
unsigned

int ← short, char, bool

▼ 对于其他双目操作(逗号操作除外),按"整型提升转换规则"和"<u>算术类型</u>转换规则"进行转换(一般是低精度类型操作数的类型转换为高精度类型操作数的类型)。

- 1) bool、char、signed char、unsigned char、short int、unsigned short int型的操作数,如果int型能够表示它们的值,则其类型转换成int,否则,转换成unsigned int。其中,bool型的操作数,false通常转换为0,true通常转换为1。
- 2) wchar\_t和枚举类型转换成下列类型中第一个能表示其所有值的类型: int、unsigned int、long int、unsigned long int。

- 1) 如果其中一个操作数类型为long double,则另一个的类型转换成long double。
- 2) 否则,如果其中一个操作数类型为double,则另一个的类型转换成double。
- 3) 否则,如果其中一个操作数类型为float,则另一个的类型转换成float。
- 4) 否则, 先对操作数进行整型提升转换, 如果转换后操作数的类型不一样, 则按下列规则再进行转换。
- 5) 如果其中一个操作数类型为unsigned long int,则另一个的类型转换成unsigned long int。
- 6) 否则,如果一个操作数类型为long int,另一个操作数类型为unsigned int,那么,如果long int能表示unsigned int的所有值,则unsigned int转换成long int,否则,两个操作数的类型都转化成unsigned long int。
- 7) 否则,如果一个操作数类型为long int,则另一个的类型转换成long int。
- 8) 否则,如果一个操作数类型为unsigned int,则另一个的类型转换成unsigned int。

- 隐式类型转换还会发生在函数调用及其值的返回过程中。
  - → C程序调用函数时,通常要求实参与形参类型一致。当函数的实参与形参类型不一致 时,系统会隐式地将实参的数据类型转换成形参的数据类型,再操作。
  - → 当函数的执行结果与函数定义的类型不同时,系统会隐式地将函数执行结果的数据 类型转换成函数定义的数据类型,再返回给调用者。

→ 类似于赋值操作



### ● 思考:

→!3 的结果为: true false

→ (-3) && true 的结果为: true false

→ true < false 的结果为: true false

→ 30 > 20 > 10的结果为: true false

→ 10 < 20 < 30的结果为: true false

## 显式类型转换的作用

隐式类型转换有时不能满足要求,于是C语言提供了显式类型转换机制,由程序员用类型关键字明确地指出要转换的类型,强制系统进行类型转换

。 → i+(int)j ♥ 例4.13 隐式类型转换存在的问题示例。

```
int main( )
                          不同类型的数据(i和j)在一起操作(算术、
                           比较操作),隐式类型转换的结果违背了常
                           识。
 int i = -10;
 unsigned int j = 3;
 if(i + j < 0) printf("-7 \n");
 else printf("error. \n"); //结果显示error
 if(i < j) printf("i<j \n");</pre>
 else printf("i>j \n"); //结果显示i>j
 return 0;
```

◎ 例4.13 隐式类型转换存在的问题示例。

```
int main( )
 int i = -10;
 unsigned int j = 3;
 if(i + (int)j < 0) printf("-7 n");
 else printf("error. \n");
 if(i < (int)j) printf("i<j \n");</pre>
 else printf("i>j \n");
 return 0;
```

利用显式类型转换,则结果可显示-**7**和i<i。

- 当把一个枚举值赋值给一个整型变量时,枚举值会隐式转换成整型,而当把一个整数赋给枚举类型的变量时,系统不会将整数转换成枚举类型数据,这时候可以用显式类型转换。
  - → 比如,

```
Weekday d;
d = (Weekday)(d + 1);
//如果写 "d = d+1;" 编译器会报错,因为d+1的结果为int类型
```

- 此外,对于一些对操作数类型有约束的操作,可以用显式类型转换保证操作的正确性。
  - ▶ 比如,C语言中的求余数运算要求操作数必须是整型数据, "int x = 10%3.4;" 应改为 "int x = 10%(int)3.4;", 否则,编译会出错。

# 伪随机数的生成-强制类型转换的应用

- ◆ 实际应用与程序设计中常常需要生成随机数。随机数的特性是产生前其值不可预测,产生后的多个数之间毫无关系。
- 真正的随机数是通过物理现象产生的,例如利用激光脉冲、噪声的强度和 掷骰子的结果等等,它们的产生对技术要求往往比较高。
- 一般情况下,通过一个固定的、可以重复的计算方法产生的伪随机数就可 以满足需求,它们具有与随机数类似的统计特征。
  - → 线性同余法是产生伪随机数的常用方法

```
● 例4.14 伪随机数的生成程序。
#define RANDOM MAX 65536
unsigned int Random();
                                     seed:
int main( )
                                              39022
  for (int i = 0; i < 10; i++)
                                              61087
     int j = Random();
     printf("%d \n", j);
            unsigned int Random()
  return 0; | {
                 static unsigned int seed = 1;
                 seed = (25163 * seed + 13859) % RANDOM MAX;
                 return seed;
```

```
unsigned long int next = 1;
unsigned int rand(void)
{
   next = next * 1103515245 + 12345;
   return (unsigned int) (next/65536) % RAND_MAX;
}
```

