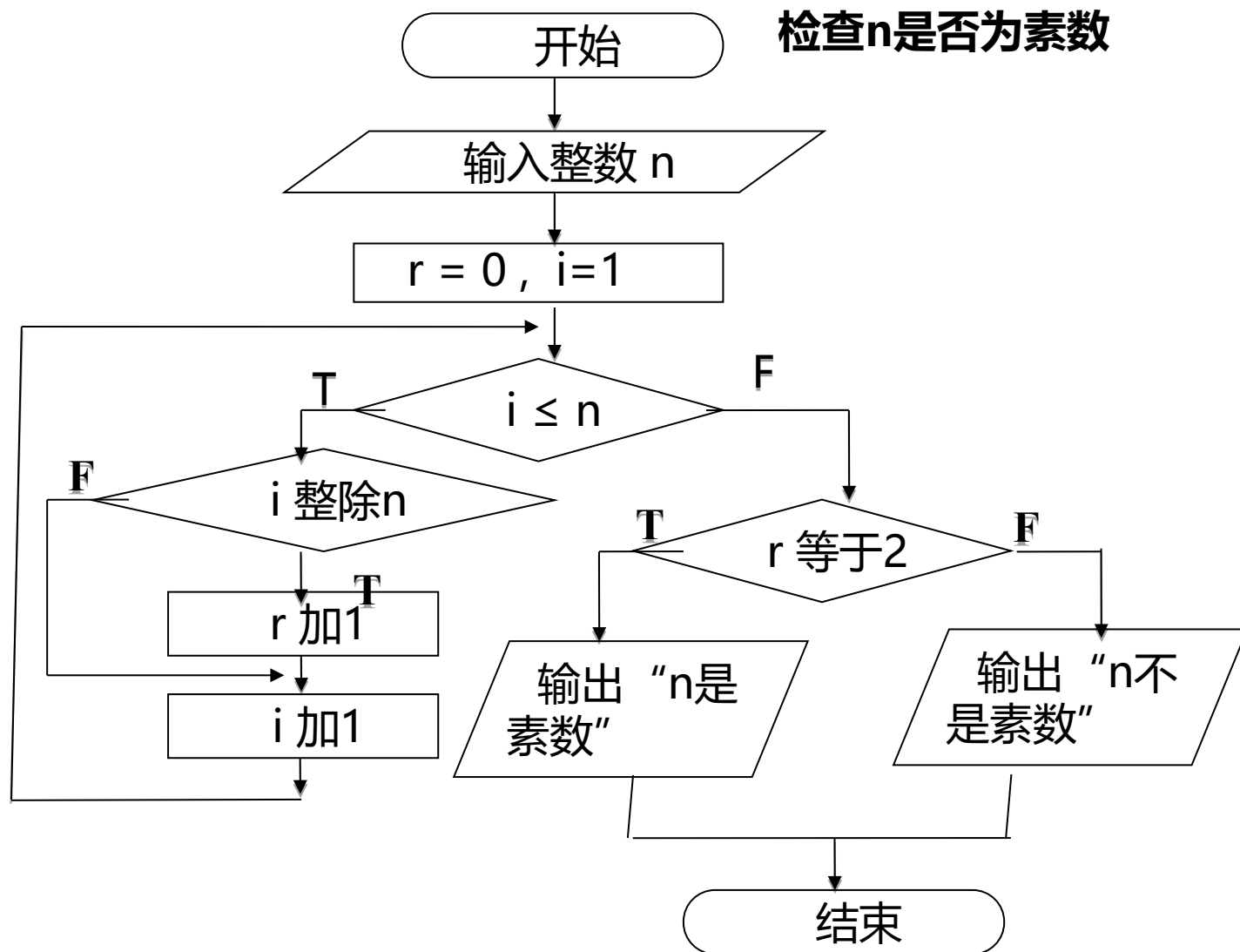




北京理工大学  
BEIJING INSTITUTE OF TECHNOLOGY

# C++程序设计基本方法

- C++程序设计思想与方法 (第3版)  
人民邮电出版社 翁惠玉 俞勇
- C++程序设计：题解与拓展 (第2版)  
清华大学出版社 翁惠玉 俞勇
- C++ Primer (第5版)  
人民邮电出版社
- C++ Primer Plus (第6版)  
人民邮电出版社
- C++大学教程 (第9版) 电子工业出版社

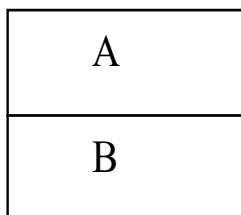


## 缺点

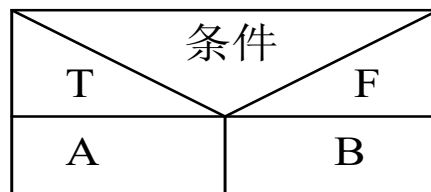
流程可随意转来转去，使人很难理解算法的逻辑，难以保证程序的正确性  
占用的篇幅也较大

## 解决方法

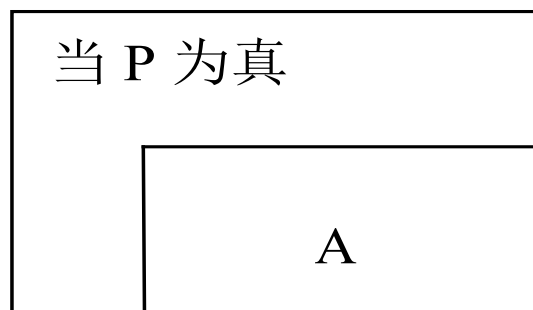
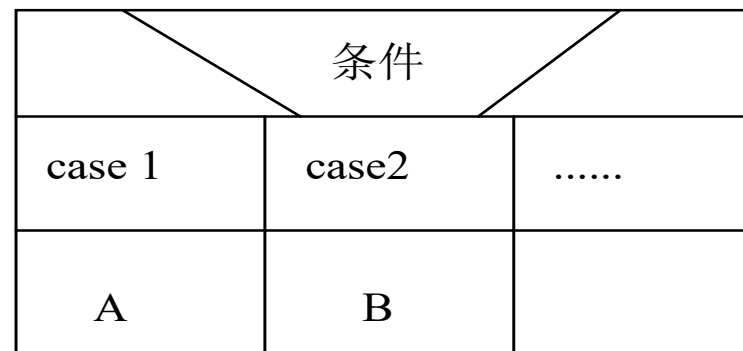
结构化程序设计



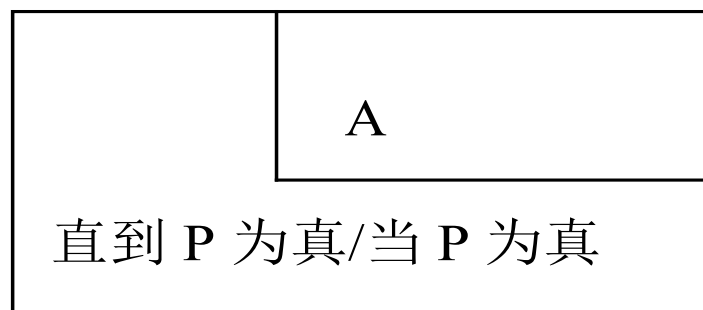
顺序结构

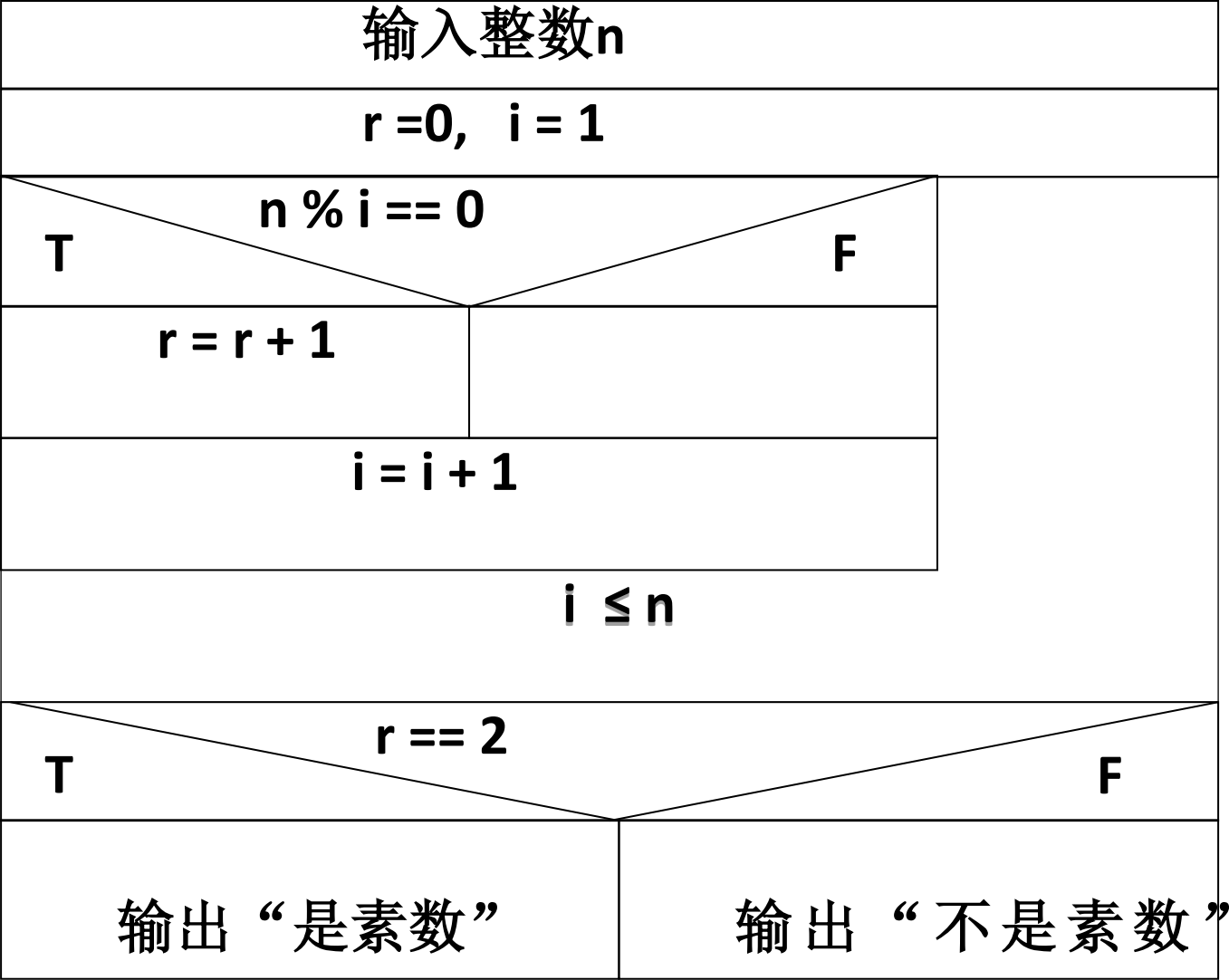


分支结构



循环结构





## 程序设计语言的控制结构 + 自然语言

输入n

设 $r = 0$

for ( $i = 1; i \leq n; ++i$ )

    if ( $n \% i == 0$ )  $++r$

if ( $r == 2$ )

    输出 “n是素数”

else 输出 “n不是素数”

用程序设计语言表示算法

C++是从C发展而来，而C又是从B语言发展而来

C语言是由贝尔实验室在B语言的基础上开发的，并有美国国家标准组织和国际标准化组织进行了标准化

C++是C的扩展，主要是提供了面向对象的功能

IDE 与编辑器？

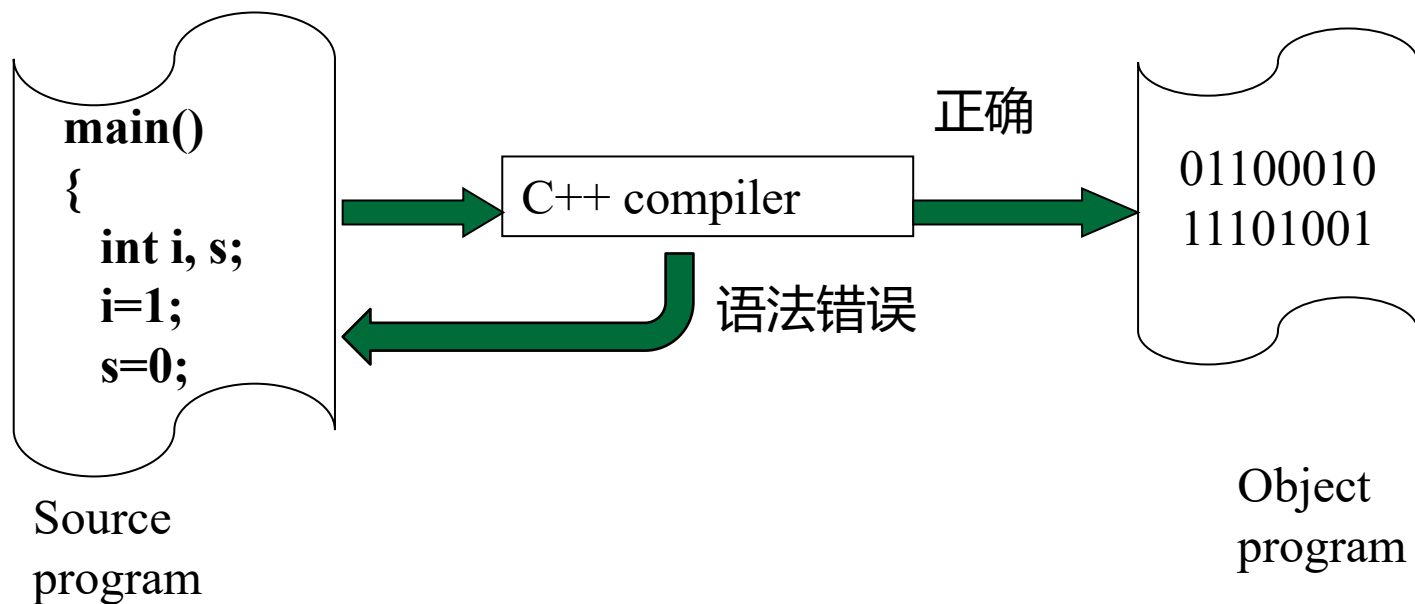


将高级语言的程序翻译成机器语言

## 翻译方式

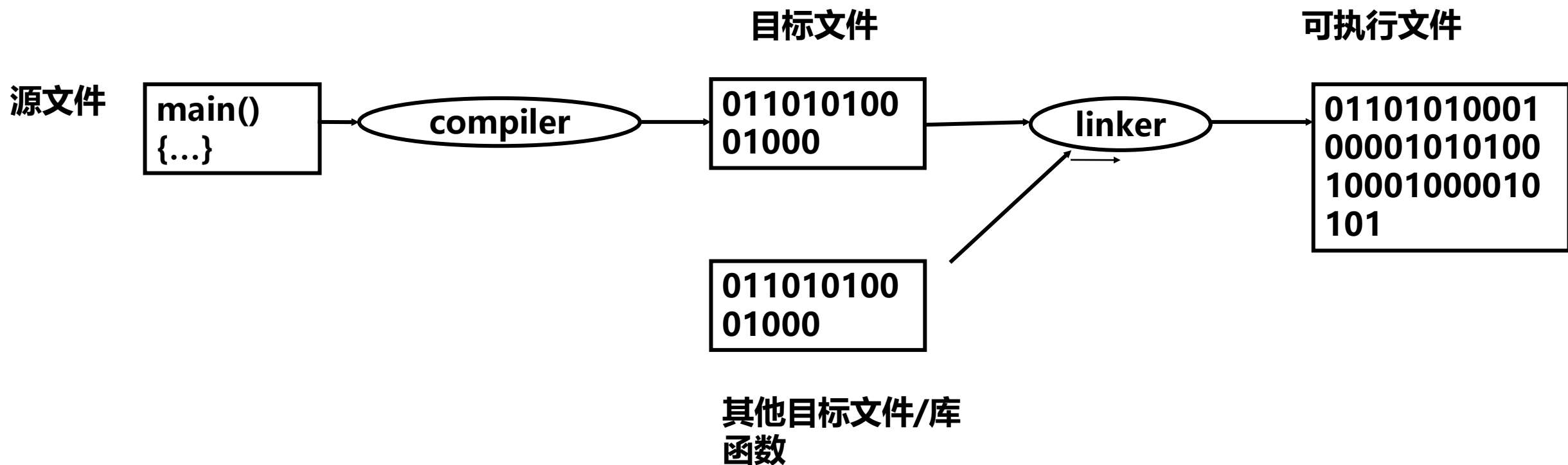
解释执行：逐句翻译并执行

编译执行：全部翻译成机器指令





将目标程序与已有的其它目标程序连接起来，产生一个可执行的程序



将程序代码放入内存，开始执行

## 逻辑错误，亦称为bug（臭虫）

没有得到期望的运行结果

程序异常终止

## 找出逻辑错误的过程称为debug

## 集成的编程环境一般都支持各种调试工具，包括

单步执行

变量跟踪

断点设置

## 选择完备的测试数据

黑盒法

白盒法

# 程序的基本结构

```
// 文件名: 2-1.cpp  
// 用标准公式求解一元二次方程  
#include <iostream>  
#include <cmath>  
using namespace std;
```

} 注释

} 编译预处理指令

} 使用名字空间

主  
程  
序

```
int main()  
{  
    double a, b, c, x1, x2, dlt;  
  
    cout << "请输入方程的3个系数: " << endl;  
    cin >> a >> b >> c;  
  
    dlt = b * b - 4 * a * c;  
    x1 = (-b + sqrt(dlt)) / 2 / a;  
    x2 = (-b - sqrt(dlt)) / 2 / a;  
  
    cout << "x1=" << x1 << " x2=" << x2 << endl;  
  
    return 0;  
}
```

C++的注释是从 // 开始到本行结束

也可以采用C风格的注释，即从/\*与\*/之间所有的文字都是注释，可以是连续的几行。

注释是写给人看的，而不是写给计算机的。

程序注释：从整体描述程序操作过程

注释也可以出现在主程序中，解释主程序中一些比较难理解的部分。

给程序添加注释是良好的程序设计风格



## C++的编译分成两个阶段

预编译

编译

**预编译处理程序中的预编译命令，即那些以#开头的指令**

## 编译预处理指令主要有

库包含：用#include实现，表示程序使用了某个库

**库是预先做好的一些工具程序**

**每个库要提供一个接口，告诉库的用户如何使用库提供的功能**

**库包含就是把库的接口文件放入源文件，以便编译器检查程序中对库的调用是否正确**

## 库包含格式

`#include <filename>`：包含了一个系统库

`#include "filename"`：包含了一个用户自定义的库



# 名字空间

## 问题

在大型的程序时，每个源文件可能由不同的开发者开发

不同的源文件中可能有同样的名字

当这些源文件连接起来形成一个可执行文件时，就会造成重名

## 名字空间

把一组程序实体组合在一起，构成的一个作用域

一个名字空间中不能有重名，不同的名字空间中可以定义相同的实体名

## 实体引用

名字空间的限定：名字空间名::实体名

**C++标准库中的名字都定义在名字空间std中。如std::cout**

# 使用名字空间的指令

## 格式

```
using namespace 名字空间名;
```

一旦用了使用名字空间的指令，该名字空间中的所有实体在引用时就不需要再加名字空间的限定了。

主程序由一个或多个函数组成

函数是一系列独立的程序步骤，这些程序步骤集合在一起，并赋予一个名字。

每个程序都必须有一个名为main的函数，它是程序的入口

```
int main()  }  函数头
```

```
{
```

```
    double a, b, c, x1, x2, dlt;
```

```
    cout << "请输入方程的3个系数: " << endl;
```

```
    cin > a >> b >> c;
```

```
    dlt = b * b - 4 * a * c;
```

```
    x1 = (-b + sqrt(dlt)) / 2 / a;
```

```
    x2 = (-b - sqrt(dlt)) / 2 / a;
```

```
    cout << "x1=" << x1 << "    x2=" << x2 << endl;
```

```
    return 0;
```

```
}
```

函数  
体

注意  
缩进

## 说明函数和外界的交流

### 形式

返回类型 函数名（参数表）

### 返回类型

是函数的输出值的类型

### 函数名

函数的名字。程序可以通过函数名执行函数体的语句

### 参数表

函数的输入

**可以把函数想象成数学中的函数。参数表是一组自变量，返回类型是函数值的类型**

函数如何完成预定功能的过程。它说明了如何从输入（参数）得到输出的（返回值）的过程。

可以把它想象成数学中的函数表达式

# 函数体的组成

```
int main()
{
    double a, b, c, x1, x2, dlt;
    cout << "请输入方程的3个系数: " << endl;
    cin >> a >> b >> c;

    dlt = b * b - 4 * a * c;
    x1 = (-b + sqrt(dlt)) / 2 / a;
    x2 = (-b - sqrt(dlt)) / 2 / a;

    cout << "x1=" << x1 << " x2=" << x2 << endl;

    return 0;
}
```

**变量定义：**为程序编写时值未知的数据预约它们的存放处

**输入阶段：**获取执行时才能确定的用户数据

**计算阶段：**由输入通常通过计算得到结果的过程，算法的体现

**输出阶段：**显示程序执行的结果



**变量，也称为对象。程序中可以变化的值**

**变量有三个重要属性**

名称、类型、值

**C++中变量定义的格式**

类型名 变量名1, 变量名2, ..., 变量名n;

或

类型名 变量名1 = 初值, ..., 变量名n (初值) ;

如:     int num1, num2 = 5, num3;

double area;

**在C++中，每个变量在使用前必须被定义，以便编译器检查变量使用的合法性**

**名字必须以字母或下划线开头名字中的其它字符必须是字母、数字或下划线**

**名字中出现的大写和小写字母被看作是不同的字符**

ABC, Abc, abc是三个独立的变量名

**名字不可以是系统的关键字**

如: int, double, for, return等, 它们在C++语言中有特殊用途

**C++没有规定过名字的长度, 但各个编译系统都有自己规定**

**名字应使读者易于明白其存储的值是什么, 做到“见名知意”**



# 自动类型推断

## 自动类型推断并赋初值

auto 变量名 = 初值;

如: int a=5;

auto b = a;

auto c = 'A' ;

## 仅推断类型

decltype( 表达式) 变量名 ;

如 int a, b;

decltype(a\*b) c;

# 自动类型推断

auto 和 decltype 都是 C++11 引入的类型推导机制，它们各自有不同的用途和意义。

以下是它们之间的区别：

auto 通常用于简单变量的类型推导，它会忽略表达式的引用和 const/volatile 修饰符。

decltype 用于推导表达式的确切类型，包括所有属性。

类型推导机制的存在有以下几个重要意义：

简化代码：减少了程序员需要编写和维护的类型信息，使得代码更加简洁。

提高代码可读性：通过自动推导类型，代码的意图更加清晰，尤其是在处理复杂类型时。

增强代码灵活性：类型推导使得代码更容易适应变化，比如在模板编程中，类型推导可以处理各种不同的模板参数。

减少错误：手动编写类型信息容易出错，类型推导可以减少这种错误。

提高编程效率：类型推导减少了编写代码时的重复性工作，提高了编程效率。

总的来说，类型推导是现代 C++ 编程中的一个重要特性，它使得 C++ 语言更加现代化和高效。

## 数据类型包括两个方面

数据的取值范围

可用的操作

## C/C++ 中的数据类型分为两大类

基本数据类型：整型、浮点型、字符型和布尔型

构造数据类型：数组、结构、联合和枚举



## 基本型 int

4 byte

表示范围  $-2^{31} \sim (2^{31} - 1)$

## 短整型 short

2 byte

表示范围  $-2^{15} \sim (2^{15} - 1)$

## 长整型 long / long int

4 byte

表示范围  $-2^{31} \sim (2^{31} - 1)$

## 长长整型 long long / long long int

8 byte

表示范围  $-2^{63} \sim (2^{63} - 1)$



如何将符号位数字化

数字的三种编码方式为

原码

反码

补码

**用符号位和数值表示带符号数，最高位为符号位**

正数的符号位为0，负数的符号位为1

数值部分用二进制表示

**如用一个字节表示数值**

$[62]_{\text{原}} = 0 \quad 0111110$

$[-62]_{\text{原}} = 1 \quad 0111110$

**表示范围**

-127 ~ +127

**正数的反码与原码相同，负数的反码为该数的绝对值的原码取反。如：**

$$[62]_{\text{反}} = 0 \ 0111110$$

$$[-62]_{\text{反}} = 1 \ 1000001$$

正数的补码与原码相同，负数的补码为该数的反码加1。如：

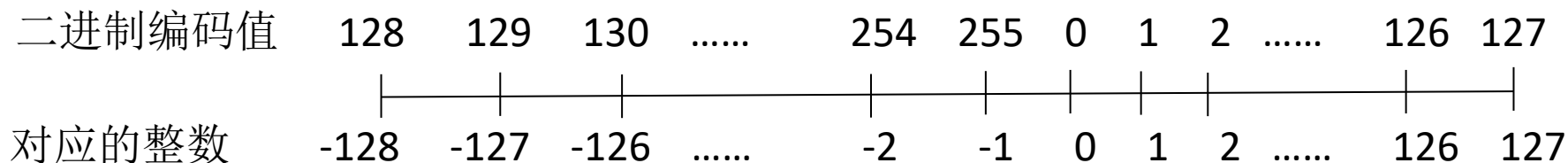
$$[62]_{\text{补}} = 0 \ 0111110$$

$$[-62]_{\text{补}} = 1 \ 1000010$$

$$\begin{array}{r} 11000001 \\ + \quad \quad 1 \\ \hline 11000010 \end{array}$$

0的补码表示是唯一的

$$\begin{array}{r} 11111111 \\ + \quad \quad 1 \\ \hline 1 \ 00000000 \end{array}$$



大多数计算机系统都用补码表示整数

## 整数运算的溢出

超出整型的表示范围

如整数用两个字节表示时，正整数 32767 加 1 的结果为 -32768

**C++ 不检查这样的错误，程序员必须自己保证程序中不出现这样的错误**

将所有的数都看成正整数，无需存储符号位

## 无符号数的定义

在各种整数类型前加上关键词unsigned

变成unsigned int, unsigned short, unsigned long, unsigned long long

unsigned int	$0 \sim 2^{32}-1$
unsigned short	$0 \sim 65535$
unsigned long	$0 \sim 2^{32}-1$

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

int main()
{
    short int a = -1;
    unsigned short b = a;

    cout << a << " " << b << endl;

    a = 32767;
    a = a + 1;
    cout << a << endl;

    return 0;
}
```

运行结果

-1 65535  
-32768



## 定点表示

小数点的位置固定不变

## 浮点表示

小数点位置不固定

如十进制数 $N=246.135$ ，可表示可为

$$\begin{aligned} N &= 246135 * 10^{-3} = 2461350 * 10^{-4} \\ &= 0.246135 * 10^3 = 0.0246135 * 10^4 \end{aligned}$$

## 实数表示

尾数 $\times 2^{\text{指数}}$

尾数和指数都是整数

## 存储实数的空间分成两个部分，分别存储尾数和指数

分配给指数的位数越多，表示数的范围越大，但分配给尾数的位数将减少，从而降低数的精度。



## C++浮点类型

float: 至少32位

double: 至少48位, 且不少于float

long double: 至少和double一样

## VS中的浮点类型

单精度 float: 占用4字节, 精确度7位, 范围 $10^{-38} \sim 10^{38}$

双精度 double: 占8个字节

## 浮点类型可以执行的运算

算术运算

比较运算

输入输出

**实型数在计算机中不能精确表示**

**截断误差**

由于尾数部分位数不够，使数值部分丢失

有时一个十进制转化成二进制数时小数点后会无限循环，因此尾数无法精确表示

**不要判两个实型数相等**

## 字符类型

处理一个字母或符号，占一个字节，存放的是字符的内码

## 字符类型名

char

## 运算

加减、比较、输入输出

## 字符的机内表示

用字符编码表示

常用的有ASCII, BCD, EBCDIC等。PC机中都用ASCII.

## ASCII码的重要特性

数字 '0' 到 '9' 是顺序编码的

字母被分成二段：大写的和小写的。大写字母是连续的，小写字母也是连续的



## 8位二进制数组成

标准：7位数据，一位校验，能编码128个字符

扩展：8位数据，能编码256个字符

## 可打印字符

小写字母、大写字母、数字、标点符号、空格等

## 非打印字符

换行和报警字符或响铃 等控制字符

## 布尔型 (bool)

标准C中没有布尔型数据

占一个字节，它的值为：true, false

## 布尔型数据的内部表示

true为1，false为0

## 布尔型数据的运算

逻辑运算

算术运算



## 方法一

`typedef 已有类型名 新类型名;`

如一旦执行了

