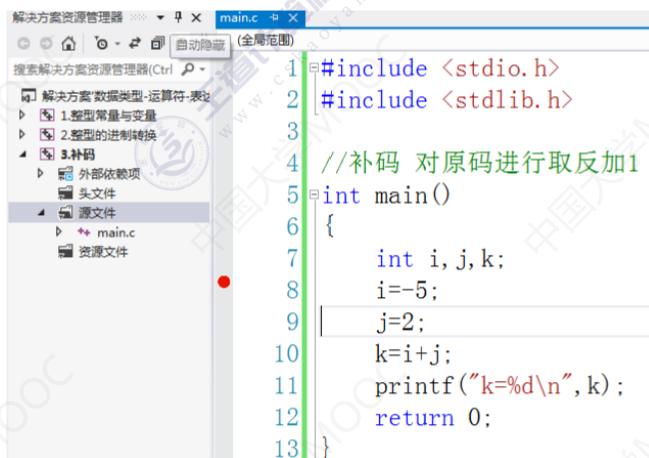


第 1 章 C 语言高级

1.1 整型，浮点型，字符型

1.1.1 补码的作用—组成原理考

计算机的 CPU 无法做减法操作，只能做加法操作。CPU 中有一个逻辑单元叫加法器。计算机所做的减法，都是通过加法器将其变化为加法实现的。那么减法具体是如何通过加法实现的呢？实现 $2 - 5$ 的方法是 $2 + (-5)$ 。由于计算机只能存储 0 和 1，因此我们编写程序来查看计算机是如何存储 -5 的，如图 1.1.1 所示，5 的二进制数为 101，称为原码。计算机用补码表示 -5 ，补码是对原码取反后加 1 的结果，即计算机表示 -5 时会对 5 的二进制数（101）取反后加 1，如图 1.1.2 所示。 -5 在内存中存储为 `0xfffffffffb`，因为对 5 取反后得 `0xfffffffffa`，加 1 后得 `0xfffffffffb`，对其加 2 后得 `0xfffffffffd`，见图 1.1.3，它就是 k 的值。当最高位为 1（代表负数）时，要得到原码才能知道 `0xfffffffffd` 的值，即对其取反后加 1（当然，也可以减 1 后取反，结果是一样的）得到 3，所以其值为 -3 。



```
#include <stdio.h>
#include <stdlib.h>
//补码 对原码进行取反加1
int main()
{
    int i, j, k;
    i=-5;
    j=2;
    k=i+j;
    printf("k=%d\n", k);
    return 0;
}
```

图 1.1.1 查看计算机如何存储 -5 的程序

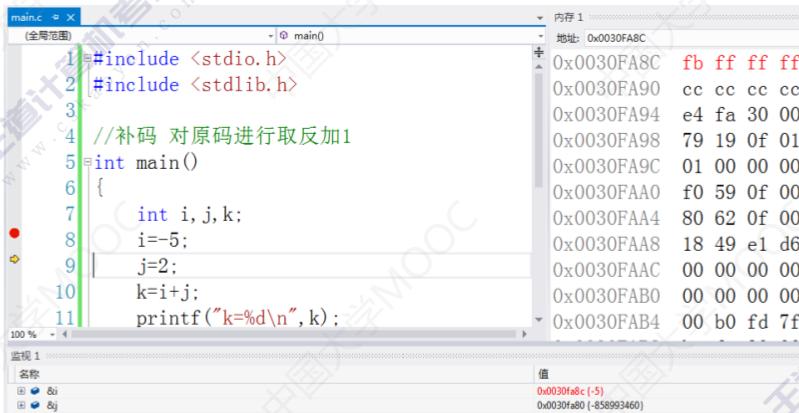


图 1.1.2-5 在内存中的存储方式

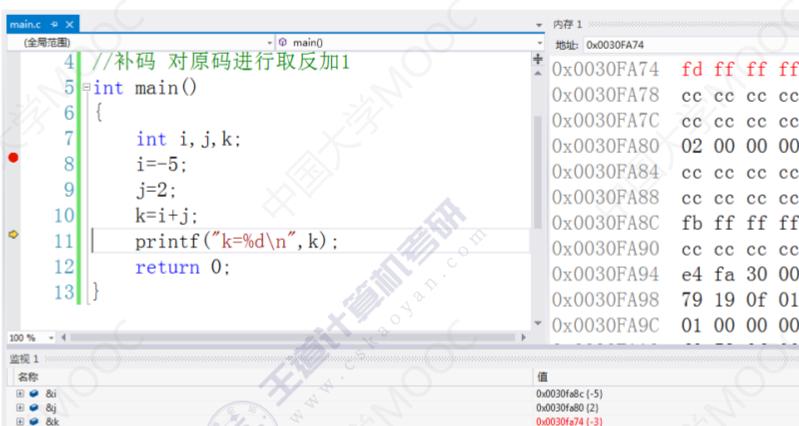


图 1.1.3-5 取反、加 1 并加 2 后的存储方式

反码:

