计算机程序设计

刘龙军 副教授

2025年9月22日

三、循环控制结构

(课本4、5、6章)

循环语句

- 用顺序结构和选择结构是否可以解决所有的问题？

计算10!

=10×9×8×7×6×5×4×3×2×1

计算100!

=100×99×98×…×3×2×1

循环结构与枚举法

- **计算机的强项是不厌其烦地做同样的操作，这是通过循环语句实现的**
- **对某些问题，在寻找它的解时需要检查所有的可能的方案，从中找出可行解。这种解决问题的方法称为枚举法**

密码忘记了，如何开锁？

上节课回顾

- 顺序结构中的 块
- 条件分支中的 逻辑
- if else 匹配 及 作用域，
- Switch 中的“常量”
- 编程 “三步走”
- 循环 “三要素” 例子： for循环
- 算法复杂性（循环次数：时间复杂度），程序的空间复杂度

- 本周四下午上机：

抄练习---边查边练----：自我闭卷编程、自我用不同方法编程、所学方法综合验证

- PTA 本地编程、调试、测试；上传反复测试不同的case。

	组长						
1	许哲诚	张家畅	陈纪云	卢柏语	杨长乐		
2	王斐然	程家玮	杨卓	郭鑫睿	曹曲一		
3	朱书林	段续韬	符兴博	方善坤	周子轩		
4	范晨昊	蔡晨霖	李天昊	崔锦程	曾强彪		
5	吴政奇	李长乐	戴志骏	王泽群			
6	夏熙淳	薛梓帆	徐剑树	张志国	皮又支		
7	宋奕歆	钱安龙	孙修源	魏无忌	江啸	田园盎然	
8	韩奕铭	殷瞳	李张辰	薛华林	王子田		
9	王帅	商赢时	杨淇钧	罗舒仪	刘欣		
10	闫贺超	胡开创	李沅宸	朱梓铭	乔钵原		
11	孟祥杭	屈景明	张巍瀚	王远棹	张颢严	苏欣艺	
12	苏维钰	夏特	许嘉柯	李礼复	李书桁		
13	臧一赫	张森博	陈浩	徐成刚	袁华卿		
14	刘砚沣	马也	田康诚	周劭杰	张栩嫣		
15	仝竣宁	彭澍	田锦玮	江知行	孙啸宇		
16	高卓然	胡凤辉	王延鹏	姜庆东	杨奕宁	王佳琳	

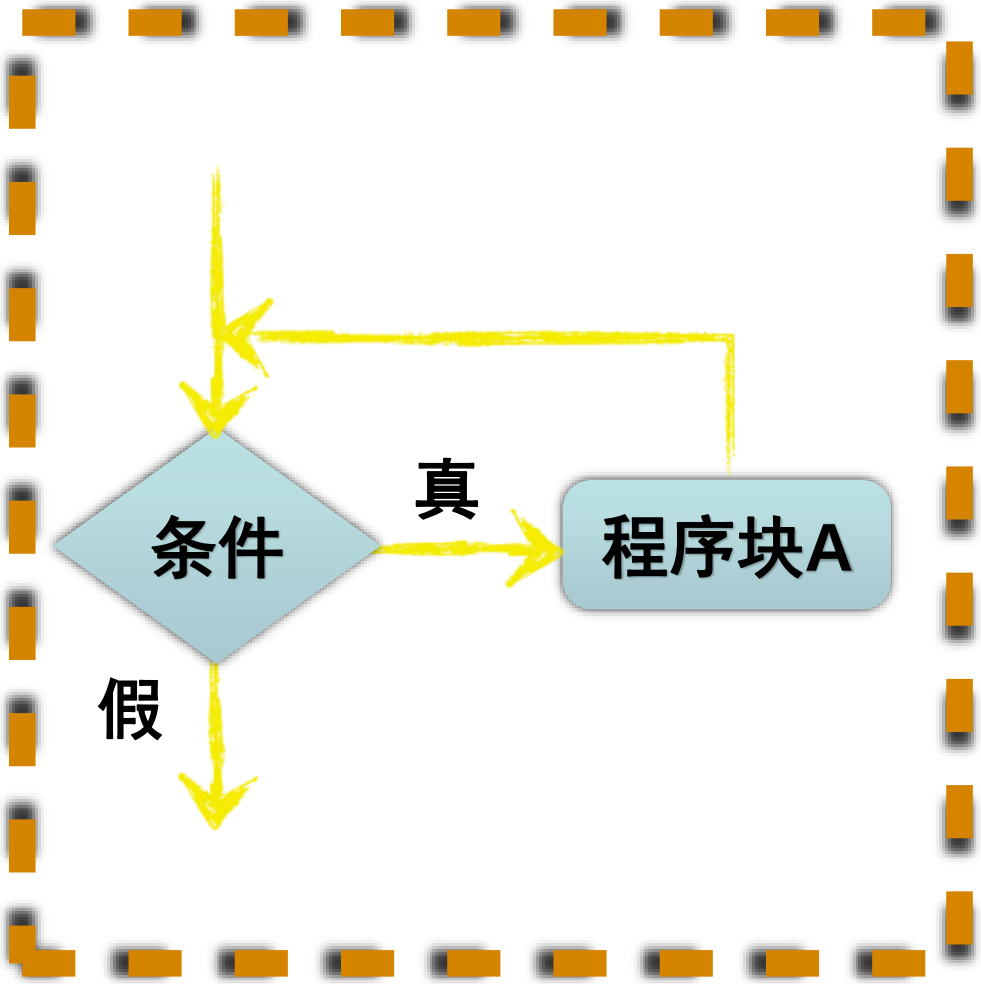
- 1-8组上机时间前两节可
- 9-16组后两节课
- 中间可调换或改动

开锁过程中的流程

- 顺序
  - 输入一个数，然后打开锁
- 选择
  - 判断锁是否打开，不开还得接着干啊……
- 循环
  - 先试000，再试001，……，直到打开为止

循环结构的含义

- 根据某一条件的判断结果，反复执行某一程序块的过程
- 进入循环结构，判断循环条件，如果循环条件的结果为真，则执行A程序块的操作，即循环一次，然后再次判断循环条件，当循环条件为假时，循环结束