```
typedef int INTEGER;
```

那么，要定义一个整型变量a，除了可以用

```
int a;
```

之外，也可以用

```
INTEGER a;
```

## 方法二

`using 别名 = 类型名;`

如：`using INTEGER = int;`



## sizeof 运算符

了解某一类型或某一表达式占用的内存量

## sizeof 运算符的用法

sizeof(类型名) 或 sizeof(表达式)

如：

sizeof(float) : float类型的变量占用的内存量

sizeof(' a' + 15) : 表达式 ' a' +15 的计算结果所占的内存量

写程序时可以确定且在程序运行过程中不会变化的值

常量必须是C++的合法类型

## 整型常量可用十进制、八进制和十六进制表示

十进制: 123, -234

八进制: 0123

十六进制: 0x123, 0x3a2f

## 整型常量的类型

### 十进制常量

默认为int, 除非超出了int的范围

超出int范围将被看成long

超出long的范围将被看成long long

## 八进制和十六进制常量

默认表示成合适范围的无符号数

## 用后缀明确指出常量类型

L或l: 长整型。如 100L

U或u: 无符号数。如100U

LU或UL: 无符号长整型

LL或ll: long long类型

LLU或uLL: 无符号long long类型

## 十进制表示

1.23 3.14 -5.988

## 浮点常量的类型

缺省是double类型

可以用 f 或者 F 明确指明是float类型

如: 1.25F

## 科学计数法

尾数e指数

$123e2=12300$   $2.25e-3=0.00225$

## 使用科学计数法需注意

尾数不能为空  $e3 \rightarrow 1e3$

指数必须为整数  $2.5e2.3$ 是非法的

## 可打印字符

用一对单引号括起来

'a' , 'S' , '2' 等

## 非打印字符：用转义序列

以 \ 开始的一串字符，表示一个字符

如：' \n ' 表示换行

' \t ' 表示水平制表符

' \a ' 表示振铃

# 常用的转义字符

字符形式	含义
\n	换行
\t	水平制表
\b	退一格
\r	回车
\f	换页
\\	\
\'	'
\"	"
\ddd	1到3位八进制数代表的字符
\xhh	1到2位十六进制数代表的字符

给有意义的常数取一个名字

符号常量的命名与变量相同，但通常用大写字母  
含义清楚，提高了程序的可读性。

在需要改变一个常量时能做到 “一改全改”



## 用编译预处理命令#define实现

#define 符号常量 字符串

#define PI 3.14

## 预编译时，用字符串取代程序中的符号常量

## 存在的问题

#define的处理只是简单的字符串的替换，可能会引起一些意想不到的错误。如

#define A 3+5

x = A \* 2的结果是13，而不是16



## 用const定义符号常量

const <类型名> <常量名> = <值>;

如:

```
const double PI = 3.1415926;
```

```
const int AA = 3 + 5;
```

```
c = 2 * AA;
```

**结果是16!**

## 符号常量的初值可以是一个变量或表达式的结果

如下面语句是合法的

```
int a ;
```

```
cin >> a;
```

```
const int x = 2 * a;
```

## 常量表达式的初值必须是编译时的常量

例如

**constexpr** int x = 5 ; 是合法的

而下列语句是非法的

```
int a ;
```

```
cin >> a;
```

**constexpr** int x = 2 \* a;

## cin 对象

键盘流入的数据流

## 流提取运算符>>

二元运算符

将键盘输入的数据存入变量

返回值是左运算数

## 格式

cin >> 变量

cin >> 变量1 >> 变量2 >> ... >> 变量n



**当程序执行到这个语句时会停下来等待用户的输入**

**用户可以输入数据，用回车（↵）结束。**

**当有多个输入数据时，一般用空白字符（空格、制表符和回车）分隔**

如：a 为整型，d 为 double，则对应于

cin >> a >> d, 用户的输入可以为

12 13.2↵

12 (tab键) 13.2↵

12↵13.2↵

## 作用

从键盘接收一个字符

可以输入空白字符

## 用法

`cin.get (ch)`

`ch = cin.get()`

## 比较

如 a, b, c为字符型变量，对应语句

```
a = cin.get();
```

```
b = cin.get();
```

```
c = cin.get();
```

如果输入 a b c↵，则a的值是a，b的值是空格，c的值是b。

如果将这个输入用于语句：

```
cin >> a >> b >> c ,
```

那么变量a、b、c的内容分别为 'a' 、 'b' 、 'c'

# 输出流对象cout

## 输出流对象cout

代表显示器

## 流插入运算符<<

二元运算符

将右边的变量或表达式的内容输出到左边对象

表达式值是左边对象

## 格式

输出一个变量的值: `cout << a;`

输出多个变量的值: `cout << a << b << c;`

输出表达式的结果: `cout << "Hello world"`

上述情况的组合: `cout << a << '+' << b << '=' << a + b << endl;`

用户的输入与程序要求不一致

出现输入错误时，程序将忽略后面所有的输入语句

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

int main()
{
    int a, b, c;
    cout << "请输入3个整数: ";
    cin >> a ;
    cout << a << endl;
    cin >> b ;
    cout << b << endl;
    cin >> c;
    cout << c << endl;
    cout << a + b + c << endl;
    return 0;
}
```

正常程序运行过程如下所示

请输入3个整数: 1 2 3

1

2

3

6

异常程序运行过程如下所示

请输入3个整数: 2 abcv

2

-858933909

-879023454

-1737957361



算术表达式

赋值表达式

关系表达式

逻辑表达式

条件表达式

逗号表达式

## 算术表达式由运算符和运算对象组成

### 算术运算符

+   -   \*   /   %

除 “-” 外，所有的算术运算符都是二元运算符。“-” 可为二元运算，也可为一元运算

### 计算次序

#### 优先级

高 \* / %

低 + -

#### 结合性

左结合

用圆括号改变优先级

### 运算对象

整型、浮点型、字符型和布尔型

**乘号不能省略**

**出现除法时注意括号的应用**

如                      应写为:  $(a + b) / (c * d)$  或  $(a + b) / c / d$   
但不能写成:  $(a + b) / c * d$  或  $a + b / c * d$

**在写算术表达式时，为使表达式更加清晰，一般在运算符前后各插一个空格**

比较     $(a+b)/c*d$                $(a + b) / c * d$

## 算术表达式中可以同时出现各种类型的数据

如:  $3.7 * 2 - 'a' + \text{true}$

**在进行运算前，将运算数转为同一类型，运算结果与运算数相同**

## 转换规则

bool、char和short这些非标准的整数在运算前都必须转换为int

表示范围小的向范围大的转换

精度小的向精度大的转换

int和float运算时，将int转换成double。

int和long运算时，将int转换成long。

int和double运算时，将int转换成double。

float和double运算时，将float转换成double。

**实型 -> 整型**

舍弃小数部分

**整型 -> 实型**

数值不变，但以浮点的形式保存

**double -> float**

截取float的有效数字存放到float变量中

**float -> double**

将有效位扩展到16位

**字符 -> 整型**

将字符型数据作为整型数据的最后一个字节

如果所用系统将字符处理成无符号量，则前面补0。如果所用系统将字符处理成有符号量，则扩展符号。

**整型 -> 字符**

取整型数据的最低8位

## 除法的结果取决于运算数

两个整数相除结果是整数

只要有一个运算数是浮点数，则会保留小数部分

如  $4 / 5 = 0$

$4.0 / 5$  或  $4 / 5.0$  或  $4.0 / 5.0 = 0.8$

## 问题

`int x = 4, y = 5;` 要想使  $x / y$  的结果为 0.8 而不是 0，该怎么办？

## 答案

用强制类型转换

## 强制类型转换格式

C风格: (类型名) 表达式

C++风格: 类型名 (表达式)

**例如，要想使两个整型变量 x 和 y 除的结果为double型，可以用下列语句**

```
double z;  
z = (double) x / y;  
z = double (x) / y;
```

**考虑：如果写为 `z = (double) (x / y)`的结果如何？**

强制类型转换在C语言中引入了一个漏洞

为了方便查找这些错误，C++提供了在强制类型转换时指明转换的性质

## 类型转换运算符

静态转换(static\_cast): 用于编译器隐式执行的任何类型转换

重解释转换(reinterpret\_cast)

常量转换(const\_cast)

动态转换(dynamic\_cast)

## 格式

转换类型 <类型名> (表达式)

```
z = static_cast<double>(x) / y;
```



C++语言中，其他的数学运算都是通过函数的形式来实现

所有的数学函数都在cmath中

要使用这些数学函数，必须在程序头上写上编译预处理命令：

```
#include <cmath>
```

# cmath的主要内容

绝对值函数	<code>int abs(int x) ;    double fabs(double x)</code>
$e^x$	<code>double exp(double x)</code>
$x^y$	<code>double pow(double x, double y)</code>
	<code>double sqrt(double x)</code>
$\ln x$	<code>double log(double x)</code>
$\log_{10}x$	<code>double log10(double x)</code>
三角函数	<code>double sin(double x)</code> <code>double cos(double x)</code> <code>double tan(double x)</code>
反三角函数	<code>double asin(double x)</code> <code>double acos(double x)</code> <code>double atan(double x)</code>

## 赋值表达式格式

<变量> = <表达式>

## 赋值表达式作用

将右边的表达式的值存入左边的变量，整个表达式的值是右边的表达式的结果

## 赋值运算符=的优先级与结合性

比算术运算符低

右结合

## 注意与代数表达式的区别

$x = x + 2$ 是正确的表达式

## 左值(lvalue)

能出现在赋值运算符左边的表达式称为左值

## 表达式语句

表达式;

## 赋值语句

赋值表达式后面加上分号

如:  $x = x + 2$

# 赋值作为表达式的意义

**将赋值表达式作为更大的表达式的一部分 (不要这样写代码)**

如:  $a = (x = 6) + (y = 7)$  是合法的表达式

等价于分别将x 和 y 的值设为6 和 7, 并将6和7相加, 结果存于变量a

**多重赋值 (不要这样写代码)**

$a = b = c = 0$

要保证所有的变量都是同类型的, 以避免在自动类型转换时出现与预期不相符的结果的可能性

如变量d定义为double,变量i定义为int, 语句

$d = i = 1.5;$

的结果是: i等于1, d等于1.0

## 复合赋值运算符

二元运算符与赋值运算符结合的运算符

## 常用的复合赋值运算符

$+=$ ,  $-=$ ,  $*=$ ,  $/=$ ,  $\%=$

## 意义

变量 op = 表达式  变量 = 变量 op 表达式;

如: `balance += deposit;`

`balance -= surcharge;`

`x /= 10;`

`salary *= 2;`

## 自增、自减运算符

**`++`, `--`**

## 作用

相当于`+=1`和`-=1`

## 用法

前缀和后缀两种

`++k`    `k++`    `--k`    `k--`

## 运算结果

前缀：被修改的变量本身

后缀：修改前的变量值

`j = ++i`            `i=4 j=4`

`j = i++`           `i=4 j=3`

`j = --i`            `i=2 j=2`

`j = i--`           `i=2 j=3`

## 一个程序有注释、编译预处理指令、主程序组成

### 主程序是一组函数

函数由函数头和函数体组成

函数体由变量定义、输入、计算、输出组成

### 程序设计风格

用注释告诉读者他们所需要知道的东西

使用缩进来区别程序不同的控制级别

使用有意义的名字

避免直接使用那些有特殊意义的常量

在适当的情况下使用标准习语和习惯



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

int main()
{
    double a, b, c, x1, x2, dlt;

    cout << "请输入方程的3个系数: " << endl;
    cin > a >> b >> c;

    dlt = b * b - 4 * a * c;
    x1 = (-b + sqrt(dlt)) / 2 / a;
    x2 = (-b - sqrt(dlt)) / 2 / a;

    cout << "x1=" << x1 << " x2=" << x2 << endl;

    return 0;
}
```

## 程序执行

有时能得到正确结果

有时会得到结果为无穷大

有时程序甚至异常终止

## 问题

没有考虑特殊情况

不是一元二次方程

$$b^2 - 4ac < 0$$

## 解决方案

检测特殊情况：关系表达式和逻辑表达式

处理特殊情况的机制：if 和 switch 语句

# 关系表达式

用来实现比较

## 关系运算符

$>$ ,  $>=$ ,  $==$ ,  $<=$ ,  $<$ ,  $!=$

## 关系表达式

用关系运算符将二个表达式连接起来,  $x < y$   
关系表达式的结果是: true 或 false

## 优先级

高于赋值运算符, 低于算术运算符

$5 + 3 > 6 - 2$   $\longleftrightarrow$   $(5 + 3) > (6 - 2)$

## 关系运算符内部

$==$ 和 $!=$ 较低

$a < b == c < d$   $\longleftrightarrow$   $(a < b) == (c < d)$

$-2 < -1 < 0$   $\longleftrightarrow$   $(-2 < -1) < 0$  **false**

## 结合性

左结合

## “等于”运算符是由两个等号组成

常见的错误是在比较相等时用一个等号

## 小心避免冗余

主要是在关系表达式中需要判别布尔型的变量的值时

判别一个布尔变量flag的值是否为true，初学者常常会用表达式

```
flag == true
```

事实上，只要用一个最简单的表达式：flag就可以了

用于区分更复杂的情况

逻辑运算符

&& (and)      || (or)      ! (not)

优先级

! > 关系运算符 > && > ||

逻辑表达式

由逻辑运算符连接起来的表达式,其结果为 “真(true)” 或 “假(false)”

# 写出下列问题的逻辑表达式

## 检查字符变量a的内容是否为字母

a是大写字母 || a是小写字母

$a \geq 'a' \ \&\& \ a \leq 'z' \ || \ a \geq 'A' \ \&\& \ a \leq 'Z'$

注意, 不能写成

$'a' \leq a \leq 'z' \ || \ 'A' \leq a \leq 'Z'$

## 整型变量m的内容是否能同时被3和5整除

$m \% 3 == 0 \ \&\& \ m \% 5 == 0$

## 逻辑运算的对象可为任意类型的数据

0为假，非0 为真。

`5 % 2 && p`

`5 > 3 && 2 || 8 < 4 - ! 0`     `true`

逻辑表达式与算术表达式计算有什么不同？？？

## 短路求值

计算逻辑表达式时，先计算左边。如左边已能决定表达式的值，则右边不执行

如  $5 < 3 \ \&\& \ 8 > 4$        $8 > 4$  没有计算

$3 < 5 \ || \ 8 < 4$        $8 < 4$  没有计算

## 优点

减少计算量：在  $\&\&$  逻辑表达式中，应把 false 可能性较大的条件放在左边

在  $\||$  表达式中，应把 true 可能性较大的条件放在左边

一个条件可以控制另一个条件，如  $m \neq 0 \ \&\& \ n / m == 9$

## 尽量避免在一个逻辑表达式中完成多项任务

如  $a = 1, b = 2, c = 2, d = 4, m = 1, n = 1.$

问执行  $(m = a > b) \ \&\& \ (n = c > d)$  后，m、n的值分别为多少？

$m=0, n=1$

# if 语句

## if语句的格式

```
if (条件测试) {语句}
```

```
if (条件测试) {语句1} else {语句2}
```

**条件测试为true时所执行的程序块叫做then子句**

**条件为false时执行的语句叫做else子句**

```
if (grade >= 60)
    { cout << "passed" ; }
if (grade >= 60)
    { cout << "passed" ; }
else
    { cout << "failed" ; }
```

**合理的缩排，使程序结构更加清晰**



条件的结果值应该是 true 或 false

事实上，条件可为任意表达式，不一定是关系表达式和逻辑表达式

0 为false，非 0 为true

## 常见的错误

条件测试是比较相等时，用一个等号

编译器并不报错

```
if (x = 3) cout << " x = 3" ;  
else cout << " x != 3" ;
```

# 判断闰年的程序

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

int main()
{
    int year;
    bool result;

    cout << "请输入所要验证的年份: ";
    cin >> year;

    result = (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;

    if (result) cout << year << "是闰年" << endl;
    else cout << year << "不是闰年" << endl;

    return 0;
}
```

不要写成 `result == true`



## if语句的嵌套

if 语句的 then 子句或 else 子句是 if 语句

## 歧义性

if 语句可以没有else子句，如

if (a > b) if (b > c) max = a; else max =b;

if (x < 100) if (x < 90) 语句1 else if (x<80) 语句2 else 语句3 else 语句4;

## 配对原则

每个else子句是和在它之前最近的一个没有else子句的if语句配对

可以清晰地表示出层次，便于程序员阅读

```
if (x < 100)
    if (x < 90) 语句1
    else if (x<80) 语句2
        else 语句3
else 语句4;
```

如果x<80没有else该怎么处理？

```
if (x < 100)
    if (x < 90) 语句1
    else { if (x<80) 语句2 }
else 语句4;
```

编写一个程序，求解一元二次方程。

最简单的解法： 利用

# 改进解一元二次方程的程序

```
int main()
{
    double a, b, c, x1, x2, dlt;

    cout << "请输入3个参数: " << endl;
    cin >> a >> b >> c;

    if (a == 0) cout << "不是一元二次方程" << endl;
    else {
        dlt = b * b - 4 * a * c;
        if (dlt >= 0) {
            x1 = (-b + sqrt(dlt)) / 2 / a;
            x2 = (-b - sqrt(dlt)) / 2 / a;
            cout << "x1=" << x1 << " x2=" << x2 << endl;
        }
        else cout << "无根" << endl;
    }

    return 0;
}
```

} 复合语句

## 作用

更加简练的用来表达条件执行的方式

## 格式

(条件) ? 表达式1 : 表达式2

## 执行过程

首先计算条件值。如果条件结果为true，则计算表达式1的值，并将它作为整个表达式的值。如果条件结果为false，则整个表达式的值为表达式2的值。

## 将x和y中值较大的一个赋值给max

```
max = (x > y) ? x : y;
```

## 输出根据某个条件有小小的不同

例如

```
if (x == 3)
```

```
    cout << "x的值是3\n" ;
```

```
else
```

```
    cout << "x的值不是3\n" ;
```

更好的做法? ? ?

```
cout << "x的值" << ( (x == 3 ? "" : "不" ) << "是3" << endl; (不要写这样的代码)
```



## 用于多分支的情况

### 格式

```
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
```

```
}
```

可以省略

### 执行过程

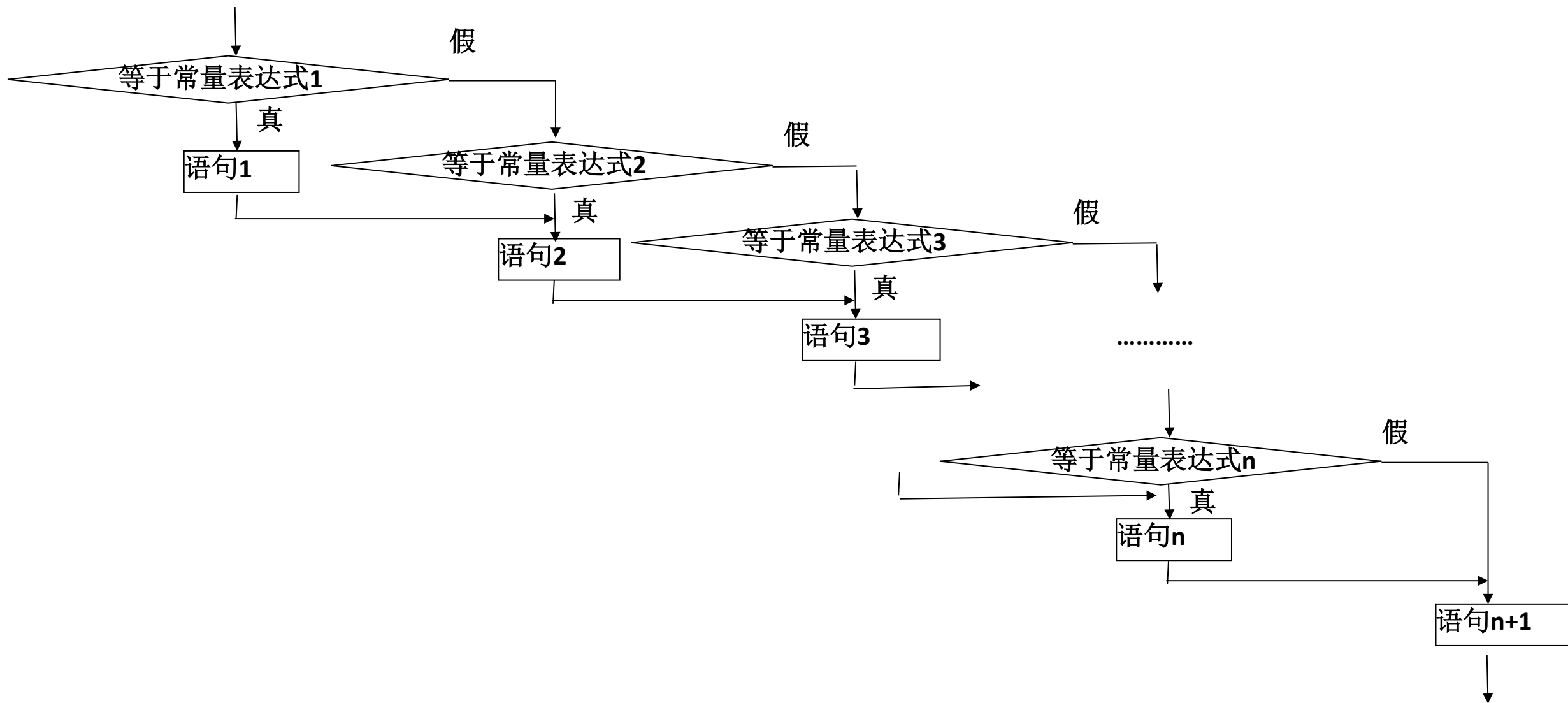
当表达式值为常量表达式1时，  
执行语句1到语句n+1；

当表达式值为常量表达式2时，  
执行语句2到语句n+1；

当表达式值为常量表达式n时，  
执行语句n到语句n+1；

否则，执行语句n+1

# switch语句





## 作用：跳出当前的switch语句

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

## 执行过程

当表达式值为常量表达式1时，  
执行语句1；

当表达式值为常量表达式2时，  
执行语句2；

- 
- 

当表达式值为常量表达式n时，  
执行语句n；

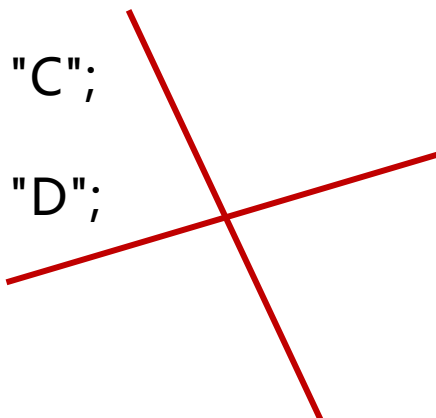
否则，执行语句n+1

条件		
case 1	case2	.....
A	B	

# 按下表将百分制成绩转换成5 级记分制

<b>score<math>\geq</math>90</b>	<b>A</b>
<b>90<math>&gt;</math>score<math>\geq</math>80</b>	<b>B</b>
<b>80<math>&gt;</math>score<math>\geq</math>70</b>	<b>C</b>
<b>70<math>&gt;</math>score<math>\geq</math>60</b>	<b>D</b>
<b>score<math>&lt;</math>60</b>	<b>E</b>

```
switch(score) {  
    case score  $\geq$  90: cout << "A";  
                    break;  
    case score  $\geq$  80: cout << "B";  
                    break;  
    case score  $\geq$  70: cout << "C";  
                    break;  
    case score  $\geq$  60: cout << "D";  
                    break;  
    default: cout << "E";  
}
```



<b>score<math>\geq</math>90</b>	<b>A</b>
<b>90&gt;score<math>\geq</math>80</b>	<b>B</b>
<b>80&gt;score<math>\geq</math>70</b>	<b>C</b>
<b>70&gt;score<math>\geq</math>60</b>	<b>D</b>
<b>score&lt;60</b>	<b>E</b>

成绩档次与个位数无关  
表达式=成绩/10

```
switch(score / 10) {  
    case 10:  
    case 9: cout << "A";  
            break;  
    case 8: cout << "B";  
            break;  
    case 7: cout << "C";  
            break;  
    case 6: cout << "D";  
            break;  
    default: cout << "E";  
}
```

# 计算机自动出四则运算计算题

要求自动出0 - 9之间的四则运算题，并批改结果

生成题目

Switch(题目类型) {

case 加法: 显示题目, 输入和的值, 判断正确与否

case 减法: 显示题目, 输入差的值, 判断正确与否

case 乘法: 显示题目, 输入积的值, 判断正确与否

case 除法: 显示题目, 输入商和余数的值, 判断正确与否

}

**如何让程序每次执行的时候都出不同的题目？即不同的运算数，不同的运算符**

## 运算数自动生成

随机数生成器rand()：能随机生成0到RAND\_MAX之间的整型数

将生成的随机数映射到0 - 9之间：

`rand() % 10`

`rand() * 10 / (RAND_MAX + 1)`

## 运算符的生成

用编码0 - 3表示四个运算符。因此题目的生成就是生成0 - 3之间的随机数。

```
#include <cstdlib>    //包含伪随机数生成函数
#include <iostream>
using namespace std;

int main()
{
    int num1, num2, op, result1, result2; //num1,num2:操作数, op:运算符, result1,result2: 结果

    num1=rand() * 10 / (RAND_MAX + 1);    // 生成运算数
    num2=rand() * 10 / (RAND_MAX + 1);    //生成运算数
    op=rand() * 4 / (RAND_MAX + 1);        // 生成运算符 0--+, 1-- -, 2--*, 3-- /
```



```

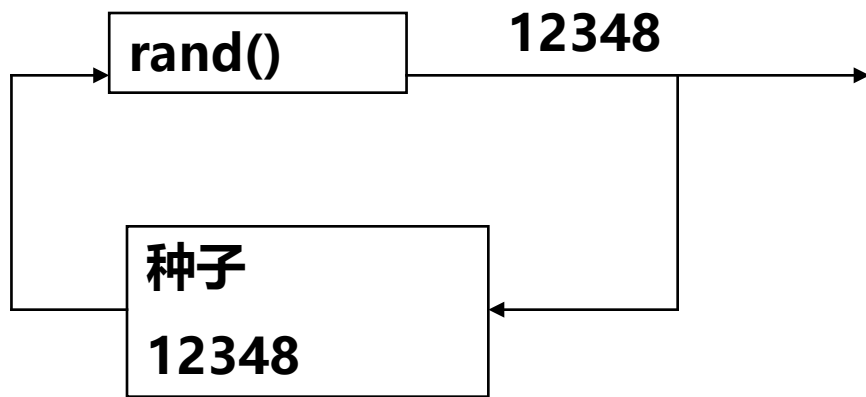
switch (op)    {
    case 0: cout << num1 << "+" << num2 << "= ?" ;    cin >> result1;
        if (num1 + num2 == result1) cout << "you are right\n";
        else cout << "you are wrong\n";
        break;
    case 1: cout << num1 << "-" << num2 << "= ?" ;    cin >> result1;
        if (num1 - num2 == result1) cout << "you are right\n";
        else cout << "you are wrong\n";
        break;
    case 2: cout << num1 << "*" << num2 << "= ?" ;    cin >> result1;
        if (num1 * num2 == result1) cout << "you are right\n";
        else cout << "you are wrong\n";
        break;
    case 3: cout << num1 << "/" << num2 << "= ?" ;        cin >> result1;
        cout << "余数为 = ?"; cin >> result2;
        if ((num1 / num2 == result1) && (num1 % num2 == result2))
            cout << "you are right\n";
        else cout << "you are wrong\n";
    }

    return 0;
}

```

每次运行出的题目  
都是相同的!!!

