第四章 循环

C++简明双链教程（李昕著，清华大学出版社）

作者：李昕

PPT制作者：廖集秀

目录

01 while 循环

while循环的基本语法格式

1. 语法格式

while的语法格式与if的单分支语法格式完全相同，但：

- if在符合条件后只能执行一次
- while在符合条件时反复执行，直到条件不符合退出循环。

```
1   while(条件)
2   {
3       代码块;
4   }
```

2. 循环体

第2-4行称为循环体，当第1行的条件符合时，循环体被反复执行，当条件不符合时，循环体不会被执行。

ps：注意第1行末尾不能有分号，分号表示空语句，形成了一个无效的空循环。

例题4.1

1~N求和。请求出100以内，1至任意数之和。

样例输入	样例输入
100	5050

```cpp
#include<iostream>
using namespace std;

int main (  )
{
    int sum=0,i=1,count;
    cin>>count;                    //循环控制变量初始化
    while(i<=count)                //循环条件
    {
        sum+=i;
        ++i;                       //循环控制变量的改变
    }
    cout<<sum<<endl;
}
```

循环三要素

1. **循环控制变量初始化；** → 定义了循环的起点

2. **循环条件；** → 界定了循环的终点

3. **循环控制变量的改变。** → 决定了循环的方向和改变的步长

每个循环必须包含这三个要素。

例题4.1

1~N求和。请求出100以内，1至任意数之和。

样例输入	样例输入
100	5050

典型的累积求和过程，注意第6行的**累积变量**必须要先**初始化为0**。

变量在使用前必须进行初始化，否则结果是不确定的。

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       int sum=0,i=1,count;
7       cin>>count;                    //循环控制变量初始化
8       while(i<=count)                //循环条件
9       {
10          sum+=i;
11          ++i;                       //循环控制变量的改变
12      }
13      cout<<sum<<endl;
14  }
```

累积变量必须要先初始化为0!

例题4.1

1~N求和。请求出100以内，1至任意数之和。

样例输入	样例输入
100	5050

使用while完成**指定次数的循环**，可以采用以下简约的形式。

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       int sum=0,i;
7       cin>>i;                    //循环控制变量初始化
8       while(i--)                 //循环条件
9       {
10          sum+=(i+1);
11      }
12      cout<<sum<<endl;
13  }
```

1. 采用反向循环，当i为0时，循环条件为false，停止循环。

2. 第8行先进行判断，然后执行了自减1操作，循环次数得到了保障。

对于这个题目而言，i既作为循环控制变量，也在第10行参与了运算。第10行的i已经是被自减1后的结果，因此要改为i+1。

知识点

索引	要点	正链	反链
T411	循环三要素的作用和基本使用方法		
T412	熟练掌握while(变量--)的循环次数控制方法	T244 T268	

02 do-while循环

do-while循环的基本语法格式

1. 语法格式

- do-while循环**先执行**循环体中的语句，然后**再判断**条件是否为真。
- while循环是**先判断再执行**。

所以while循环可能一遍也不执行，而do-while循环的第一遍是一定要执行。

```
1   do
2   {
3       代码块;
4   } while (条件);
```

ps：注意第4行最后一定要有一个分号；，表示do-while循环的结束。

例题4.2

猜数游戏。要求猜一个介于1～10之间的数字，根据用户猜测的数与标准值进行对比，并给出提示，以便下次猜测能接近标准值，直到猜中为止。

样例输入	样例输入
3 8 5	太小 太大 答案是：5

```
1  #include<iostream>
2  #include<ctime>
3  using namespace std;
4
5  int main (  )
6  {
7      srand(time(NULL));                    //随机数的种子
8      int magic=rand()%10+1;                //将随机数控制在1-10之间
9      int guess;
10     do
11     {
12         cin>>guess;
13         if (guess > magic)
14             cout<<"太大\n";
15         else if (guess < magic)
16             cout<<"太小\n";
17     }while (guess != magic);
18     cout<<"答案是: "<<guess<<endl;
19 }
```

1. 第7行是随机数种子，当种子相同时，随机数产生的序列总是相同的。

2. 第8行通过取余和加1操作，将rand产生的随机整数控制到1-10的范围内。

3. 采用do-while循环，保证玩者至少要猜一次。

随堂练习

产生一个在80-100范围内的随机整数。

知识点

索引	要点	正链	反链
T421	掌握do-while的使用方法，至少执行一次，结束处的分号不能缺失		

03 for循环

for循环的基本语法格式

前文提到了循环三要素，当**已知循环范围**时，采用for循环书写更加简洁清晰。

```
1   for(初始化; 循环控制条件; 循环控制变量的改变)
2   {
3       语句块;
4   }
```

例题4.3

求n的阶乘。

样例输入	样例输入
5	120

与代码4.1进行对比，实现方法非常类似，但书写上明显简洁了很多。
代码4.1如下：

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       int sum=0,i=1,count;
7       cin>>count;                 //循环控制变量初始化
8       while(i<=count)             //循环条件
9       {
10          sum+=i;
11          ++i;                    //循环控制变量的改变
12      }
13      cout<<sum<<endl;
14  }
```

与代码4.1进行对照，第9行初始化语句只执行了一遍，然后执行条件判断，然后执行循环体，最后执行++i;操作。再次执行条件判断、循环体、++i;操作。

如果条件不成立，则循环退出。

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       int n;
7       cin>>n;
8       int factorial=1;
9       for(int i=1;i<=n;++i)
10          factorial*=i;
11      cout<<factorial<<endl;
12  }
```

例题4.3

求n的阶乘。

样例输入	样例输入
5	120

与代码4.1进行对比，实现方法非常类似，但书写上明显简洁了很多。
代码4.1如下：

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       int sum=0,i=1,count;
7       cin>>count;                 //循环控制变量初始化
8       while(i<=count)             //循环条件
9       {
10          sum+=i;
11          ++i;                    //循环控制变量的改变
12      }
13      cout<<sum<<endl;
14  }
```

	执行次数
初试化语句	1 遍
循环体	n 遍
++i;	n 遍
条件判断	n+1 遍　(n次成立+1次不成立)

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       int n;
7       cin>>n;
8       int factorial=1;
9       for(int i=1;i<=n;++i)
10          factorial*=i;
11      cout<<factorial<<endl;
12  }
```

循环结束时，i为n+1，表示第一次条件不成立时i的值

例题4.4

输入一串由小写字母组成的句子，将其中的所有小写字母转换为大写字母输出。

样例输入	样例输入
this is a lower string	THIS IS A LOWER STRING

C++中专门提供了一种**基于范围的循环**，比传统的for循环语法简单很多。

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       string s;
7       getline(cin,s);
8       for(auto ch:s)
9           cout.put(toupper(ch));
10  }
```

1. 第8行的for循环，:后表示一个容器变量，从该容器变量里，逐个取出元素，从头到尾进行循环。

例题4.4

输入一串由小写字母组成的句子，将其中的所有小写字母转换为大写字母输出。

样例输入	样例输入
this is a lower string	THIS IS A LOWER STRING

ps： auto不是一种数据类型，它通过判断自动解析变量的类型。

例如auto a=5;，a就被自动解析为整型。

C++中专门提供了一种**基于范围的循环**，比传统的for循环语法简单很多。

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       string s;
7       getline(cin,s);
8       for(auto ch:s)
9           cout.put(toupper(ch));
10  }
```