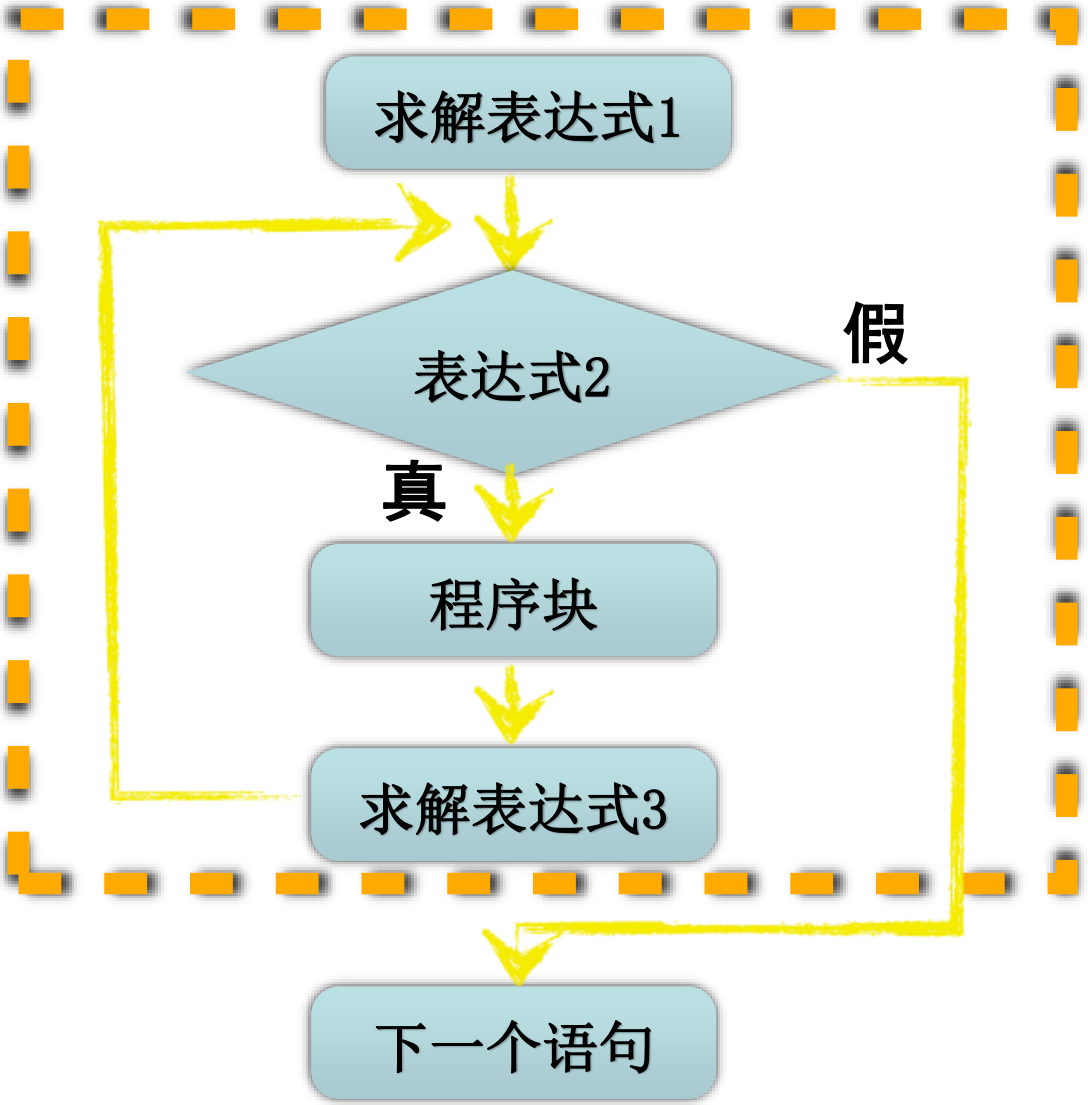
循环语句---for语句

- 表达式1：循环变量赋初值;
- 表达式2：循环条件;　　三要素
- 表达式3：循环变量增值;

```
for(表达式1; 表达式2; 表达式3)
    程序块
```

```
int sum=0;
for(i=1;i<=100;i++)
{
        sum += i;
}
```

for语句的流程图




```
int sum=0;
for(i=1;i<=100;i++)
{
        sum += i;
}
```

for语句的注意事项

```
int sum=0;
for(i=1;i<=100;i++)
{
        sum += i;
}
```

(1)for的所有表达式均可省略，但要防止进入死循环

```
for(;;)
        程序块
```

(2)省略表达式1

```
i=1;
for( ;i<=10;i++)
{
    sum += i;
}
```

(3)省略表达式2， 循环无法停止，必须在循环内设置停止条件

```
for(i=1; ;i++)
{
    sum += i;
        if(...) 程序块
}
```