计算机产生的随机数称为**伪**随机数，它是根据一个算法计算出来的



编译器为每个程序、每次执行指定的随机数的种子都是相同的  
因此程序每次执行生成的随机数序列都是相同的

## 设置种子的函数srand

srand (种子)

**如何让程序每次执行时选择的种子都不一样呢？**

选择系统时间为种子

time(NULL) 取当前的系统时间

```
#include <cstdlib>      //包含伪随机数生成函数
#include <ctime>        //包含取系统时间的函数
#include <iostream>
using namespace std;

int main()
{
    int num1, num2, op, result1, result2;    //num1,num2:操作数, op:运算符, result1,result2: 结果

    srand(time(NULL));    //随机数种子初始化

    num1=rand() * 10 / (RAND_MAX + 1);    // 生成运算数
    num2=rand() * 10 / (RAND_MAX + 1);    //生成运算数
    op=rand() * 4 / (RAND_MAX + 1);        // 生成运算符 0--+, 1-- -, 2--*, 3-- /
```

```
switch (op) {
    case 0: cout << num1 << "+" << num2 << "= ?" ;    cin >> result1;
        if (num1 + num2 == result1) cout << "you are right\n";
        else cout << "you are wrong\n";
        break;
    case 1: cout << num1 << "-" << num2 << "= ?" ;    cin >> result1;
        if (num1 - num2 == result1) cout << "you are right\n";
        else cout << "you are wrong\n";
        break;
    case 2: cout << num1 << "*" << num2 << "= ?" ;    cin >> result1;
        if (num1 * num2 == result1) cout << "you are right\n";
        else cout << "you are wrong\n";
        break;
    case 3: cout << num1 << "/" << num2 << "= ?" ;        cin >> result1;
        cout << "余数为 = ?"; cin >> result2;
        if ((num1 / num2 == result1) && (num1 % num2 == result2))
            cout << "you are right\n";
        else cout << "you are wrong\n";
}

return 0;
}
```

# 程序的缺陷

每次执行只能出一道题

减法可能出现负值

除法可能出现除0

结果太单调

# 计数循环

某一组语句重复执行指定的次数

通常用 for 语句实现，如计算 1 到 100 之和可写为：

```
s=0;  
for (int i=1; i<=100; ++i)    s+=i;
```

循环条件

i 称为循环变量

每次循环后循环变量的修正

for(循环变量赋初值; 是否达到循环次数; 循环变量增值)

符合循环条件时的执行语句

## 格式

```
for (表达式1; 表达式2; 表达式3) {  
    语句  
}
```

循环控制行

循环体

## 执行过程

执行表达式1

执行表达式2

如果表达式2的结果为“true”，则执行循环体和表达式3，然后回到2，否则for语句执行结束

**完整执行一次循环体称为一个循环周期**



**某班级有100个学生，设计一程序统计该班级某门考试成绩中的最高分、最低分 and 平均分**

## 方案一

先输入100个整型数，保存在各自的变量中

依次检查这100个数，找出最大的和最小的

在找的过程中顺便可以把所有的数都加起来。最后将总和除100就得到了平均值

## 缺点

需要定义100个变量

需要100个输入语句输入100个变量的值

从100个变量中找出最大者，需要100个if 语句

从100个变量中找出最小者，需要100个if 语句

将这100个变量加起来需要一个长长的算术表达式

## 思想

每个学生的分数在处理过后就没用了，为此，可以用一个变量保存当前正在处理的分数  
每次输入分数的同时将它们加起来：70加40等于110，110加80等于190……。并记住最低分的和最高分的值。上述过程重复100次

## 实现

变量定义： `int value, total, max, min;`

过程：当输入每个成绩时必须执行下面的步骤，这可以用for循环实现

- 请求用户输入一个整数值，将它存储在变量 `value` 中

- 将 `value` 加入到保存当前和的变量 `total` 中

- 如果 `value` 大于 `max`，将 `value` 存于 `max`

- 如果 `value` 小于 `min`，将 `value` 存于 `min`

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

int main()
{
    int value, total, max, min, i;    //value: 当前输入数据, i为循环变量
    total = 0; max = 0; min = 100;    //变量的初始化
    for (i=1; i<=100; ++i) {
        cout << "\n请输入第" << i << "个人的成绩: "; cin >> value;
        total += value;
        if (value > max) max = value;
        if (value < min) min = value;
    }

    cout << "\n最高分: " << max << endl;
    cout << "最低分: " << min << endl;
    cout << "平均分: " << total / 100 << endl;

    return 0;
}
```

注意  
缩进

## for循环的三个表达式可以是任意表达式

### 三个表达式都是可选的

如果循环不需要任何初始化工作，则表达式1可以缺省。如循环前需要做多个初始化工作，可以将多个初始化工作组合成一个逗号表达式，作为表达式1

如果不需要判断循环是否结束，表达式2可省略

如果不需要修改循环变量，表达式3可以省略。如果需要做多项工作，可以用逗号表达式



# 逗号表达式

## 格式

表达式1, 表达式2, ..., 表达式n

## 执行过程

先执行表达式1, 再执行表达式2, ..., 再执行表达式n

整个表达式的计算结果为最后一个表达式的值

如a的初值为0, 则表达式

$a += 1, a += 2, a += 3, a += 4, a += 5$

的结果为 15

## 优先级

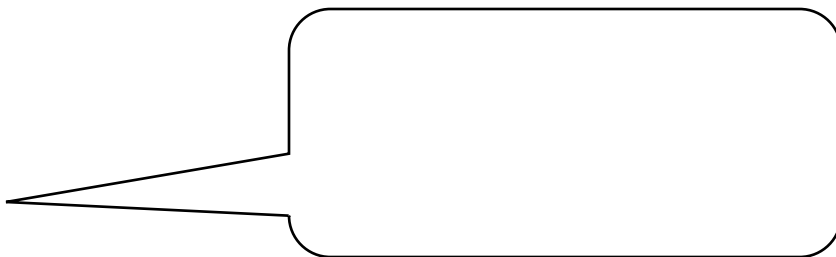
所有运算符中最低的

# 表达式1可选

## 从1加到100的问题

```
s=0;
```

```
for (i=1; i<=100; ++i) s += i;
```



## 将所有的初始化都放在循环内

```
for (s=0, i=1; i<=100; ++i) s += i;
```

## 将所有的初始化都放在循环外

```
i=1; s=0;
```

```
for ( ; i<=100; ++i) s += i;
```

## 表达式2不一定是关系表达式

可以是逻辑表达式，甚至可以是算术表达式

当表达式2是算术表达式时，表达式的值为非0，执行循环体，表达式的值为0时退出循环

## 表达式2省略

不判断循环条件，循环将无终止地进行下去

## 无终止的循环称为“死循环”

最简单的死循环是 `for (;;) ;`

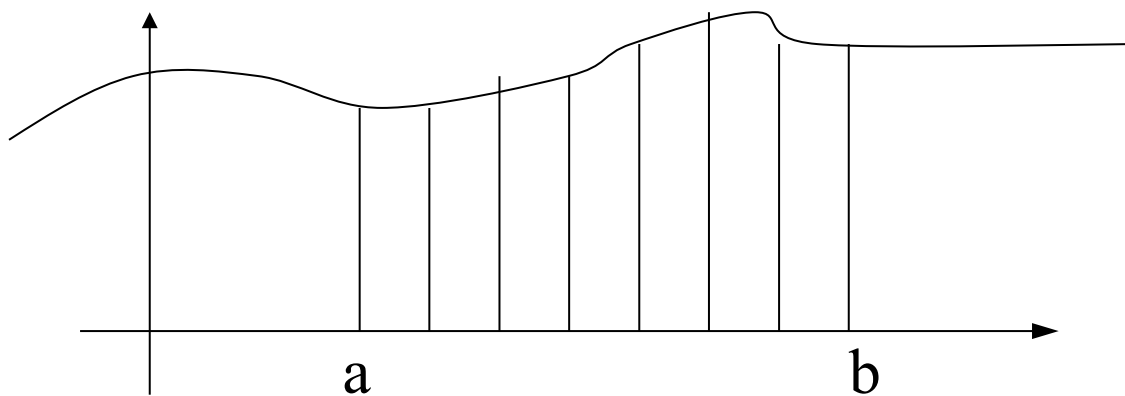
结束一个无限循环，必须从键盘上输入特殊的命令以中断程序执行并强制退出

求函数  $f(x) = x^2 + 5x + 1$  在区间  $[a, b]$  之间的定积分

## 实现思想

定积分是函数与x轴围成的区域的面积

可以通过将这块面积分解成一连串的小矩形，计算各小矩形的面积的和而得到





```
int main()
{
    double a, b, dlt, integral = 0;

    cout << "请输入定积分的区间: ";
    cin >> a >> b;
    cout << "请输入小矩形的宽度: ";
    cin >> dlt;

    for (double x = a + dlt / 2; x < b; x += dlt)
        integral += (x * x + 5 * x + 1) * dlt;
    cout << "积分值为: " << integral << endl;

    return 0;
}
```

**表达式3可以是任何表达式**

一般为赋值表达式或逗号表达式

**表达式3可以省略**

执行完循环体后直接执行表达式2

**如从1加到100，可以写为**

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

```
s=0;
for (i=1; i<=100; s += i, i++);
```

**循环语句的循环体包含一个循环**

**内层的循环在外层循环的每一个周期中都将执行它的所有的周期**

**每个循环都要有一个自己的循环变量以避免循环变量间的互相干扰**

# 打印九九乘法表

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

int main()
{
    int i, j;

    for ( i=1; i<=9; ++i ) {
        for ( j=1; j<=9; ++j )
            cout << i*j << '\t';
        cout << endl;
    }

    return 0;
}
```

- C++11新增加的功能
- 传统的for循环，循环变量的变化是有规律的
  - 计算1到100之和： `s = 0; for ( i=1; i <=100; ++i) s+=i;`
  - 计算1到100中的奇数和： `s = 0; for ( i=1; i <100; i+=2 ) s+=i;`
- 作用：循环变量是取某几个值，但这些值之间并无明确的变化规律
- 格式： `for (循环变量: { 数据列表 } ) 循环体;`  
求1、9、6、8、3的和，可用  
`s = 0;`  
`for ( int i : {1,9,6,8,3} ) s += i;`

- for循环的结束条件是表达式2为假

- 循环体遇到特殊情况需要立即结束循环

- 例如，检测素数

- 特殊情况：1 不是素数，2 是素数，除2之外的偶数都不是素数

- 检测3到n-2之间的奇数，一旦检测到一个因子就可以说明n不是素数，退出循环

- 格式： break

# break语句实例

```
int main()
{
    int num, k;

    cout << "请输入要检测的数: ";
    cin >> num;

    if (num == 2) { cout << num << "是素数\n "; return 0; }
    if (num == 1 || num % 2 == 0) { cout << num << "不是素数\n "; return 0; }

    for (k = 3; k < num; k += 2)
        if (num % k == 0) break;
    if (k < num) cout << num << "不是素数\n";
    else cout << num << "是素数\n";

    return 0;
}
```

□跳出当前循环周期

□例：输出3个字母A、B、C的所有排列方式

- 枚举每个位置的可能值
- 不同位置不能有相同的值

□格式：continue



# continue语句实例

```
int main()
{
    char ch1, ch2, ch3;

    for (ch1 = 'A'; ch1 <= 'C'; ++ch1)           // 第一个位置的值
        for (ch2 = 'A'; ch2 <= 'C'; ++ch2)       // 第二个位置的值
            if (ch2 == ch1) continue;
            else for (ch3 = 'A'; ch3 <= 'C'; ++ch3) // 第三个位置的值
                if (ch3 == ch1 || ch3 == ch2) continue;
                else cout << ch1 << ch2 << ch3 << '\t'; // 输出一个合法的
排列

    return 0;
}
```

# 基于哨兵的循环

## 问题

如何对不同人数的班级完成分数统计任务？

## 方法一

在程序的开始部分请求用户输入数据个数，并将之存放在某个变量中，以此来替换for语句控制行中使用的常量100

## 方法二

定义一个特殊的输入数据，用户可以通过输入该数据来标识输入序列的结束。这个数据称为哨兵

## 方法二需要另外一种的循环控制结构

while循环和do.....while循环

## 格式

while (表达式) 语句

## 执行过程

计算条件表达式的值

如果是false，循环终止，并接着执行在整个while循环之后的语句

如果是true，执行循环体，而后再回到循环控制行，再次对条件进行检查。

## 用途

用于循环次数不定的循环。循环是否结束取决于某一个条件是否成立

设计一个程序，统计某个班级某门考试成绩中的最高分、最低分和平均分。

当输入的分数为-1时，输入结束

```
int main()
{
    int value, total, max, min, noOfInput;
    total = 0; max = 0; min = 100; noOfInput = 0;
    cout << "请输入第1位学生的成绩: ";
    cin >> value;
    while (value != -1) {
        ++noOfInput;
        total += value;
        if (value > max) max = value;
        if (value < min) min = value;
        cout << "\n请输入第" << noOfInput + 1 << "个人的成绩: ";
        cin >> value;
    }
    cout << "\n最高分: " << max << endl;
    cout << "最低分: " << min << endl;
    cout << "平均分: " << total / noOfInput << endl;
    return 0;
}
```

# While语句实例

求  $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$   $\frac{x^n}{n!} < 0.000001$  时结束

```
ex=0;
p = 1;
while ( p>0.000001 ) {
    ex += p;
    计算新的p;
}
```

## 问题

如何计算p?

计算第i个p, 需要两个i次的循环

第一个循环计算 $x^i$

第二个循环计算 $i!$

## 优化方案

从前一项计算后一项。

如果p是第i项的值, 则第  $i+1$  项的值为  $p * x / (i+1)$

```
int main()
{
    double ex = 0, x, p = 1;    //ex存储e^x的值, p保存当前项的值
    int i = 0;

    cout << "请输入x: ";
    cin >> x;

    while (p > 1e-6) {
        ex += p;
        ++i;
        p = p * x / i;
    }

    cout << "e的" << x << "次方等于: " << ex << endl;

    return 0;
}
```

# While语句实例：输入异常检测

输入异常时，>> 操作返回false

通常可用做while循环的条件

```
#include <iostream>
using namespace std;
int main()
{
    int sum = 0, input;
    cout << "Enter numbers: ";
    while (cin >> input)
        sum += input;
    cout << "Last value entered = " << input << endl;
    cout << "Sum = " << sum << endl;

    return 0;
}
```

某次运行过程如下

Enter numbers: 10 -50 -123M 87

Last value entered = -123

Sum = -163



# 输入异常检测函数

eof(): 读到EOF

bad(): I/O失败或一些无法诊断的错误

fail(): 没能读到预期的数据

运行结果

Enter numbers: 10 -50 -123M 87

没有得到正确输入

Last value entered = -123

Sum = -163

```
#include <iostream>
using namespace std;
int main()
{
    int sum = 0, input;

    cout << "Enter numbers: ";
    while (cin >> input)
        sum += input;
    if (cin.eof())    cout << "遇到文件结束\n";
    if (cin.fail())   cout << "没有得到正确输入\n";
    if (cin.bad())    cout << "其他错误\n";
    cout << "Last value entered = " << input << endl;
    cout << "Sum = " << sum << endl;

    return 0;
}
```

# 输入异常恢复: cin .clear()

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

int main()
{
    int sum = 0, input;
    cout << "Enter numbers: ";
    while (cin >> input)
        sum += input;

    cin.clear();
    cin.get();
    cin >> input;

    cout << "Last value entered = " << input << endl;
    cout << "Sum = " << sum << endl;

    return 0;
}
```

某次运行过程如下

Enter numbers: 10 -50 -123M 87

Last value entered = -123

Sum = -163

运行结果

Enter numbers: 10 -50 -123M 87

Last value entered = 87

Sum = -163

# Do...While 循环语句

## 格式

do 语句 while (表达式) ;

## 执行过程

先执行循环体，然后判断循环条件。如条件成立，继续循环，直到条件为假

**例：将输入的整数回显在显示器上，直到输入0为止。**

```
do {  
    cin >> value ;  
    cout << value;  
} while ( value != 0 );
```

# do ..... while实例

计算方程 $f(x)$ 在区间 $[a, b]$ 之间的根

注意,  $f(x)$ 是单调连续的,  $f(a)$ 和 $f(b)$ 必须异号

假设方程为  $x^3 + 2x^2 + 5x - 1 = 0$  , 区间为 $[-1, 1]$

令  $x_1 = a, x_2 = b$

连接  $(x_1, f(x_1))$  和  $(x_2, f(x_2))$  的弦交与  $x$  轴的坐标点可用如下公式求出

$$x = \frac{x_1 * f(x_2) - x_2 * f(x_1)}{f(x_2) - f(x_1)}$$

若  $f(x)$  与  $f(x_1)$  同符号，则方程的根在  $(x, x_2)$  之间，将  $x$  作为新的  $x_1$

否则根在  $(x_1, x)$  之间，将  $x$  设为新的  $x_2$ 。

重复步骤(2)和(3)，直到  $f(x)$  小于某个指定的精度为止。此时的  $x$  为方程  $f(x) = 0$  的根

```
int main()
{
    double  x,  x1 = -1,  x2 = 1,  fx1,  fx2,  fx;

    do {
        fx1 = x1 * x1 * x1 + 2 * x1 * x1 + 5 * x1 -1;
        fx2 = x2 * x2 * x2 + 2 * x2 * x2 + 5 * x2 -1;
        x = (x1 * fx2 - x2 * fx1) / (fx2 - fx1);
        fx = x * x * x + 2 * x * x + 5 * x -1;
        if ( fx * fx1 > 0 ) x1 = x; else x2 = x;
    } while ( fabs( fx ) > 1e-7 );

    cout << "方程的根为: " << x << endl;

    return 0;
}
```

**考虑一个读入数据直到读到标志值的问题**

**如用自然语言描述，基于标志的循环的结构由以下步骤组成**

读入一个值

如果读入值与标志值相等，则退出循环

执行在读入那个特定值情况下需要执行的语句

**循环的中途退出问题**

当一个循环中有一些操作必须在条件测试之前执行时

## 方案一

修改循环结构被改为

读入一个值

```
while (读入值与标志值不相等) {
```

```
    数据处理
```

```
    读入一个值
```

```
}
```

## 方案二

用break语句：跳出循环

```
while (true) {
```

```
    提示用户并读入数据
```

```
    if (value == 标志) break;
```

```
    数据处理
```

```
}
```



对所有可能的情况一种一种去尝试，直到找到正确的答案。

枚举法的实现基础是循环。

# 阶梯问题

有一个长阶梯，若每步上两个台阶，最后剩一阶。若每步上3阶，最后剩2阶。若每步上5阶，最后剩4阶。若每步上6阶，最后剩5阶。每步上7阶，最后正好1阶都不剩。编一程序，寻找该楼梯至少有多少阶。

## 解题思路

枚举1、2、3、.....，寻找满足条件的数

能否优化？？？

枚举7、14、21、.....

```
int main()
{
    int n;

    for (n = 7; (n % 2 != 1 || n % 3 != 2 || n % 5 != 4 || n % 6 != 5); n += 7);

    for (n = 7; ; n += 7)
        if (n % 2 == 1 && n % 3 == 2 && n % 5 == 4 && n % 6 == 5) break;

    cout << "满足条件的最短的阶梯长度是: " << n << endl;

    return 0;
}
```

## 执行结果

满足条件的最短的阶梯长度是: 119

# 实例二 水果问题

用150元钱买了3种水果。各种水果加起来一共100个  
西瓜10元一个，苹果3元一个，橘子1元1个  
设计一个程序，输出每种水果各买了几个

## 它有两个约束条件

第一是三种水果一共100个；  
第二是三种水果一共花了150元

## 解决方案

可以按一个约束条件列出所有可行的情况，然后对每个可能解检查它是否满足第二个约束条件  
也可以用第二个约束条件列出所有情况，然后对每个可能解检查它是否满足第一个约束条件

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

int main()
{
    int mellon, apple, orange;    //分别表示西瓜数、苹果数和橘子数

    for (mellon = 1; mellon < 15; ++mellon)
        for (apple = 1; apple < 150 - 10 * mellon; ++apple) {
            orange = 150 - 10 * mellon - 3 * apple;
            if (mellon + apple + orange == 100) {
                cout << "mellon:" << mellon << ' ';
                cout << "apple:" << apple << ' ';
                cout << "orange:" << orange << endl;
            }
        }

    return 0;
}
```

### 执行结果

```
Mellon: 2 apple: 16 orange: 82
Mellon: 4 apple: 7 orange: 89
```

## 目的

寻求问题最优解

## 适合情况

解决问题的过程由多个阶段组成

## 基本过程

在求解过程的每一阶段都选取一个局部最优的策略，把问题规模缩小  
最后把每一步的结果合并起来形成一个全局解



# 贪婪法实例

给定一组不重复的个位数，例如5、6、2、9、4、1，找出由其中3个数字组成的最大的3位数。

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

int main()
{
    int num = 0, max = 10, current;

    for (int digit = 100; digit > 0; digit /= 10) {
        current = 0;
        for (int n: {5, 6, 2, 4, 9, 1})
            if (n > current && n < max) current = n;
        num += digit * current;
        max = current;
    }

    cout << num << '\t';

    return 0;
}
```

函数是程序设计语言中最重要的部分，是模块化设计的主要工具

函数可以将完成某个任务的一段程序封装起来

使用函数的程序只需要知道函数可以解决什么问题，而不需要知道如何解决

C++程序由一组函数组成，其中必须有一个main() 函数

在C++语言中，字符处理、字符串处理和数学计算都是用函数的方式提供的



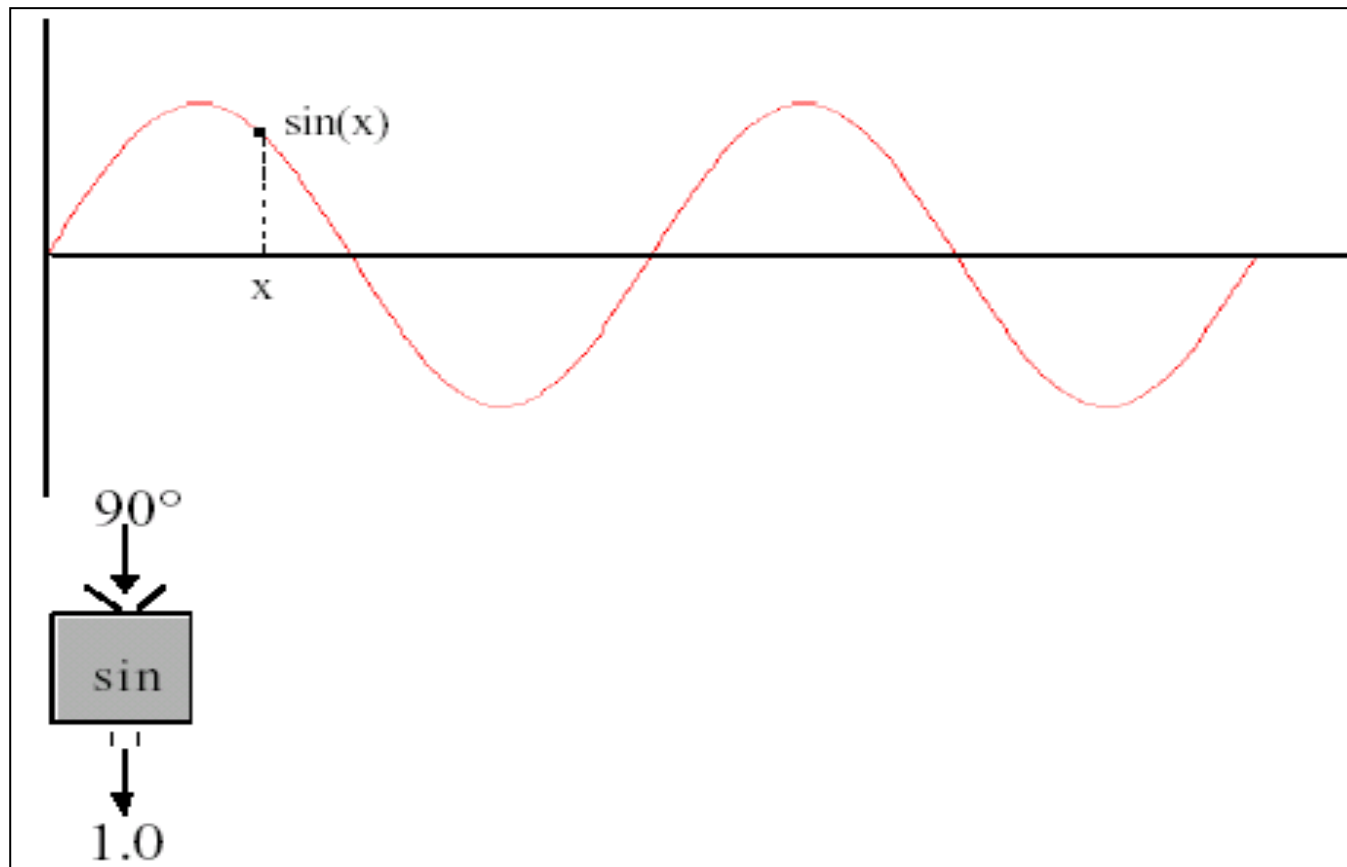
# 函数的例子

可以将函数想象成一个黑盒

在它的上面放入一个“值”，下面就会掉出“结果”

上面的值称为参数

下面的值称为返回值



```
total = sin(30) + sin(25) + sin(68) + sin(87);
```



# 函数的用途

简化了解决问题的主流程

减少了程序中的重复代码

容易保证程序的正确性

每个函数有独立的、明确的功能

不要将多个功能放入一个函数

每个函数的执行结果仅依赖于参数

## 函数定义

```
类型  函数名 (形式参数表)
{
    变量定义部分
    语句部分
}
```

} 函数体

## 如何返回执行结果

```
return 返回值; 或 return (返回值);
int max(int a, int b)
{ if (a>b) return(a) else return(b); }
```

**一个函数没有返回值，类型用void。没有返回值的函数也称为过程**



# 函数的命名

函数名是一个标识符，符合标识符命名规范

函数名要有意义

函数名一般是一个动词短语，表示函数的行为

# 函数举例—无参数、无返回值的函数

打印一个由五行组成的三角形

```
    *
   ***
  *****
 *****
*****
```

```
void printstar()
{
    cout << "  *\n" ;
    cout << "   ***\n" ;
    cout << "  *****\n" ;
    cout << " *****\n" ;
    cout << "*****\n" ;
}
```

# 函数举例—有参数、无返回值的函数

打印一个由n行组成的三角形

```
void printstar(int numOfLine)
{
    int i , j;

    for (i = 1; i <= numOfLine; ++i) {
        cout << endl;
        for (j = 1; j <= numOfLine - i; ++j)
            cout << ' ';
        for (j = 1; j <= 2 * i - 1; ++j)
            cout << "*" ;
    }
}
```

```
      *
     ***
    *****
   *********
  ***********
```

# 函数举例—无参数、有返回值的函数

从键盘获取一个1 – 10之间的整型数

```
int getInput()
{
    int num;
    while (true) {
        cin >> num;
        if (num >= 1 && num <= 10)
            return num;
    }
}
```



# 函数举例—有参数、有返回值的函数

计算n!

```
int p( int n )  
{  
    int s=1, i;  
  
    if ( n < 0 ) return(0);  
    for ( i = 1; i <= n; ++i )  
        s *= i;  
  
    return ( s );  
}
```

# 函数举例—返回布尔量的函数

判断某一年是否为闰年的函数

```
bool IsLeapYear( int year )  
{  
    bool leapyear;  
  
    leapyear = ((( year %4 == 0 )  && ( year % 100 != 0 )) || (year % 400 == 0));  
  
    return ( leapyear );  
}
```

希望编译器自动推断函数返回类型

```
??? add(int a , int b)
{
    return a+b;
}
```

```
decltype( a + b ) add(int a , int b)
{
    return a+b;
}
```



```
decltype(5+7) add(int a , int b)
{
    return a+b;
}
```



## 尾置返回类型

函数返回类型用auto表示

在形式参数表后用 “->类型” 指出真正的返回类型

```
auto add(int a , int b) -> decltype( a + b )
{
    return a+b;
}
```

# 函数的使用

函数声明

函数调用

让编译器知道程序中的函数调用是否正确

函数声明包括下列内容：函数名、函数的参数类型、函数的返回类型

函数的声明形式为： 返回类型 函数名（参数表）；

如：

```
int max(int, int);
```

```
int max(int a, int b);
```

库函数在调用前需要 #include 相应的头文件

## 函数调用形式

函数名 (实际参数表)

eg. `max( x, y);`

## 注意

形式参数和实际参数的个数、排列次序、类型要完全相同。

实际参数可以是常量、变量、表达式，甚至是另一个函数调用  
值传递：实际参数作为形式参数的初值。

## 调用方式

作为语句：`printstar();`

作为表达式的一部分

如要计算  $5! + 4! + 7!$

$x = p(5) + p(4) + p(7)$

# 函数使用实例

```
#include <iostream>
using namespace std;
int max(int a, int b);
```

函数原型说明

```
int main()
```

```
{
```

```
    int x, y;
```

```
    cin >> x >> y;
```

```
    cout << max(x + 5, y - 3);
```

函数调用

```
    return 0
```

```
}
```

```
int max(int a, int b)
```

```
{
```

```
    if (a > b) return(a); else return(b);
```

```
}
```

函数实现



# 函数使用实例

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

int max(int a, int b)
{
    if (a > b) return(a); else return(b);
}

int main()
{
    int x, y;

    cin >> x >> y;
    cout << max(x + 5, y - 3);

    return 0;
}
```

建议用前一种方式！！





# 函数执行过程

在主程序中计算每个实际参数值

用实际参数值初始化形式参数

依次执行函数体的每个语句，直到遇见return语句或函数体结束

计算return后面的表达式的值，用表达式的值构造一个临时变量

回到调用函数，用临时变量置换函数调用，继续主程序的执行

# 函数执行过程

```
int p( int );  
int max( int a, int b )
```

```
int main()  
{  
    int x, y;  
    cin >> x >> y;  
    cout << max(x, y);  
    return 0;  
}
```

```
int p( int n )  
{  
    int s = 1, i;  
    if (n < 0) return(0);  
    for (i=1; i<=n; ++i)  
        s*=i;  
    return(s);  
}
```

main	x(2)		y(3)	
max	a(2)	b(3)	n1(2)	n2(6)
p	n(3)		s	i

```
int max( int a, int b )  
{  
    int n1, n2;  
    n1=p(a);  
    n2=p(b);  
    return (n1>n2? n1: n2);  
}
```

# 变量的作用域

## 变量的作用域

一个变量能被访问的程序部分

## 局部变量

在块内定义的变量称为局部变量

函数的形式参数是局部变量

即使是main函数中定义的变量也是局部的

## 全局变量

在所有的函数外面定义的变量称为全局变量

作用范围：从定义位置后的函数都能使用此变量

## 全局变量的作用

方便函数间的数据传递

```
int a = 2;
int main()
{
    int a = 2, b = 3;
    cout << a << b;
    {
        int a = 4;
        cout << a << b;
    }
    cout << a << b;

    return 0;
}
```

当内部块与外部块有同名标识符时，在内部块中屏蔽外部块的同名标识符

# 写出下面程序的执行结果

```
void f1(int);
int f2(int);
int g = 15;

int main()
{
    cout << g << endl;
    f1(5);
    cout << f2(1) << endl;
    f1(3);
    cout << f2(1) << endl;

    return 0;
}
```

```
void f1(int s)
{
    g = s;
    cout << g << endl;
}

int f2(int s)
{
    return g * s;
}
```

输出：

15  
5  
5  
3  
3

两次调用f2(1)的结果  
是不同的!!!