反码是一种在计算机中数的机器码表示。对于单个数值（二进制的 0 和 1）而言，对其进行取反操作就是将 0 变为 1，1 变为 0。正数的反码和原码一样，负数的反码就是在原码的基础上符号位保持不变，其他位取反。

十进制	原码	反码
6	0000 0110	0000 0110
-3	1000 0011	1111 1100

1.1.2 整型变量

整型变量包括 6 种类型，如图 1.1.4 所示，其中有符号基本整型与无符号基本整型的最高位所代表的意义不同，如图 1.1.5 所示。不同整型变量表示的整型数的范围如表 1.1.1 所示，超出范围会发生溢出现象，导致计算出错。

有符号基本整型 `(signed)int`
 有符号短整型 `(signed)short(int)`
 有符号长整型 `(signed)long(int)`
 无符号基本整型 `unsigned int`
 无符号短整型 `unsigned short(int)`
 无符号长整型 `unsigned long(int)`
 注意：括号表示其中的内容是可选的。

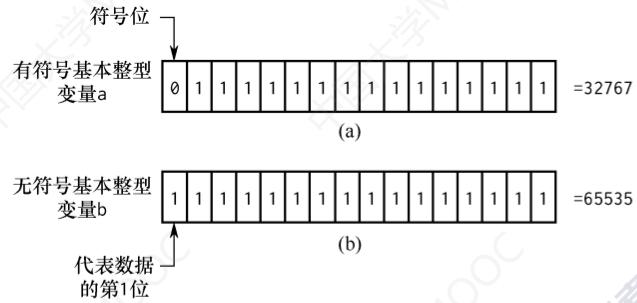


图 1.1.4 整型变量的 6 种类型

图 1.1.5 有符号基本整型与无符号基本整型的最高位所代表的意义

表 1.1.1 不同整型变量表示的整型数的范围

类 型	类型说明符	长 度	整型数的范围
基本整型	<code>int</code>	4 字节	$-2^{31} \sim (2^{31}-1)$
短整型	<code>short</code>	2 字节	$-2^{15} \sim (2^{15}-1)$
长整型	<code>long</code>	4 字节 (64 位为 8 字节)	$-2^{31} \sim (2^{31}-1)$ 或 $-2^{63} \sim (2^{63}-1)$
无符号整型	<code>unsigned int</code>	4 字节	$0 \sim (2^{32}-1)$
无符号短整型	<code>unsigned short</code>	2 字节	$0 \sim 65535$
无符号长整型	<code>unsigned long</code>	4 字节 (64 位为 8 字节)	$0 \sim (2^{32}-1)$ 或 $0 \sim (2^{64}-1)$