● 许多编译器的stdlib头文件中定义了生成伪随机数的函数rand/srand和预先定义为 32768的宏RAND\_MAX,用户可以直接调用。

```
unsigned long int next = 1; //全局变量next用来传递随机数的种子
unsigned int rand (void)
   next = next * 1103515245 + 12345; //两个常数是素数的成绩
   return (unsigned int) (next/65536) % RAND MAX;
void srand(unsigned int seed)
   next = seed;
```

● 函数中运用了显示类型转换。所生成的随机数的范围是0~RAND\_MAX-1。随机数种子 next是在另一个库函数srand中通过参数seed设置的

- ◆ 实际上,可以利用系统的时间函数time(0)的返回值来设置seed。
- 例4.15 调用库函数生成伪随机数。

```
#include <ctime>
#include <cstdlib>
int main( )
{ srand(time(0)); //time(0)取出的是从1970年1月1日到程序运行时刻的秒数
 for (int i = 0; i < 10; i++) //产生10个1~10之间的随机数
    int j = 1 + (int)(10.0 * rand() / RAND MAX);
    printf("%d \n", j);
 return 0;
```

```
先单独调用了一次 rand,相当于丢掉一个伪随机数,这是为了避免短期内产生的每
组伪随机数的第一个数都相同,因为短期内时间的变化不足以引起函数返回值的改变 */
                return (unsigned int) (next/65536) % RAND MAX;
#include <ctime>
#include <cstdlib>
int main( )
{ srand(time(0)); //time(0)取出的是从1970年1月1日到程序运行时刻的秒数
 rand();
 for (int i = 0; i < 10; i++) //产生10个1~10之间的随机数
    int j = 1 + (int)(10.0 * rand() / RAND MAX);
    printf("%d \n", j);
 return 0;
```

```
int main()
    int n;
    printf("请输入一个正整数:");
    scanf("%d", &n); //从键盘输入一个正整数
    for(int i=2; i<n; i++) //判断i是否为质数
        int j=2;
        while(j < i && i%j != 0) //判断j是否能被i整除
             j++;
        if(j == i) //i是质数
             printf("%d \t", i);
    printf(" \n");
    return 0;
                        偶数有必要判断是否为质数?
```

```
int main()
     int n;
    printf("请输入一个正整数:");
     scanf("%d", &n);
     for(int i=2; i<n; i++)
          int j=2;
          while(j < i && i%j != 0)
               j++;
          if(j == i)
               printf("%d \t", i);
    printf(" \n");
     return 0;
```

```
if(n < 2) return -1;
printf("2 \t");
for(int i=3; i<n; i+=2)</pre>
```

```
int main()
     int n;
     printf("请输入一个正整数:");
     scanf("%d", &n);
     if (n < 2) return -1;
     printf("2 \t");
     for(int i=3; i<n; i+=2)</pre>
          int j=2;
          while(j < i && i%j != 0)
               j++;
          if(j == i)
               printf("%d \t", i);
     printf(" \n");
     return 0;
```

j有必要到i-1?

```
int main()
     int n;
     printf("请输入一个正整数:");
     scanf("%d", &n);
     if (n < 2) return -1;
     printf("2 \t");
                              while(j <= sqrt(i)+1 && i%j!=0)
     for(int i=3; i<n; i+=2)
          int j=2;
          while(j < i && i%j != 0)
               j++;
                               if(j > sqrt(i)+1)
          if(j == i)
               printf("%d \t", i);
     printf(" \n");
     return 0;
```

```
int main()
     int n;
     printf("请输入一个正整数:");
     scanf("%d", &n);
     if (n < 2) return -1;
     printf("2 \t");
     for(int i=3; i<n; i+=2)
          int j=2;
          while(j <= sqrt(i)+1 && i%j != 0)
               j++;
          if(j > sqrt(i)+1)
               printf("%d \t", i);
     printf(" \n");
     return 0;
```

sqrt(i)被重复计算!

```
int main()
     int n;
    printf("请输入一个正整数:");
     scanf("%d", &n);
     if (n < 2) return -1;
                                     , k = sqrt((float)i)+1;
    printf("2 \t");
     for(int i=3; i<n; i+=2)
          int j=2, k = sqrt(i)+1;
          while(j <= sqrt(i)+1 && i%j != 0)
               j++;
                            while(j<=k && i%j!=0)
          if(j > sqrt(i)+1)
             printf("%d \t", i);
if(j > k)
    printf(" \n");
     return 0;
```

## 优化

```
int main()
    int n;
    printf("请输入一个正整数:");
     scanf("%d", &n);
     if (n < 2) return -1;
    printf("2 \t");
     for(int i=3; i<n; i+=2)
         if( prime(i) )
               printf("%d \t", i);
    printf(" \n");
     return 0;
```

```
long double sqrt(long double);
double sqrt(double);
float sqrt(float);
int main( )
    • • •
    int n;
    double sn = sqrt( n );
    • • •
                           double sn = sqrt( (double)n );
```