全局变量破坏了模块化，建议尽量少使用

当全局变量和局部变量同名时，在局部变量的作用范围中全局变量被屏蔽

全局变量的使用将在模块化设计中详细介绍

## 存储类别

变量所存储的区域

## 完整的变量定义

存储类型 数据类型 变量名;

## 存储类别

自动变量: auto

寄存器变量: register

外部变量: extern

静态变量: static

## 在函数内或块内定义的变量缺省时是auto

如 `auto int i;`  
存储在栈工作区

## 行为

进入块时，分配空间。退出块时，释放空间

## C++11的新特性

C++11中自动变量不能加auto  
auto用于变量定义时自动推断变量类型

## 存储在寄存器

代替自动变量或形参，提高变量的访问速度

## 注意

指定register只是一种意向

如无合适的寄存器可用，则编译器把它设为自动变量



## 不在本模块作用范围内的全局变量

### 声明格式

extern 类型 变量;

如:      extern int num;

num为另一个源文件中的全局变量或定义在本语句后面的全局变量

### 用途

在某函数中引用了一个声明在本函数后的全局变量时，需要在函数内用extern声明此全局变量

当一个程序有多个源文件组成时，用extern可引用另一文件中的全局变量

```
#include <iostream>
using namespace std;
void f();

int main()
{
    extern int x;
    f();
    cout << "in main(): x= " << x << endl;
    return 0;
}

int x;

void f()
{
    cout << "in f(): x= " << x << endl;
}
```

不是正常的用法。

正常用法是将全局变量的定义放在源文件的最前面

# 通常的用法

```
//file1.cpp
#include <iostream>
using namespace std;

void f();
extern int x;    //外部变量的声明

int main()
{
    f();
    cout << "in main(): x=  " << x << endl;

    return 0;
}
```

能不能用  
**int x;**

```
//file2.cpp
#include <iostream>
using namespace std;

int x;    //全局变量的定义

void f()
{
    cout << "in f(): x=  " << x << endl;
}
```

符号常量不能声明为extern

符号常量只在一个源文件中有效

整个程序的运行期间都存在、但访问被限定在程序的某一范围内

## 两类静态变量

静态的局部变量

静态的外部变量

# 静态的局部变量

第一次调用函数时定义，程序运行时始终存在，但只能在被定义的函数内使用

函数执行结束时不消亡

再次调用函数时不重新定义，继续沿用原有空间

静态变量定义时如果没有赋初值，默认初值为0

运行结果为：7 8 9

eg.

```
int f(int a)
{
    int b=0;
    static int c=3;

    b=b+1;
    c=c+1;

    return(a+b+c);
}
int main()
{
    for ( int i=0; i<3; ++i)
        cout << f(2);

    return 0;
}
```

## 只能被本源文件使用的全局变量

其他源文件不能用extern引用它

## 函数也能被声明为静态的

该函数只能被用于本源文件中，其他源文件不能调用此函数

```
//file1.cpp
#include <iostream>
using namespace std;

void f();
extern int x;    //出错

int main()
{
    f();
    cout << "in main(): x=  " << x << endl;

    return 0;
}
```

```
//file2.cpp
#include <iostream>
using namespace std;

static int x;

void f()
{
    cout << "in f(): x=  " << x << endl;
}
```





## 参数默认值

声明函数时指定

类型 函数名 (形式参数类型 形式参数名 = 初值) ;

## 作用

如果调用函数时没有为它指定实际参数时，编译器自动将默认值赋给形式参数

例如，将print函数声明为

```
void print(int value, int base=10);
```

调用print(20) 等价于 print(20, 10)

## 缺省参数无论有几个，都必须放在参数序列的最后

例如：Int SaveName (char \*first, char \*second = "" ,char \*third = "" , char \*fouth = "" );

函数调用时，若某个参数省略，则其后的参数皆应省略而取其缺省值

## 参数默认值的指定在函数声明处

因为函数的默认值是提供给调用者使用的

好处： 在不同的源文件中，可以对函数的参数指定不同的默认值

## 目的

为了提高执行效率

## 编译器处理方式

直接用内联函数的代码替换函数调用

省去了函数调用的开销

## 注意

内联函数必须定义在调用的前面

## 内联函数的定义

在函数头部前加保留词inline

```
#include <iostream>
using namespace std;
inline float cube(float s)
{
    return s*s*s;
}

int main()
{
    float side;
    cin >> side;
    cout << cube(side) << endl;

    return 0;
}
```

side \* side \* side

# 慎用内联函数

**内联以代码复制(膨胀)为代价，省去了函数调用的开销，提高函数的执行效率**

## 以下情况不宜用内联

如果函数体内的代码比较长，使用内联将导致内存消耗代价较高

如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大

可用于（但不一定能用于）常量表达式中的函数

## 常量表达式函数定义

在函数的返回类型前加上保留字constexpr。如

```
constexpr int f1() {return 10;}
```

## 常量表达式中可以包含常量表达式函数调用

如： `constexpr int x = 1 + f1();`

```
int f3() { return 10;}
```

```
constexpr int x = 1 + f3();
```



# 常量表达式函数使用注意事项

➤ 只能有一个return语句

```
constexpr int f2(int n)
{
    if (n % 2) return n+10; else return 11;
}
```



➤ 被隐式地指定为内联函数

➤ 常量表达式函数的返回值可以不是常量

```
constexpr int f2(int n)
{
    return (n % 2) ? n+10: 11;
}
```



在C语言中，不允许出现同名函数

当要求写一组功能类似、参数类型或参数个数不同的函数时，必须给它们取不同的函数名

例如某个程序要求找出一组数据中的最大值，这组数据最多有5个数据。我们必须写四个函数：求两个值中的最大值、求三个值中的最大值、求四个值中的最大值和求五个值中的最大值。我们必须为这四个函数取四个不同的函数名，例如：max2, max3, max4和max5。



# 函数重载

## 一组同名函数

### 重载条件

通过函数原型能区分不同函数，如参数个数、参数类型

如    `int max(int a1, int a2);`  
      `int max(int a1, int a2, int a3);`  
      `int max(int a1, int a2, int a3, int a4);`  
      `int max(int a1, int a2, int a3, int a4, int a5);`

### 重载实现

编译器首先为每个函数取一个不同的内部名字

当发生函数调用时，编译器根据实际参数和形式参数的匹配情况确定具体调用的是那个函数，将这个函数的内部函数名取代重载的函数名



## 函数模板

类型的参数化（泛型化）

即把函数中某些形式参数的类型定义成参数，称为模板参数

## 用途

一组仅仅是参数的类型不一样，程序的逻辑完全一样重载函数，可以写成一个函数模板

## 执行

在函数调用时，编译器根据实际参数的类型确定模板参数的值，生成不同的模板函数

# 函数模板的定义

## 定义形式

```
template<模板形式参数表>  
返回类型 函数名(形式参数表)  
{  
    //函数体  
}
```

## 模板形式参数表

class 标识符, class 标识符, .....

如

```
template<class T>  
T max(T a, T b)  
{  
    return a>b ? a : b;  
}
```

# 函数模板实例

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

```
template <class T> T max(T a, T b);
```

```
int main()
{
    cout << "max(3,5) = " << max( 3 , 5 ) <<endl;
    cout << "max(3.3, 2.5) = " << max( 3.3, 2.5 ) <<endl;
    cout << "max('d', 'r') = " << max( 'd', 'r' ) <<endl;

    return 0;
}
```

```
template <class T>
T max(T a, T b)
{ return a>b ? a : b; }
```

函数模板声明

函数模板调用

函数模板定义

```
int max(int , int )
double max(double , double )
char max(char , char )
```



## 用途

某些模板参数在函数的形式参数表中没有出现，编译器就无法推断模板实际参数的类型

如

```
template <class T1, class T2, class T3>
```

```
T1 calc(T2 x, T3 y)
```

```
{ return x + y; }
```

调用calc(5, 'a' ), 则编译器无法推断T3是什么类型。

## 显式实例化

告诉编译器这3个模板参数的实参类型。模板的实际参数放在一对尖括号中，紧接在函数名后面

如函数调用

```
calc<char, int, char>(5, 'a' );    结果是' f'
```

```
calc<int, int, char>(5, 'a' );    结果是102
```

## 只指定部分的类型

如cala的T2和T3都可以自动推断，只需要指定T1

```
calc<int>(5 , 'a');
```

当模板实际参数个数小于形式参数个数时，编译器按从左到右的次序依次匹配

第一个实际参数对应第一个形式参数，第二个实际参数对应第二个形式参数， .....

没有得到实际参数的形式参数对应的类型由编译器自动推断

# 利用尾置返回类型自动推断

```
template <class T1, class T2>  
auto calc(T1 x, T2 y) -> decltype (x + y)  
{ return x + y; }
```



# 函数模板的特化

某些特定的类型无法使用通用的算法

如

```
template <class T>  
T add(T a, T b)  
{  
    return a+b;  
}
```

如过希望只允许数字字符参加运算

调用add( '2' , '3' )返回值是' 5'

其他不合法的字符或结果大于' 9' 返回值都是' 0'

为此类型定义一个特定的版本

调用add( ' a' , ' b' )

结果是没有意义的

# 函数模板的特化

```
template <class T>
T add(T a, T b)
{
    return a+b;
}
```

add(3, 5) 调用普通的模板

add( '2' , '3' ) 调用特化版

```
template <>
char add(char a, char b)
{
    if (a < '0' && a > '9' || b < '0' && b > '9' || a - '0' + b > '9')
        return '0';
    return a - '0' + b;
}
```



## 递归

将问题分解成同类的小问题，小问题的解组成大问题的解

如完成筹集1 000 000元的善款的任务

```
void CollectContributions(int n)
```

```
{
```

```
    if (n <= 100) 从一个捐赠人处收集资金;
```

```
    else {
```

```
        找10个志愿者;
```

```
        让每个志愿者收集n/10元; CollectContributions(n/10);
```

```
        把所有志愿者收集的资金相加;
```

```
    }
```

```
}
```

## 递归函数

在函数中直接或间接地调用函数本身

## 递归要求

必须有递归终止的条件

函数有与递归终止条件相关的参数

在递归过程中，决定终止条件的参数有规律地递增或递减

# 递归函数的常见模式

有可对函数的入口进行测试的基本情况

if (递归终止条件)

不需要递归的简单答案;

else {

递归调用同一函数;

构建答案;

}

# 递归函数实例—阶乘函数

$$n! = \underbrace{1 * 2 * 3 * 4 * \dots * (n-1)}_{(n-1)!} * n$$

```
long p(int n)
{
    if (n == 0) return 1;
    else return n * p(n-1);
}
```

递归函数

递归终止条件

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & \text{其他} \end{cases}$$

# Fibonacci函数

0	1	2	3	4	5	6
0	1	1	2	3	5	8

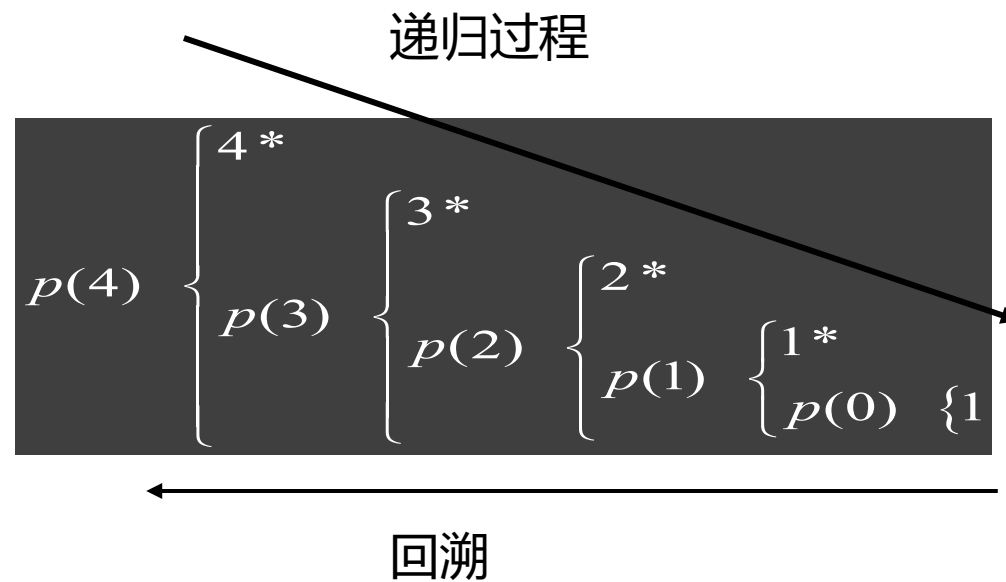
$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & \text{其他} \end{cases}$$

```
int f(int n)
{
    if (n==0) return 0;
    else if (n==1) return 1;
    else return (f(n-1)+f(n-2));
}
```

# 递归函数的执行

```
int p(int n)
{
    if( n == 0)
        return 1;
    else
        return n * p(n-1);
}
```

求 $p(4)$



# 递归与非递归的选择

大多数常用的递归都有等价的非递归实现

究竟使用哪一种，凭你的经验选择

非递归程序复杂，但效率高

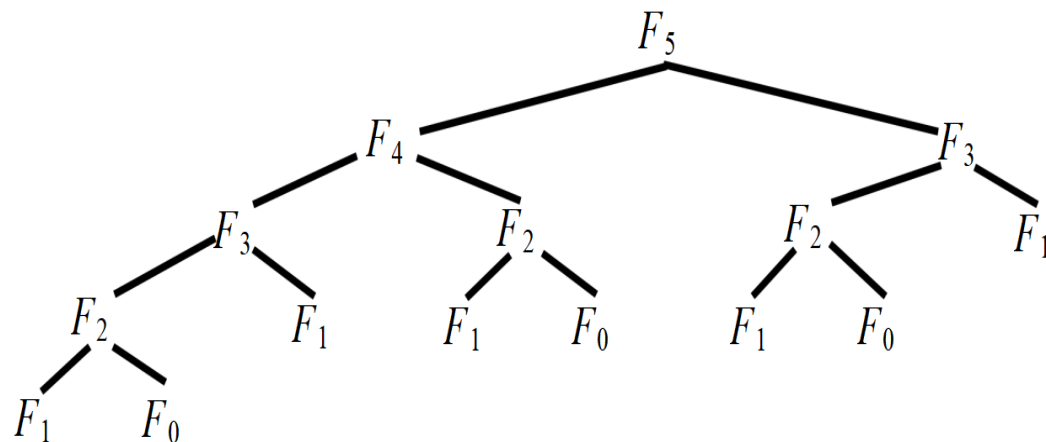
递归程序逻辑清晰，但往往效率较低

如果递归调用会因其重复工作，则不能用递归

如Fibonacci函数的递归实现

```
int f(int n)
{
    if (n==0) return 0;
    else if (n==1) return 1;
    else return (f(n-1)+f(n-2));
}
```

运行时间是灾难性的!!!



# Fibonacci函数的非递归实现

```
int f( int n )
{
    int i, fn, fn_1 = 0, fn_2 = 1;

    if (n == 0) return 0;
    if (n == 1) return 1;
    for ( i = 2; i<=n; ++i) {
        fn = fn_1 + fn_2;
        fn_2 = fn_1;
        fn_1 = fn;
    }

    return fn;
}
```

消耗的时间：执行n次加法和3n次赋值！！！！



# 递归实例 -- 打印三角形

设计一函数，可以在屏幕上的任意地方显示一个任意大小的由任意字符（如：  
\*）组成的倒三角形

## 函数的原型

```
void display( char symbol, int offset, int length)
```

如调用display( '\*' , 0, 11), 显示如下图像

```
*****  
*****  
***
```

**按递归的观点，在某一位置输出一倒三角形可以分成两步**

在指定位置输出长度为length的一行符号

在指定位置的下一行、下二列输出一个大小为 length-4 的有同样的符号组成的倒三角形

## 实现代码

如果有一个函数

draw(symbol , length)

输出 length 个 symbol

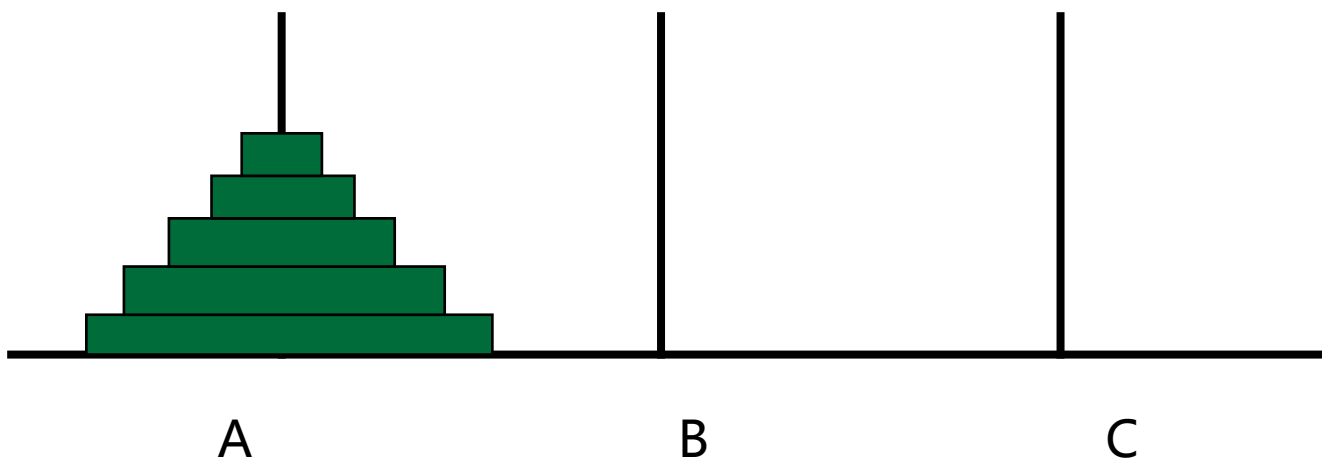
```
void display(char symbol, int offset, int length )
{
    if ( length > 0 ) {
        draw( ' ', offset );
        draw(symbol , length);
        cout << endl;
        display( symbol , offset+2 , length-4 );
    }
}
```

## 递归设计思想

输出k个符号可以看成先输出一个符号，然后再输出k-1个字符组成的一行

```
void draw(char c , int k)
{
    if ( k > 0 ) {
        cout << c;
        draw(c , k-1 );
    }
}
```

# 递归实例—Hanoi塔问题



## 目标

将A上的盘子全部移到B上

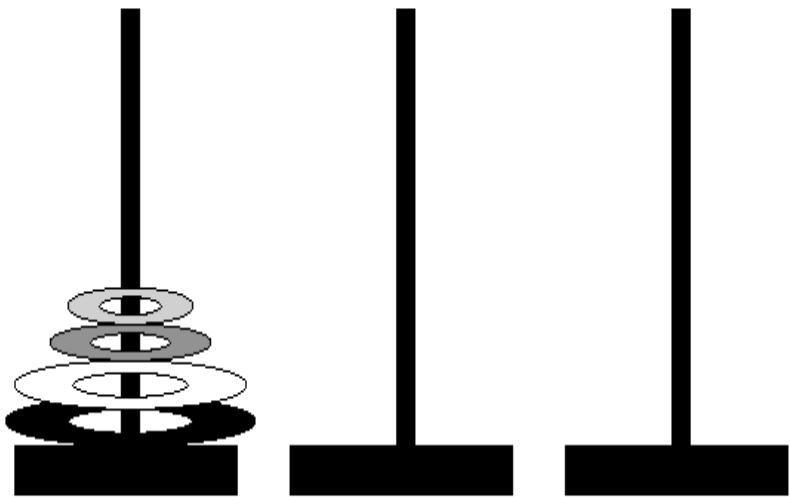
## 规则

每次只能移动一个盘子

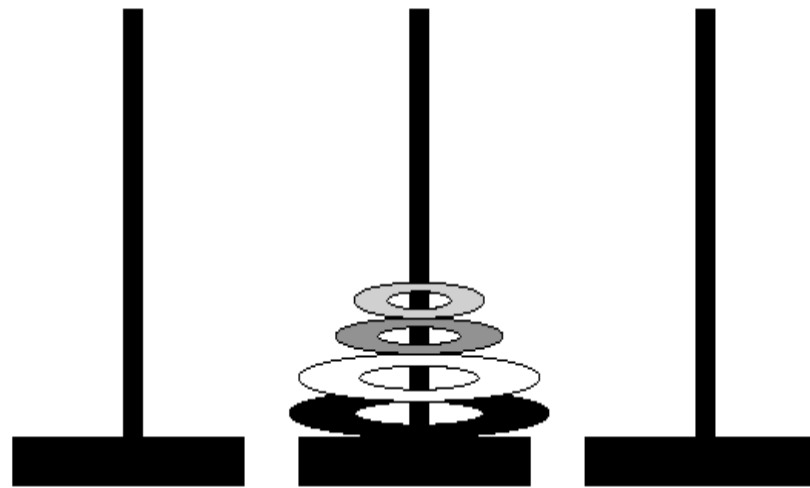
不允许大盘子放在小盘子上

# Hanoi塔

$n = 4$  (最开始的情况)



$n = 4$  (完成情况)

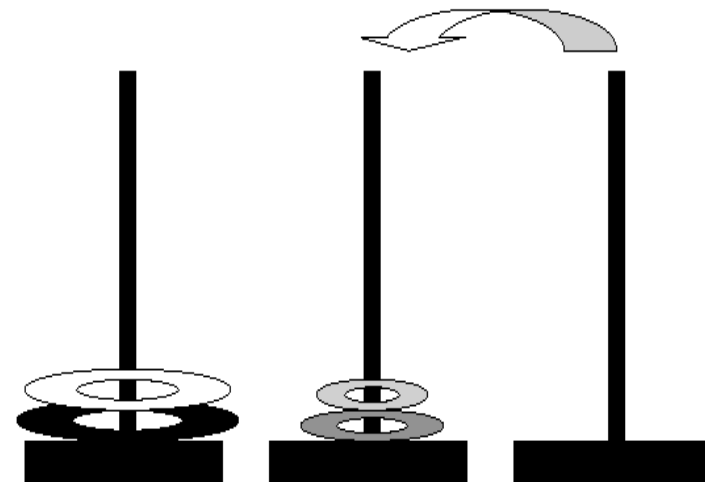
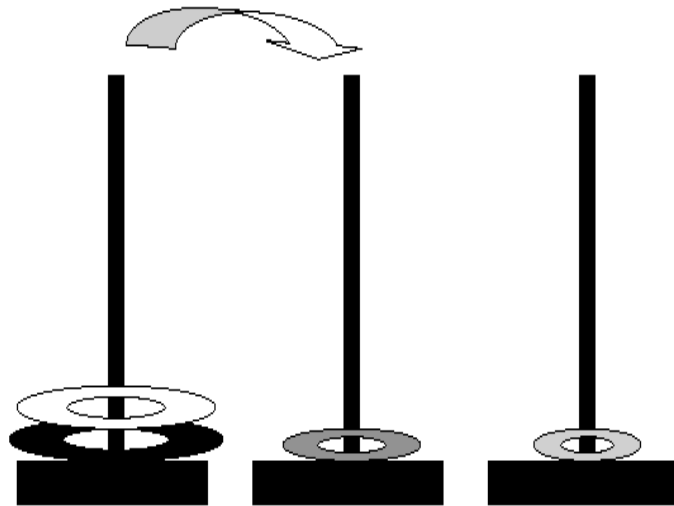
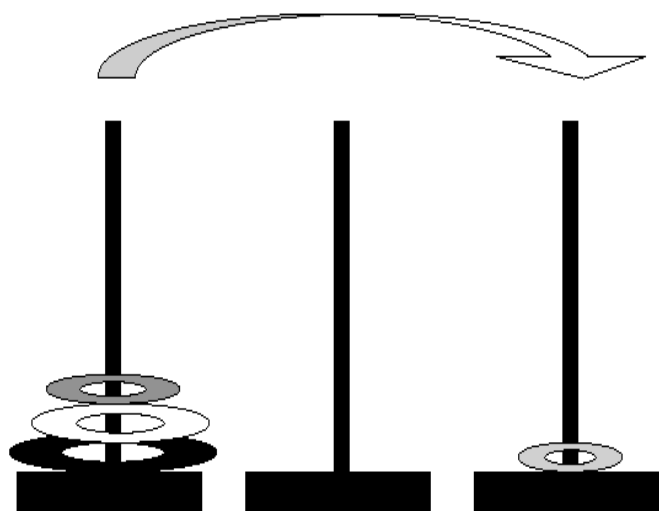


# Hanoi塔

第1步：从开始的杆到辅助杆

第2步：从开始杆到目的杆

第3步：从辅助杆到目的杆

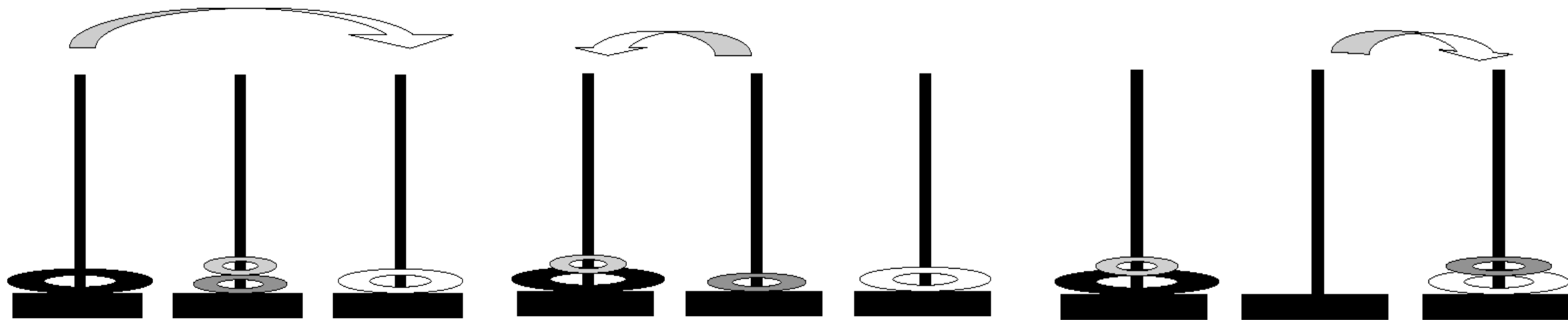


# Hanoi塔

第4步：从开始的杆到辅助杆

第5步：从目的杆到开始杆

第6步：从目的杆到辅助杆

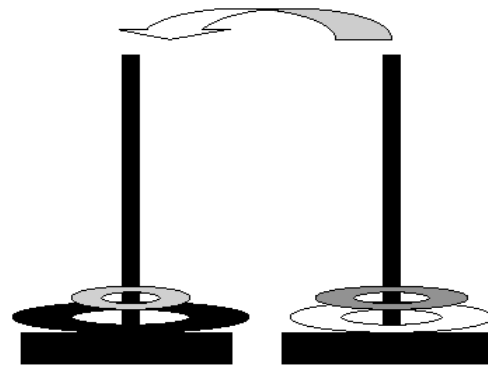
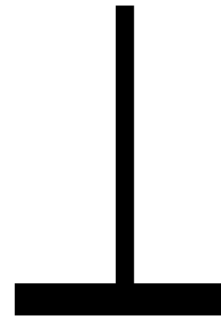
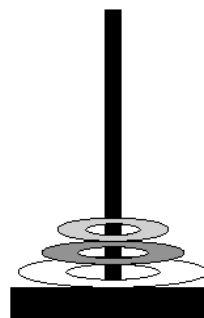
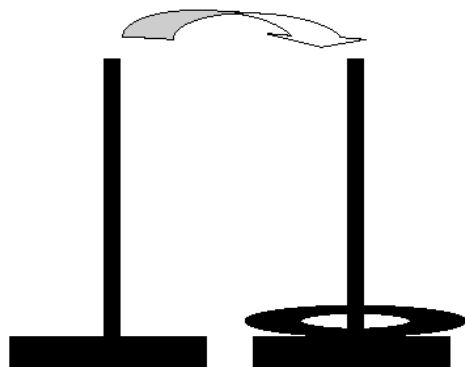
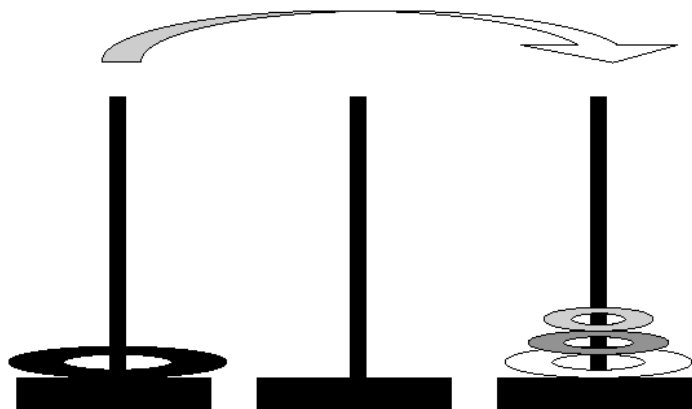