由图 1.1.6 可以看出不同类型的整型变量的定义方法。下面介绍溢出。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 //不同整型变量的定义方法; 溢出, 防止溢出
4 int main()
5 {
6     int i=10;
7     short a=32767;//2字节
8     short b=0;
9     long c=100;
10    unsigned int m=3;
11    unsigned short n=4;
12    unsigned long k=5;
13    b=a+1;
14    printf("b=%d\n", b); //发生溢出
15    system("pause");

```

图 1.1.6 不同类型的整型变量的定义方法

有符号短整型数可以表示的最大值为 32767，当我们对其加 1 时，`b` 的值会变为多少呢？实际运行打印得到的是 -32768。为什么会这样？因为 32767 对应的十六进制数为 `0x7fff`，加 1 后变为 `0x8000`，其首位为 1，因此变成了一个负数。取这个负数的原码后，就是其本身，值为 32768，所以 `0x8000` 是最小的负数，即 -32768。这时就发生了溢出，我们对 32767 加 1，希望得到的值是 32768，但结果却是 -32768，因此导致计算结果错误。在使用整型变量时，一定要注意数值的大小，数值不能超过对应整型数的表示范围。有的读者可能会问在编写的程序中数值大于 $2^{64}-1$ 时怎么办？答案是可以自行实现大整数加法，详见后面介绍数组时的说明。

图 1.1.6 中代码的执行结果如图 1.1.7 所示。

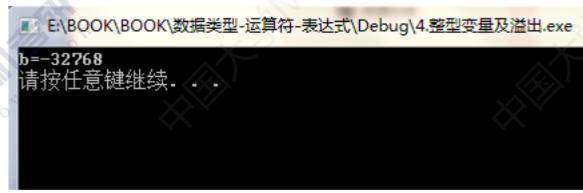


图 1.1.7 图 1.1.6 中代码的执行结果

思考题：针对上例中的溢出问题，如何修改程序才能让打印的 b 值是 32768 而不是 -32768？

1.1.3 浮点型变量

在 C 语言中，要使用 `float` 关键字或 `double` 关键字定义浮点型变量。`float` 型变量占用的内存空间为 4 字节，`double` 型变量占用的内存空间为 8 字节。与整型数据的存储方式不同，浮点

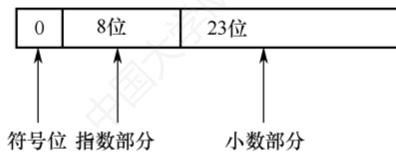


图 1.1.8 浮点型数据的组成

型数据是按照指数形式存储的。系统把一个浮点型数据分成小数部分（用 M 表示）和指数部分（用 E 表示）并分别存放。指数部分采用规范化的指数形式，指数也分正、负（符号位，用 S 表示），如图 1.1.8 所示。

数符占 1 位，是 0 时代表正数，是 1 时代表负数。表 1.1.2 是 IEEE-754 浮点型变量存储标准。

表 1.1.2 IEEE-754 浮点型变量存储标准

格式	SEEEEEEE	EMMMMMMM	MMMMMMMM	MMMMMMMM
二进制数	01000000	10010000	00000000	00000000
十六进制数	40	90	00	00

S：S 是符号位，用来表示正、负，是 1 时代表负数，是 0 时代表正数。

E：E 代表指数部分，指数部分运算前都要减去 127（这是 IEEE-754 的规定），因为还要表示负指数。这里的 10000001 转换为十进制数为 129， $129 - 127 = 2$ ，即实际指数部分为 2。

M：M 代表小数部分，这里为 0010 0000 0000 0000 0000 0000。底数左边省略存储了一个 1，使用的实际底数表示为 1.00100000000000000000000000000000。

下面以浮点数 4.5（十进制数）为例具体介绍。

计算机并不能计算 10 的幂次，指数值为 2，代表 2 的 2 次幂，因此将 1.001 向左移动 2 位即可，也就是 100.1；然后转换为十进制数，整数部分是 4，小数部分是 2^{-1} ，刚好等于 0.5，因此十进制数为 4.5。浮点数的小数部分是通过 $2^{-1} + 2^{-2} + 2^{-3} + \dots$ 来近似一个小数的。

下面介绍浮点数的精度控制。

浮点型变量分为单精度 (`float`) 型、双精度 (`double`) 型和长双精度 (`long double`) 型三类。如表 1.1.3 所示，因为浮点数使用的是指数表示法，所以我们不用担心数值的范围，也不用去看浮点数的内存。我们需要注意的是浮点数的精度问题，如图 2.5.2 所示，我们赋给 a 的值为 1.23456789e10，加 20 后，应该得到的值为 1.234567892e10，但结果却是 1.23456788e+010，变得更小了。我们将这种现象称为精度丢失，因为 `float` 型数据能够表示的有效数字为 7 位，最多只保证 1.234567e10 的正确性，要使结果正确，就需要把 a 和 b 均改为 `double` 型。

表 1.1.3 浮点数的数值范围与有效数字

类 型	位 数	数 值 范 围	有 效 数 字
float	32	$10^{-37} \sim 10^{38}$	6 ~ 7 位
double	64	$10^{-307} \sim 10^{308}$	15 ~ 16 位
long double	128	$10^{-4931} \sim 10^{4932}$	18 ~ 19 位