2. auto表示自动根据容器中每个元素的类型自动解析元素的数据类型。

例题4.4

输入一串由小写字母组成的句子，将其中的所有小写字母转换为大写字母输出。

样例输入	样例输入
this is a lower string	THIS IS A LOWER STRING

C++中专门提供了一种**基于范围的循环**，比传统的for循环语法简单很多。

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       string s;
7       getline(cin,s);
8       for(auto ch:s)
9           cout.put(toupper(ch));
10  }
```

3. 第9行的toupper是C/C++中自带的小写转大写函数，同理大写转小写函数为tolower。

随堂练习

输出1-10每个数的阶乘。

知识点

索引	要点	正链	反链
T431	for循环的使用方法，用两个分号确定循环三要素，掌握每个要素的执行时间和执行次数		
T432	当循环次数确定时，建议使用for；当循环次数不确定时，建议使用while	T411	
T433	掌握for(auto 变量 : 容器)的循环形式		

04 嵌套循环

4.1 嵌套循环基本方法

嵌套循环体现了一个**笛卡尔积**的概念，是内外循环的循环次数的乘积。这一节用*构造图形展现嵌套循环。

这些图形在实际使用中用处不大，但是对初学者理解嵌套循环有很大意义。

例题4.5

用 * 组成三行四列的矩形并输出。

样例输入	样例输入
无	**** **** ****

1. 循环准则

嵌套循环最重要的准则是**内循环先循环**，当i=0时，内循环执行一遍，然后i=1时再执行一遍，以此类推，共执行三遍。

2. 控制变量

内循环和外循环的循环**控制变量不能相同**。

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       for(int i=0;i<3;i++){
7           for(int j=0;j<4;j++)
8               cout.put('*');
9           cout<<endl;
10      }
11  }
```

每行输出结束的时候，第9行输出一个回车，进入下一行。

例题4.6

用 * 组成底为4，高为4的左对齐的直角三角形并输出。

样例输入	样例输入
无	* ** *** ****

当执行内循环时，外循环的循环控制变量是不变的，因此可以形成变长控制。

```
#include<iostream>
using namespace std;

int main (  )
{
    for(int i=0;i<4;i++){
        for(int j=0;j<=i;j++)
            cout.put('*');
        cout<<endl;
    }
}
```

例题4.7

用 * 组成底为4，高为4的右对齐的直角三角形并输出。

样例输入	样例输入
无	* ** *** ****

标准输出是按文本行制定的，不能控制输出的位置。因此在每行输出*前，控制输出空格的数量，以此达到右对齐的目的。

```cpp
#include<iostream>
using namespace std;

int main (  )
{
    for(int i=0;i<4;i++){
        for(int j=0;j<4-1-i;j++)
            cout.put(' ');
        for(int j=0;j<=i;j++)
            cout.put('*');
        cout<<endl;
    }
}
```

例题4.7

用 * 组成底为4，高为4的右对齐的直角三角形并输出。

样例输入	样例输入
无	* ** *** ****

其实可以看到，每一行输出的字符总数量是相等的，因此可以改成简单的矩形输出，控制输出字符，也可以达到相同的目的。

特殊字符图像输出模板：

```cpp
#include<iostream>
using namespace std;

int main ( )
{
    for(int i=0;i<4;i++){                    //外循环控制行
        for(int j=0;j<4;j++)                 //内循环控制列
            cout.put(j<4-1-i?' ':'*');       //根据特定条件输出不同字符
        cout<<endl;
    }
}
```

例题4.8

用 * 组成总高为5的“X形”并输出。

样例输入	样例输入
无	* * * * * * * * *

这种方法可以进一步推广，用来输出更加复杂的图形。例如”X”形等，可自行进行尝试。

```
#include<iostream>
using namespace std;

int main (  )
{
    for(int i=0;i<5;i++){
        for(int j=0;j<5;j++)
            cout.put(j==i||j+i==5-1?'*':' ');
        cout<<endl;
    }
}
```

随堂练习

实现九九乘法表。

实现方法与输出*组成的直角三角形完全相同。

注意，控制输出格式%-4d。其中-表示左对齐，4表示占4个字符的位置。

知识点

索引	要点	正链	反链
T441	掌握嵌套循环的基本使用方法，内循环先循环		
T442	掌握代码4.9所示的特殊字符图像输出模板		

4.2 内循环变量的初始化

在使用嵌套循环时，经常会出现某些变量只出现在内循环中。对于这些变量的初始化要特别小心。

下面的代码希望输出一个“由数字组成的直角三角行”，特别注意第8行。

输出数字组成的直角三角形：

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       int i=1,j=1,num;
7       cin>>num;
8       while(i<=num){
9           j=1;
10          while(j<=i){
11              cout<<j<<'\t';
12              ++j;
13          }
14          cout<<endl;
15          ++i;
16      }
17  }
```

样例输入	实际输入
5	1 2 3 4 5

只有将第9行的注释去掉，才能得到期望的结果。

样例输入	样例输入
5	1 1 2 1 2 3 1 2 3 4 1 2 3 4 5

内循环的循环控制变量每次都需要**重新进行初始化**

- 内循环可尽量采用for循环，其语法结构会提醒编程者进行初始化
- 内循环控制变量的初始化要写在外循环的里面，内循环的外面

知识点

索引	要点	正链	反链
T443	内循环控制变量的初始化要写在外循环的里面，内循环的外面	T217	

05 break和continue

5.1 死循环与break

在书写循环时，尽力**避免死循环**，因为死循环永无终止的。在C/C++中：

- 经典的死循环写法：while(1){}

  其中1表示true，因为永远为真，所以会一直循环下去。

- 退出当前循环：break

  注意break只退出一层循环。

死循环加break可以构建未知循环次数的基本结构。

例题4.9

输入一系列整数，-1表示结束。

```
1   while(1)
2   {
3       cin>>num;
4       if(num==-1)
5           break;
6       ...
7   }
```

在**确定循环次数**的情况下，建议使用for循环；

循环次数未知时，建议使用while循环。

知识点

索引	要点	正链	反链
T451	break可以退出当前循环，但是只退出一层循环		T472

5.2 循环与continue

1. 循环与continue语法规则

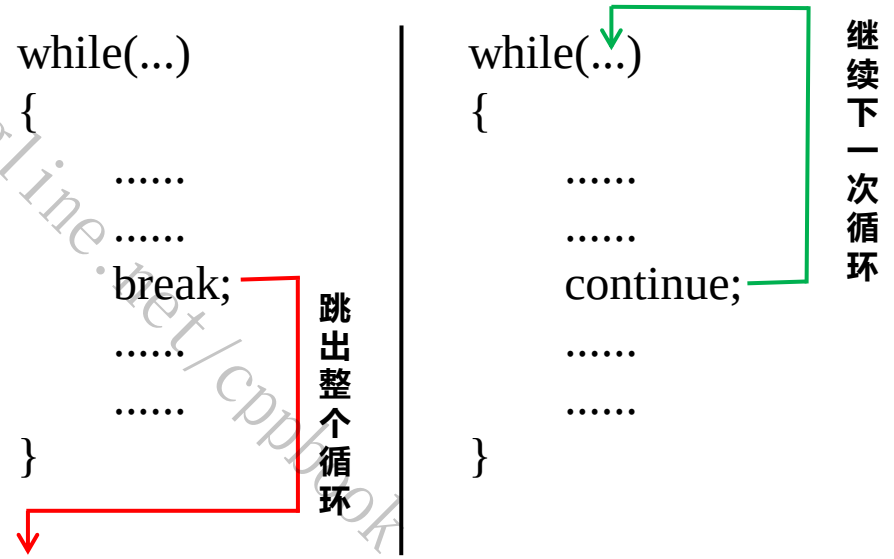
如果条件成立，语句块2不会被执行，跳转到第1行，继续执行下一次循环。