# Hanoi塔

第7步：从开始杆到目的杆

第8步：从开始杆到目的杆

第9步：从辅助杆到目的杆



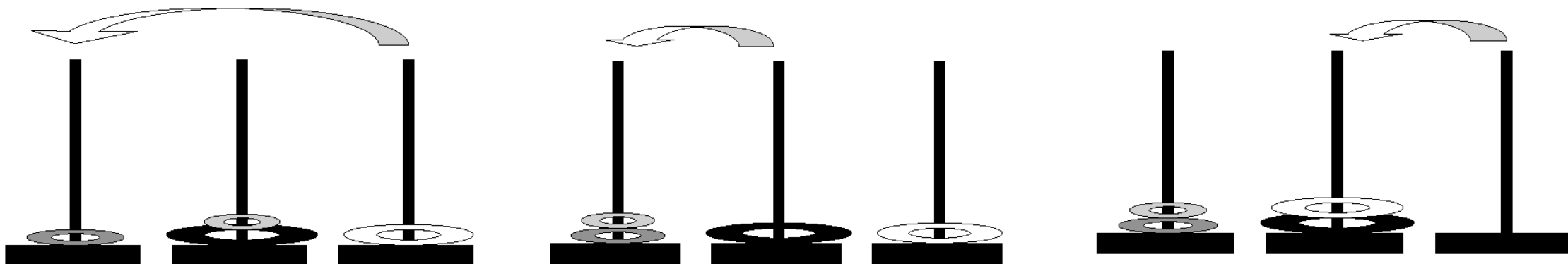


# Hanoi塔

第10步：从辅助杆到开始的杆

第11步：从目的杆到开始杆

第12步：从辅助杆到目的杆

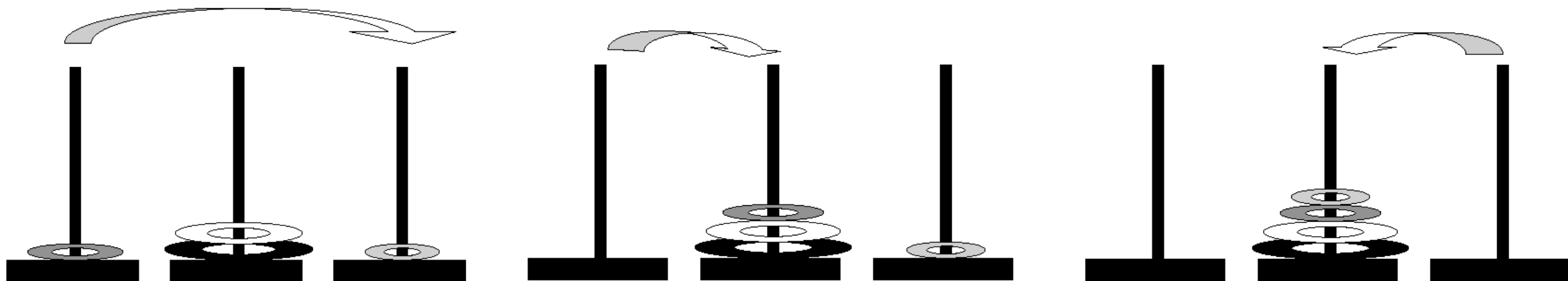


# Hanoi塔

第13步：从开始的杆到辅助杆

第14步：从开始杆到目的杆

第15步：从辅助杆到目的杆



## 最简单的情况，只有一个盘子

将盘子直接从 A 移到 B

## 大于一个盘子的情况

将除了最下面一个盘子外的所有盘子从 A 移到 C

将最下面的盘子从 A 移到 B

将 C 上的盘子移回 B

# Hanoi 塔函数

```
void Hanoi(int n, char start, char finish, char temp)
{
    if (n==1)
        cout << start << "->" << finish << '\t';
    else {
        Hanoi(n-1, start, temp, finish);
        cout << start << "->" << finish << '\t';
        Hanoi(n-1, temp, finish, start);
    }
}
```

如果C++只提供输出一个字符的函数put，设计一个输出非负整型数的函数

## 递归实现

最简单的情况：1位数

将这位数转换成数字输出

$n + '0'$

大于1位的情况

先打印去除个位数以后的数字

打印个位数

$n / 10$

递归

$n \% 10$

如果C++只提供输出一个字符的函数put，设计一个输出非负整型数的函数

```
#include <iostream>
using namespace std;
void printInt(int); //输出一个非负整型数
int main()
{
    int num;
    cout << "请输入一个整型数: " << endl;
    cin >> num;
    printInt(num);
    cout << endl;
    return 0;
}
```

```
void printInt(int num)
{
    if (num < 10)
        cout.put(num+'0');
    else {
        printInt(num/10);
        cout.put(num%10 + '0');
    }
}
```

函数可以将一段完成独立功能的程序封装起来

通过函数名就可执行这一段功能

使用函数可以将程序模块化

C++的程序是由一组函数组成。每个程序必须有一个名为main的函数，它对应于一般的程序设计语言中主程序

函数也可以调用自己，这样的函数称为递归函数

基于递归的算法

**在统计100个学生的考试成绩时，不仅要计算均值还要计算方差**

## 解决方案

定义100个 int 型的变量  $n_1, \dots, n_{100}$

将100个数相加后除100，得到均值

再通过这100个数和均值求得方差

## 缺点

程序只能用顺序结构。程序很长、很啰嗦

如果人数发生变化，程序就得重写



**数组是保存一组同类元素的数据类型**

**数组的两个特征**

数组元素是有序排列的

数组元素是同类的

**定义数组**

数组名字

数组元素的类型

数组的大小

# 数组的定义

## 格式

类型 数组名[元素个数];

其中，元素个数必须是常量。如：

```
int intarray[10];
```

但 `int n=10;`

`int intarray[n];` 是错误的

## 常用的方法

```
#define NumOfElement 10
```

```
int intarray[ NumOfElement ]; 相当于
```

```
int intarray[ 10 ];
```

## 数组初始化

```
float x[5] = { -1.1, 0.2, 33.0, 4.4, 5.05 };
```

**初始化表的长度可以短于要被初始化的数组元素数目**

剩余元素被初始化为0

**带有初始化的数组可以不定义数组规模**

编译器根据初值的个数决定数组的大小

```
int a[]={1, 2, 3, 4, 5}; 则默认数组大小为5
```

## 数组访问是访问数组元素

## 数组元素的表示

数组名[序号]

如intarray[2]

当数组的大小为n时，元素的序号为 $0 - n-1$

## 下标

元素的序号

程序中，下标可为整数、整型变量或结果为整型的任意表达式

# 数组的输入输出

```
int main()
{
    int a[10], idx;

    for (idx = 0; idx <= 9; ++idx)
        cin >> a[ idx ];
    cout << endl;

    for ( idx = 0; idx <= 9; ++idx)
        cout << a[ idx ];

    return 0;
}
```

for (int x : a)  
cin >> x;



for (int x : a)  
cout << x;

for (auto x : a)  
cout << x;

# 数组在内存中

定义数组就是定义了一块连续的空间

空间的大小等于元素数\*每个元素所占的空间大小。

数组元素按序存放在这块空间中

**数组名记录了这块空间的起始地址**

# 数组在内存中

如有定义: `int intarray[5];`

占用了20个字节, 因为每个整型数占四个字节。

如给`intarray[3]`赋值为3, 如果这块空间的起始地址为100, 那么在内存中的情况是

随机值	随机值	随机值	3	随机值
100	103	104	107	108
111	112	115	116	119

访问变量`intarray[idx]`

编译器计算它的地址 $100 + \text{idx} * 4$ , 对该地址的内容进行操作

# 数组下标超界问题

## C/C++语言不检查数组下标的超界

如定义数组 `int intarray[10]`; 合法的下标范围是0 – 9

但引用`intarray[10]`，系统不会报错

如数组`intarray` 的起始地址是1000，当引用`intarray[10]`时，对1040号内存进行操作。而1040可能是另一个变量的地址

## 解决方法

由程序员自己控制

在对下标变量进行操作前，先检查下标的合法性



# 数组应用—求均值和方差

```
int main()
{
    double num[10], avg = 0, dev = 0;
    int i;
    for (i = 0; i < 10; ++i) {
        cout << "请输入第" << i << "个数: ";
        cin >> num[i];
    }
    for (i = 0; i < 10; ++i) avg += num[i];
    avg /= 10;
    for (i = 0; i < 10; ++i)
        dev += (num[i] - avg) * (num[i] - avg);
    dev /= 10;
    cout << "均值为: " << avg << endl;
    cout << "方差为: " << dev << endl;
    return 0;
}
```

# 使均值方差问题的程序更通用

元素个数不确定怎么办？

## 方案一

可以将被统计的数字的个数定义成一个符号常量  
需要时，可以修改这个符号常量的值，重新编译

## 方案二

定义一个足够大的数组存放被统计数字的信息  
定义一个输入结束标志，用while循环解决这个问题  
可参照分数统计程序。不需要重新编译程序

**从键盘输入一串字符，直到输入非字母，统计字符串中个字母出现的次数**

## 解决方法一

用26个整型变量计数26个字母，对输入字符串中的每一字符用switch语句分别计数

## 解决方法二

用一个26个元素的数组，如num[26]，表示计数

num[0]存放a的个数, num[1]存放b的个数....

对每一个字符不必用switch，而只需用一个简单的计算：

```
++num[ toupper(ch) - ' A' ];
```

```
#include <iostream>
#include <ctype.h>
using namespace std;
```

```
int main()
{
    int count[26] = {0}, i;
    char ch;

    ch = toupper(cin.get());
    while (ch >= 'A' && ch <= 'Z') {
        ++count[ch - 'A'];
        ch = toupper(cin.get());
    }

    for (i=0; i< 26; ++i)
        cout << count[i] << '\t';

    return 0;
}
```

```
while (ch = toupper(cin.get()))
    if( ch < 'A' || ch > 'Z' ) break;
    else ++count[ch - 'A'];
```

# 一维数组作为函数的参数

**设计一函数，统计10位同学的平均成绩**

## 设计考虑

如何设计参数和返回值

返回值是平均成绩

## 参数设计方法一

用10个整型的形式参数

## 参数设计方法二

一个10个元素的整型数组

**第二种方法更加简练**

# 统计函数的实现

```
int average(int array[10])    ? ? ?  
{  
    int sum = 0;  
  
    for (int i = 0; i < 10; ++i)  
        sum += array[i];  
  
    return sum / 10;  
}
```

# average函数的使用

```
int main()
{
    int score[10];

    cout << "请输入10个成绩: " << endl;
    for (int i = 0; i < 10; i++)
        cin >> score[i];

    cout << "平均成绩是: " << average(score)
        << endl;

    return 0;
}
```

请输入10个成绩:

90 70 60 80 65 89 77 98 60 88

平均成绩是: 77

注意：形式参数是数组，实际参数也是一个数组

# 一个有趣的现象

在函数average的return语句前增加一个对array[3]赋值的语句，如array[3] = 90。

在main函数的return语句前增加一个输出score[3]的语句

结果是什么？ ？ ？



# 统计函数的实现

```
int average(int array[10])
{
    int sum = 0;

    for (int i = 0; i < 10; ++i)
        sum += array[i];

    array[3] = 90;

    return sum / 10;
}
```

# average函数的使用

```
int main()
{
    int score[10];

    cout << "请输入10个成绩: " << endl;
    for (int i = 0; i < 10; i++)
        cin >> score[i];
    cout << "平均成绩是: " << average(score) << endl;
    cout << score[3] << endl;

    return 0;
}
```

请输入10个成绩:

90 70 60 80 65 89 77 98 60 88

平均成绩是: 77

????

90

# 数组参数的传递机制

C++语言规定，数组名是数组的起始地址

参数传递时，实际参数是数组名，形式参数也是数组名

参数传递时，用score 初始化形式参数数组array

如 score 的首地址为1000，在函数中形参数组array的首地址也为1000。

形式参数和实际参数是同一数组！！！！

# 数组作为函数的参数

在函数中并没有定义新的数组

对形式参数数组指定规模是没有意义的

形式参数数组不需要指定大小，所以方括号中为空

函数如何知道数组的规模？用另一个整型参数表示

总结：数组传递需要两个参数，数组名和数组规模

# 均分纸牌问题

有 $n$ 堆纸牌，编号分别为 $0, 1, 2, \dots, n-1$ 。每堆上有若干张纸牌，但纸牌总数必为 $n$ 的倍数。相邻堆之间可以一次移动若干张纸牌。找出使每堆上纸牌数相同的最少移动次数

	0	1	2	3
例如： $n=4$ ，4堆纸牌分别为：	9	8	17	6

移动三次可以达到目的：

从2取4张牌放到3；

再从2取3张牌放到1；

最后从1取1张牌放到0

## 存储设计

每堆纸牌存放的是一个数字， $n$ 堆纸牌可以用一个 $n$ 个元素的整型数组 $num$ 存储

## 求解过程

移动后，每堆中的牌数相同，所以先求出平均数 $v$

移动过程由 $n-1$ 个阶段组成，每个阶段评估 $i$ 和 $i+1$ 两个堆之间的移动

- 若 $num[i] > v$ ，将 $num[i] - v$ 张牌从第 $i$ 堆移动到第 $i+1$ 堆；
- 若 $num[i] < v$ ，将 $v - num[i]$ 张牌从第 $i+1$ 堆移动到第 $i$ 堆。

```
int cardAvg(int num[], int size)
{
    int avg = 0, sum = 0;

    for (int i = 0; i < size; ++i)
        avg += num[i];
    avg /= size;

    for (int i = 0; i < size - 1; ++i)
        if (num[i] != avg) {
            ++sum;
            num[i+1] -= avg - num[i];
        }

    return sum;
}
```

想一想：

如果某次移动时牌不够，会不会影响移动次数？？？

如何输出移动过程？

# 排序和查找

**顺序查找**

**二分查找**

**选择排序法**

**气泡排序法**



## 被查找的数存放在一个数组中

## 过程

从数组的第一个元素开始，依次往下比较，直到找到要找的元素为止。

## 顺序查找函数模板

模板参数：元素类型

函数参数：数组、被查找元素

返回值：被查找元素在数组中的位置，-1表示不存在

```
template <class T>
int find(T a[], int size, T x)
{
    for (int k = 0; k < size; ++k)
        if (x == a[k]) return k;

    return -1;
}
```

数组元素已按某一顺序排序，如数字的大小顺序、字母的字母序等

以下讨论中都假设是按升序排序

## 过程

设定查找范围的上下界：lh, rh

找出中间元素的位置： $\text{mid} = (\text{lh} + \text{rh}) / 2$

比较中间元素与欲查找的元素 key

如 key 等于中间元素，则 mid 就是要查找的元素的位置；

如  $\text{key} >$  中间元素，则从 lh - mid 的这些元素不可能是要查找的元素，修正查找范围为  $\text{lh} = \text{mid} + 1$  到 rh；

如  $\text{key} <$  中间元素，则从 mid - rh 的这些元素不可能是要查找的元素，修正查找范围为 lh 到  $\text{rh} = \text{mid} - 1$ ；

如  $\text{lh} > \text{rh}$ ，则要查找的元素不存在，否则返回第二步

0	1	2	3	4	5	6	7	8	9	10
12	14	18	20	21	22	24	26	28	30	33

## 查找28

查找区间: [0, 10] [6, 10]

Mid: 5 8 找到

## 查找23

查找区间: [0, 10] [6, 10] [6, 7] [6, 5]

Mid: 5 8 6 没找到

# 二分查找程序

```
template <class T>
int binarySearch (T a[], int size, T x)
{
    int low = 0, high = size - 1, mid;

    while ( low <= high ) {
        mid = ( low + high ) / 2;
        if ( x == a[mid] ) return mid;
        if ( x < a[mid])
            high = mid - 1;
        else low = mid + 1;
    }

    return -1;
}
```

## 顺序搜索的平均比较次数

$$(1 + 2 + 3 + \dots + n) / n = (n + 1) / 2$$

## 二分查找的最坏情况的比较次数

$$n / 2 / 2 \dots / 2 / 2 = 1$$



K次

$$k = \log_2 n$$

# N 和 $\log_2 N$ 的值

N	$\log_2 N$
10	3
100	7
1000	10
1,000,000	20
1,000,000,000	30

## 过程

在所有元素中找到最小的元素放在数组的第0个位置

在剩余元素中找出最小的放在第一个位置

以此类推，直到所有元素都放在适当的位置

## 用伪代码表示

```
int lh, rh, array;  
输入要排序的元素，存入array;  
for (lh = 0; lh < n; lh++) {  
    在 array 的从 lh 到 n - 1 的元素之间找出最小的放入 rh;  
    交换下标 lh 和 rh 中的值;  
}  
输出排好序的元素;
```

# 直接选择排序实例

31	41	59	26	53	58	97	93
----	----	----	----	----	----	----	----

26	41	59	31	53	58	97	93
----	----	----	----	----	----	----	----

已正确定位

26	31	59	41	53	58	97	93
----	----	----	----	----	----	----	----

已正确定位

26	31	41	59	53	58	97	93
----	----	----	----	----	----	----	----

已正确定位



# 直接选择排序的实现

```
template <class T>
void selectSort(T a[], int size)
{
    for (int lh = 0; lh < size; ++lh) {
        int min = lh;
        for (int k = lh; k < size; ++k)
            if ( a[k] < a[min] ) min = k;
        T tmp = a[lh];
        a[lh] = a[min];
        a[min] = tmp;
    }
}
```

# 直接选择排序的效率

如果要排序 $n$ 个元素

找出第一个元素要比较 $n-1$ 次

找出第二个元素比较 $n-2$ 次

...

找出第 $n-1$ 个元素比较1次

总的比较次数为

$$1 + 2 + 3 + \dots + (n-1) = n(n-1)/2$$

# 冒泡排序法

## 过程

对数组元素进行  $n-1$  次冒泡

第一遍冒泡冒出一个最大的气泡，放入最后一个位置

对剩余元素再进行第二次冒泡，冒出最大的泡放入倒数第二个位置

依次执行到最后一个元素

## 伪代码表示

for ( $i=1; i < n; ++i$ )

    从元素 0 到元素  $n-i$  进行冒泡，最大的泡放入元素  $n-i$ ;

## 将待冒泡的数据从头到尾依次处理

比较相邻的两个元素，如果大的在前小的在后，就交换这两个元素

这样经过从头到尾的检查，最大的一个就被交换到最后了

**如果在一次起泡中没有发生交换，则表示数据都已排好序，不需要再进行起泡**

## 进一步细化

```
for (i=1; i<n; ++i) {  
    for (j = 0; j <n-i; ++ j)  
        if (a[j] > a [j+1]) j 交换;  
    if (没有发生过数据交换) break;  
}
```

待冒泡的元素

5	7	3	0	4	2	1	9	6	8
---	---	---	---	---	---	---	---	---	---

待冒泡的元素

5	3	0	4	2	1	7	6	8	9
---	---	---	---	---	---	---	---	---	---

待冒泡的元素

3	0	4	2	1	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

待冒泡的元素

0	3	2	1	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

待冒泡的元素

0	2	1	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

待冒泡的元素

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

```
template <class T>
void bubbleSort (T a[], int size)
{
    for (int i = 1; i < size; ++i) {
        bool flag = false;
        for (int j = 0; j < size-i; ++j)
            if (a[j+1] < a[j]) {
                T tmp = a[j];
                a[j] = a[j+1];
                a[j+1] = tmp;
                flag = true;
            }
        if (!flag) break;
    }
}
```

# 冒泡排序的效率

## 最好情况

$n$ 次比较

0次交换

## 最坏情况

$n(n-1)/2$ 次比较和交换

**数组的每一个元素又是数组的数组称为多维数组**

**最常用的多维数组是二维数组，通常用来表示矩阵**

## **二维数组的定义格式**

类型 数组名[常量表达式1][常量表达式2]

常量表达式1表示行数，常量表达式2表示列数

## **二维数组元素表示**

数组名[行号][列号]



eg. `int a[4][5];`

## 解读一

4行5列的矩阵，矩阵的每个元素是整数

可以通过 “数组名[行号][列号] ” 访问矩阵的元素

## 解读二

定义了一个4个元素的一维数组a[0]到a[3]

a[i] 是一个5个元素组成的一维数组的名字，即二维数组的一行

由于 a[i] 是这个数组的名字，访问该数组的第 j 个元素可以表示为 a[i][j]。j 的值为 0-4

## 按行序排列

a[0][0]

a[0][1]

...

a[0][4]

a[1][0]

...

a[3][4]

## 格式

类型说明 数组名[常量表达式1] [常量表达式2]={.....};

## 给所有的元素赋初值

```
int a[3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

也可以通过花括号把每一行括起来使这种初始化方法表示得更加清晰。

```
int a[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

1	2	3	4
5	0		
0	0		

# 多维数组的初始化

## 对部分元素赋值

```
int a[3][4] = { 1, 2 , 3, 4, 5 };
```

## 为每一行的部分元素赋初值

```
int a[3][4] = { {1, 2}, {3, 4}, {5}};
```

矩阵乘法  $C=A*B$

$A[L][M]$ ,  $B[M][N]$

$C[L][N]$

处理过程

- 输入A, B
- 相乘
- 输出C

```
#define MAX_SIZE 10    //矩阵的最大规模
int main()
{
    int a[MAX_SIZE][MAX_SIZE];
    int b[MAX_SIZE][MAX_SIZE];
    int c[MAX_SIZE][MAX_SIZE];
    int i, j, k;
    int NumOfRowA, NumOfColA, NumOfColB;

    cout << "\n输入A的行数、列数和B的列数: ";
    cin >> NumOfRowA >> NumOfColA >> NumOfColB;

    //输入数组A
    cout << "\n输入数组A:\n";
    for (i=0; i< NumOfRowA; ++i)
        for (j=0; j < NumOfColA; ++j) {
            cout << "a[" << i << "][" << j << "] = ";
            cin >> a[i][j];
        }
}
```

```
//输入数组B
cout << "\n输入数组B:\n";
for (i=0; i< NumOfColA; ++i)
    for (j=0; j< NumOfColB; ++j) {
        cout << "b[" << i << "][" << j << "] = ";
        cin >> b[i][j];
    }

//执行A*B
for (i=0; i< NumOfRowA; ++i)
    for (j=0; j< NumOfColB; ++j) {
        c[i][j] = 0;
        for (k=0; k<NumOfColA; ++k)    c[i][j] += a[i][k] * b[k][j];
    }

//输出数组C
cout << "\n输出数组C:";
for (i=0; i < NumOfRowA; ++i) {
    cout << endl;
    for (j=0; j< NumOfColB; ++j)
        cout << c[i][j] << '\t';
}
return 0;
}
```

# 程序举例--打印N阶魔阵

8	1	6
3	5	7
4	9	2

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

## 填写过程

第一个元素：第一行中间一列

下一单元：行-1，列+1

如行-1，列+1有内容，则下一单元为“行+1，列不变”



# 填写魔阵的思想

```
row = 0; col = N/2;  
magic[row][col] = 1;  
for (i=2; i<=N*N; ++i) {  
    if (上一行、下一列有空)  
        设置上一行、下一列为当前位置;  
    else 设置当前列的下一行为当前位置;  
    将i 放入当前位置  
}
```

## 如何表示某个位置有空？

赋一个特殊值 0

## 如何实现回绕

找下一列  $\text{col} = (\text{col} + 1) \% \text{scale}$

找下一行  $\text{row} = (\text{row} + 1) \% \text{scale}$

找上一行  $\text{row} = (\text{row} - 1 + \text{scale}) \% \text{scale}$

```
#include <iostream>
using namespace std;
#define MAX 15      //最高位15阶
int main()
{
    int magic[ MAX ][ MAX ] = { 0 };
    int row, col, count, scale;

    // 输入阶数scale
    cout << "input scale\n";
    cin >> scale;
```

//生成魔阵

```
row=0; col = (scale - 1) / 2; magic[row][col] = 1;
for (count = 2; count <= scale * scale; count++) {
    if (magic[(row - 1 + scale) % scale][(col + 1) % scale] == 0) {
        row = ( row - 1 + scale ) % scale;
        col = ( col + 1 ) % scale;
    }
    else row = ( row + 1 ) % scale;
    magic[row][col] = count;
}
```

// 输出

```
for (row=0; row<scale; row++) {
    for (col=0; col<scale; col++)
        cout << magic[ row ][ col] << '\t';
    cout << endl;
}
return 0;
}
```

# 二维数组作为函数参数

二维数组的传递也是传递起始地址

第二个下标一定要指定，而且必须是编译时的常量

试设计两个函数，分别完成列数为5的整型二维数组的输入和输出

## 函数原型

函数参数：一个列数为5的二维数组

返回类型：void

```
void inputMatrix(int a[][5], int row)
{
    for (int i = 0; i < row; ++i) {
        cout << "\n请输入第" << i << "行的5个元素: ";
        for (int j = 0; j < 5; ++j)
            cin >> a[i][j];
    }
}
```

```
void printMatrix(int a[][5], int row)
{
    for (int i = 0; i < row; ++i) {
        cout << endl;
        for (int j = 0; j < 5; ++j)
            cout << a[i][j] << '\t' ;
    }
}
```

```
int main()
{
    int array[3][5];

    inputMatrix(array,2);
    printMatrix(array,2);
    inputMatrix(array,3);
    printMatrix(array,3);

    return 0;
}
```

由一系列字符组成的一个数据称为字符串

在C++中，字符串常量用一对双引号括起来。如" Hello,world"

## 空字符串

不包含任何字符的字符串称为空字符串，表示为 ""

空字符串占用的空间为1个字节，存储 '\0'

## C风格的字符串存储

用字符类型的数组

所需的存储空间比实际的字符串长度大1

如要将字符串" Hello,world" 保存在一个数组中，该数组的长度为12

# 字符串变量的定义及初始化

```
char ch[] = { 'H' , ' e' , ' l' , ' l' , ' o' , ' ' , ' w' , ' o' , ' r' , ' l' , ' d' , ' \0' };
```

```
char ch[] = {" Hello,world" };
```

```
char ch [] = " Hello,world" ;
```



**逐个字符的输入输出：这种做法和普通的数组操作一样**

```
char str[5];  
for (k = 0; k < 4; ++k)    cin >> str[k];  
str[4] = '\0' ;  
  
for (k = 0; a[k] != '\0' ++k)  
    cout << str[k];
```

**将整个字符串一次性地用 cin 和 cout 输入或输出**

**通过对象的成员函数函数 get 或 getline**

如定义了一个字符数组ch

输出ch的内容

```
cout << ch;
```

从 ch 的第一个字符开始输出，直到遇到' \0'

输入一个字符串放在ch中

```
cin >> ch;
```

**注意**

cin 输入是以空格、回车或Tab键作为结束符。因此无法输入包含空白字符的字符串

无法控制输入的字符串的长度不超过数组的长度

最好在输出的提示信息中告知允许的最长字符串长度

# 函数cin.get()和cin.getline()

从键盘输入一个包含任意字符的字符串

## 格式

cin.get (字符数组, 长度, 结束字符)

cin.getline (字符数组, 长度, 结束字符)

如: cin.get (s, 80, '\n' )

## 区别

cin.get 将结束字符留在输入流中, 而 cin.getline 将结束字符从输入流中删除

字符串不能直接用系统的内置运算符进行操作

C语言设计了一个cstring库，提供了一些用来处理字符串的函数

函数	作用
strcpy(dst, src)	将字符从 src拷贝到dst。函数的返回值是dst的地址
strncpy(dst, src, n)	至多从 src 拷贝n个字符到dst。函数的返回值是dst的地址
strcat(dst, src)	将 src 接到 dst 后。函数的返回值是dst的地址
strncat(dst, src, n)	从 src 至多取 n 个字符接到 dst 后。函数的返回值是dst的地址
strlen(s)	返回s的长度
strcmp(s1, s2)	比较 s1 和 s2。如 $s1 > s2$ 返回值为正数， $s1 = s2$ 返回值为0， $s1 < s2$ 返回值为负数
strncmp(s1, s2, n)	如 strcmp，但至多比较n个字符
strchr(s, ch)	返回一个指向s中第一次出现ch的地址
strrchr(s, ch)	返回一个指向s中最后一次出现ch的地址
strstr(s1, s2)	返回一个指向s1中第一次出现s2的地址

统计一组输入整数的和。输入时，整数之间用空格分开。这组整数可以是以八进制、十进制或十六进制表示。八进制以0开头，如075。十六进制以0x或0X开头，如0x1F9。其他均为十进制。输入以回车作为结束符。例如输入为：123 021 0x2F 30，输出为237，即 $123 + 17 + 47 + 30$ 。

## 关键问题

如何输入整数

当做字符串输入

如何区分一个个数字

检查空格

如何将字符串表示的不同数制的整数转换成十进制数

用循环

# 基本处理过程

输入str

跳过字符串开头的空格

```
while (str[i] != '\0') {
```

    区分基数，存入flag

```
    switch (flag) {
```

```
        case 10: 转换十进制数，存入data;        break;
```

```
        case 8:  转换八进制数，存入data ;        break;
```

```
        case 16: 转换十六进制数，存入data ;
```

```
    }
```

```
    sum += data;
```

```
    while (str[i] == ' ' ) ++i;          // 跳过空格
```

```
}
```

# 完整程序

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

int main()
{
    char str[81];
    int sum = 0, data, i = 0, flag;

    cin.getline(str,81);
    while (str[i] == ' ') ++i;

    while (str[i] != '\0') {
        if (str[i] != '0' ) flag = 10;
        else {
            if (str[i+1] == 'x' || str[i+1] == 'X') {
                flag = 16; i += 2;
            }
            else { flag = 8; ++i; }
        }
    }
}
```