(4)省略表达式3，应设法保证循环能正常结束

```
for(i=1;i<=100;)
{
        sum+=i;
        i++;
}
```

for语句的注意事项

- (5) 表达式1和3可以是逗号表达式, 包含一个以上的简单的表达式,中间用逗号隔开,建议少用这种方式

```
for(sum=0,i=1; i<=100; i++)
{
        sum+=i;
}
```

- (6) 表达式2一般是关系表达式(i<=100)或者逻辑表达式 (a<b && x<y)

循环的嵌套

- 一个循环内又包含另一个完整的循环结构称为**循环嵌套**

```
for(表达式1；表达式2；表达式3)
{
        语句块1;
        for(表达式4；表达式5；表达式6)
        {
                语句块2;
        }
        语句块3;
}
```

程序设计例子-----抓交通肇事犯

- 某日，某人骑车出交大，在南门口被一小车撞倒，小车撞人后逃逸。现场有A、B、C三同学目击此事，他们向警察描述了车号的一些特征

  A说：牌照的前两位数字是相同的；

  B说：牌照的后两位数字是相同的，但与前两位不同；

  C说：四位的车号刚好是一个整数的平方；

- 请根据以上线索编程找出车号。

问题分析

- 找出一个四位数，满足
  - (1)前两位数相同;
  - (2)后两位数同,且与前两位不同;
  - (3)该整数是另一个整数的平方;

第一种方法

- **枚举法：**

```c
#include <stdio.h>
#include <math.h>
/* 抓交通肇事犯演示 */
int main()
{
    int i,k;            // 定义循环变量
    int number=0;       // 结果
    int n1,n2,n3,n4;    // 千位,百位,十位,个位
    int flag=0;         // 条件成立的标志
    int count=0;
    for(i=1; i<=9999; i++){
        number =i;
        n1 =  (number/1000);
        n2 =  (number/100 )%10;
        n3 =  (number/10  )%10;
        n4 =  number%10;
        flag = (n1 == n2) && (n3 == n4) && (n1 != n3);

        if(flag){
            for(k=1; k*k<=number; k++) {  //判断该数是否是另一个整数的平方
                if(k*k==number){
            printf("The plate No. is %d.\n", number);  //若是,打印结果
        }
        count++;
    }
    }
    printf("Count: %d.\n", count);  ////打印循环次数
    return 0;
}
```

第二种方法

```
#include <stdio.h>
#include <math.h>
/* 抓交通肇事犯 */
int main()
{
    int i,j,k;
    int number=0;
    int count=0;                    //记录循环次数

    for(i=1; i<=9; i++)             //i:车号前二位取值
     {
         for(j=1; j<=9; j++)        //j:车号后二位取值
         {
             if(i!=j)               //j:判断两位数字是否不同
      {
          number = i*1000 + i*100 + j*10 +j;    //计算出可能的整数
          for(k=1; k*k<=number; k++)             //判断该数是否是另一个整数的
平方
          {
            count++;
            if(k*k==number)
              {
              printf("The plate No. is %d.\n", number);  //若是,打印结果
            }
        }
       }
        }
    }
     printf("Count: %d.\n", count);  ////打印循环次数
   return 0;
}
```

第三种方法

```c
#include <stdio.h>

int main()
{
    int num;
    int a, b, c, d;  //千位，百位，十位，个位
    int count=0;     //循环次数
    int i;
    for(i=32;i<100;i++) //四位数
    {
       num = i*i;
       a = num/1000;
       b = (num - a*1000)/100;
       c = (num- a*1000 - b*100)/10;
       d = num%10;
       if (a == b && c == d && a!=c)
          printf("The plate No.is %d.\n", num);//若是，打印结果
    count++;
    }
    printf("count = %d\n",count);
    return 0;
}
```

循环69次

第四种方法

```c
#include <stdio.h>
/*抓交通肇事犯的最快方法*/
int main()
{
    int i,k;
    int number;
    int n1,n2,n3,n4;
    int flag;
    int count=0;              //记录循环次数/

    for(i=1;i<=9;i++)
    {
        k=i*11;               //车牌号的平方根定然是十一的倍数
        number=k*k;           //车牌号是k的平方
        n1 =  (number/1000);      //千位
        n2 =  (number/100 )%10;   //百位
        n3 =  (number/10  )%10;   //十位
        n4 =  number%10;          //个位
        flag = (n1 == n2) && (n3 == n4) && (n1 != n3);
        if(flag)
        {
           printf("The plate No.is %d.\n", number);//若是，打印结果
        }
        count++;
    }
    printf("count:%d\n",count); //打印循环次数
    return 0;
}
```

循环9次

算法学习

- 一： 文字性理解
- 二： 具象化、可视化理解
- 三： 代码翻译
- 四：测试、巩固
- https://developer.aliyun.com/article/762909
- https://www.totuma.cn/

while语句

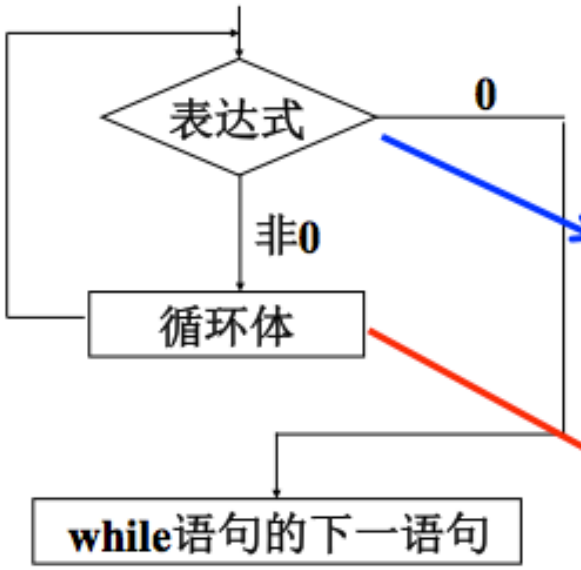
- 一版形式

```
while(表达式)
        程序块1
```

```
int i=0;
int sum=0;
while(i<=100)
{
        sum += i;
        i++;
}
```