```
1   while(1)
2   {
3       语句块1;
4       if(条件)
5           continue;
6       语句块2;
7   }
8
```

2. break与continue对比

continue与break类似，都是跳出循环。

但continue只是退出本次循环，不执行循环体中的后继语句，直接转到下一次循环，**并非完全跳出**循环。



实践练习

组织n个同学一起玩数7游戏。n个同学围成圆圈，从1开始报数。7的倍数和末尾为7的同学请击掌，不要报数。有报数错误的同学，游戏终止。在这个游戏中，理想状态下是一个无限循环，每个同学都要执行报数操作。击掌同学执行的是continue操作，跳过了报数，但是进入下一次循环。二报数错误的同学执行了break操作，终止了循环。

例题4.10

化工12-1班有30名同学，学号能被3整除的为女生，请输出该班的女生。

continue使用示例：

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       for(int i=1; i<=30; i++)
7       {
8           if(i%3!=0)
9               continue;
10          cout<<i<<"是漂亮女生"<<endl;
11      }
12  }
```

男生的学号符合第8行的条件，因此第10行不会被执行。只有女生的学号会执行第10行。

break和continue都只能出现在循环体中，虽然没有明确的语法规定，但是它们都要跟if合用。

如果没有条件判断，就意味着循环体中该语句后继部分永远不会被执行。

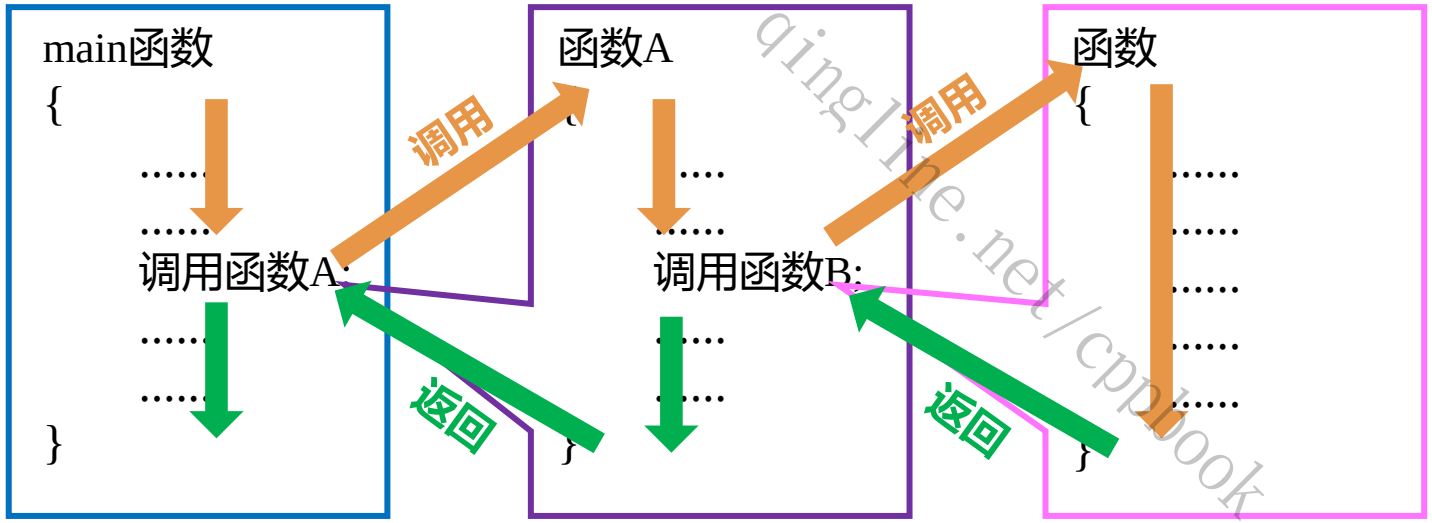
知识点

索引	要点	正链	反链
T452	continue只是退出本次循环，不执行循环体中的后继语句，直接转到下一次循环，并非完全跳出循环		

06 循环与递归

6.1 递归的演化

函数是可以嵌套调用的，如下图所示，main可以调用函数A，函数A继续调用函数B。



6.1 递归的演化

其代码形式如下：

在函数A中调用函数B

```
1   int A(int x)
2   {
3       int y,z;
4       代码块1;
5       z=B(y);
6       代码块2;
7       return 2*z;
8   }
```

函数B

```
1   int B(int t)
2   {
3       int a,c;
4       代码块1;
5       代码块3;
6       代码块2;
7       return 3+c;
8   }
```

如果函数A调用了函数B，同时函数B也调用了函数A，就形成了**间接递归调用**：

在函数A中调用函数B

```
1   int A(int x)
2   {
3       int y,z;
4       代码块1;
5       z=B(y);
6       代码块2;
7       return 2*z;
8   }
```

在函数B中调用函数A

```
1   int B(int t)
2   {
3       int a,c;
4       代码块1;
5       c=A(a);
6       代码块2;
7       return 3+c;
8   }
```

6.1 递归的演化

再考虑一种特殊形式，如果A和B两个函数除了函数名之外，其他部分完全相同：

在函数A中调用函数B

```
1   int A(int x)
2   {
3       int y,z;
4       代码块1;
5       z=B(y);
6       代码块2;
7       return 2*z;
8   }
```

在函数B中调用函数A

```
1   int B(int x)
2   {
3       int y,z;
4       代码块1;
5       z=A(y);
6       代码块2;
7       return 2*z;
8   }
```

这也是间接递归调用，但是很显然，没有必要把完全相同的函数体定义成两个函数

因此只保留函数A，依旧可以达到以上代码的效果。这就是**直接递归调用**：

```
1   int A(int x)
2   {
3       int y,z;
4       代码块1;
5       z=A(y);
6       代码块2;
7       return 2*z;
8   }
```

其中z=A(y);
调用了自身，
称为递归调用。

当进行递归调用时，将y作为参数从头执行函数A，反复迭代下去，效果相当于死循环。

知识点

索引	要点	正链	反链
T461	在一个函数内重新调用自身，则实现了递归调用		

6.2 简单递归

为了让代码执行有限次，递归调用之前需要加一个条件判断，在何种条件下停止递归递归调用。

因此一个递归函数最重要的两个部分是：

1. **建立递归终止条件；**
2. **形成递归调用。**

任何递归函数都必须出现这两个部分。

以下是递归函数的通用模板：

```
递归函数f(参数列表)
{
    if(参数符合特定条件)终止递归;
    进行递归调用，参数发生变化，向终止条件靠拢
}
```

例题4.11

求n的阶乘n!

$$n! = \begin{cases} 1, & \text{if } n = 1 \\ \text{n} * (n-1)!, & \text{if } n = 1 \end{cases}$$

```cpp
#include<iostream>
using namespace std;

long long factn(int n)                  //递归函数
{
    long long fac;
    if (n == 1)                         //循环终止条件
        return 1;
    fac = n * factn(n-1);               //递归调用
    return fac;
}

int main ( )
{
    int n;
    cin>>n;
    cout<<factn(n)<<endl;
}
```

以求4！为例，递归调用执行过程如下：

```
#include<stdio.h>
long factn(int);
int main()
{
        long fac;
        fac = factn(4);
        printf("4!=%1d\n", fac);
        return 0;
}
```
返回24

```
long factn(4)
{
        long fac;
        ......
        fac = 4*factn(3);
        return (fac);
}
```
返回6

```
long factn(3)
{
        long fac;
        ......
        fac = 3*factn(2);
        return (fac);
}
```
返回2

```
long factn(2)
{
        long fac;
        ......
        fac = 2*factn(1);
        return (fac);
}
```
返回1

```
long factn(1)
{
        long fac;
        if(n==1)
                return (1);
        ......
}
```