```
data = 0;
switch (flag) {
    case 10: while (str[i] != ' ' && str[i] != '\0 ') data = data * 10 + str[i++] - '0';
              break;
    case 8: while (str[i] != ' ' && str[i] != '\0 ') data = data * 8 + str[i++] - '0' ;
            break;
    case 16: while (str[i] != ' ' && str[i] != '\0') {
              data = data * 16;
              if (str[i] >= 'A' && str[i] <= 'F') data += str[i++] - 'A' + 10;
              else if (str[i] >= 'a' && str[i] <= 'f') data += str[i++] - 'a' + 10;
              else data += str[i++] - '0';
            }
}
sum += data;
while (str[i] == ' ') ++i;
}
cout << sum << endl;
return 0;
}
```



**设计一函数，统计字符串中的单词数。单词和单词之间用空格分开**

## 函数原型

参数：字符数组，但不需要传长度

返回值：整型

## 解题关键

单词的数目可以由单词间的空格决定

## 解题思路

设置一个计数器num表示单词个数。开始时，num=0。

从头到尾扫描字符串。当发现当前字符为非空格，而当前字符以前的字符是空格，则表示找到了一个新的单词，num加1。

当整个字符串扫描结束后，num中的值就是单词数。

# 字符串作为函数参数

```
int count( char sentence[ ] )
{
    int i, num = 0;
    char prev = ' ';

    for (i = 0; sentence[i] != '\0'; ++i) {
        if ( prev == ' ' && sentence[i] != ' ' ) ++num;
        prev = sentence[i];
    }

    return num;
}
```

**有没有其他  
的解决方  
案？？？**

C++专门处理字符串的库。可以将字符串变量定义为string 类型

## 优点

可以用运算符实现字符串的操作

不需要关心存储空间的问题

# string类与字符数组的相同处

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    char charr1[20], charr2[20] = "C language";
    string str1, str2 = "C++ language";
    cout << "输入C风格字符串: "; cin >> charr1;
    cout << "输入C++风格字符串: "; cin >> str1;
    cout << "输出两个字符串:\n";
    cout << charr1 << " " << charr2 << " " << str1 << " " << str2 << endl;
    cout << charr2 << "中第3个字符是 " << charr2[2] << endl;
    cout << str2 << "的第3个字符是 " << str2[2] << endl;
    return 0;
}
```

# 用运算符操作string类对象

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

int main()
{
    string str1, str3, str2 = "str2";

    str1 = str2;
    cout << str1 << " " << str1.size() << endl;
    cout << (str1 == str2 ? "true" : "false") << endl;
    cin >> str1;
    getline(cin, str2);
    cout << str1 << " " << str1.size() << endl << str2 << " " << str2.size() << endl;
    str1 += str2;    cout << str1 << " " << str1.size() << endl;
    str3 = str1 + str2;    cout << str3 << " " << str3.size() << endl;

    return 0;
}
```

## 某次执行结果

```
str2  4
true
aaa bbb ccc ddd
aaa  3
bbb ccc ddd  12
aaa bbb ccc ddd  15
aaa bbb ccc ddd bbb ccc ddd  27
```

回溯法

分治法

动态规划

## 用途

找某一问题的可行解

## 算法思想

首先暂时放弃问题规模大小的限制，先从最小规模开始

将问题的候选解按某种顺序逐一枚举和检验，选择一个可行解，如果找不到可行解，则回到前一规模

如果没有达到规模要求，继续扩大当前候选解的规模，并继续试探

如果当前的候选解满足包括问题规模在内的所有要求时，该候选解就是问题的一个解

## 回溯

找不到可行解，则回到前一规模

## 向前探测

扩大当前候选解的规模，并继续试探的



## 问题

有编号为0, 1, 2, 3, 4的5本书,  
准备分给5个人A, B, C, D, E,  
每个人的阅读兴趣用一个二维数组描述:

$\text{Like}[i][j] = \text{true}$  i 喜欢书 j

$\text{Like}[i][j] = \text{false}$  i 不喜欢书 j

写一个程序, 输出一个皆大欢喜的分书方案

## like矩阵

0	0	1	1	0
1	1	0	0	1
0	1	1	0	1
0	0	0	1	0
0	1	0	0	1

## 阅读兴趣

用一个二维数组 like 存储

## 分配方案

用一个一维数组 take

$\text{take}[i] = j$  表示第  $i$  本书分给了第  $j$  个人

如果第  $i$  本书尚未被分配, 给  $\text{take}[i]$  一个特殊值, 如 -1

函数 `trynext(i)`: 在已经给 0 到  $i-1$  个人成功分数后, 给第  $i$  个人分书

依次尝试把书  $j$  分给人  $i$

如果第  $i$  个人不喜欢第  $j$  本书, 则尝试下一本书,

如果喜欢, 并且第  $j$  本书尚未分配, 则把书  $j$  分配给  $i$

如果  $i$  是最后一个人, 则方案数加1, 输出该方案, 返回true

否则调用 `trynext(i+1)` 为第  $i+1$  个人分书

如果 `trynext(i+1)` 成功, 返回true

如果 `trynext(i+1)` 失败, 让第  $i$  个人退回书  $j$ , 尝试下一个  $j$ , 即寻找下一个可行的方案

返回false

**将like, take作为全局变量, 以免每次函数调用时都要带一大串参数**

# 找一个可行方案

```
bool trynext(int i)
{
    int j, k;

    for (j=0; j<5; ++j) {
        if ( like[ i ][ j ] && take[ j ] == -1) {           // 如果 i 喜欢 j, 并且 j 未被分配
            take[ j ] = i;                                   // j 分给 i
            if (i == 4) {                                     // 找到一种新方案, 输出此方案
                cout << " 书\t人" << endl;
                for (k=0; k<5; k++)
                    cout << k << '\t' << char(take[k] + 'A') << endl;
                return true;
            }
            else if ( trynext( i+1 ) ) return true; // 为下一个人分书且成功
            else take[j] = -1;                     // 回溯, 尝试找下一方案
        }
    }
    return false;
}
```

# 找所有可行方案

```
void trynext(int i)
{
    int j, k;

    for (j=0; j<5; ++j) {
        if ( like[ i ][ j ] && take[ j ] == -1) {           // 如果 i 喜欢 j, 并且 j 未被分配
            take[j] = i;                                     // j 分给 i
            if (i == 4) {                                     // 找到一种新方案, 输出此方案
                n++;
                cout << "\n第" << n << "种方案: " << endl;
                cout << " 书\t人" << endl;
                for (k=0; k<5; k++)
                    cout << k << "\t" << char(take[k] +'A') << endl;
            }
            else trynext(i+1);                                //为下一个人分书
            take[j] = -1;    //尝试找下一方案
        }
    }
}
```

当like矩阵的值为

<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>

找所有可行方案的结果为  
找一个可行方案的结果为

第1种方案:

书	人
0	B
1	C
2	A
3	D
4	E

第2种方案:

书	人
0	B
1	E
2	A
3	D
4	C



## 问题

在一个 $8 \times 8$ 的棋盘上放8个皇后，使8个皇后中没有两个以上的皇后会在同一行、同一列或同一对角线上

## 求解思想

设计一个函数`queen(i)`，在前  $i-1$  列上的皇后都合理安置后安置第  $i$  列的皇后

依次枚举第 1 行到第 8 行，检查是否能放置皇后

如果没有合适位置，返回false

找到一个合适位置后递归调用`queen(i+1)`

如果`queen(i+1)`返回true，返回true。否则将皇后从当前位置取走，继续寻找其他合适的位置

```
bool queen (k)
{
    for (i = 1; i <= 8; ++i)
        if ( 皇后放在第 i 行是可行的 ) {
            在第i行放入皇后;
            if (k == 8) 输出解, 返回true;
            else if (queen (k+1)) 返回true;
            else 恢复该位置为空;
        }
    返回 false;
}
```



## 棋盘设计

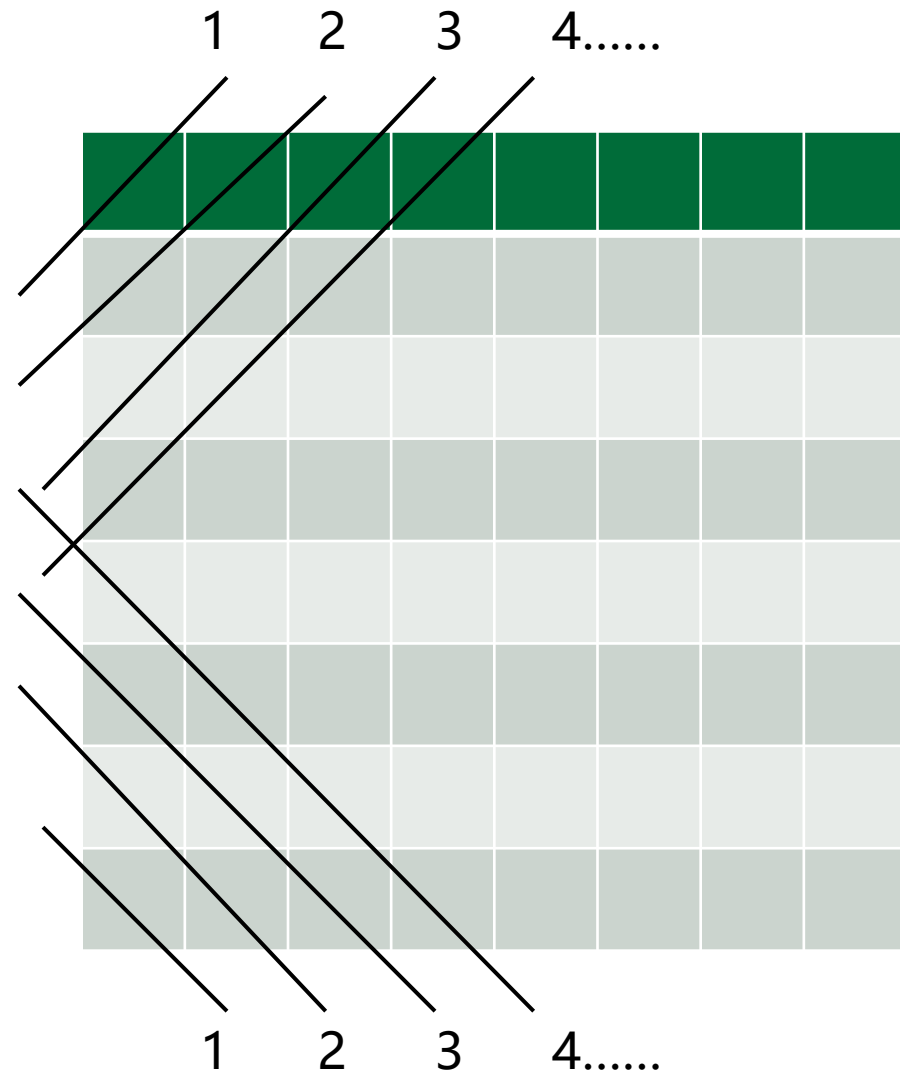
一维数组  $col[9]$ ，值  $col[j]$  表示在棋盘第  $j$  列上的皇后位置  
 $col[0]$  的初值为0。当回溯到第0列时，说明程序已求得全部解(或无解)，结束程序执行

## 如何检查合理性

$row[9]$ :  $row[A]=true$  表示第  $A$  行上没有皇后

$digLeft[16]$ :  $digLeft[A]=true$  表示第  $A$  条右高左低斜线上没有皇后；从左上角依次编到右下角(1-15)

$digRight[16]$ :  $digRight[A]=true$  表示第  $A$  条左高右低斜线上没有皇后。从左下角依次编到右上角(1-15)



$i$  行  $k$  列对应的对角线下标  
 $digLeft[k+i-1]$   
 $digRight[8+k-i]$

# 找一个可行解

```
bool queen(int k)
{
    int i, j;

    for (i = 1; i < 9; i++)
        if (row[i] && digLeft[k+i-1] && digRight[8+k-i]) {
            col[k] = i;
            row[i] = digLeft[k+i-1] = digRight[8+k-i] = false;
            if (k == 8) {
                for (j = 1; j <= 8; j++)
                    cout << j << " " << col[j] << "\t" ;
                return true ;
            }
            if (queen(k+1)) return true ;
            row[i] = digLeft[k+i-1] = digRight[8+k-i] = true;
        }
    return false;
}
```

```
int col[9];
bool row[9], digLeft[17], digRight[17];
```

```
int main()
{
    int j;

    for(j = 0; j <= 8; j++)
        row[j] = true;
    for(j = 0; j <= 16; j++)
        digLeft[j] = digRight[j] = true;
    queen (1);

    return 0;
}
```

Q							
						Q	
				Q			
							Q
	Q						
			Q				
					Q		
		Q					

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

在找到一个可行方案后，不要停止寻找，继续寻找其他可行的位置。

```
void queen_a11(int k)
{
    int i, j;

    for (i = 1; i < 9; i++)
        if (row[i] && digLeft[k+i-1] && digRight[8+k-i]) {
            col[k] = i;
            row[i] = digLeft[k+i-1] = digRight[8+k-i] = 0;
            if (k == 8) {
                cout << '\n' << ++n << "\t\t" ;
                for (j = 1; j <= 8; j++)
                    cout << j << ' ' << col[j] << '\t' ;
            }
            else queen_a11(k+1);
            row[i] = digLeft[k+i-1] = digRight[8+k-i] = 1;
        }
}
```

## 分治法

分：分成较小的可以递归解决的问题

治：从子问题的解形成原始问题的解

**分治算法通常都是高效的递归算法**

## 思路

将待排序的数据放入数组a中，数据为a[low], ... , a[high]

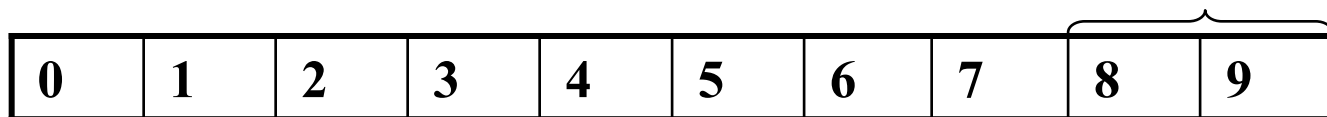
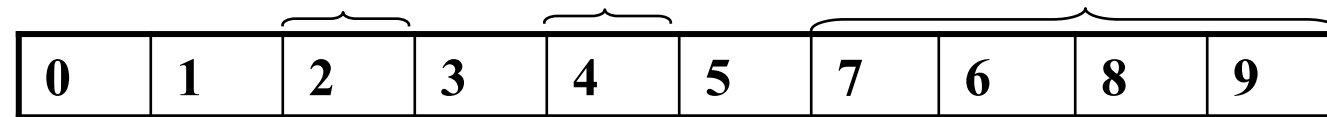
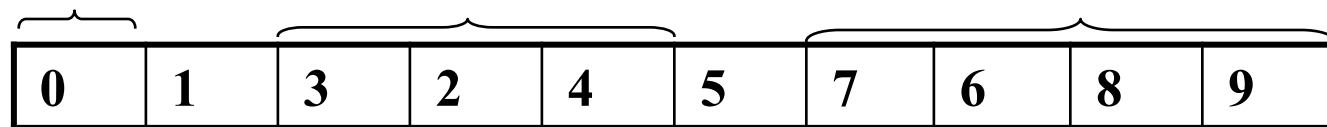
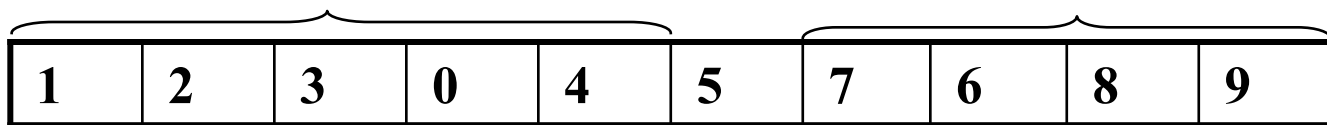
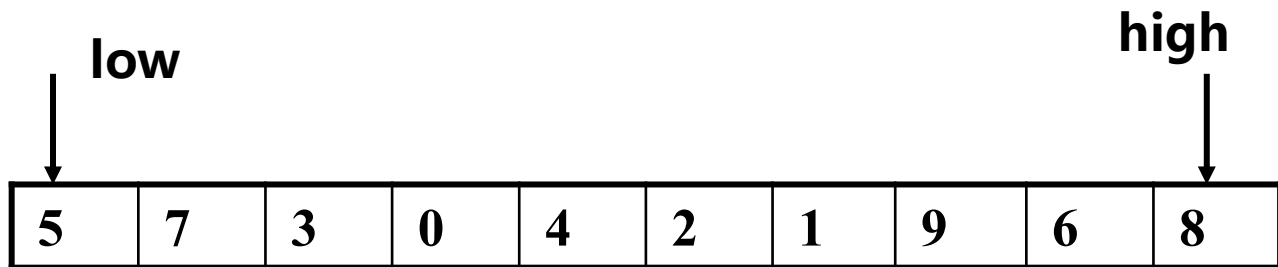
从待排序的数据中任意选择一个，如a[low]，将它放入变量k

将待排序的数据分成两组，比k小的放入数组的前一半；比k大的放入数组的后一半；将k放入中间位置

对前半一半和后半一半分别重复上述方法

## 最好时间效率

$O(n \log n)$



# 快速排序要解决的问题

## 如何选择作为分段基准的元素？

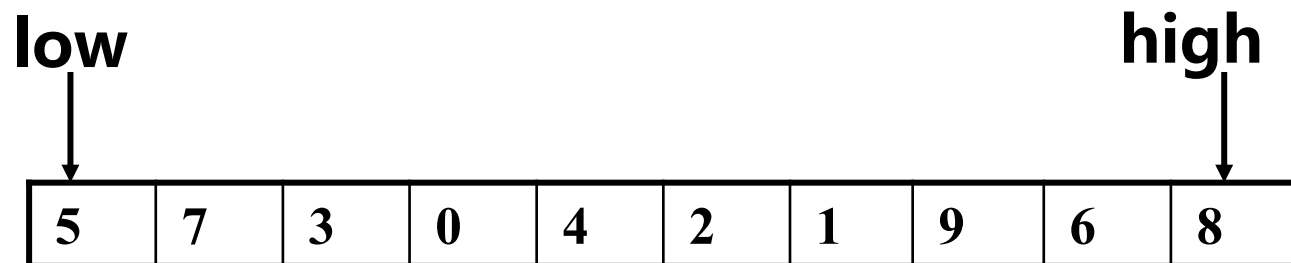
采用第一个元素

选取第一个、中间一个和最后一个中的中间元素

## 如何分段？

考虑空间问题



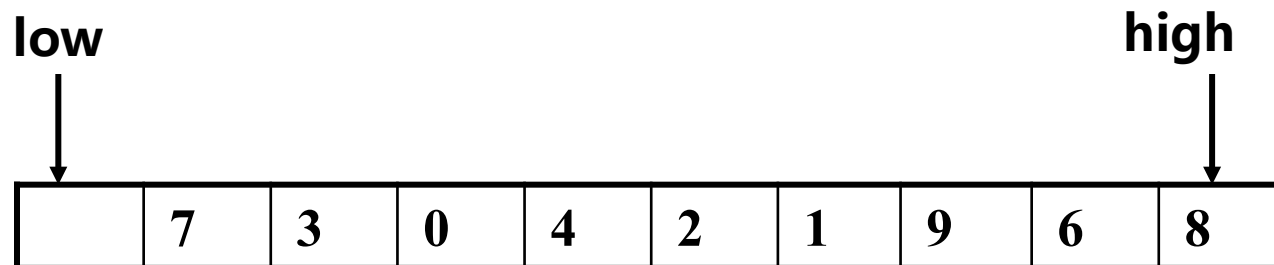


K=5

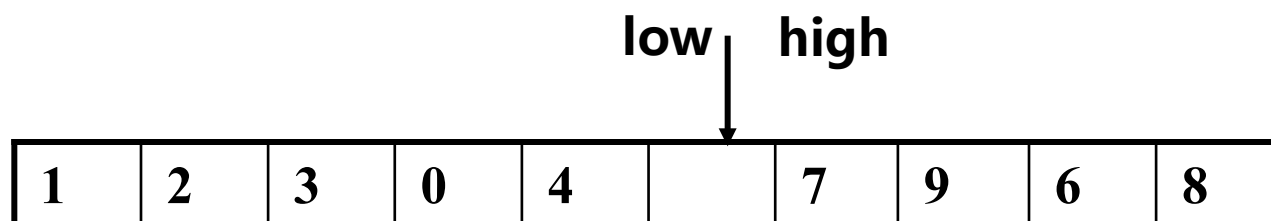
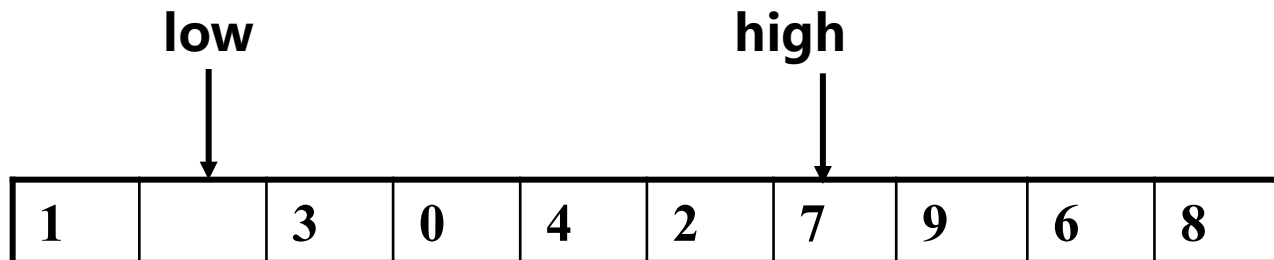
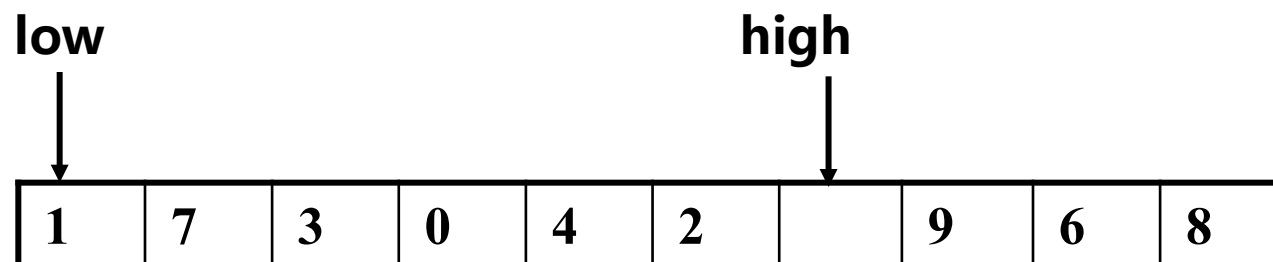
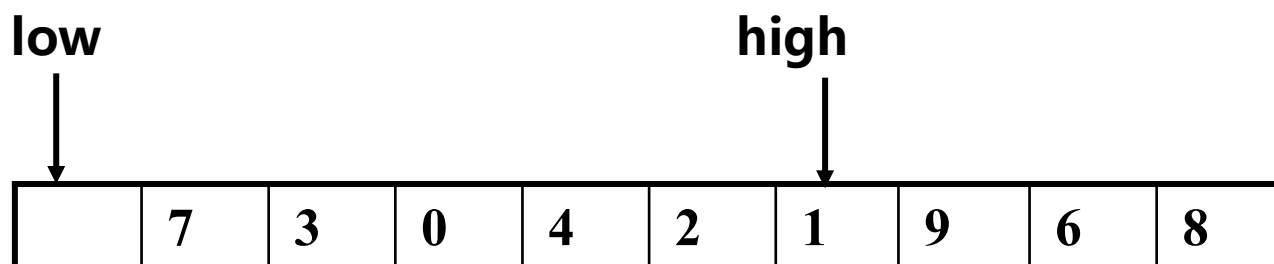
**重复执行下列过程，直到 low 等于 high**

从右向左开始检查。如果high的值大于k，high减1，继续往前检查，直到遇到一个小于k的值  
将大于k的这个值放入low的位置。然后从low位置开始从左向右检查，直到遇到一个大于k的值  
将low位置的值放入high位置

**将k放入low和high重叠的位置**



K=5



# Divide函数

```
int divide( int a[], int low, int high)
{
    int k = a[low];
    do {
        while (low < high && a[high] >= k) --high;
        if (low < high) {
            a[low] = a[high];
            ++low;
        }
        while (low < high && a[low] <= k) ++low;
        if (low < high) a[high] = a[low];
    } while (low != high);
    a[low] = k;

    return low;
}
```

# 快速排序函数

```
void quicksort(int a[], int low, int high)
{
    int mid;

    if (low >= high) return;
    mid = divide(a, low, high);
    quicksort( a, low, mid-1);
    quicksort( a, mid+1, high);
}
```

# 动态规划的思想

## 使用场合

貌似可以用分治法

但会分解出一系列重叠的子问题

如果用分治法的话会使得递归调用的次数呈指数增长

如Fibonacci数列的计算，第  $i$  个 Fibonacci 数是前两个 Fibonacci 数之和

## 动态规划

按从小到大的次序解决小问题

记录小问题的解

在解决大问题时不需要递归，只需要取出记录的小问题的解

对于一种货币，有面值为 $C_1, C_2, \dots, C_N$ (分)的硬币，最少需要多少个硬币来找出 $K$ 分钱的零钱。

## 贪婪法

不断使用可能的最大面值的硬币

如：硬币有1、5、10和25分的面值，需要找零63分钱

可以使用2个25分、一个10分的硬币以及三个1分，一共是6个硬币

## 贪婪法不一定能给出最优解

如果包含一个21分硬币时，贪心算法仍然给出一个用六个硬币的解

但是最佳的解是用三个硬币（三个都是21分的硬币）

## 枚举所有可能的分解情况，选出最优方案

如果可以用一个硬币找零，最优解为一个硬币

否则，对于每个可能的值  $i$ ，计算找  $i$  零分钱和找零  $K-i$  分钱需要的最小硬币数，选择和值最小的  $i$

## 例如，找零63分钱，有1、5、10、21和25分面值的硬币

找出1分钱零钱和62分钱零钱分别需要的硬币数是1和4。因此，63分钱需要使用五个硬币

找出2分钱和61分钱分别需要2和4个硬币，一共是六个硬币

继续尝试所有的可能性，看到一个21分和42分的分解，分别用一个和两个硬币来找开，因此，这个找零问题就可以用三个硬币解决

需要尝试的最后一种分解是31分和32分，可以用两个硬币找出31分零钱，用三个硬币找出32分零钱，一共是五个硬币

因此最小值是三个硬币

# 伪代码

```
int coin( int k )
{
    int i, tmp, int coinNum = k;

    if (能用一个硬币找零) return 1;
    for (i=1; i<k; ++i)
        if ((tmp = coin(i) + coin(k-i)) < coinNum)
            coinNum = tmp;
    return coinNum;
}
```

找零63

1     62

2     61

3     60

4     59

.....

62     1

找零62

1     61

2     60

3     59

4     58

.....

61     1

大量重复计算，效率低



## 尝试指定其中的一个硬币来递归地简化问题

```
int coin( int k )  
{  
    int i, tmp, int coinNum = k;  
  
    if (能用一个硬币找零) return 1;  
    for (每一个硬币)  
        if ((tmp = 1+ coin(k-i硬币值) < coinNum)  
            coinNum = tmp;  
    return coinNum;  
}
```

找零63

1     62

5     58

10   53

21   42

25   38

找零62

1     61

5     57

10   52

21   41

25   37

依然有大量重复计算

# 动态规划解

## 解决方案：动态规划

从小到大解决问题，将子问题的答案存放起来，当再次遇到此子问题时就不用重复计算了

## 存放子问题的解

数组 `coinsUsed[i]`：代表了找  $i$  分零钱所需的最小硬币数

## 算法

先找出0分钱的找零方法，把最小硬币数存入 `coinUsed[0]`

依次找出1分钱、2分钱...的找零方法，直到到达要找零的钱为止

对每个要找的零钱  $i$ ，把  $i$  分解成某个 `coins[j]` 和  $i - \text{coins}[j]$

所需硬币数为 `coinUsed[i-coins[j]]+1`

对所有的  $j$ ，取最小的 `coinUsed[i - coins[j]] + 1` 作为  $i$  分钱找零的的答案

```
void makechange( int coins[], int differentCoins, int maxChange, int coinUsed [] )
```

coins存放所有不同的硬币值，不同的硬币个数为 differentCoins。

maxChange为 要找的零钱数