```
main.c 9 x
(全局范围) ① main()
4 //浮点数-防止精度丢失
5 int main()
6 {
7     float n=1.456; //近似值, 判断时要用减法
8     long double m=1.456;
9     double e=1e-6; //8字节
10    float a=1.23456789e10;
11    float b;
12    b=a+20;
13    printf("b=%f\n", b);
14    system("pause");
}
100 % - 4
监视 1
名称 值
n 1.45599997
m 1.4560000000000000
b 1.23456788e+010
```

图 1.1.9 验证精度丢失现象的程序

图 1.1.9 中程序的执行结果如图 1.1.10 所示。



图 1.1.10 图 1.1.9 中程序的执行结果

思考题：把上例程序中的 a 和 b 都改为 double 型，实际求和后 b 的值是否正确？

1.1.4 字符型常量

用单引号括起来的一个字符是字符型常量，且只能包含一个字符！例如，'a'、'A'、'1'、' '是正确的字符型常量，而'abc'、'a'、" "是错误的字符型常量。表 1.1.4 中给出了各种转义字符及其作用。以“\”开头的特殊字符称为转义字符，转义字符用来表示回车、退格等功能键。

表 1.1.4 各种转义字符及其作用

转义字符	作 用
\n	换行
\t	横向跳格
\r	回车
\\\	反斜杠
\b	退格
\0	空字符，用于标示字符串的结尾，它不是空格，无法打印
\ddd	ddd 表示 1~3 位八进制数，用处不大
\xhh	hh 表示 1~2 位十六进制数，用处不大

为便于理解转义字符,请读者按自己的想法编写图 1.1.11 所示的程序并运行,查看打印效果。

```
char d, e='c'; // 占用1字节
char e='\t'; // 横向跳格, 显示4个空格
char f='\' ;
char h='\\' ;
char i='\0'; //字符串的结束符, 空字符
d=c-32;
printf("\123\n");//转义八进制数, 变为十进制数是83, 打印的是字符S
printf("\x40\n");//转义十六进制数, 变为十进制数是64, 打印的是字符C
printf("d=%c\n", d); //得到大写字母C
printf("c=%d\n", c); //以十进制数形式输出字符的ASCII码值
printf("e=%chaha\n", e);
printf("abc\rd\n"); //r回到行首, 所以得到dbc
printf("abc%cd\n", f); //b是向前退一格
printf("%c\\\\n", h); //要输出一个\,需要转义
printf("i=%cb\b\n", i); //空字符什么都没有
cout<<"noone".
```

图 1.1.11 理解转义字符的程序示例

图 1.1.11 中程序的执行效果如图 1.1.12 所示。

```
S
e
d=C
c=99
e=      haha
dbc
abd
\\
i= b
请按任意键继续... .
```

图 1.1.12 图 1.1.11 中程序的执行效果

思考题: 为什么 abc\rd 打印出的效果是 dbc?

1.2 混合运算

字符型 (char)、整型 (包括 int、short、long)、浮点型 (包括 float、double) 数据可以

混合运算。运算时, 不同类型的数据首先要转换为同一类型, 然后进行运算。不同类型数据的转换级别如图 1.2.1 所示, 从短字节到长字节的类型转换是由系统自动进行的, 编译时不会给出警告; 若反向进行, 则编译时编译器会给出警告。

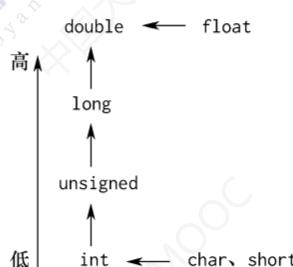


图 1.2.1 不同类型数据的转换级别

1.2.1 数值按 int 型运算

C 语言中的整型数算术运算总是以默认整型类型的精度进行的。为了获得这个精度, 表达式中的字符型和短整型操作数在使用之前会被转换为基本整型 (int 型) 操作数, 这种

转换被称为整型提升 (Integral Promotion)。例如,

```
char a,b,c;
a = b + c;
```

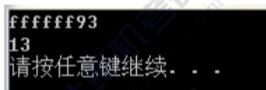
其中，**b** 和 **c** 的值首先被提升为基本整型数，然后执行加法运算。加法运算的结果将被截短，然后存放到 **a** 中。这个例子的结果和使用 8 位算术运算的结果相同，但在下面的例 1.2.1 中，结果不再相同。

【例 1.2.1】 整型常量默认按 int 型运算实例。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char b = 0x93<<1>>1;
    printf("%x\n",b);//fffff93
    b = 0x93<<1;
    b = b>>1;
    printf("%x\n",b);//13
    system("pause");
}
```

执行结果如图 1.2.2 所示。



```
ffffff93
13
请按任意键继续...
```

图 1.2.2 例 1.2.1 中程序的执行结果

为什么采用十六进制数打印呢？这是因为输出时 `%x` 是取 4 字节进行输出的；**b** 中存储的只有 93，为什么前面却打印出了 3 字节的 ff 也就是 6 个 ff 呢？如果用 `%d` 输出，那么可以得到一个负值，当我们用 `%x` 输出一个少于 4 字节的数时，前面补的字节是按照对应数据的最高位来看的，因为字符 **b** 的最高位为 1，所以其他 3 字节补的都是 1。