- 只要表达式的值不为0，就重复执行程序块1，直到表达式的值为0时止;
- 选for还是while，取决于个人偏好。

while执行流程




```
int i=0;
int sum=0;
while(i<=100)
{
    sum += i;
    i++;
}
```

例子---累加器

■ 求 1+2+3+……+100

```
void main(void)
{   int  i=1,sum=0;   //定义变量，初始化
    while(i<=100)    //构造循环
    {    sum=sum+i;   // 循环体，多次执行
         i=i+1;
    }
   cout<<“sum=”<<sum<<endl;   //输出结果
}
```

实际上是将i不停地累加到一起

sum

i

循环结束!!

sum＝5050

- **注意红色字体，精确到每一位，为false时 其它值的变化 状态**

循环条件	初值	真	真	真	真	真	真	真	假
循环次数		1	2	3	4	.....	99	100	101
sum	0	1	3	6	10			5050	
i	1	2	3	4	5		100	101	25

例子程序

■ 从屏幕输入n个数，输出这n个数的和

```
// sum.cpp
// sum up N numbers
#include <iostream>
int main (int argc, const char * argv[])
{
    using namespace std;
    int n;
    int sum;
    int i,no;
    cout<<"Please input n:";
    cin>>n;
    cout<<"Please input the numbers:\n";
    i=1;                          //i为循环次数控制，初始值为1
    sum=0;                        //求和之后的值
    while (i<=n) {                //执行n次
        cin>>no;          //循环体
        sum+=no;                  //求和 sum=sum+no;
        i++;                      //增加已经循环的次数
    }
    cout<<"The sum is "<<sum<<endl;
    return 0;
}
```

```
Please input n:5
Please input the numbers:
33
11
2
9
11
The sum is 66
```

例子程序

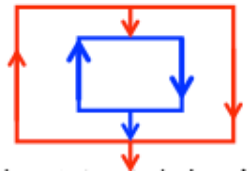
- 从键盘连续输入字符，直到输入“回车”符为止， 统计输入的字符个数

```
// counterchar.cpp
// counting the number of characters in a inputted sentence
#include <iostream>

int main (int argc, const char * argv[]) {
    using namespace std;
    char c;
    int cnt;

    cout<<"Please enter your sentence:\n";

    cin.get(c);          //取第一个字符
    cnt=1;                    //计数初始值

    while (c!='\n') {         //循环终止条件
        cnt++;                //计数增加
        cin.get(c);           //取下一个字符，把‘\n’统计在内
    }
    cout<<"There are "<<cnt<<" characters in this sentence.\n";

    return 0;
}
```

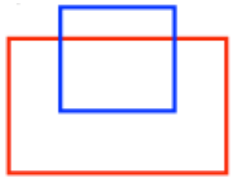
```
Please enter your sentence:
we are in class
There are 15 characters in this sentence.
```

循环嵌套

- 在循环体语句中又包含另一个循环语句时，称为**循环嵌套**
- 在多重循环中，处于内部的循环称为内循环，处于外部的循环称为外循环



- 内循环必须完全嵌套与内循环中，内外循环不能交叉，并且内外循环的循环控制变量不能重名



例子

打印99乘法表

```
// times99.cpp

#include <iostream>

int main (int argc, const char * argv[]) {
    using namespace std;
    int i, j;
    for(i=1;i<10;i++){
        for(j=1;j<10;j++){
            cout<<i<<"*"<<j<<"="<<i*j<<"\t";
        }
        cout<<endl;
    }
    return 0;
}
```

```
1*1=1   1*2=2   1*3=3   1*4=4   1*5=5   1*6=6   1*7=7   1*8=8   1*9=9
2*1=2   2*2=4   2*3=6   2*4=8   2*5=10  2*6=12  2*7=14  2*8=16  2*9=18
3*1=3   3*2=6   3*3=9   3*4=12  3*5=15  3*6=18  3*7=21  3*8=24  3*9=27
4*1=4   4*2=8   4*3=12  4*4=16  4*5=20  4*6=24  4*7=28  4*8=32  4*9=36
5*1=5   5*2=10  5*3=15  5*4=20  5*5=25  5*6=30  5*7=35  5*8=40  5*9=45
6*1=6   6*2=12  6*3=18  6*4=24  6*5=30  6*6=36  6*7=42  6*8=48  6*9=54
7*1=7   7*2=14  7*3=21  7*4=28  7*5=35  7*6=42  7*7=49  7*8=56  7*9=63
8*1=8   8*2=16  8*3=24  8*4=32  8*5=40  8*6=48  8*7=56  8*8=64  8*9=72
9*1=9   9*2=18  9*3=27  9*4=36  9*5=45  9*6=54  9*7=63  9*8=72  9*9=81
```