```
void makechange( int  coins[ ],  int  differentCoins, int  maxChange, int  coinUsed[] )
{
    coinUsed[0] = 0;
    for (int cents = 1; cents <= maxChange; cents++) {
        int minCoins = cents;
        for (int j = 1; j < differentCoins; j++) {
            if (coins[j] > cents) continue;
            if (coinUsed[ cents - coins[j] ] + 1 < minCoins)
                minCoins = coinUsed[ cents - coins[ j] ] + 1;
        }
        coinUsed[cents] = minCoins;
    }
}
```

数组通常用来存储具有同一数据类型并且按顺序排列的一系列数据

数组中的每一个值称作元素，通常用下标值表示它在数组中的位置

数组中的元素用数组名后加用方括号括起来的下标表示

在C++中，下标是从0开始的

定义一个数组时，必须定义数组的大小，而且它必须是常量

数组元素通常是连续存储的。第一个元素的地址称为基地址

数组的下标可以是任意的计算结果可以自动转换成整型数的表达式

数组可以是多维的。多维数组可以看成数组的数组

**指针概念及指针变量的定义**

**指针运算**

**指针与数组**

**动态内存分配**

**字符串与指针**

**const与指针**

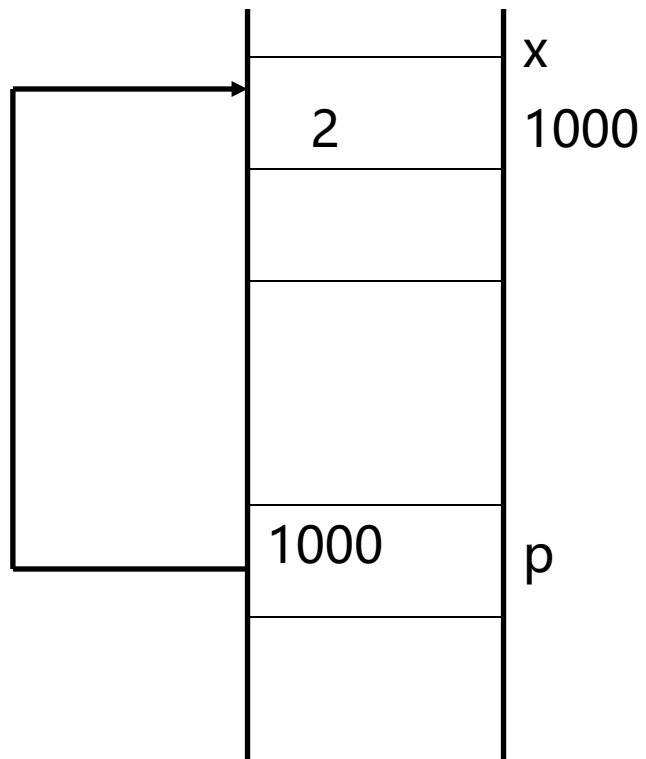
## 指针

地址作为数据处理

## 用途

提供间接访问

实现动态内存分配



## 指针变量定义

该变量中存放的是一个地址

指针变量的主要用途是提供间接访问，因此也需要知道指针指向的单元的数据类型

## 指针变量的定义格式

类型标识符 \*指针变量;

如: `int *intp;`

`double *doublep;`

`int *p, x, *q;`

## 统配指针 `void *`

可以指向任何类型



**不指向任何地址**

**空指针表示**

用符号常量NULL，NULL值为0;

用符号常量nullptr：C++11的风格

**两种表示方法的区别**

NULL的值是整数0，与函数重载配合时可能会出问题。如有两函数

```
void f(int *);
```

```
void f(int);
```

f(NULL)会调用f(int)

nullptr 不存在到整型的隐式转换

## 赋值

### 让指针指向某一变量

取地址运算符 “&”：如表达式 “&x” 返回的是变量 x 的地址。如：intp = &x;

& 运算符后面不能跟常量或表达式。如 &2 是没有意义的，&(m \* n + p)。也是没有意义的

### 同类指针互相赋值

## 间接访问

解引用运算符 “\*”：\*intp 表示的是 intp 指向的这个单元。如：\*intp = 5 等价于 x = 5

**在对 intp 使用引用运算之前，必须先对 intp 赋值**

# 指针实例

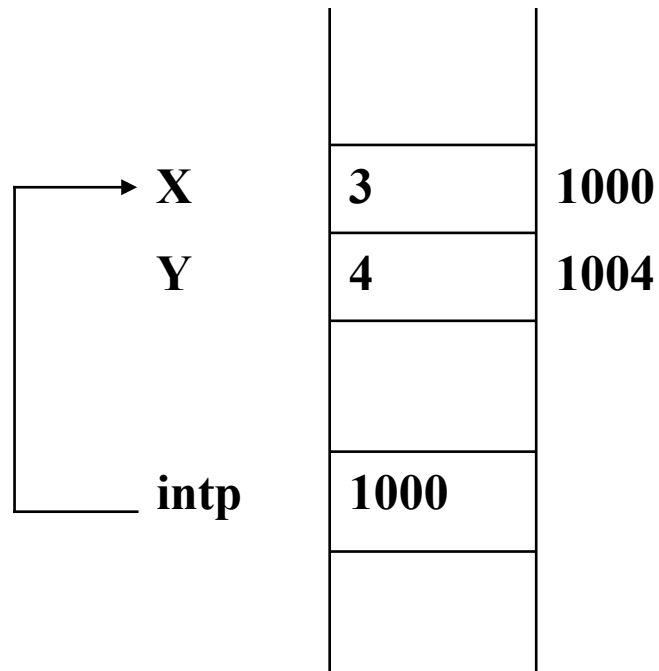
如有：

```
int X, *intp, Y;
```

```
X=3;
```

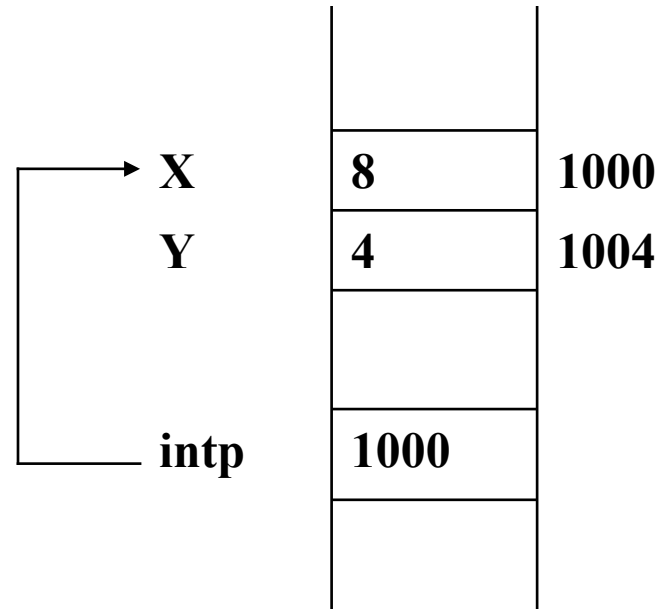
```
Y=4;
```

```
intp=&X;
```



如执行：

```
*intp=Y+4;
```



注意：不能用 `intp=100;`

因为我们不知道存储变量的真实地址

# 指针变量的使用

设有定义

```
int x, y;
int *p1,*p2;
```

执行语句：

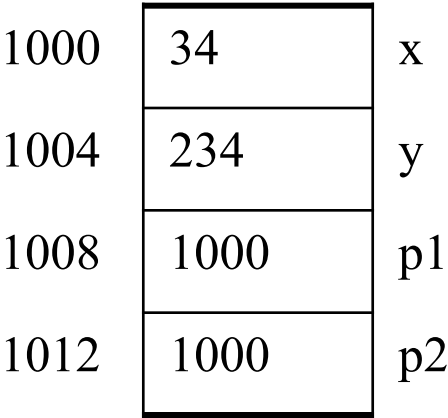
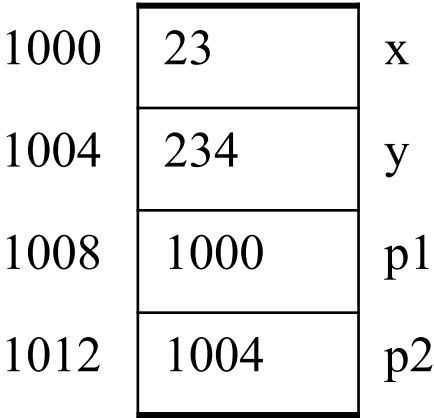
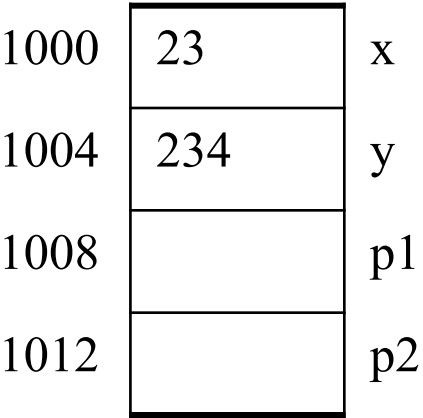
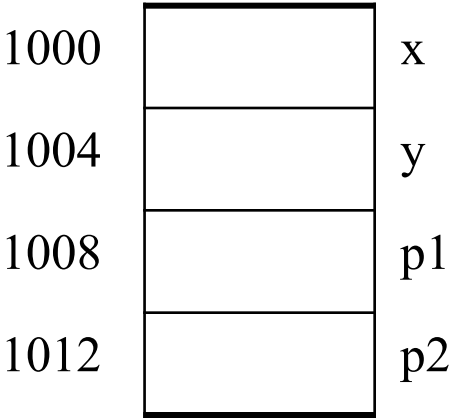
```
x=23;
y=234;
```

执行语句：

```
p1=&x;
p2=&y;
```

执行语句：

```
*p1=34;
p2=p1;
```



## □ 指针只能指向同类变量

- `int x, *p = &x; // 正确`
- `float *fp = &x; // 错误`

危险!!!

## □ 作用

- 重新解释内存中的二进制比特串
- 如将 `int` 的 32 位二进制数解释成 `float` 类型的指针
- `float *fp = reinterpret_cast < float * > &x; //正确`

# 指向数组元素的指针

## 指向数组元素的指针

$p = \&a[1], p = \&a[i]$

## 数组元素的地址是通过数组首地址计算的

如数组的首地址是 1000, 则第  $i$  个元素的地址是  $1000 + i * \text{每个数组元素所占的空间长度}$

## 数组名是常量指针

如有 `int array[10], *p;`

可以执行 `p = array;`



# 指针与数组

**一旦执行了`p=array`，则`p`与`array`是等价的**

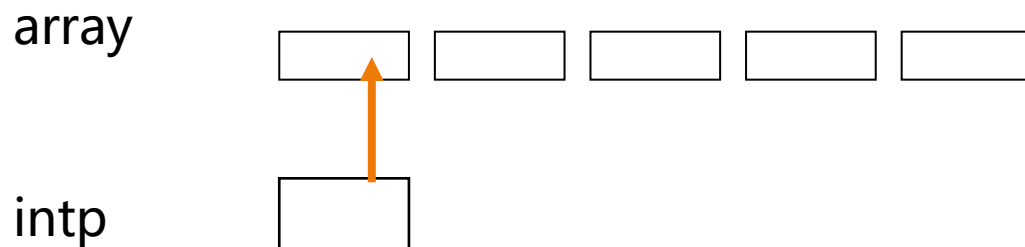
对该指针可以进行任何有关数组下标的操作

访问`array`的10个元素，可用下列循环

```
for ( i=0; i<10; ++i )
```

```
    cout << p[i];
```

**注意，数组和指针是完全不同的！**



当执行了  
`intp = array`  
后

# 指针的算术运算

**指针保存的是一个地址，地址是一个整型数，因此可以进行各种算术运算**

指针只能做加减运算

指针加建是加减一个基类型的长度

仅当指针指向数组时，加减运算才有意义

## 指针运算的意义

指针+1表示数组中指针指向元素的下一元素地址；

指针-1表示数组中指针指向元素的上一元素地址；

合法的指针操作： $p + k$ ,  $p - k$ ,  $p1 - p2$



# 数组元素的指针表示

## 数组名是指针

array[k]地址是  $\text{array} + k$

array[k]等价于  $\text{*(array} + k)$

## 执行 `intp = array` 后

array[k]地址是  $\text{intp} + k$

array[k]等价于  $\text{*(intp} + k)$

array[k]等价于 `intp[k]`

# 访问数组a的10个元素

## 方法1:

```
for ( i=0; i<10; ++i )  
    cout << a[i];
```

## 方法3:

```
for ( p=a; p < a+10; ++p )  
    cout << *p ;
```

## 方法5:

```
for ( p=a, i=0; i<10; ++i )  
    cout << p[i] ;
```

## 方法2:

```
for ( i=0; i<10; ++i )  
    cout << *(a+i);
```

## 方法4:

```
for ( p=a, i=0; i<10; ++i )  
    cout << *(p+i);
```

## 下列程序段 有无问题?

```
for ( i=0; i<10; ++i ) {  
    cout << *a ;  
    ++a;  
}
```

程序运行过程中申请变量，可以是简单变量或一维数组

## 实现方法

定义一个指针

申请一块内存，地址存入指针

通过指针间接访问动态申请的内存

如：

```
int *scores;  
scores = 内存的起始地址;
```

# 动态内存分配与回收

## 申请动态变量

申请动态变量: `p = new type;`

申请动态数组: `p = new type[size];`

申请动态变量并初始化: `p = new type(初值);`

申请动态数组并初始化: `p = new int[5]{1,2,3,4,5};`

## 释放动态变量的空间

动态变量的空间必须由程序释放

释放动态变量: `delete p;`

释放动态数组: `delete [] p;`

字符数组可以不加方括号

## 内存泄漏

没有释放动态变量的空间

# 动态内存分配与回收

```
int main()
{
    int *p;
    char *q;

    p = new int(99);      //动态分配内存，并将99作为初始化值赋给它
    q = new char[10];
    strcpy(q, "abcde");
    cout << *p << q << endl;
    cout << p << '\t' << (void *) q << endl;
    cout << &p << '\t' << &q << endl;
    delete p;
    delete q;

    return 0;
}
```

输出结果:

99abcde

005879D0      0045EF08

0042F8A0      0042F8A4

# 动态分配是否成功

new 操作的结果是申请到的空间的地址

当系统空间用完时，new 操作可能失败

new 操作失败时，返回空指针

new操作后最好检查一下是否成功

# 检查动态内存分配是否成功

```
int main()
{
    int *p;

    p = new int;
    if ( !p ) {
        cout << "allocation failure\n";
        return 1;
    }
    *p = 20;
    cout << *p;
    delete p;

    return 0;
}
```

assert ( ) 宏在标准头文件cassert中

## 格式

assert( 逻辑表达式 )

## 作用

如果表达式为假，则在发出一个错误消息后程序会终止

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

```
int main()
{
    int *p;

    p = new int;
    assert (p != 0);
    *p=20;
    cout << *p;
    delete p;

    return 0;
}
```



# 内存分配的进一步介绍

OS
Program
Heap
Stack
Globe variables

动态分配

自动分配

静态分配

## 堆

动态变量

## 栈

自动变量

函数被调用时，空间被分配；函数执行结束后，空间被释放

## 全局变量区

全局变量和静态变量

这些空间在整个程序运行期间都存在

## 问题

设计一个计算某次考试成绩的均值和均方差程序

程序运行时，先输入学生数，然后输入每位学生的成绩，最后程序给出均值和均方差

## 解决方案一

开设一个足够大的数组，每次运行时只使用一部分

缺点：浪费空间

## 解决方案二

用动态内存分配根据输入的数据量申请一个动态数组

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    int *score, num, i;
    double average = 0, variance = 0;

    cout << "请输入参加考试的人数: ";    cin >> num;

    score = new int[num];
    cout << "请输入成绩: \n";
    for (i = 0; i < num; ++i)    cin >> score[i];
    for (i = 0; i < num; ++i)    average += score[i];
    average = average / num;
    for (i = 0; i < num; ++i)    variance += (average - score[i]) * (average - score[i]);
    variance = sqrt(variance) / num;

    cout << "平均分是: " << average << "\n均方差是: " << variance << endl;
    delete [ ] score;

    return 0;
}
```

## 字符串的常用表示是指向字符的指针

### 用法

```
char *String;  
String = "abcde" ;
```

```
char *String, ss[ ] = "abcdef" ;  
String = ss;
```

```
char *String;  
String = new char[10];  
strcpy(String, "abc" );
```

# String = "abcde"

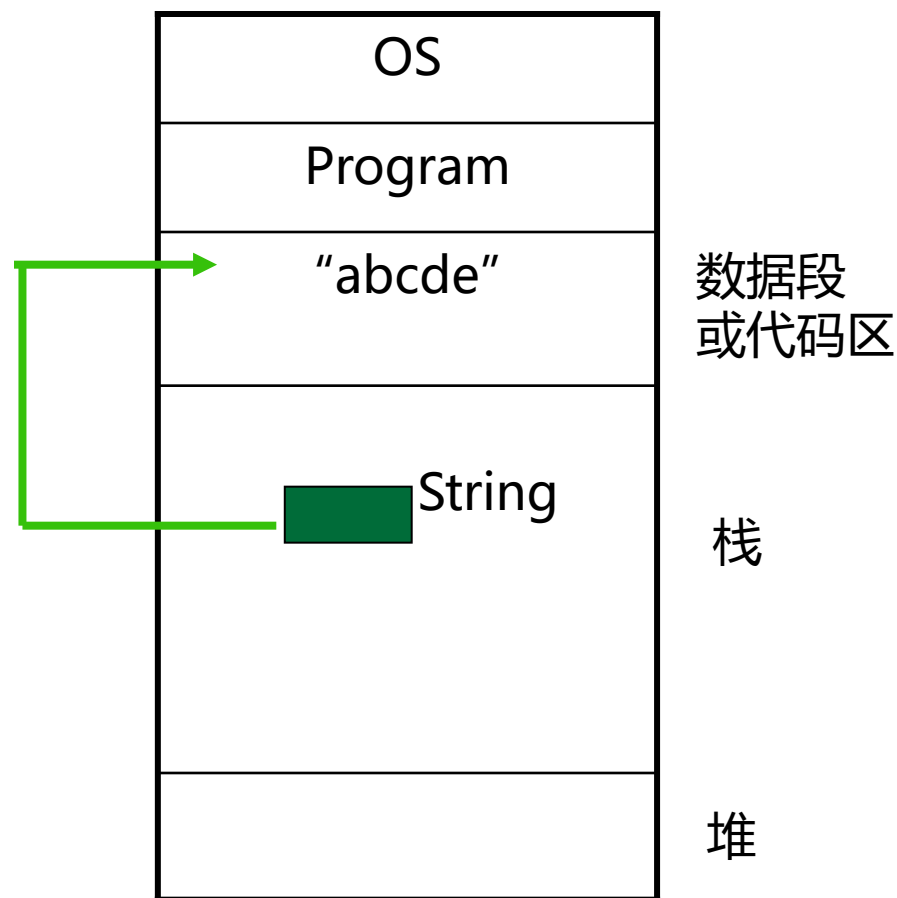
字符串常量存储在内存中称为数据段的区域里  
将存储字符串" abcde" 的内存的首地址赋给  
指针变量String

## 注意

不能将string作为strcpy或strcat的第一个参数

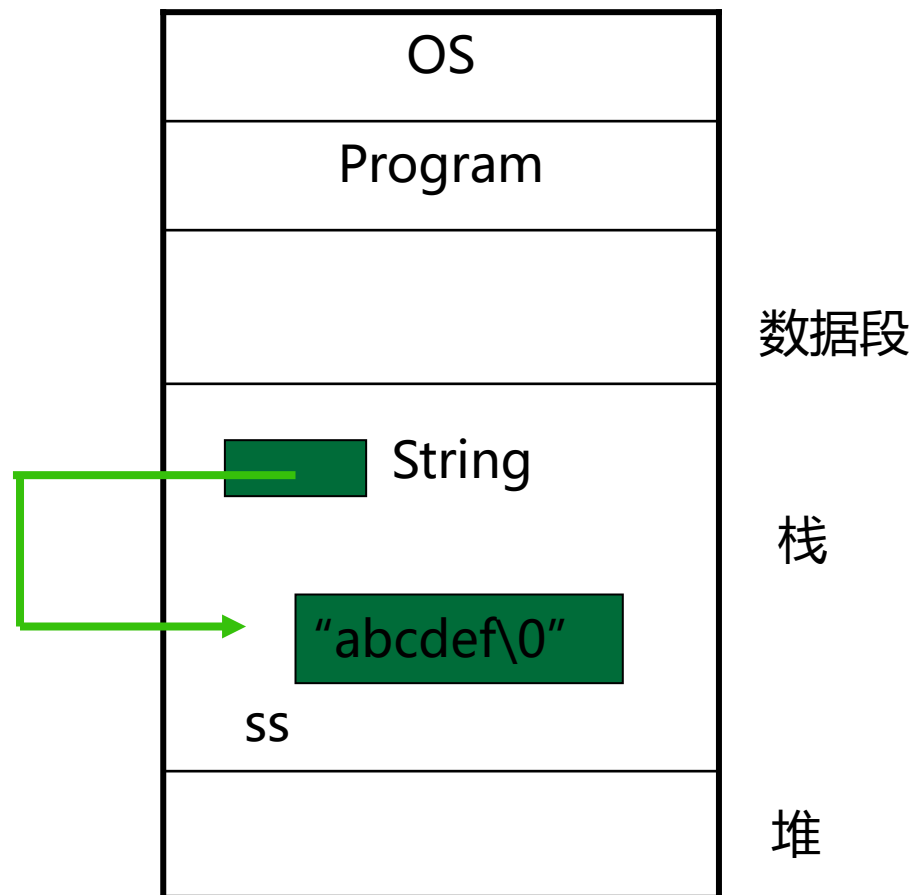
可以通过下标读取字符串中的字符。如string[3]的值是' d'

不可以通过下标变量赋值。如，string[3] = 'w' 是错误的



```
char *String, ss[ ] = "abcdef" ;  
String = ss
```

将字符数组ss的起始地址存入String

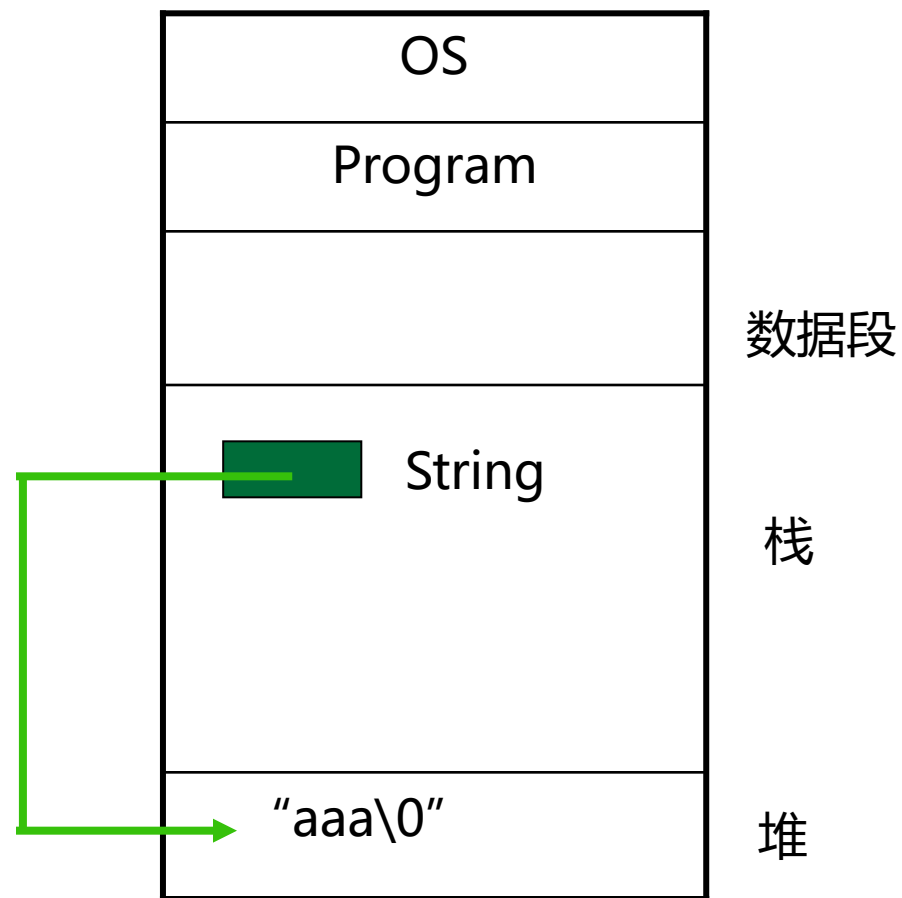


```
String = new char[5];  
strcpy(String, "aaa" )
```

string在栈工作区

字符串存储在堆工作区

string记录了字符串"aaa" 的内存的首地址



## 问题

编写一个统计字符串中单词的个数的函数

## 关键技术

如何传递一个字符串

## 字符串传递

作为字符数组传递

作为指向字符的指针传递

**字符串作为字符数组传递时不需要指定长度。因为字符串操作的结束是依据 ‘\0’**



```
#include <ctype>
using namespace std;
int word_cnt( const char *s )
{
    int cnt = 0;

    while (*s != '\0') {
        while ( isspace(*s) ) ++s;    //跳过空白字符
        if (*s != '\0') {
            ++cnt;                    //找到一个单词
            while ( !isspace(*s) && *s != '\0 ' ) ++s;    //跳过单词
        }
    }

    return cnt;
}
```

**const int \*p;**

不能通过p修改它指向的单元的内容，但p可以指向不同的变量。

**int \*const p = &x;**

可以通过p修改指向的对象，但p的值不能变  
即p永远只能指向x。p定义时必须给出初值

**const int \*const p = &x;**

p的值不能修改，\*p的值也不能修改

**提高安全性**

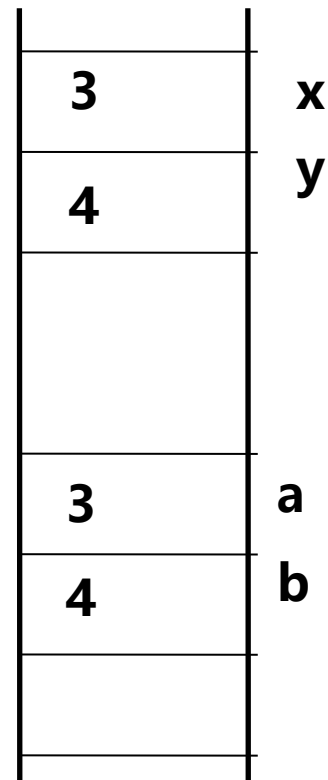
# 指针作为函数参数

例：编一函数，交换二个参数值

新手可能会编出如下的函数：

```
void swap( int a, int b )  
{  
    int c;  
    c=a;  a=b;  b=c;  
}
```

希望通过调用swap( x, y)交换变量 x 和 y 的值



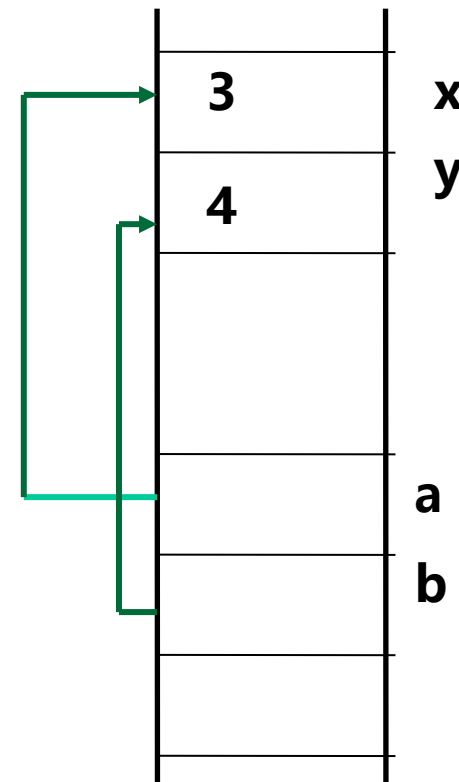
因为C++采用的是值传递机制，函数中a、b值的交换不会影响实际参数x和y的值

# 正确的方法

```
void swap(int *a, int *b)
{
    int c;
    c=*a;  *a= *b;   *b=c;
}
```

交换x和y的值，可以调用swap(&x, &y)

用指针作为参数可以在函数中修改主调程序的变量值，即实现变量传递。必须小心使用！！！！



# 解一元二次方程的函数

## 问题

如何让此函数返回二个根

## 解决方案

采用指针作为函数的参数

由调用程序准备好存放两个根的变量，将变量地址传给函数，函数将两个根的值分别放入这两个地址

## 函数原型

`void SolveQuadratic(double a, double b, double c, double *px1, double *px2)`

## 函数调用

`SolveQuadratic(1.3, 4.5, 2.1, &x1, &x2)`

`SolveQuadratic(a, b, c, &x1, &x2)`

## 两类函数参数

输入参数：用值传递

输出参数：用指针传递

在参数表中，输入参数放在前面，输出参数放在后面

## 如何获知方程有根、没根？

并不是每个一元二次方程都有两个不同根，有的可能有两个等根，有的可能没有根。函数的调用者如何知道  $x_1$  和  $x_2$  中包含的是否是有效的解？

## 解决方案

让函数返回一个整型数。该整型数表示解的情况

0：两个解

1：一个解

2：无解

3：不是一元二次方程



# 完整的函数

```
int SolveQuadratic (double a, double b, double c, double *px1, double *px2)
{
    double disc, sqrtDisc;

    if(a == 0) return 3;      //不是一元二次方程

    disc = b * b - 4 * a * c;
    if( disc < 0 ) return 2;   //无根

    if ( disc == 0 ) { *px1 = -b / (2 * a); return 1;} //等根

    //两个不等根
    sqrtDisc = sqrt(disc);
    *px1 = (-b + sqrtDisc) / (2 * a);
    *px2 = (-b - sqrtDisc) / (2 * a);

    return 0;
}
```





# 函数的调用

```
int main()
{
    double a,b,c,x1,x2;
    int result;

    cout << "请输入a,b,c: ";
    cin >> a >> b >> c;

    result = SolveQuadratic(a, b, c, &x1, &x2);
    switch (result)    {
        case 0: cout << "方程有两个不同的根: x1 = " << x1 << " x2 = " << x2; break;
        case 1: cout << "方程有两个等根: " << x1; break;
        case 2: cout << "方程无根"; break;
        case 3: cout << "不是一元二次方程";
    }

    return 0;
}
```