为什么把操作分成两步后，**b** 的值就为 13 呢？因为 `0x93` 左移一位时，虽然按 4 字节进行，但是最低一字节的值为 `0x26`，赋值给 **b** 后，**b** 内存储的就是 `0x26`，这时再对 **b** 进行右移时，单字节拿到寄存器中是按 4 字节运算的，但是因为 **b** 的最高位为零，因此拿到寄存器中按 4 字节运算，前面补的都是零，再右移一位表示除以 2，因此得到的值是 13。

另外一种场景是我们对两个整型常量做乘法，并赋值给一个长整型变量，对编译器来讲这是按照 `int` 型进行的。如例 1.2.2 所示，打印结果为 0，无论是在 VS 中新建 Win32 控制台应用程序并在 32 位下执行，还是在 Linux 下将其编译为 64 位的可执行程序，执行结果均为 0。那么怎么解决这种类型问题呢？我们来看例 1.2.2。

【例 1.2.2】 两个较大常量相乘溢出实例。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    long l;
```

```
I = 131072*131072;  
printf("%ld\n",I);  
system("pause");  
}
```

我们可以在做乘法前，将整型数强制转换为 long 型，32 位控制台应用程序代码如例 1.2.3 所示，在 32 位操作系统下，long long 型只占 8 字节而 long 型占 4 字节。

【例 1.2.3】 32 位程序两个较大的常量相乘不会溢出实例。

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main()  
{  
    long long I;  
    I = (long long)131072*131072;  
    printf("%lld\n",I);  
    system("pause");  
}
```

上例代码的输出结果如图 1.2.3 所示，如果是 64 位程序，如例 1.2.3 所示，那么转化为 long 型即可。

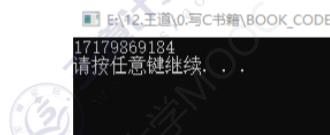


图 1.2.3 例 1.2.3 程序的输出结果

【例 1.2.4】 64 位程序两个较大的常量相乘不会溢出实例（Linux 系统下）。

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main()  
{  
    long I;  
    I = (long)131072*131072;  
    printf("%ld\n",I);  
    system("pause");  
}
```

1.2.2 浮点型常量默认按 double 型运算

浮点型常量默认按 8 字节运算，如例 1.2.5 所示。

【例 1.2.5】 浮点型常量运算实例。

```
#include <stdio.h>
```

```
#include <stdlib.h>
//浮点型常量默认按 8 字节运算
int main()
{
    float f = 12345678900.0+1;
    double d = f;
    printf("%f\n",f);
    printf("%f\n",12345678900.0+1);
    system("pause");
    return 0;
}
```

输出结果如图 1.2.4 所示。



```
12345678848.000000
12345678901.000000
请按任意键继续...
```

图 1.2.4 例 1.2.5 程序的输出结果

第一个打印的值只有 7 位精度，原因是单精度浮点数 `f` 只有 4 字节的存储空间，能够表示的精度是 6 ~ 7 位，所以只保证 1 ~ 7 位是正确的，后面的都是近似值。第二个打印的值是正确的浮点型常量，它是按 8 字节即 `double` 型进行运算的，同时 `%f` 会访问寄存器 8 字节的空间进行浮点运算，因此可以正常输出。

1.3 位运算符

位运算符 `<<`、`>>`、`~`、`|`、`^`、`&` 依次是左移、右移、按位取反、按位或、按位异或、按位与。

左移：高位丢弃，低位补 0，相当于乘以 2。工作中很多时候申请内存时会用左移，例如要申请 1GB 大小的空间，可以使用 `malloc(1<<30)`。`malloc` 函数的使用将在后面的章节中介绍。

右移：低位丢弃，正数的高位补 0（无符号数我们认为是正数），负数的高位补 1，相当于除以 2。移位比乘法和除法的效率要高，负数右移，对偶数来说是除以 2，但对奇数来说是先减 1 后除以 2。例如，`-8>>1`，得到的是 `-4`，但 `-7>>1` 得到的并不是 `-3` 而是 `-4`。另外，对于 `-1` 来说，无论右移多少位，值永远为 `-1`。

异或：相同的数进行异或时，结果为 0，任何数和 0 异或的结果是其本身。

按位取反：数位上的数是 1 变为 0，0 变为 1。

按位与和按位或：用两个数的每一位进行与和或。

图 1.3.1 所给出的例子的执行结果如图 1.3.2 所示，有兴趣的读者可以自己改动一下。



图 1.3.1 位运算符应用实例