当main函数开始调用factn(4)时，开始递归过程，依次调用factn(3)，factn(2), factn(1)。

这个过程就是函数的嵌套调用，比较容易理解。

以求4！为例，递归调用执行过程如下：

```
#include<stdio.h>
long factn(int);
int main()
{
        long fac;
        fac = factn(4);
        printf("4!=%1d\n", fac);
        return 0;
}
```
返回24

```
long factn(4)
{
        long fac;
        ......
        fac = 4*factn(3);
        return (fac);
}
```
返回6

```
long factn(3)
{
        long fac;
        ......
        fac = 3*factn(2);
        return (fac);
}
```
返回2

```
long factn(2)
{
        long fac;
        ......
        fac = 2*factn(1);
        return (fac);
}
```
返回1

```
long factn(1)
{
        long fac;
        if(n==1)
                return (1);
        ......
}
```

最重要的是当执行factn(1)时，因为满足n==1，执行return(1)，递归调用被终止，但程序没有结束。

它将返回值1传递给调用它的factn(2)，以此类推，最终factn(4)将计算结果24返回给main函数，得到期望的结果。

以求4！为例，递归调用执行过程如下：

```
#include<stdio.h>
long factn(int);
int main()
{
        long fac;
        fac = factn(4);
        printf("4!=%1d\n", fac);
        return 0;
}
```
返回24

```
long factn(4)
{
        long fac;
        ......
        fac = 4*factn(3);
        return (fac);
}
```
返回6

```
long factn(3)
{
        long fac;
        ......
        fac = 3*factn(2);
        return (fac);
}
```
返回2

```
long factn(2)
{
        long fac;
        ......
        fac = 2*factn(1);
        return (fac);
}
```
返回1

```
long factn(1)
{
        long fac;
        if(n==1)
                return (1);
        ......
}
```

总而言之，调用的过程容易被理解，但是初学者往往容易忽视**返回的过程**。

循环与递归

从前面的知识可以获知，求阶乘操作是可以用循环进行计算的，递归函数也能完成相同的功能。理论上，循环和递归是可以**相互替换**的。有时用循环求解相关问题，逻辑上会比较复杂。

- 递归优点：思路简洁清晰（因为具体的实施过程不需要编程者思考）
- 递归缺点：效率有时比循环低（因为**函数调用需要代价消耗**）

因此，在一些在线评测题目中，如果对**时间效率**的要求比较高，可以采用循环代替递归。

随堂练习

1. 用递归方法求2的n次幂。

2. 输入一个字符串，用递归方法逆序输出每个字符。

3. 斐波那契数列（Fibonacci sequence），又称黄金分割数列，因数学家莱昂纳多·斐波那契（Leonardo Fibonacci）以兔子繁殖为例子而引入，故又称为“兔子数列”，指的是这样一个数列：0、1、1、2、3、5、8、13、21、34、……在数学上，斐波那契数列以如下被以递推的方法定义：F(0)=0，F(1)=1, F(n)=F(n - 1)+F(n - 2)（n ≥ 2，n 为正整数）。试用递归方法求解该数列的第n项。

知识点

索引	要点	正链	反链
T462	掌握递归函数的使用，理解终止条件，理解返回路径的执行。掌握递归书写模板	T332 T334	

6.3 分类递归

对于一些特定的场景，需要先进行分类，然后对每个类别进行递归后汇总。

以一个例子展开这个问题：

例题4.12

将m个相同的小球放到n个相同的袋子里，允许空袋，共有多少种放法？1≤m, n < 100。允许空袋。（CSP-J2019年真题）

【输入格式】

一行，两个数据m和n，分别表示小球的数量和袋子的数量

【输出格式】

一个整数，表示共有多少种放法

样例输入	样例输入
8 5	18

在这个问题中，因为允许空袋的存在，很难形成递推公式。

但是可以分别考虑1个袋子，2个袋子，多个袋子非空的情况，最后把这些情况汇总到一起，就得到了需要的答案。

因此得到主函数如下：

```cpp
#include <iostream>
using namespace std;

int place(int m,int n);

int main ()
{
    int m,n;
    cin>>m>>n;
    int sum=0;
    for(int i=1; i<=min(m,n); i++)  //遍历非空袋子的可能性
        sum+=place(m,i);            //对每一种非空袋子数量进行递归求解
    cout<<sum<<endl;
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

int place(int m,int n);

int main ()
{
    int m,n;
    cin>>m>>n;
    int sum=0;
    for(int i=1; i<=min(m,n); i++)  //遍历非空袋子的可能性
        sum+=place(m,i);            //对每一种非空袋子数量进行递归求解
    cout<<sum<<endl;
    return 0;
}
```

place函数计算将m个球放到n个非空袋子中的情况。

如果m<n，那么将m个球放到n个非空的袋子中是不可能的。因此第11行取m和n的最小值。

min是std的库函数，可以直接调用，同样std也提供了max函数。

- 因为袋子都是完全相同的，因此放法可能存在重复，不失一般性，只考虑非递减序列，即每个袋子中小球数量都大于等于前一个袋子，这样就保证了放法的唯一性。

- 考虑三种递归终止条件：m<n时，放法为0；n=1或m=n时，只有1种放法
- 对于递归，分为两种情况进行考虑：

1. 第一个袋子只放一个小球，这样就需要计算m-1个球放到n-1个袋子中的情况，即place(m-1,n-1)；

2. 第一个袋子不止放一个小球，因为是非递减序列，当第一个袋子放入一个小球后，其他袋子也至少放置一个小球，n个小球已经被放置，因此需要考虑m-n个小球放到n个非空袋子中的情况，即place(m-n,n)。

综上所述，就得到了递归函数place的写法：

```
1   int place(int m,int n)
2   {
3       if(n==1||m==n){ return 1; }
4       if(m<n){ return 0; }
5       return place(m-1,n-1)+place(m-n,n);
6   }
```

例题4.12

将m个相同的小球放到n个相同的袋子里，允许空袋，共有多少种放法？1≤m, n < 100。允许空袋。（CSP-J2019年真题）

【输入格式】

一行，两个数据m和n，分别表示小球的数量和袋子的数量

【输出格式】

一个整数，表示共有多少种放法

样例输入	样例输入
8 5	18

以上方法如果理解比较困难，也可以考虑**纵向递归**，即逐个考虑每个袋子的情况。

- 每个袋子最少放置小球的数量与前一个袋子相同，最多放置小球的数量不超过[m/n]，m/n向下取整。

```cpp
int place(int m,int n,int start)        //start表示该袋子中最少放置的小球数
{
    if(n==1){ return 1; }
    int sum = 0;
    for(int j=start; j<=m/n; j++)
        sum+=place(m-j,n-1,j);          //下一个带袋子中起始的小球数为当前袋子中的小球数
    return sum;
}

int main ()
{
    int m,n;
    cin>>m>>n;
    int sum=0;
    for(int i=1; i<=min(m,n); i++)      //遍历非空袋子的可能性
        sum+=place(m,i,1);              //第一个袋子从1个小球开始考虑
    cout<<sum<<endl;
    return 0;
}
```

例题4.12

将m个相同的小球放到n个相同的袋子里，允许空袋，共有多少种放法？1≤m, n < 100。允许空袋。（CSP-J2019年真题）

【输入格式】

一行，两个数据m和n，分别表示小球的数量和袋子的数量

【输出格式】

一个整数，表示共有多少种放法

样例输入	样例输入
8 5	18

- 一旦超过了[m/n]，就破坏了序列的非递减性。
- 当n=1时，因为前面计算保证了序列的非递减性，因此是一定能够放下的。

由此得到了一下递归算法：

```
int place(int m,int n,int start)        //start表示该袋子中最少放置的小球数
{
    if(n==1){ return 1; }
    int sum = 0;
    for(int j=start; j<=m/n; j++)
        sum+=place(m-j,n-1,j);          //下一个带袋子中起始的小球数为当前袋子中的小球数
    return sum;
}

int main ()
{
    int m,n;
    cin>>m>>n;
    int sum=0;
    for(int i=1; i<=min(m,n); i++)      //遍历非空袋子的可能性
        sum+=place(m,i,1);              //第一个袋子从1个小球开始考虑
    cout<<sum<<endl;
    return 0;
}
```