do while语句

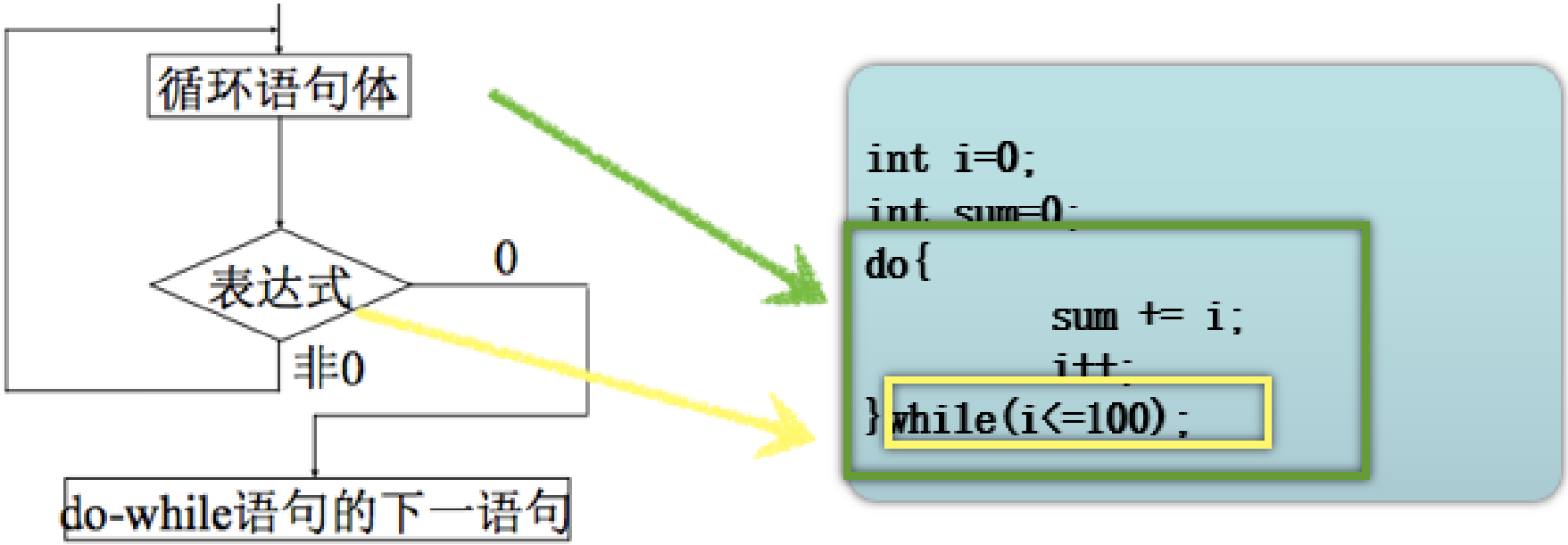
- 一般形式

```
Do
                程序块1
while(表达式);
```

```
int i=0;
int sum=0;
do{
          sum += i;
          i++;
}while(i<=100);
```