```
5
i&f=5
i|f=7
~i=-6
请按任意键继续... . . .
```

图 1.3.2 图 1.3.1 中程序的执行结果

思考题:

- 有两个变量 **a** 与 **b**, 在不使用第三个变量的情况下, 通过异或操作来交换这两个变量的值, 这种交换相对于之前的加法交换有何优势?
- 如何用位运算找到一个数的最低位为 1 的那一位 (不可使用循环)? 要求复杂度为 $O(1)$.
- C 语言中未提供循环移位的运算符, 有兴趣的读者可以自行尝试实现一个能够循环移位的函数。

1.4 选择与循环

1.4.1 switch 语句

判断的一个变量可以等于几个值或几十个值时, 使用 **if** 和 **else if** 语句会导致 **else if** 分支非常多, 这时可以考虑使用 **switch** 语句, **switch** 语句的语法格式如下:

```
switch (表达式)
{case 常量表达式 1:语句 1
 case 常量表达式 2:语句 2
 ...
 case 常量表达式 n :语句 n
 default      :语句 n+1
}
```

下面来看一个使用 **switch** 语句的例子。如例 1.4.1 所示, 输入一个年份和月份, 然后打印对应月份的天数, 如输入一个闰年和 2 月, 则输出为 29 天。具体代码如下所示, 对应的电子附件项目名称为“**switch 月份 1**”, 读者会发现, **switch** 语句中 **case** 后面的常量表达式的值不是

按照 1 到 12 的顺序排列的，这里要说明的是，switch 语句匹配并不需要常量表达式的值有序排列，输入值等于哪个常量表达式的值，就执行其后的语句，每条语句后需要加上 break 语句，代表匹配成功一个常量表达式时就不再匹配并跳出 switch 语句。

【例 1.4.1】switch 语句的使用。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int mon,year;
    while(scanf("%d%d",&year,&mon)!=EOF)
    {
        switch (mon)
        {
            case 2:printf("mon=%d is %d days\n",mon,28+(year%4==0&&year%100!=0||year%400==0));break;
            case 1:printf("mon=%d is 31days\n",mon);break;
            case 3:printf("mon=%d is 31days\n",mon);break;
            case 5:printf("mon=%d is 31days\n",mon);break;
            case 7:printf("mon=%d is 31days\n",mon);break;
            case 8:printf("mon=%d is 31days\n",mon);break;
            case 10:printf("mon=%d is 31days\n",mon);break;
            case 12:printf("mon=%d is 31days\n",mon);break;
            case 4:printf("mon=%d is 30days\n",mon);break;
            case 6:printf("mon=%d is 30days\n",mon);break;
            case 9:printf("mon=%d is 30days\n",mon);break;
            case 11:printf("mon=%d is 30days\n",mon);break;
            default:
                printf("error mon\n");
        }
    }
    system("pause");
    return 0;
}
```

如果一个 case 语句后面没有 break 语句，那么程序会继续匹配下面的 case 常量表达式。

例 1.4.2 中的代码是对例 1.4.1 中代码的优化，对应的电子附件项目名称为“switch 月份 2”。

例 1.4.2 中的代码执行效果和上面的代码执行效果一致，原理是只要匹配到 1、3、5、7、8、10、12 中的任何一个，就不再拿 mon 与 case 后的常量表达式的值进行比较，而执行语句 printf("mon=%d is 31days\n",mon)，完毕后执行 break 语句跳出 switch 语句。switch 语句最后加入 default 的目的是，在所有 case 后的常量表达式的值都未匹配时，打印输出错误标志或者一些提醒，以便让程序员快速掌握代码的执行情况。

【例 1.4.2】日期实例改进。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int mon,year;
    while(scanf("%d%d",&year,&mon)!=EOF)
    {
        switch (mon)
        {
            case 2:printf("mon=%d is %d days\n",mon,28+(year%4==0&&year%100!=0||year%400==0));break;
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:printf("mon=%d is 31days\n",mon);break;
            case 4:
            case 6:
            case 9:
            case 11:printf("mon=%d is 30days\n",mon);break;
            default:
                printf("error mon\n");
        }
    }
    system("pause");
    return 0;
}
```

1.4.2 do while 循环

do while 语句的特点是：先执行循环体，后判断循环条件是否成立。其一般形式为