知识点

索引	要点	正链	反链
T463	分类递归就是先分类，在不同类别上执行递归，如何划分类别需要仔细思考		

07 经典循环问题

7.1 整数分解和倒序重组

整数分解和倒序重组：

```
1   int reverse(int n)
2   {
3       int ret=0;
4       while(n){                   //如果n大于0则循环
5           ret = ret*10+n%10;      //取出n的各位数，并添加到最终结果ret中
6           n/=10;                  //n整除10，通过循环不断减少，最终变为0退出循环
7       }
8       return ret;
9   }
```

随堂练习

若一个数（首位不为零）从左向右读与从右向左读都一样，将其称之为回文数。对于任意的正整数，判断其是否为回文数。

随堂练习

给你一个 32 位的有符号整数 x，返回将 x 中的数字部分反转后的结果。如果反转后整数超过 32 位的有符号整数的范围 [−231，231 − 1]，就返回 0。假设环境不允许存储 64 位整数（力扣7题）。

提示：为了防止数据溢出，要进行提前判断，符合要求的才能组合。

知识点

索引	要点	正链	反链
T471	通过循环达成整数分解和倒序重组，实质上就是除10取余法	T225 T265	T477

7.2 素数判断

解决思路：对于一个整数n，如果在2~n-1范围内都没有因子，则该数为素数。

1. 标记法素数判断：

```
1   int prime(int n)
2   {
3       bool flag=true;              //定义一个标记变量
4       for(int i=2;i<n;++i)
5           if(n%i==0){              //如果能整除，则不是素数
6               flag=false;          //修改标记变量
7               break;               //有一个整除，则表示n不是素数，退出循环
8           }
9       return flag;                 //必须循环结束之后，才能判定n没有因子
10  }
```

7.2 素数判断

解决思路：对于一个整数n，如果在2~n-1范围内都没有因子，则该数为素数。

2. 循环次数法素数判断：

```
1   int prime(int n)
2   {
3       int i=2;                //i一定要定义在循环前，因为在循环后要使用
4       for(;i<n;++i)           //for的循环变量初始化可以为空，但是;不可省略
5           if(n%i==0)          //如果能整除，则不是素数
6               break;          //有一个整除，则表示n不是素数，退出循环
7       return i==n;            //i与n相等，表示循环正常退出。如果执行了break，则i一定小于n
8   }
```

7.2 素数判断

解决思路：对于一个整数n，如果在2~n-1范围内都没有因子，则该数为素数。

3. return法素数判断：

```
int prime(int n)
{
    for(int i=2;i<n;++i)
        if(n%i==0)                  //如果能整除，则不是素数
            return false;           //有一个整除，则表示n不是素数，退出函数
    return true;                    //如果程序能执行到这里，表示n肯定没有因子
}
```

实际上，从数学角度，循环次数可以进一步优化。如果2~n/2范围内没有因子，则n/2~n-1范围内肯定没有因子。更进一步，如果2~n范围内没有因子，则n~n-1范围内肯定因子。

例题4.13

证明：假定a*b=n，如果b=$\sqrt{n}$

则a一定等于$\sqrt{n}$，如果b>$\sqrt{n}$

则a一定小于$\sqrt{n}$。

因为开方计算的复杂度较高，另一方面开方结果也会产生浮点数，造成不精确比较的问题。因此可以用平方进行代替。

素数判断的最优方式如下：

```
1   int prime(int n)
2   {
3       for(int i=2;i*i<=n;++i)      //优化循环次数
4           if(n%i==0)               //如果能整除，则不是素数
5               return false;        //有一个整除，则表示n不是素数，退出函数
6       return true;                 //如果程序能执行到这里，表示n肯定没有因子
7   }
```

知识点

索引	要点	正链	反链
T472	掌握素数判断的方法，重点掌握这种循环模板：一种结果在循环中，一种结果在循环后	T451	
T473	循环次数的优化是循环控制的重中之重		T475

7.3 穷举法

对于多个变量，多个等式进行求解时，通常使用解方程的方法。

但是对于计算机而言，解方程是无法接受的。但是计算机的特点就是运算速度快，因此可以用**穷举法遍历**所有的可能解，从而找到答案。

穷举法是计算机求解的基本方法之一，它的基本架构是：

```
循环所有的可能解
{
    if(候选解满足题目要求)
    {输出答案}
}
```

例题4.14

用50元钱买了三种水果。

各种水果加起来一共100个。西瓜5元一个，苹果1元一个，桔子1元3个，设计一程序输出每种水果各买了几个。

样例输入	样例输入
无	melon:0,apple:25,orange:75 melon:1,apple:18,orange:81 melon:2,apple:11,orange:87 melon:3,apple:4,orange:93

这是一个非常经典的穷举法题目。

双重循环，每重循环遍历所有的可能值。

- 西瓜最多有50/5=10个；计算苹果最大可能个数时，排除掉已经购买西瓜的金额；
- 而由于等式约束，桔子的数量直接计算，不需要循环，即完成了穷举

要根据实际情况和数学表达，最大可能的减少遍历的次数。

```
#include<iostream>
using namespace std;

int main (  )
{
    int  melon, apple, orange; //分别表示西瓜数、苹果数和桔子数
    for (melon=0; melon<=10; melon++){ // 对每种可能的西瓜数
        for( apple=0; apple <=50-5*melon; apple++){
            orange = 3*(50-5*melon-apple);  // 剩下的钱全买了桔子
            if (melon+apple+orange == 100)
                cout<<"melon:"<<melon<<",apple:"<<apple<<",orange:"<<orange<<endl;
        }
    }
}
```

例题4.14

用50元钱买了三种水果。

各种水果加起来一共100个。西瓜5元一个，苹果1元一个，桔子1元3个，设计一程序输出每种水果各买了几个。

样例输入	样例输入
无	melon:0,apple:25,orange:75 melon:1,apple:18,orange:81 melon:2,apple:11,orange:87 melon:3,apple:4,orange:93

第9-10行体现了计算思维。

计算机中无法精确表达浮点数，因此这不用除法，采用**扩大倍数**达到相同目的，避免除法可能产生的浮点结果。

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       int  melon, apple, orange; //分别表示西瓜数、苹果数和桔子数
7       for (melon=0; melon<=10; melon++){ // 对每种可能的西瓜数
8           for( apple=0; apple <=50-5*melon; apple++){
9               orange = 3*(50-5*melon-apple);  // 剩下的钱全买了桔子
10              if (melon+apple+orange == 100)
11                  cout<<"melon:"<<melon<<",apple:"<<apple<<",orange:"<<orange<<endl;
12          }
13      }
14  }
```

知识点

索引	要点	正链	反链
T474	穷举法遍历所有可能解，是计算思维的主要特征之一	T431 T441	

7.4 对称数判断--例题4.15

输入一个正整数n，求该数是否为对称数。如果n只有1位，则为对称数，否则要求前后对应位上数值相同。

样例输入	样例输入
22	true
123	false

因为涉及到每个数位上值的判断，因此需要进行**整数分解**。

但字符串中每个字符是自然分解的，因此可以用字符串的方式进行对称判断。

1. 第7行中采用了两个变量，分别指向字符串的头和尾，然后向中间靠拢。这是典型的**多变量循环**的写法。

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4   int main(){
5       string n;
6       cin>>n;
7       for(int i=0,j=n.size()-1;i<j;i++,j--){
8           if(n[i]!=n[j]){
9               cout<<boolalpha<<false<<endl;
10              return 0;
11          }
12      }
13      cout<<boolalpha<<true<<endl;
14      return 0;
15  }
```