## 类型转换后的数据精度问题

- 操作数类型转换后,有的精度不受损失,有的则会损失精度。损失精度的 隐式类型转换会得到编译器的警告。
- eg. 隐式类型转换中,对于赋值运算与函数返回值,右操作数的类型转换为左边变量定义的类型,有可能会损失精度;对于其他运算,按"整型提升转换规则"和"算术类型转换规则"进行转换,一般精度不受损失。

```
int main( )
          double a=3.3, b=1.1;
double
          int i = a/b;
          printf("%d \n", i);
          return 0;
```

```
//...
int main()
{
    double a=3.3, b=1.1;
    printf("%.0f \n", a/b);
    return 0;
}
```

3

printf("%.17f\n",3.3/1.1);

2.999999999999960

要点:设计程序时,防止因为浮点数的不精确带来的问题

printf("%f\n",3.3/1.1);

3.000000

默认6位小数,且四舍五入

# 基本类型的选用

- 为了选择合适的基本类型定义变量或函数,需要注意考虑以下几个方面:
  - → 表达是否自然,比如将一个表示人数的变量定义成float型显然不合适;
  - → 可参与的操作与实际操作是否相符,比如需要对两个变量进行求余数运算,那么把 其中任一变量定义成double型都不合适;
  - → 值域与实际需求是否协调(是否浪费空间或溢出),比如将一批书的总价定义成 long double型或float型可能没有double型合适。

#### sizeof

- ◆ 在不同规格的计算机中,各种数据类型的数据实际占用空间的大小可能不同。
- C语言提供了操作符sizeof(操作数)来计算操作数实际占用内存空间的字节数,它是一个单目操作符,其中的操作数可以是各种表达式,也可以是表示基本类型的关键字。比如,

```
→ printf(("%d \n", sizeof(long));

→ printf(("%d \n", sizeof(3));

→ double x;

printf(("%d \n", sizeof(x+3));
```

● 对于sizeof(类型名)和sizeof(常量表达式),编译时就能确定其值。

# 复杂数据的描述方法简介

- C语言不仅提供了内置的基本类型来描述简单的数据,还提供了用基本类型构造新类型的机制,这些构造出来的新类型又叫派生类型,可以用来描述复杂数据,包括:
  - → 数组
  - → 指针
    - 字符串
  - → 结构及联合
    - 链表
    - 文件
- 第5章~第9章将分别介绍这几种派生类型的构造形式及其应用。

# 复杂数据的描述方法简介

- 程序语言一般不提供直接描述复杂数据的关键字。C语言没有完整的用来定义派生类型变量的类型关键字,需要在程序中用已有关键字或符号(例如int、struct,[],\*等)先构造类型,然后再定义相应的变量。
- 派生类型变量的存储方式和可取的值往往比较复杂,一般不能直接参与基本操作,需要程序员综合运用基本操作符、流程控制方法和模块设计方法设计特别的算法来处理。
- 更为复杂的数据则需要用专门的方法(比如图模型)来描述和处理。
- 在大数据时代,我们更加需要了解和掌握数据是怎么描述的,以便更好地组织和利用。

# 类型名的自定义

- C语言允许在程序中用关键字 typedef 将已有的类型名定义成另一个类型标识符。比如,
  - → typedef unsigned int Uint; //Uint 是unsigned int的别名
  - → typedef float Real;
  - → typedef double Speedt, Sumt; //Speedt和Sumt都是double的别名
- 类型的别名可以用来定义变量:
  - → Uint x; //等价于unsigned int x;
  - → Real y; //等价于float y;
  - → Speedt speed1, speed2; //等价于double speed1, speed2;
  - → Sumt sum1, sum2, sum3; //等价于double sum1, sum2, sum3;
- 实际上, typedef只是给已有数据类型取别名,并没有定义新类型。其作用是使程序简明、清晰,便于程序的阅读、编写和修改,增强程序的可移植性。特别是对于一些形式比较复杂,易于混淆、出错的类型(比如派生类型),可以用 typedef 定义成一个容易理解的别名,避免使程序晦涩难懂。

## 小结

- 将数据分类描述:
  - → 有助于合理分配内存空间 bool
  - → 便于计算 %
  - →保护数据 \*
  - ◆基本类型的数据通常可以由基本操作符直接操作,除逻辑型与枚举类型数据之外可以由库函数直接输入、输出。
  - → 基本数据类型在参加基本操作时,有可能按一定规则进行隐式或显式的类型转换。
  - → C语言不仅提供了内置的基本类型来描述简单的数据,还提供了用基本类型构造新 类型的机制,这些构造出来的新类型又叫派生类型,可以用来描述复杂数据

- 要求:
  - → 了解基本类型的值域与操作集
  - → 掌握基本类型的变量与函数定义方法,恰当选用基本类型实现简单的任务
    - 一个程序代码量≈30行
  - → 继续保持良好的编程习惯

- → 双目操作
- → 単目操作
  - ++/--(自增/减)
  - -! (逻辑非)
  - -~ (位非)
  - +/- (取正/负)
  - sizeof(算字节数)
  - **()** (强制类型转换)
  - **–** ...
- → 三目操作
  - -?: (条件)

# Thanks!