- 只要表达式的值不为0，就重复执行程序块1，直到表达式的值为0时止;
- 选for还是do while，取决于个人偏好。

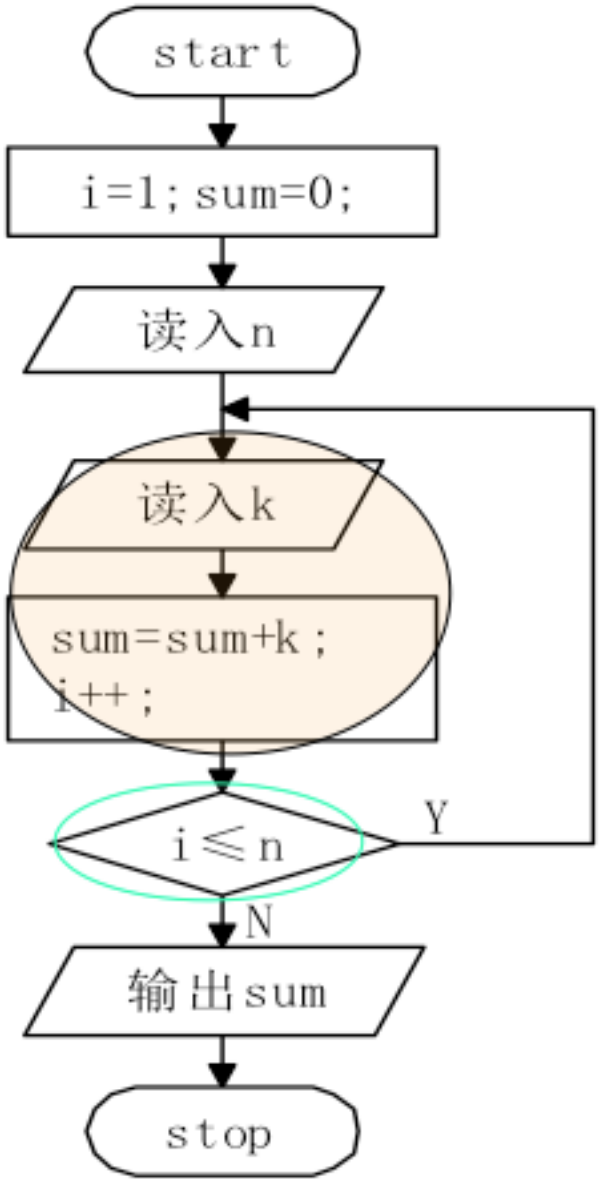
do while 执行流程

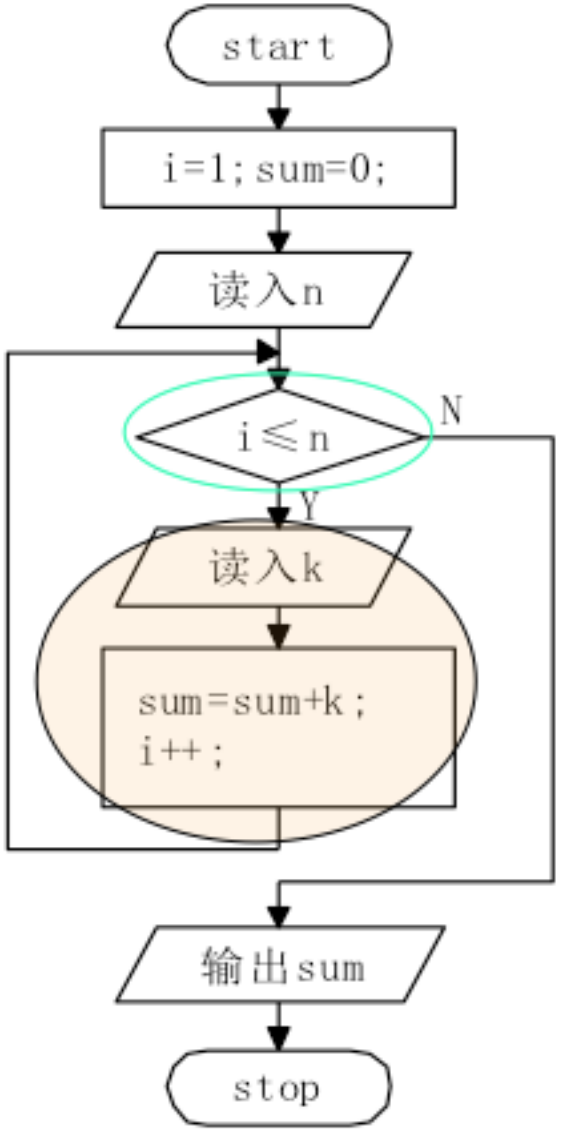


while和do-while区别

- do-while语句与while语句的区别:
  - do-while语句是在判断条件是否成立之前，先执行循环体语句一次；
  - while语句则是先判断条件是否成立，如果条件成立才执行循环体;
- 因此，while语句的循环体可能一次都不执行；而do-while语句的循环体至少被执行一次，这是while语句和do-while语句的**根本区别。**

while和do-while区别





例子程序

■ 从屏幕输入n个数，输出这n个数的和

```
// sum2.cpp
// sum up N numbers
#include <iostream>

int main (int argc, const char * argv[])
{
    using namespace std;
    int n;
    int sum;
    int i,no;
    cout<<"Please input n:";
    cin>>n;
    cout<<"Please input the numbers:\n";
    i=1;                          //i为循环次数控制，初始值为1
    sum=0;                        //求和之后的值
    do{              //执行n次
        cin>>no;          //循环体
        sum+=no;                  //求和 sum=sum+no;
        i++;                      //增加已经循环的次数
    }while(i<=n);

    cout<<"The sum is "<<sum<<endl;
    return 0;
}
```

```
Please input n:5
Please input the numbers:
33
11
2
9
11
The sum is 66
```

例子

- 从键盘连续输入字符，直到输入“回车”符为止， 统计输入的字符个数
- 使用do - while

- 注意：
- 1、循环体如果为一个以上的语句，用{ }括起。
- 2、循环体内或表达式中必须有使循环结束的条件，即一定有一个循环变量。

```
void main(void)
{   int  num=0;
    while(num<=2)
    {    num++;
         cout<<num<<endl;
     }
}
```

1
2
3

num	0	1	2	3
循环条件	真	真	真	假
输出	1<CR>	2<CR>	3<CR>	无

```
void main(void)
{
    int  y=10;
    while (y--);
   cout<<“y=”<<y<<endl;
}
```

输出是什么？
循环几次？

y	10	9	……	1	0
条件	真	真	真	真	假
输出	无	无	无	无	－1

输出：y=-1　　循环：10次

- 以下语句，循环退出时x为多少？

```
x=10;  while (x!=0)  x--;
x=10;  while (x)  x--;
x=10;  while(x--);
x=10;  while(--x);
```

x=0

x=0

x=-1

x=0

上节课回顾

- **算法入门：**

  如何设计算法：分析问题、举例答案、具象化、编程设计

  循环 “三要素” 例子： for、 While 、 do-While 语句互换

  算法复杂性（循环次数：时间复杂度），程序的空间复杂度

break和continue

- 对循环结构进行内部手术
- break语句：退出当前循环
- continue语句:中断此次循环的执行，开始下一次循环

continue/break示例（Debug）

```cpp
//break.cpp
// 演示break和continue的效果
#include <iostream>

int main ()
{
    using namespace std;
    int i, j;

    for (i = 0; i < 3; i++){
        for (j = 0; j < 3; j++){
            cout<<"i = "<<i<<"\tj= "<<j<<endl;
            continue;
            //break;
            cout<<"Next------------>\n";
        }
    }
    return 0;
}
```

break和continue注意事项

- 区别:
  - continue语句只结束本次循环，而不是中止整个循环的执行。
  - break语句则是结束循环，不再进行条件判断。
- 注意小心使用break和continue，他们增加了循环执行的分支，break更增加了循环的出口。

goto语句

- 使用goto可以让程序清晰，但是要谨慎使用
- 标号
  - error:
  - 同变量、函数的命名规则一样，后面加上一个冒号，一般顶格书写

```
for (...)
    for (...) {
        ...
        if (出现错误条件)
            goto error;
    }
error:
    出错处理语句
```