7.4 对称数判断--例题4.15

输入一个正整数n，求该数是否为对称数。如果n只有1位，则为对称数，否则要求前后对应位上数值相同。

样例输入	样例输入
22	true
123	false

因为涉及到每个数位上值的判断，因此需要进行**整数分解**。

但字符串中每个字符是自然分解的，因此可以用字符串的方式进行对称判断。

2. 一个结果在循环中产生，另外一个结果在循环后，这与素数问题的判断逻辑相同。

```
#include <iostream>
#include <string>
using namespace std;
int main(){
    string n;
    cin>>n;
    for(int i=0,j=n.size()-1;i<j;i++,j--){
        if(n[i]!=n[j]){
            cout<<boolalpha<<false<<endl;
            return 0;
        }
    }
    cout<<boolalpha<<true<<endl;
    return 0;
}
```

知识点

索引	要点	正链	反链
T475	掌握对称判断的方法	T245 T472	T524
T476	掌握多变量循环的方法		T524
T477	掌握利用字符串达成整数自然分解的方法	T265 T471	

7.5 二进制中1的个数--例题4.16

输入一个整数，输出该数二进制表示中1的个数。

（Leetcode剑指Offer15）

样例输入	样例输入
22	3

因为涉及到二进制形式的运算，因此位运算是非常好的方法。以下两种方法分别通过**左移位**和**右移位**达到相同的目的。

```cpp
#include <iostream>
using namespace std;
int main(){
    int n;
    cin>>n;
    int num = 0;
    for(int flag=1;flag<=n;flag<<=1)
        num += !!(n&flag);
    cout<<num<<endl;
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;
int main(){
    int n;
    cin>>n;
    int num = 0;
    for(;n>0;n>>=1)
        num += n&1;
    cout<<num<<endl;
    return 0;
}
```

左侧代码第8行通过两次取反将一个非0数转换为1，形成一次简洁的判断。

7.5 二进制中1的个数--例题4.16

输入一个整数，输出该数二进制表示中1的个数。

（Leetcode剑指Offer15）

样例输入	样例输入
22	3

因为涉及到二进制形式的运算，因此位运算是非常好的方法。以下两种方法分别通过**左移位**和**右移位**达到相同的目的。

```cpp
#include <iostream>
using namespace std;
int main(){
    int n;
    cin>>n;
    int num = 0;
    for(int flag=1;flag<=n;flag<<=1)
        num += !!(n&flag);
    cout<<num<<endl;
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;
int main(){
    int n;
    cin>>n;
    int num = 0;
    for(;n>0;n>>=1)
        num += n&1;
    cout<<num<<endl;
    return 0;
}
```

以上方法无论对应位置上是1/0，都要进行一次判定，当位数较多时，会**增加循环次数**。可以用**数学方法简化**这个过程。

2m和2m-1进行位与操作，结果一定为0；

而且n和n-1只在n的二进制形式中最右侧一个1向右的部分不同；因此形成了以下方法，每次消除**n中最右侧**的一个1：

```cpp
#include <iostream>
using namespace std;
int main(){
    int n;
    cin>>n;
    int num = 0;
    for(;n>0;n&=n-1)
        ++num;
    cout<<num<<endl;
    return 0;
}
```

以 22=0b10110 为例：

- 22&21 = 0b10110&0b10101 = 0b10100 = 20

消除了0b10110中最右侧的1；

- 20&19 = 0b10100&0b10011 = 0b10000 = 16

再次消除最右侧的1；

- 16&15 = 0b10000&0b1111 = 0

从这个过程中可以看到，n的二进制形式中有几个1，循环就会执行几次。

知识点

索引	要点	正链	反链
T478	掌握循环与位运算的结合	T26A T473	

7.6 乘法的加法实现*

计算机中只有**加法器**，减法是通过补码实现的，而乘法和除法是通过**反复调用**加法器实现的，下面以乘法为例给出其实现方式。

7.6 乘法的加法实现*

为了简化运算过程，这里假定x和y大于0。

```
1   #include<iostream>
2   using namespace std;
3
4   int multi(int x, int y) {
5       int result = 0;
6       while (y) {
7           if (y & 1)
8               result += x;
9           x <<= 1;
10          y >>= 1;
11      }
12      return result;
13  }
14  int main (){
15      cout<<multi(3,11)<<endl;
16  }
```

3*11可以分解为3*1+3*2+3*8，即3<<0+3<<1+3<<3，

- 第10行计算被乘数3移动相应位数后的值，第10行循环将每一位都变成个位
- 第7-8行判断如果个位上为1，就累加到最后的结果中。

从而用加法实现了乘法操作。

例题4.17

用上述类似方法实现幂函数X^n。

幂函数X^n是一个用途非常广泛的函数，其实现方式与上面代码中用加法器实现乘法的方式类型。

```
1  double pow(double x, int n) {
2      double result = 1;
3      int minus = 1;
4
5      if (n < 0) {
6          minus = -1;
7          n = -n;
8      }
9
10     if (0 == n) {
11         return 1;
12     } else if (0 == x) {
13         return 0;
14     }
15
16     while (n) {
17         if (n & 1)
18             result *= x;
19         x *= x;
20         n >>= 1;
21     }
22
23     return minus<0?1.0/result:result;
24 }
```

- 第2-3行进行初始化工作
- 第5-8,23行对n为负数进行了特殊的处理
- 第10-14行对n和x的特殊值进行了处理

例题4.17

用上述类似方法实现幂函数X^n。

幂函数X^n是一个用途非常广泛的函数，其实现方式与上面代码中用加法器实现乘法的方式类型。

```
1  double pow(double x, int n) {
2      double result = 1;
3      int minus = 1;
4
5      if (n < 0) {
6          minus = -1;
7          n = -n;
8      }
9
10     if (0 == n) {
11         return 1;
12     } else if (0 == x) {
13         return 0;
14     }
15
16     while (n) {
17         if (n & 1)
18             result *= x;
19         x *= x;
20         n >>= 1;
21     }
22
23     return minus<0?1.0/result:result;
24 }
```

- 第16-21行以311为例：

  311 = 38*32*31

  第19行循环计算每位的权重，第17行如果n的个位为1，表示对应的权重有效，在第18行中将其累乘到结果中，每次计算后n向右移动一位。

知识点

索引	要点	正链	反链
T479	以加法实现乘法，以乘法实现幂运算，发挥位运算的计算效率优势	T26A	

08 循环与输入

8.1 输入重定向

cin从标准输入stdin中读取数据

cout将结果写入标准输出stdout进行显示

cin与stdin总是**保持同步**的，也就是说这两种方法可以混用，而不必担心文件指针混乱。（cout和stdout也一样）

正因为这个兼容性的特性，导致cin有许多额外的开销。

解决方法：在所有cin之前添加一条控制语句**cin.sync_with_stdio(false);**

（解除cin与stdin的同步，这样**读取速度**将会极大加快，对于大批量数据读入的问题，可以解决运行超时错误。）

8.1 输入重定向

测试样例的数据较多时，可以通过**freopen("文件名","r",stdin);**的方式，将输入进行重定位，不再从stdin中读取数据，而是要求程序从**指定文件**读取数据。