```
do
{
    循环体语句;
}
while (表达式);
```

执行过程如下：首先执行一次指定的循环体语句，然后判断表达式，当表达式的值为非零（真）时，返回重新执行循环体语句，如此反复，直到表达式的值等于 0 为止。图 1.4.1 是使用 do while 语句计算 1 到 100 之间所有整数之和的例子，do while 语句与 while 语句的差别是，do while 语句第一次执行循环体语句之前不会判断表达式的值，也就是如果 i 的初值为 101，那么依然会进入循环体。实际工作中 do while 语句应用较少。

```
int main()
{
    int i=1;
    int total=0;
    do
    {
        total=total+i;
        i++;
    }while(i<=100); //这里必须加分号,否则编译不通
    printf("total=%d\n",total);
    system("pause");
    return 0;
}
```

图 1.4.1 do while 语句计算 1 到 100 之间的所有整数之和

下面来看一个某公司关于 do while 语句的面试题，如例 1.4.3 所示。

【例 1.4.3】 do while 语句的面试题。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int i=1;
    do{
        printf("%d\n",i);
        i++;
        if(i<15)
        {
            continue;
        }
    }while(0);
    system("pause");
}
```

例 1.4.3 中代码的输出结果是多少？输出结果如图 1.4.2 所示。为什么只会输出 1？这是很多读者容易犯的错误，即以为执行了 continue 语句就会跳过 while 内的表达式判断，其实不然，continue 语句只会跳过其下面的代码部分，依然要进行 while 内表达式的判断。因为 while 内的表达式为假，所以代码只会执行一次。

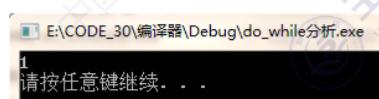
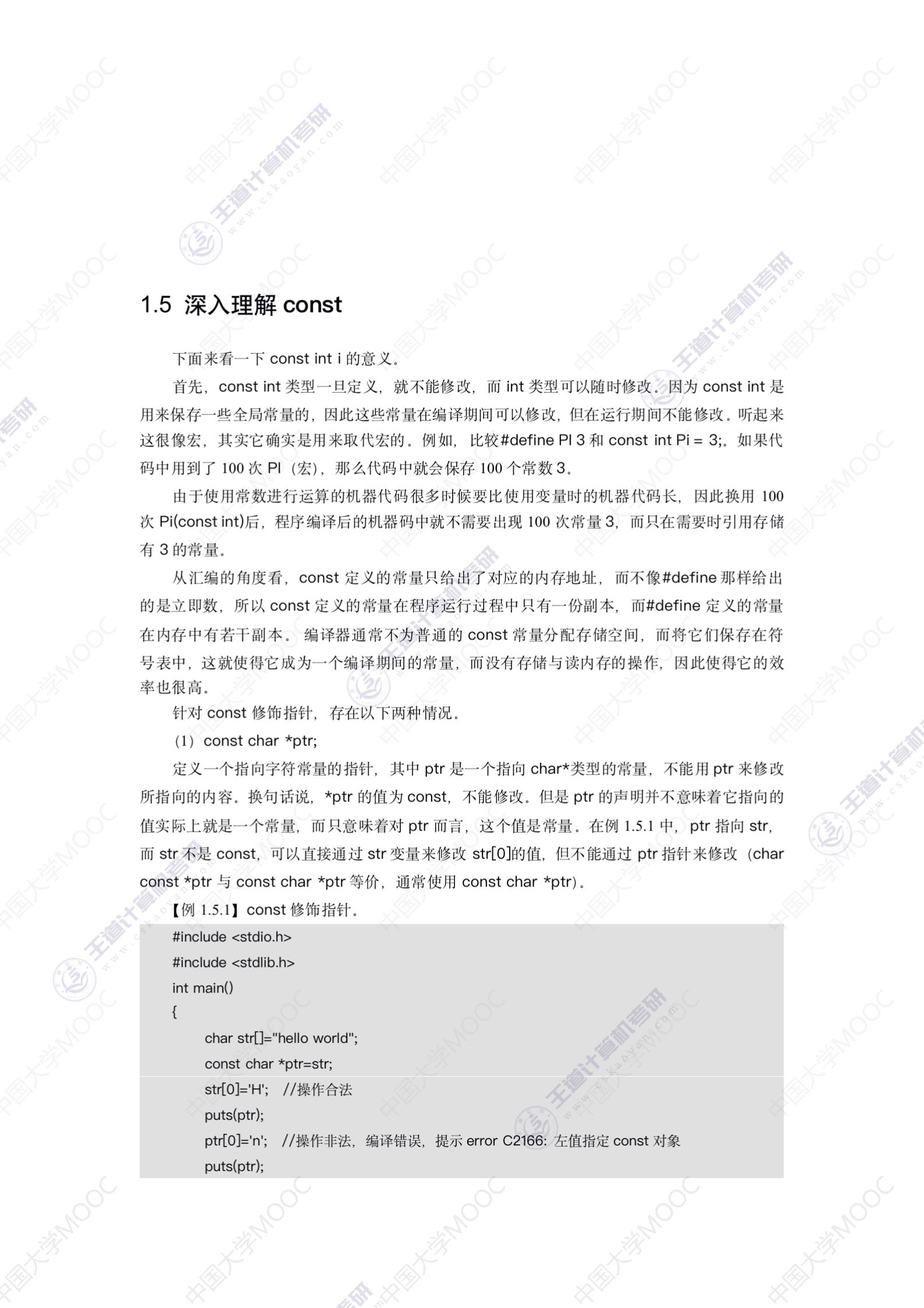