goto简化程序

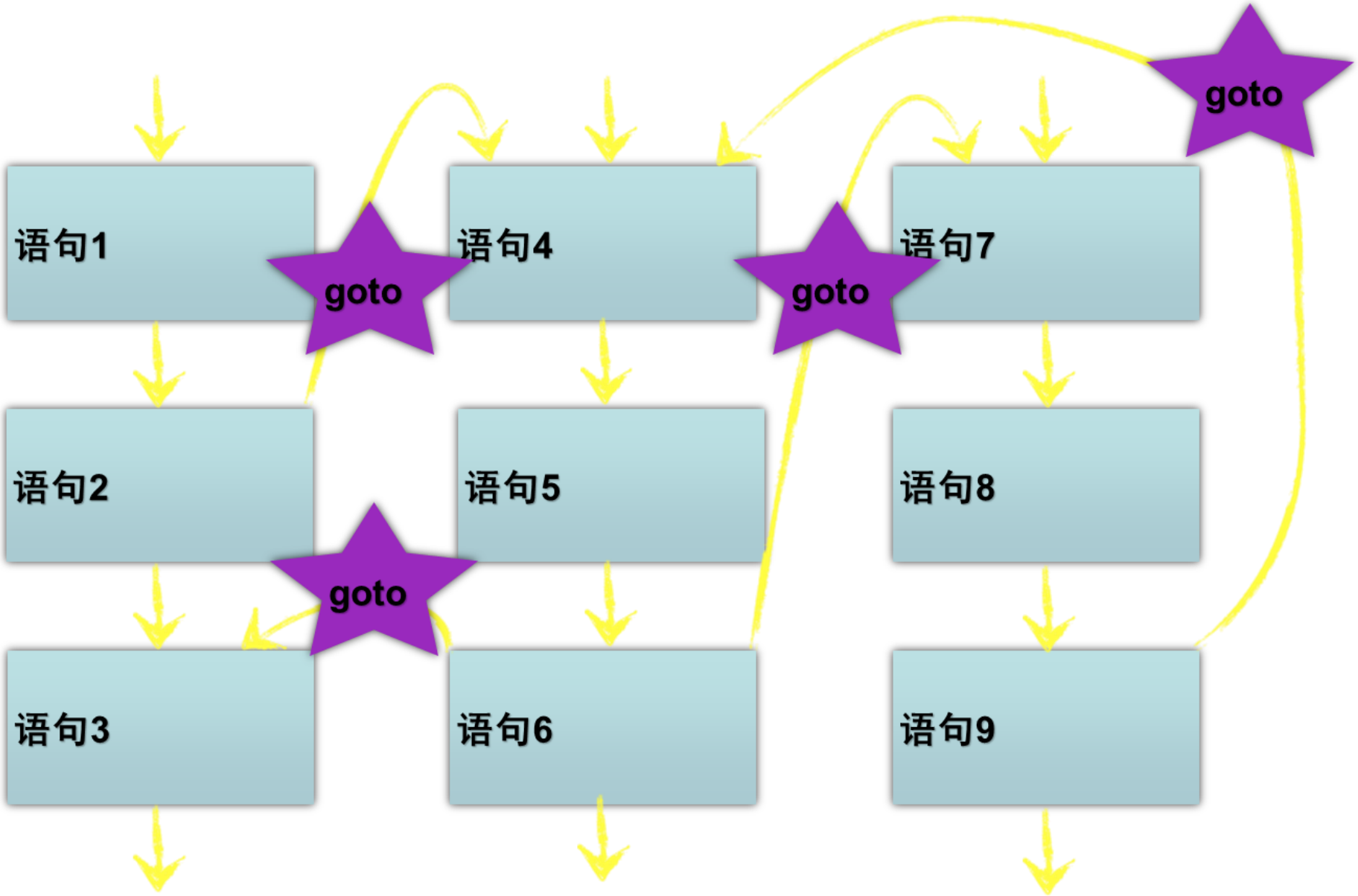
```
for (...)
    for (...) {
        ...
        if (出现错误条件)
            goto error;
    }
error:
    出错处理语句
```

```
int flag = 0;
for (...) {
    for (...) {
        ...
        if (出现错误条件) {
            flag = 1;
            break;
        }
    }
    if (flag)
        break;
}
if (flag)
    出错处理;
```

使用goto的原则

- 使用之后，程序仍然是单入口、单出口
- 不要使用一个以上的标号
- 不要用goto往回走，要往下走
- 不要让goto制造出永远不会被执行的代码

程序员的个人舞台



结构化/模块化 程序设计方法

- 1968年, E. W. Dijkstra发表文章“Goto Statement Considered Harmful”反对使用goto,提出了结构化程序设计思想。

Letters to the Editor

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (*if B* **then** *A*), alternative clauses (*if B* **then** *A1* **else** *A2*), choice clauses as introduced by C. A. R. Hoare (*case*[*i*] *of* (*A1*, *A2*, ···, *An*)), or conditional expressions as introduced by J. McCarthy (*B1* → *E1*, *B2* → *E2*, ···, *Bn* → *En*), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, *n* say, of people in an initially empty room, we can achieve this by increasing *n* by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of *n*, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, *n* equals the number of persons in the room minus one!