ps：但将代码提交到在线测试平台前，要将freopen语句**注释**掉，否则在线测试平台是找不到用户指定的问题就的。

忘记注释时，可进一步采用**ONLINE_JUDGE宏**的判断，来解决问题。

8.1 输入重定向

具体解决步骤如下：

1）在与源代码相同的目录下，新建一个文本文件，例如data.txt。

2）双击打开这个文件，粘贴上程序需要输入的数据，例如：

样例输入	样例输入
3 1 2 3	6

3）仿照以下代码书写程序

输入重定向代码示例：

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       cin.sync_with_stdio(false); //cin和stdin解除同步，在输入数据量比较小时不需要
7   #ifndef ONLINE_JUDGE
8       freopen("data.txt","r",stdin);
9   #endif // ONLINE_JUDGE
10      int sum=0;
11      int n;
12      cin>>n;
13      for(int i=0; i<n; ++i)
14      {
15          int num;
16          cin>>num;
17          sum += num;
18      }
19      cout<<sum<<endl;
20  }
```

- 第8行的**freopen**用data.txt文件代替stdin进行数据输入传递给cin。
- 其中**data.txt表示文件名**，这里使用了相对路径，即输入数据文件和源代码文件在相同的目录下。
- **r表示“读”，stdin表示标准输入。**

输入重定向代码示例：

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       cin.sync_with_stdio(false); //cin和stdin解除同步，在输入数据量比较小时不需要
7   #ifndef ONLINE_JUDGE
8       freopen("data.txt","r",stdin);
9   #endif // ONLINE_JUDGE
10      int sum=0;
11      int n;
12      cin>>n;
13      for(int i=0; i<n; ++i)
14      {
15          int num;
16          cin>>num;
17          sum += num;
18      }
19      cout<<sum<<endl;
20  }
```

- 第7行的**ifndef中n表示not，def表示define**，因此ifndef表示如果没有定义。
- 第9行的endif与ifndef相对应，构成一个**宏定义块**。
- 整体来说，如果程序预先没有定义宏ONLINE_JUDGE，则第8行被编译，否则第8行将会被忽略。

输入重定向代码示例：

```
#include<iostream>
using namespace std;

int main (  )
{
    cin.sync_with_stdio(false); //cin和stdin解除同步，在输入数据量比较小时不需要
#ifndef ONLINE_JUDGE
    freopen("data.txt","r",stdin);
#endif // ONLINE_JUDGE
    int sum=0;
    int n;
    cin>>n;
    for(int i=0; i<n; ++i)
    {
        int num;
        cin>>num;
        sum += num;
    }
    cout<<sum<<endl;
}
```

- 所有的宏都是以#开头，末尾不加;，因为**宏不是语句**。
- 由此推导，#include也是宏，表示将对应的库文件的内容在此位置展开。

输入重定向代码示例：

```
1   #include<iostream>
2   using namespace std;
3
4   int main (  )
5   {
6       cin.sync_with_stdio(false); //cin和stdin解除同步，在输入数据量比较小时不需要
7   #ifndef ONLINE_JUDGE
8       freopen("data.txt","r",stdin);
9   #endif // ONLINE_JUDGE
10      int sum=0;
11      int n;
12      cin>>n;
13      for(int i=0; i<n; ++i)
14      {
15          int num;
16          cin>>num;
17          sum += num;
18      }
19      cout<<sum<<endl;
20  }
```

正常来说，在用户电脑上一般都没有定义ONLINE_JUDGE宏，但是所有的在线评测系统上都定义了宏ONLINE_JUDGE。

因此在本机运行时，第8行将会被编译执行，但是提交到在线评测系统上，第8行语句将会被**自动忽略**。

这就自动解决了用户再向在线评测系统提交时，忘记注释freopen语句的尴尬。

知识点

索引	要点	正链	反链
T481	掌握输入重定向的方法，了解如何用输入重定向解决大数据量输入的问题，特别注意当输入数据量比较大时，需要用cin.sync_with_stdio(false); 解除同步，防超时		

8.2 数量不确定输入

对于**没有告知用户输入的确定数量**的题目：

解决方法：通过cin不断读取

- 当cin能正确读取数据时，返回true；
- 当读取到文件尾，会遇到一个特殊的**文件结束符EOF**，这时cin返回false，表示读取的结束。

ps：因为在线评测系统中，所有的输入数据都是以**文件形式**存在的。

例题4.18

有多行数据，每行有两个整数，输出每行两个整数的和。

样例输入	样例输入
3 5 7 9 12 8	8 16 20

不确定数据的输入：

```
1   #include<iostream>
2   using namespace std;
3   int main (  )
4   {
5   #ifndef ONLINE_JUDGE
6       freopen("data.txt","r",stdin);
7   #endif // ONLINE_JUDGE
8       int a,b;
9       while(cin>>a>>b){
10          cout<<a+b<<endl;
11      }
12  }
```

如果没有第5~7行（即没有使用文件重定向），而是用户通过键盘进行标准输入。那么在最后一行时：

- 在Windows系统中，需要输入Ctrl+z，然后回车
- 在 UNIX/Linux/Mac OS 系统中，Ctrl+d 代表输入结束

这样也会产生一个结束符号，否则以上程序会一直运行下去。

知识点

索引	要点	正链	反链
T482	掌握数量不确定输入问题的解决方法，了解文件结束符发挥的作用		T483

8.3 多级数量不确定输入

因为**空格**和**回车**都是空白符，都可以作为**cin的分割符**，所以每行的数据量不确定时，不能一体化输入，要区分为两个层次：**行和行内数据**。

- 可以用**getline**将输入分割为多行，但对字符串内的数据进行分割时，依旧非常复杂。
- 建议采用**istringstream**将字符串转换为**流**，采用流的方式将数据进行自然分割，极大地降低了处理的复杂性。

例题4.19

现在为若干组整数分别计算平均值。已知这些整数的绝对值都小于100，每组整数的数量不少于1个，不大于20个。

【输入格式】每一行输入一组数据（至少有一组数据），两个数据之间有1到3个空格。

【输出格式】对于每一组数据，输出数据均值。输出的均值只输出整数部

样例输入	样例输入
10 30 20 40 -10 17 10 10 9	25 5 9

多级数量不确定输入：

```
1   #include <iostream>
2   #include <string.h>
3   #include<sstream>
4   using namespace std;
5   int main() {
6       string line;
7       while(getline(cin,line)){
8           istringstream iss(line);
9           int i=0,num,sum=0;
10          for(;iss>>num;++i)
11              sum+=num;
12          cout<<sum/i<<endl;
13      }
14  }
```

知识点

索引	要点	正链	反链
T483	掌握多级数量不确定输入的方法	T482	
T484	掌握用流的方式分解字符串	T271	

09 程序优化案例

例题4.20

计算I,I+1,I+2,...r的异或和，

即I^(I+1)^(I+2)...^r。

【输入格式】输入包括两个整数I和r，空格分隔，$1 \le I < r \le 10^{18}$

【输出格式】输出题目描述中的异或和。

样例输入	样例输入
3 6	4

基础解法：

```cpp
#include<iostream>
using namespace std;

int main()
{
    long long n1,n2,sum=0;
    cin >> n1 >> n2;
    for(long long i=n1;i<=n2;i++)
        sum ^= i;
    cout << sum << endl;
    return 0;
}
```

如果给定范围比较小，以上代码能够完成题目需求。但是题目给定的数值范围为10^{18}，循环次数过多，在线评测系统中将会引发运行超时。

例题4.20

计算l,l+1,l+2,...r的异或和，

即l^(l+1)^(l+2)...^r。

【输入格式】输入包括两个整数l和r，空格分隔，$1 \le l < r \le 10^{18}$

【输出格式】输出题目描述中的异或和。

样例输入	样例输入
3 6	4

基础解法：

```cpp
#include<iostream>
using namespace std;

int main()
{
    long long n1,n2,sum=0;
    cin >> n1 >> n2;
    for(long long i=n1;i<=n2;i++)
        sum ^= i;
    cout << sum << endl;
    return 0;
}
```