图 1.4.2 例 1.4.3 中代码的执行结果



1.5 深入理解 `const`

下面来看一下 `const int i` 的意义。

首先，`const int` 类型一旦定义，就不能修改，而 `int` 类型可以随时修改。因为 `const int` 是用来保存一些全局常量的，因此这些常量在编译期间可以修改，但在运行期间不能修改。听起来这很像宏，其实它确实是用来取代宏的。例如，比较`#define PI 3` 和 `const int Pi = 3;`。如果代码中用到了 100 次 `PI`（宏），那么代码中就会保存 100 个常数 3。

由于使用常数进行运算的机器代码很多时候要比使用变量时的机器代码长，因此换用 100 次 `PI(const int)` 后，程序编译后的机器码中就不需要出现 100 次常量 3，而只在需要时引用存储有 3 的常量。

从汇编的角度看，`const` 定义的常量只给出了对应的内存地址，而不像`#define` 那样给出的是立即数，所以 `const` 定义的常量在程序运行过程中只有一份副本，而`#define` 定义的常量在内存中有若干副本。编译器通常不为普通的 `const` 常量分配存储空间，而将它们保存在符号表中，这就使得它成为一个编译期间的常量，而没有存储与读内存的操作，因此使得它的效率也很高。

针对 `const` 修饰指针，存在以下两种情况。

(1) `const char *ptr;`

定义一个指向字符常量的指针，其中 `ptr` 是一个指向 `char*`类型的常量，不能用 `ptr` 来修改所指向的内容。换句话说，`*ptr` 的值为 `const`，不能修改。但是 `ptr` 的声明并不意味着它指向的值实际上就是一个常量，而只意味着对 `ptr` 而言，这个值是常量。在例 1.5.1 中，`ptr` 指向 `str`，而 `str` 不是 `const`，可以直接通过 `str` 变量来修改 `str[0]`的值，但不能通过 `ptr` 指针来修改 (`char const *ptr` 与 `const char *ptr` 等价，通常使用 `const char *ptr`)。

【例 1.5.1】 `const` 修饰指针。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char str[]="hello world";
    const char *ptr=str;
    str[0]='H'; //操作合法
    puts(ptr);
    ptr[0]='n'; //操作非法，编译错误，提示 error C2166: 左值指定 const 对象
    puts(ptr);
```

```
        system("pause");
    }
```

(2) char * const ptr;

定义一个指向字符的指针常量，即 const 指针。由例 1.5.2 可知，不能修改 ptr 指针，但可以修改该指针指向的内容。

【例 1.5.2】const 修饰变量。

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char str[]="hello world";
    char str1[]="how do you do";
    char * const ptr=str;
    str[0]='H';
    puts(ptr);
    ptr[0]='n'; //合法
    puts(ptr);
    ptr=str1; //非法，编译错误，error C2166: 左值指定 const 对象
    system("pause");
}
```

由例 1.5.1 可以看到，const 直接修饰指针时，指针 ptr 指向的内容可以修改，但是指针 ptr 在第一次初始化后，后面不能再对 ptr 进行赋值，否则会出现编译报错，当然这种场景使用得并不多。

1.6 结构体对齐原理

数据类型自身的对齐值如下：

对于 char 型数据，其自身对齐值为 1，对于 short 型为 2，对于 int,float,double 类型，其自身对齐值为 4，单位字节。