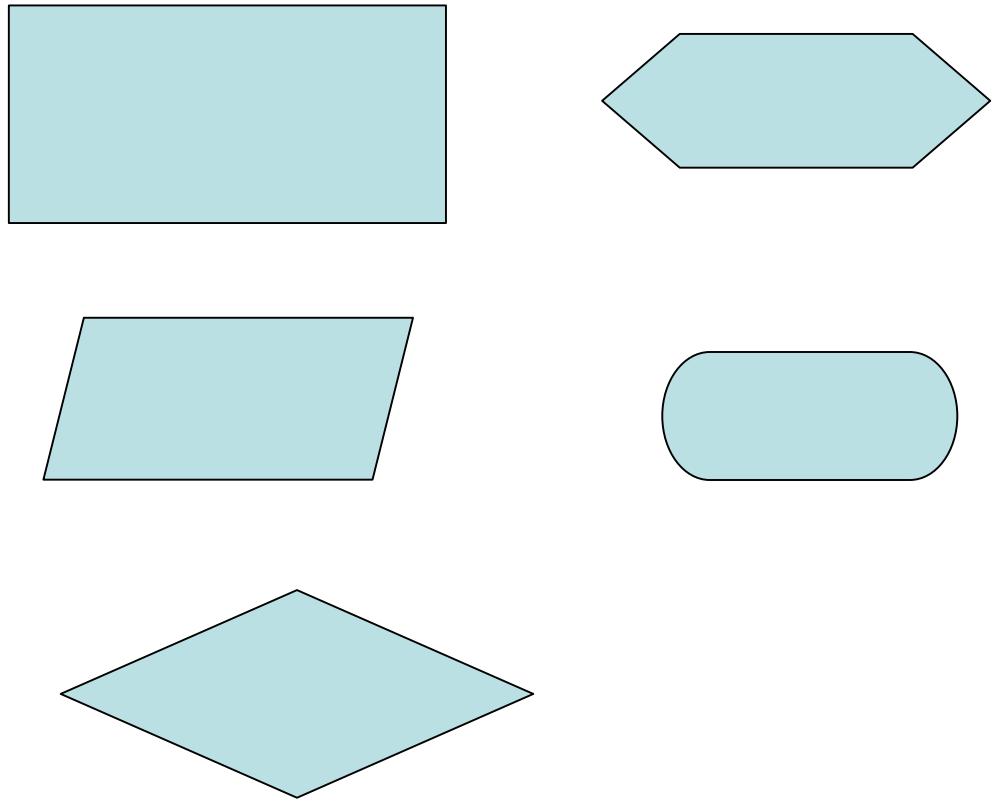
The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

程序流程图绘制（PPT里图形）

西安交通大学 XI'AN JIAOTONG UNIVERSITY

- **要会画流程图：**



请画出下列语句的流程图

```
while (condition) {
    statements
}
```

```
if (condition) {
    statements
}
else {
    statements
}
```

```
do {
    statements
} while (condition)
```

请画出下列语句的流程图

```
switch (expression) {
    case constant:
        statements
        break;
    case constant:
        statements
        break;
    default:
        break;
}
```

```
for (initial; condition; increment) {
    statements
}
```

练习

- 编程实现以下函数的求解
  - y=3*x+1, (x>0)
  - y=0, (x=0)
  - y=x^2-1, (x<0)
- 要求，从屏幕输入x，然后输出y

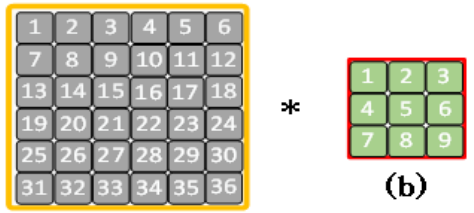
西 安 交 通 大 学 考 试 题

四、当前推动人工智能发展进入新的一轮高潮的一个重要因素之一是深度学习的突破性进展，而深度学习中卷积神经网络起到的作用尤为重要。请基于下列提示，自动动手写一个简单的卷积运算：

最基本的卷积神经网络是对二维矩阵进行乘加运算，如图 1（a）称为特征图，图 1（b）称为卷积窗，本例子中特征图是 6x6 的矩阵，卷积窗是 3x3 的矩阵。卷积神经网络中卷积窗一般是在特征图上以步长为 1 或 2 进行从左到右、从上到下滑动计算。为了简化，本例子中只要做四次卷积窗与特征图的运算，如图 2 所示，只将 3x3 的卷积窗与特征图四个角对应位置的矩阵元素一一相乘，再把 9 个点相加即可，结果如图 3 所示，再将图 3 中的四个点相加得到一个值，如图 4 所示。再经过 relu 函数得到最终结果，如图 5 所示，

Rulu 运算规则为： y = x （if x>0）； y=0 (if x<=0).

请编程实现并输出卷积运算最终结果

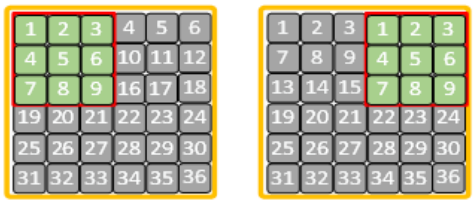


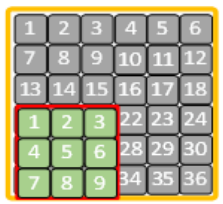
(a)

*

(b)

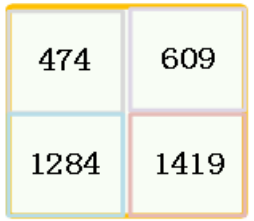
(1)





(2)

(2)



(3)

(4)

Relu

(5)

用例 1：

输入：

1 2 3 4 5 6 7 8 9

输入

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

输出

3786

用例 2：

输入

-1 2 3 4 -5 6 -7 -8 -9

输入

1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6

输出

0

- **打印10！的值**

- **打印1!+2!+…+10!的值**

本章结束！