- 异或操作的特点：对于任意偶数n，n^(n+1)=1，而1^1=0，进一步得到n^(n+1)^(n+2)^(n+3)=0。

也就是说，从任意一个偶数开始连续4个整数的异或结果为0，对于连续数值的异或和计算，绝大部分的计算都是毫无意义的耗时。

例题4.20

计算l,l+1,l+2,...r的异或和，

即l^(l+1)^(l+2)...^r。

【输入格式】输入包括两个整数l和r，空格分隔，$1 \le l < r \le 10^{18}$

【输出格式】输出题目描述中的异或和。

样例输入	样例输入
3 6	4

因此可以寻找4的倍数，只对两端多出来的部分进行异或操作即可。

利用4的倍数：

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       long long n1,n2,sum=0;
6       cin >> n1 >> n2;
7       long long a = 4 - n1%4,b = n2 %4;
8       for (long long int i = n1; i < n1 + a; i++)
9           sum = i ^ sum;       //对于头部多出来的部分进行异或
10      for (long long int i = n2 - b; i <= n2; i++)
11          sum = i ^ sum;       //对于尾部多出来的部分进行异或
12      cout << sum << endl;
13      return 0;
14  }
```

以上程序寻找头尾两端4的倍数，第7行中的a表示到头部第一个4的倍数的距离，b表示尾部到最后一个4的倍数的距离。

利用4的倍数：

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       long long n1,n2,sum=0;
6       cin >> n1 >> n2;
7       long long a = 4 - n1%4,b = n2 %4;
8       for (long long int i = n1; i < n1 + a; i++)
9           sum = i ^ sum;          //对于头部多出来的部分进行异或
10      for (long long int i = n2 - b; i <= n2; i++)
11          sum = i ^ sum;          //对于尾部多出来的部分进行异或
12      cout << sum << endl;
13      return 0;
14  }
```

- 第8行循环时不包括第一个4的倍数，最多循环3次；
- 第10行循环时包括尾部最后一个4的倍数，最多循环4次。
- 但是如果是循环次数为4时，实际尾部的计算结果为0，第2个循环没有意义。

- 如果n1=n2，以上程序依旧会循环5次，其中的4次为4的倍数开始的连续4个数，没有意义，最终结果会与n1和n2相等。这是因为4的倍数具有周期性而形成的结果。

while优化：

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       long long n1,n2,sum=0;
6       cin >> n1 >> n2;
7       while (n1<=n2 && n1&3)          //n1&0b11或n1%4
8           sum ^= n1++;
9       while(n1<=n2 && (n2+1)&3)       //(n2+1)&0b11或(n2+1)%4
10          sum ^=n2--;
11      cout << sum << endl;
12      return 0;
13  }
```

- 第7, 9行中的&3操作与对4取余等价，因为3的二进制为0b11，进行“位与”操作，相当于取二进制中的最后两位，也就是对4取余的结果。

while优化：

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       long long n1,n2,sum=0;
6       cin >> n1 >> n2;
7       while (n1<=n2 && n1&3)          //n1&0b11或n1%4
8           sum ^= n1++;
9       while(n1<=n2 && (n2+1)&3)       //(n2+1)&0b11或(n2+1)%4
10          sum ^=n2--;
11      cout << sum << endl;
12      return 0;
13  }
```

- 第7-8行的循环也是对第一个4的倍数前的部分进行异或操作，但是因为有n1<=n2判断，即使n1与n2相等，也不会产生额外操作。

- 第9-10行的循环也是从最后一个4的倍数开始向后进行运算，巧妙的使用了n2+1操作，这样即包含了最后一个4的倍数，但循环次数最多为3次。

计数while：

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       long long n1,n2,sum=0;
6       cin >> n1 >> n2;
7       if (n1&1)                   //n1&0b1或n1%2
8           sum ^= n1++;
9       int k=(n2-n1+1)&3;          //(n2-n1+1)&0b11或(n2-n1+1)%4
10      while(k--)
11          sum ^= n2--;
12      cout << sum << endl;
13      return 0;
14  }
```

- 事实上，只要从偶数开始即可，不需要一定从4的倍数开始，因此第7行判定如果为奇数，则将n1加到异或和中，并将n1加1，这样保证n1为偶数。第7行用n1&1计算n1的奇偶性，位操作的速度更快。

- 第二个循环采用了计数方式。第7行计算了n2与n1之间刨除以偶数开始的连续4个整数之后剩余数值的数量，第8-9行对这些数值进行了异或计算。

偶奇对计数：

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       long long n1,n2,sum=0;
6       cin >> n1 >> n2;
7       if(n1&1)
8           sum^=n1++;
9       if(n2&1){
10          sum ^=((n2+1-n1)>>1)&1;          //或((n2+1-n1)/2)%2
11      }else{
12          sum ^=((n2-n1)>>1)&1;            //或((n2-n1)/2)%2
13          sum ^=n2;
14      }
15      cout << sum << endl;
16      return 0;
17  }
```

- 第9行如果n2为奇数，则表示尾部没有冗余。
- 首先计算偶奇对的数量(n2+1-n1)，然后对偶奇对的数量除2并判定奇偶性。
- 如果偶奇对的数量为奇数，则让结果与1异或，否则与0异或。这里用右移1位代替除2操作，二者是等价的。

从偶数开始的n和n+1构成一对偶奇对，异或结果为1，因此中间部分的偶奇对数量为奇数则和为1，为偶数则和为0。

偶奇对计数：

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       long long n1,n2,sum=0;
6       cin >> n1 >> n2;
7       if(n1&1)
8           sum^=n1++;
9       if(n2&1){
10          sum ^=((n2+1-n1)>>1)&1;          //或((n2+1-n1)/2)%2
11      }else{
12          sum ^=((n2-n1)>>1)&1;            //或((n2-n1)/2)%2
13          sum ^=n2;
14      }
15      cout << sum << endl;
16      return 0;
17  }
```

- 第11行如果n2为偶数，尾部有一个冗余。
- 同样计算偶奇对的数量，并对最后一个元素进行异或操作。
- 此外，第12行时n2和n1都是偶数，因此(n2-n1)>>1和(n2+1-n1)>>1的结果是相同的，右移1位后，多加的1会被自动忽略掉。也就是说，第10行和第12行的代码可以完全相同，提取到选择结构外部。变成如图所示。

从偶数开始的n和n+1构成一对偶奇对，异或结果为1，因此中间部分的偶奇对数量为奇数则和为1，为偶数则和为0。

简化选择结构：

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       long long n1,n2,sum=0;
6       cin >> n1 >> n2;
7       if(n1&1)
8           sum^=n1++;
9       sum ^=((n2+1-n1)>>1)&1;
10      if(!(n2&1))
11          sum ^=n2;
12      cout << sum << endl;
13      return 0;
14  }
```

- 第11行如果n2为偶数，尾部有一个冗余。
- 同样计算偶奇对的数量，并对最后一个元素进行异或操作。

- 此外，第12行时n2和n1都是偶数，因此(n2-n1)>>1和(n2+1-n1)>>1的结果是相同的，右移1位后，多加的1会被自动忽略掉。也就是说，第10行和第12行的代码可以完全相同，提取到选择结构外部。变成如图所示。

去除简单条件语句：

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       long long n1,n2,sum=0;
6       cin >> n1 >> n2;
7       bool odd1 = n1&1;
8       sum^=n1*odd1;
9       n1+=odd1;
10      sum ^=((n2+1-n1)>>1)&1;
11      sum ^=n2*!(n2&1);
12      cout << sum << endl;
13      return 0;
14  }
```

- 在并行运算中，如果每个子进程中都没有选择结构，将会进一步提高程序性能。
  左侧代码展示了如何**去除简单的条件语句**。

- 第7行计算了条件的布尔值，利用这个结果，第8行和第11行用乘法，第9行用加分，控制了相关单元是否参与运算，取代了条件语句。

THANKS