# 数组传递的进一步讨论

## 数组传递的本质是地址传递

形参和实参可以使用数组名，也可以使用指针

数组传递是函数原型可写为：

```
type fun(type a[], int size);
```

也可写为

```
type fun(type *p, int size);
```

但在函数内部，a 和 p 都能当作数组使用

调用时，对这两种形式都可用数组名或指针作为实参

## 建议

如果传递的是数组，用第一种形式；如果传递的是普通的指针，用第二种形式

# 验证数组传递本质是指针传递

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

void f(int arr[ ], int k)
{
    cout << sizeof(arr) << "  " << sizeof(k) << endl;
}

void main()
{
    int a[10]={1,2,3,4,5,6,7,8,9,0};

    cout << sizeof(a) << endl;
    f(a,10);
}
```

**输出:**

```
4 0
4   4
```

即在main中，a是数组，占用了40个字节。而在函数f中，arr是一个指针

# 数组传递的灵活性

```
void sort(int p[ ], int n)
{ ... }
```

```
int main()
{
    int a[100];
    ...
    sort(a, 100); //排序整个数组
    sort(a, 50); //排序数组的前50个元素
    sort(a+50, 50); //排序数组的后50个元素
    ...
}
```

**函数的返回值可以是一个指针**

**返回指针的函数原型**

类型 \*函数名 (形式参数表) ;

**当函数的返回值是指针时，返回地址对应的变量不能是函数的自动局部变量**

## 问题

设计一个函数从一个字符串中取出一个子串

## 原型设计

参数：从哪一个字符串中取子串、起点和终点

返回值：取出的子串

字符串可以用一个指向字符的指针表示，所以函数的返回值是一个指向字符的指针

## 返回值指针指向的空间必须在返回后还存在

可以用动态字符数组

```
char *subString( const char *s, int start, int end )
{
    int len = strlen( s );
    if ( start < 0 || start >= len || end < 0 || end >= len || start > end ) {
        cout << "起始或终止位置错" << endl;
        return NULL;
    }
    char *sub = new char[end - start + 2];
    strncpy(sub, s + start, end - start + 1);

    return sub;
}
```

### 调用subString的函数必须释放空间

如：

```
char *pc = subString(str, 3, 10);
cout << pc << endl;
delete pc;
```

## 引用的意义

给某个变量取一个别名,使一个内存单元可以通过不同的变量名来访问

## 定义格式

类型 &变量名 = 变量名;

例: `int i;`

`int &j = i;`

j 是 i 的别名, i 与 j 是同一个内存单元。这个绑定关系在 j 的生命周期中不能改变

## 引入引用的主要目的是将引用作为函数的参数



## 代替指针传递

### 指针参数

```
void swap(int *m, int *n)
{
    int temp;
    temp=*m;
    *m=*n;
    *n=temp;
}
调用: swap(&x, &y)
```

### 引用参数

```
void swap(int &m, int &n)
{
    int temp;
    temp=m;
    m=n;
    n=temp;
}
调用: swap( x, y)
```

相当于发生了变量定义

`int &m = x`

`int &n = y`

**注意：实参必须是变量，而不能是一个表达式或常量**

```
void f( int & r )
{
    cout <<  "r= " << r <<endl;
    cout <<  "&r= " << &r <<endl;
    r= 5;
    cout <<  "r= " << r <<endl;
}

int main( )
{
    int x=47;
    cout <<  "x= " << x <<endl;
    cout <<  "&x= " << &x <<endl;
    f(x);
    cout <<  "x= " << x <<endl;
}
```

## 执行结果

```
x = 47
&x = 0065FE00
r = 47
&r = 0065FE00
r=5
x = 5
```

**在C++中，函数参数一般都采用引用传递**

## **引用传递的作用**

减少函数调用时的开销

作为输出参数

函数调用是return后面的变量的别名

## 用途

将函数用于赋值运算符的左边，即作为左值  
减少函数返回时的开销

## 实例

```
int a[] = {1, 3, 5, 7, 9};  
int &index(int); //声明返回引用的函数  
void main()  
{  
    index(2) = 25; //将a[2]重新赋值为25  
    cout << index(2);  
}  
int &index(int j)  
{ return a[j]; } //函数是a[j]的一个引用
```

## 定义格式

`const 类型 &变量名1 = 变量名2;`

例: `int i;`

`const int &j = i;`

## 常量引用的意义

控制变量的修改，不管被引用对象是常量还是变量，用此名字是不能赋值

如 `i = 5` 是合法的，`j = 5` 是非法的

**const引用的初值可以是常量或表达式。如**

`const int &y = 2+5;`

用来代替值传递

```
void f(const double &);
```

实际参数可以是常量，也可以是变量或表达式

实际参数是变量时，形式参数是实际参数的引用

实际参数是常量时，函数中会创建一个临时变量

# 返回常量引用的函数

## 用途

仅减少函数返回时的开销，保证返回变量不被修改

## 实例

```
int a[] = {1, 3, 5, 7, 9};  
const int &index(int); //声明返回引用的函数  
void main()  
{  
    index(2) = 25;    //错  
    cout << index(2); // 正确  
}  
const int &index(int j)  
{ return a[j]; }    //函数是a[j]的一个常量引用
```

- ❑ 修改一些指针/引用的const权限
- ❑ 将 `const type *` 转换为 `type *`, 将 `const type &` 转换为 `type &`
- ❑ 可以通过指针或引用修改常量值



# 修改const的结果是不确定的

```
int main()
{
    const int con = 5;
    int *p = const_cast<int *> (&con);
    int &cc = const_cast<int &> (con);

    ++(*p);
    cout << *p << endl;
    cout << con << endl;

    ++cc;
    cout << cc << endl;
    cout << *p << endl;
    cout << con << endl;

    cout << &cc << endl;
    cout << p << endl;
    cout << &con << endl;

    return 0;
}
```

输出:

6  
5  
7  
7  
5  
0018FF44  
0018FF44  
0018FF44

- 调用了一个形式参数不是const的函数，而实际参数是const
- 但是这个函数是不会修改形式参数
- 于是就需要使用const\_cast去除const限定，以便函数能够接受这个实际参数

# 右值引用

**右值是指表达式结束后就不再存在的临时对象，该对象被使用后就消亡了**  
如

$x = y + z;$

x、y、z都是左值，而 $x+y$ 的结果则是右值

**右值引用接管所引用的对象的资源。用&& 来表示**

例如

`int &&a = 10;`            合法

`int x = 10;`

`int &&y = x+9;`            合法

`int &&z = x;`            非法，是左值引用

**好处**

节省时间

指针数组

多级指针

动态二维数组

指向函数的指针

## 指针数组

每个元素均为指针

## 一维指针数组的定义形式

类型名 \*数组名[数组长度];

例如,

```
char *String[10];
```

定义了一个名为String的指针数组, 该数组有10个元素

数组的每个元素是一个指向字符的指针

## 指向字符的指针数组

用来存储一组字符串

## 实例

写一个函数用二分法查找某一个字符串

用递归实现

## 关键问题

一组字符串的存储：用指向字符的指针数组

查找时的比较：用字符串比较函数

```
//该函数用二分查找在table中查找name是否出现
//lh和rh表示查找范围，返回出现的位置
int binarySearch( const char *table[], int lh, int rh, char *name)
{
    int mid, result;

    if (lh <= rh) {
        mid = (lh + rh) / 2;
        result= strcmp(table[mid], name);
        if (result == 0) return mid;    //找到
        else if (result > 0)
            return binarySearch(table, lh, mid - 1, name);
        else return binarySearch(table, mid + 1, rh, name);
    }

    return -1; //没有找到
}
```

# 函数的应用

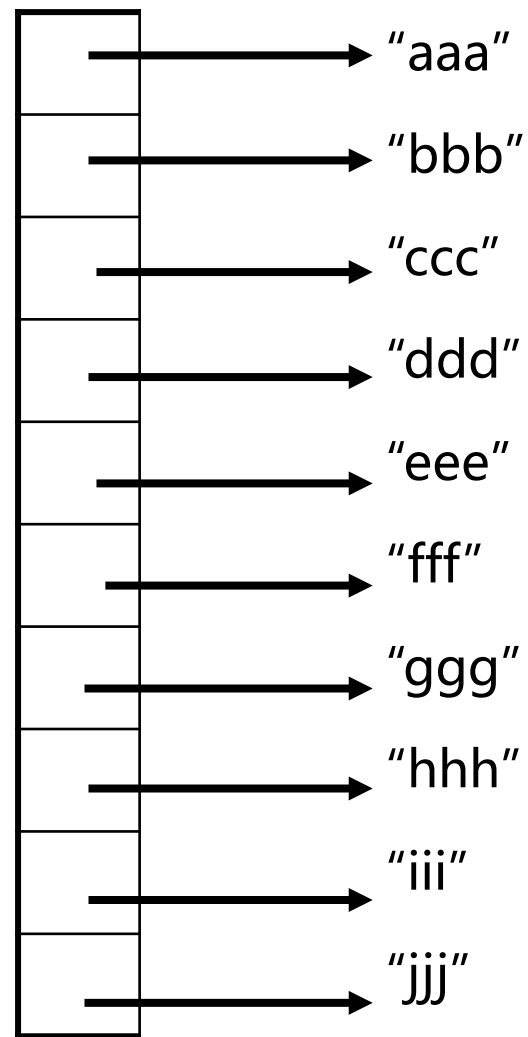
```
#include <iostream>
using namespace std;
int binarySearch(char *table[ ], int lh, int rh, char *name);

int main()
{
    char *string[10] = {"aaa", "bbb", "ccc", "ddd", "eee", "fff", "ggg", "hhh", "iii", "jjj"};
    char tmp[10];

    while (cin >> tmp)
        cout << binarySearch(string, 0, 9, tmp) << endl;

    return 0;
}
```

string





## 命令行参数

如 `cp filea fileb`  
`dir C:\`

## 如何将参数传递给程序？

main函数的参数

## main函数有二个形式参数

`int argc` : 参数的数目（包括命令名本身）

`char *argv[]`: 指向每个参数的指针，是一个指向字符串的指针数组

# 把参数传递给main()

```
int main(int argc, char *argv[])
{
    int i;

    cout << "argc= " << argc << endl;
    for(i=0; i<argc; ++i)
        cout << "argv[ " << i << "]= "
            << argv[i] << endl;

    return 0;
}
```

假设生成的执行文件myprogram.exe

**在命令行输入： myprogram**

输出结果： argc=1

argv[0]=myprogram

**在命令行输入： myprogram try this**

输出结果： argc=3

argv[0]=myprogram

argv[1]=try

argv[2]=this

## 问题

编写一个求任意  $n$  个正整数的平均数的程序

## 要求

如果该程序对应的可执行文件名为aveg，则可以在命令行中输入

```
aveg 10 30 50 20 40 ↵
```

表示求10、30、50、20和40的平均值，对应的输出为30

## 将这些数据作为命令行的参数

从argc得到数据的个数

从argv得到每一个数值，但注意数值是以字符串表示，要进行计算，必须把它转换成真正的数值

## 设计一个字符串到整数的函数

# 字符串形式的数字转换到真正的数值

```
int ConvertStringToInt( char *s )
{
    int num = 0;

    while(*s) {
        num = num * 10 + *s - '0';
        ++s;
    }

    return num;
}
```

```
int main( int argc, char *argv[] )  
{  
    int sum = 0;  
  
    for (int i = 1; i < argc; ++i)  
        sum += ConvertStringToInt(argv[i]);  
  
    cout << sum / (argc - 1) << endl;  
  
    return 0;  
}
```

如有定义： `char *string[10];` `string`的值是什么类型？

`string`是指向`string[0]`的指针

`string[0]`的值是一个指针

`string`是指向一个指针的指针

## 多级指针

指向的内容是指针的指针

# 多级指针的定义

## 两级指针

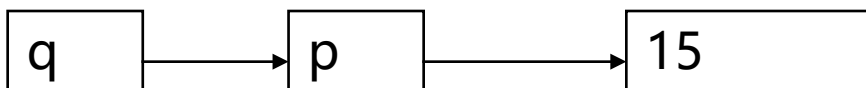
类型名 \*\*变量名;

## 三级指针

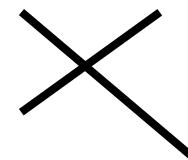
类型名 \*\*\*变量名;

如: `int **q;`

表示q指向的内容是一个指向整型的指针。可以这样使用: `int x=15, *p=&x;      q = &p;`

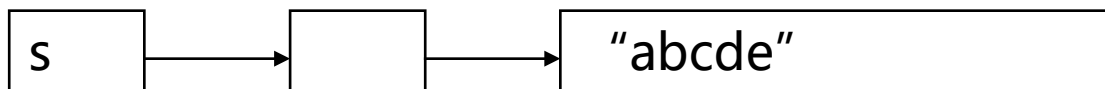


`q = &x;`



同样: `char **s;`

表示s指向的内容是一个指向字符的指针







# 多级指针的应用

用指向指针的指针访问指针数组的元素。如

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

```
int main()
{
    char *city[] = {"aaa", "bbb", "ccc", "ddd", "eee"};
    char **p;

    for (p = city; p < city + 5; ++p)
        cout << *p << endl;

    return 0;
}
```

**输出结果：**

```
aaa
bbb
ccc
ddd
eee
```

**C++不能直接申请动态二维数组，如`new int[5][9]`是非法的表达式**

## 解决方案一

用一维动态数组

将它按行序转换成一维数组，用动态的一维数组存储

如一个3行4列的矩阵 a 可以存储为12个元素的一维数组

访问 i 行 j 列的元素转换成访问一维数组的第  $4 * i + j$  个元素

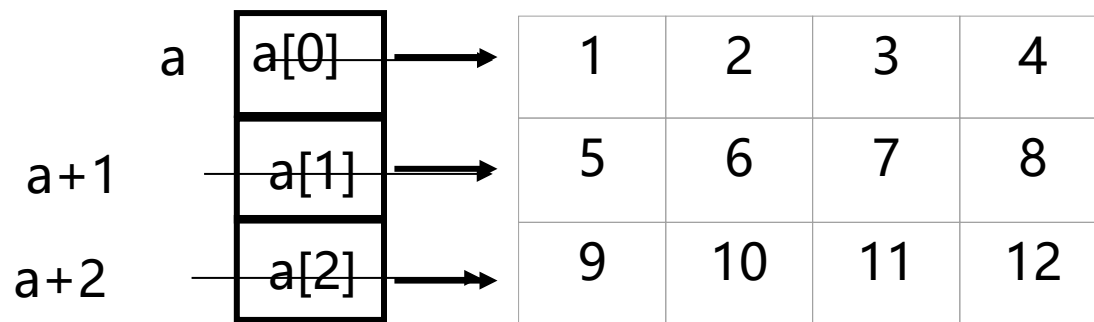
## 缺点

无法用`a[i][j]`的形式访问数组

# 动态的二维数组

## 解决方法二

利用数组和指针的关系



## 具体过程

数组名 `a` 用二级指针表示

`a` 数组是一个指针数组

每个 `a[i]` 指向一个一维数组

可以用 `a[i][j]` 访问

```
int main()
{
    int **a, i, j, k = 0;

    a = new int *[3];           // 申请指针数组
    for (i = 0; i < 3; ++i)      // 申请每一行空间
        a[i] = new int[4];
    for (i = 0; i < 3; ++i)      // 二维数组赋值
        for (j = 0; j < 4; ++j)
            a[i][j] = k++;
    for (i = 0; i < 3; ++i) {    // 二维数组输出
        cout << endl;
        for (j = 0; j < 4; ++j)
            cout << a[i][j] << '\t';
    }
    for (i = 0; i < 3; ++i)      // 二维数组空间释放
        delete [] a[i];
    delete [] a;
    return 0;
}
```

# 指向函数的指针

## 函数的指针

指向函数代码的起始地址

## 定义

返回类型 (\*指针变量名)(形式参数表);

## 赋值

```
eg. int isdigit(int n, int k) { ... }  
    int (*p)(int, int);  
    p=isdigit;  
    auto p = isdigit;
```

} 原型一致

## 引用

a=p(n,k)

## 用途

菜单选择  
函数参数

# 函数指针用于菜单选择

**例如，在一个工资管理系统中有如下功能**

添加员工

删除员工

修改员工信息

打印工资单

打印汇总表

## 设计考虑

把每个功能设计成一个函数

添加员工的函数为add

删除员工的函数为delete

修改员工信息的函数为modify

打印工资单的函数为printSalary

打印汇总表函数为printReport

主程序是一个循环，显示所有功能和它的编号，请用户输入编号，根据编号调用相应的函数。

```
int main()
{
    int select;
    while(true) {
        cout << "1--add \n";
        cout << "2--delete\n";
        cout << "3--modify\n";
        cout << "4--print salary\n";
        cout << "5--print report\n";
        cout << "0--quit\n";
        cin >> select;

        switch(select) {
            case 0: return 0;
            case 1: add(); break;
            case 2: erase(); break;
            case 3: modify(); break;
            case 4: printSalary(); break;
            case 5: printReport(); break;
            default: cout << "input error\n";
        }
    }
}
```

## 缺点

需要一个长长的switch

# 利用指向函数的指针

```
int main()
{
    int select;
    void ( *func[6] )( ) = { NULL, add, erase, modify, printSalary, printReport };
    while(1) {
        cout << "1--add \n";
        cout << "2--delete\n";
        cout << "3--modify\n";
        cout << "4--print salary\n";
        cout << "5--print report\n";
        cout << "0--quit\n";
        cin >> select;

        if (select == 0) return 0;
        if (select > 5) cout << "input error\n"; else func[select]();
    }
}
```



# 函数指针作为函数参数

## 问题

设计一个通用的冒泡排序函数，可以排序任何类型的数据

## 关键问题

如何解决任意类型数据的存问题？

将快速排序设计成一个函数模板，将待排序的数据类型设计成模板参数

如何解决不同类型的数据有不同的比较方式？

向排序函数传递一个比较函数来解决

```

void sort(int a[], int size)
{
    bool flag;
    int i, j;
        for (i = 1; i < size; ++i) {
            flag = false;
            for (j = 0; j < size - i; ++j)
                if (a[j+1] < a[j]) {
                    int tmp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = tmp;
                    flag = true;
                }
            if (!flag) break;
        }
}

```

```

template <class T>
void sort(T a[], int size, bool (*f)(T,T))

```

```

if ( f( a[j ], a[j +1 ] )) {
    T tmp = a[j];

```

```
bool decreaseInt( int x, int y ) { return x < y; }
```

```
bool increaseString( char *x, char *y ) { return strcmp( x, y ) > 0; }
```

} 有没有更好的解决方案?

```
int main()
```

```
{
```

```
    int a[] = {3,1,4,2,5,8,6,7,0,9}, i;
```

```
    char *b[] = {"aaa", "bbb", "fff", "ttt", "hhh", "ddd", "ggg", "www", "rrr ", "vvv "};
```

```
    sort( a, 10, decreaseInt );
```

```
    for ( i = 0; i < 10; ++i) cout << a[i] << "\t";
```

```
    cout << endl;
```

```
    sort(b, 10, increaseString );
```

```
    for (i = 0; i < 10; ++i) cout << b[i] << "\t";
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```

## 亦称Lambda表达式

可以理解成一个未命名的内联函数，用来代替一些简单函数  
通过指向函数的指针访问

## 与函数的不同处

可以定义在函数内部

可以访问所在函数的局部变量

## **[捕获列表](参数表)->返回类型{函数体}**

捕获列表是一个lambda 表达式所在的函数中定义的局部变量列表（通常为空）

返回类型、参数表和函数体与普通函数相同

与普通函数不同，lambda 函数必须使用尾置返回类型

# lambda 表达式定义实例

**最正常的:** `[ ](int x, int y) -> int { int z = x + y; return z; }`

**返回类型非常明确时可以不指定类型:** `[ ](int x, int y) { return x + y; }`

**可以使用所在函数中的变量 x:** `[x](int y)->int {return x+y; }`

**可以指定按引用访问 x:** `[&x](int y)->int {return x + = y; }`

**可以指定按引用访问函数中的任意变量:** `[&](int y)->int {return x += y; }`

**可以指定按值访问函数中的任意变量:** `[=](int y)->int {return x + y; }`

**可以指定按值访问x, 按引用访问 函数中的其他变量:**

`[&, x](int y)->int {return z = x+y; }`

**注意:**

捕获列表中指定为值访问时, 变量值是定义时的值捕获

列表中指定为引用访问时, 变量值是当前值

# Lambda表达式的使用

**Lambda表达式可以赋值给一个函数指针**

```
auto f = [](int x, int y) { return x + y; };
```

则可通过 f(3, 6) 调用该lambda函数, 可得结果9

**作为函数的实际参数**

```

int main()
{
    int a[] = {3, 1, 4, 2, 5, 8, 6, 7, 0, 9}, i;
    char *b[] = { "aaa", "bbb", "fff", "ttt", "hhh ", "ddd", "ggg", "www", "rrr", "vvv " };
    sort<int> (a, 10, [ ](int x, int y)->bool {return x>y; } );
    for (i = 0; i < 10; ++i) cout << a[i] << "\t";
    cout << endl;
    sort<int>(a, 10, [ ](int x, int y)->bool { return x < y; } );
    for ( i = 0; i < 10; ++i) cout << a[i] << "\t";
    cout << endl;

    sort<char *>(b, 10, [ ]( char *x, char *y )->bool { return strcmp(x, y) > 0; } );
    for (i = 0; i < 10; ++i) cout << b[i] << "\t";
    cout << endl;

    sort<char *>(b, 10, [ ]( char *x, char *y ) ->bool { return strcmp(x, y) < 0; } );
    for ( i = 0; i < 10; ++i) cout << b[i] << "\t";
    cout << endl;
    return 0;
}

```



# STL库函数中Lambda表达式的使用

STL的很多库函数都允许带一个函数指针的参数，例如对一组随机数进行各种统计

```
#include <iostream>
```

```
.....
```

```
const long Size = 390000L;
```

```
int main()
```

```
{
```

Sample size = 390000

```
    vector<int> numbers(Size);
```

```
    srand(std::time(0));
```

```
    generate( numbers.begin(), numbers.end(), rand );
```

```
    cout << "Sample size = " << Size << '\n';
```

# STL库函数中Lambda表达式的使用

```
int count3 = count_if ( numbers.begin(), numbers.end(),
                      [](int x){return x % 3 == 0;} );
cout << "Count of numbers divisible by 3: " << count3 << '\n';
int count13 = 0;
for_each ( numbers.begin(), numbers.end(),
          [&count13](int x){count13 += x % 13 == 0;} );
cout << "Count of numbers divisible by 13: " << count13 << '\n';

count3 = count13 = 0;
for_each ( numbers.begin(), numbers.end(),
          [&](int x){count3 += x % 3 == 0; count13 += x % 13 == 0;});
cout << "Count of numbers divisible by 3: " << count3 << '\n';
cout << "Count of numbers divisible by 13: " << count13 << '\n';

return 0;
}
```

```
Count of numbers divisible by 3: 130274
Count of numbers divisible by 13: 30009
Count of numbers divisible by 3: 130274
Count of numbers divisible by 13: 30009
```

打印学生成绩单，格式如下：

学号	姓名	语文成绩	数学成绩	英语成绩.
00001	张三	96	94	88
00003	李四	89	70	76
00004	王五	90	87	78

如何在程序中表示这组学生信息？

## 用二维的数组来表示？

不可行，因为这些信息有不同的类型

## 并联数组

每一列用一个一维数组来表示

要保证每位学生信息的正确性很难

将一组相关信息聚集在一起

学号	姓名	语文成绩	数学成绩	英语成绩.
00001	张三	96	94	88
00003	李四	89	70	76
00004	王五	90	87	78

学生

student1

student2

student3

允许程序员把一些分量聚合成一个整体，用一个变量表示  
各个分量都有名字，把这些分量称为成员(member)。

由于结构体的成员可以是各种类型的，程序员能创建适合于问题的数据聚合

# 结构体的使用

定义一个新的结构体类型

定义新类型的变量

访问结构体变量

## 定义结构体类型

说明结构体中包括哪些分量

## 格式

```
struct 结构体类型名{  
    字段声明;  
};
```

如:

```
struct studentT {  
    char no[10];  
    char name[10];  
    int chinese;  
    int math;  
    int english;  
};
```



不同的结构体中可以有相同的成员名

结构体成员的类型可以是任意类型，当然也可以是结构体类型

```
struct dateT
{
    int month;
    int day;
    int year;
};
```

```
struct studentT
{
    ...
    dateT birthday;
};
```

# 结构体变量的定义

## 和普通的变量定义一样

如定义了结构体类型studentT，就可以定义结构体变量：

```
studentT student1, sArray[10], *sptr;
```

申请动态变量： `sptr = new studentT;`

```
sptr = new studentT[10];
```

也可以加存储类别

```
static studentT s2;
```

可以在定义时赋初值

```
studentT student1= { "00001" , "张三" , 87, 90, 77};
```

# 结构体变量在内存中的映像

在分配内存时就会分配一块连续的空间，依次存放它的每一个分量

这块空间总的名字就是结构体变量的名字。内部还有各自的名字

student1

no

name

chinese

math

english

## 结构体变量的访问

对结构体类型变量的引用一般为引用他的成员

### 成员表示

结构变量名.成员名

如: student1.name

**如结构中还有结构，则一级一级用“.” 分开， 如**

如: student1.birthday.year

# 结构变量的赋值

## 无法直接对整个结构体赋值

结构体是一个统称。每个结构体类型在使用前都要先定义自己有哪些分量。系统事先无法知道如何处理他

## 结构体赋值

### 为它的每一个成员的赋值

如：输入student1的内容可用：

```
cin >> student1.no >> student1.name >> student1.chinese >> student1.math  
>> student1.english
```

### 同类型的结构变量之间可以相互赋值

如

```
student1 = student2;
```

将student2的成员对应赋给student1的成员

## 输出它的每一个成员

如：输出student1的内容可用：

```
cout << student1.no << student1.name << student1.chinese << student1.math  
    << student1.english
```

## 给结构体指针赋值

如: `sp = &student1;`  
`sp = new studentT;`

## 结构体指针的引用

<code>(*指针).成员</code>	如: <code>(*sp).name</code>	}	<code>student1.成员</code>
<code>指针-&gt;成员</code>	如: <code>sp-&gt;name</code>		

- `->` 是所有运算符中优先级最高的

通常程序员习惯使用第二种方法

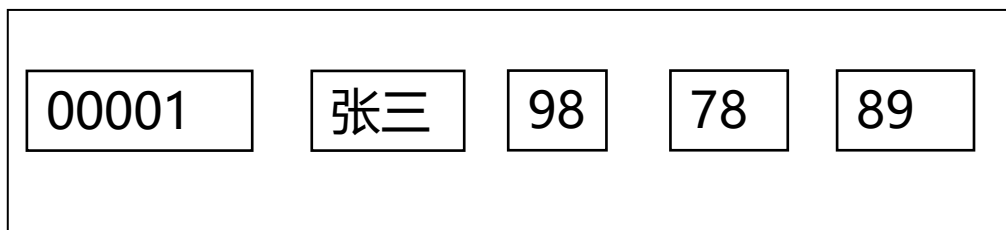
# 结构体作为函数参数

## 传递结构体

值传递

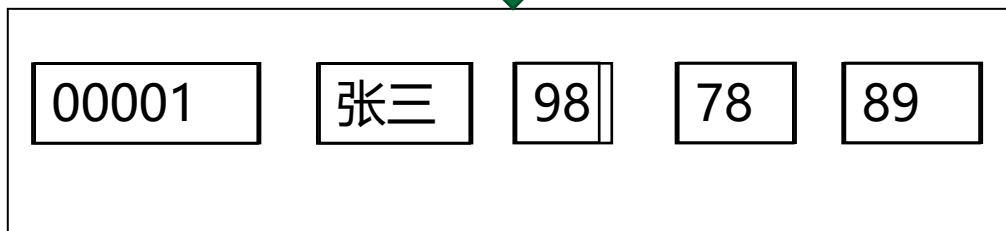
main函数

student1



函数PrintStudent中

s



```
void PrintStudent(studentT s)
```

```
PrintStudent( student1 );
```



# 指向结构体的指针作为参数

## 值传递缺点

结构体一般占用的内存量都比较大

浪费时间和空间

## 指针传递或引用传递

优点：效率高

缺点：函数可以修改实际参数

## 通常采用引用传递或const的引用传递

设计一函数，打印学生信息

```
void PrintStudent(const studentT &s)
{
    cout << s.no << '\t' << s.name << '\t' << s.chinese << '\t'
        << s.math << '\t' << s.english << endl;
}
```

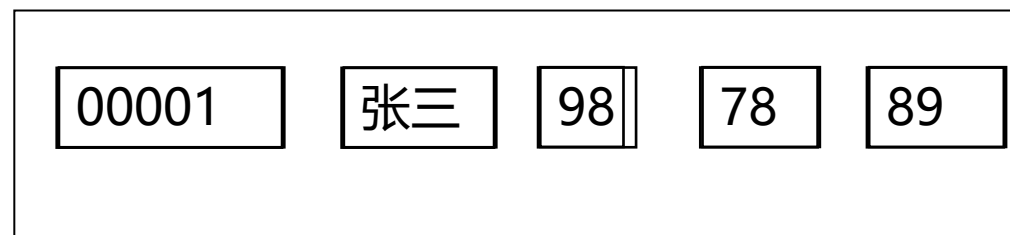
特点：节约内存，提高函数调用速度，可靠

# 返回结构体类型的函数

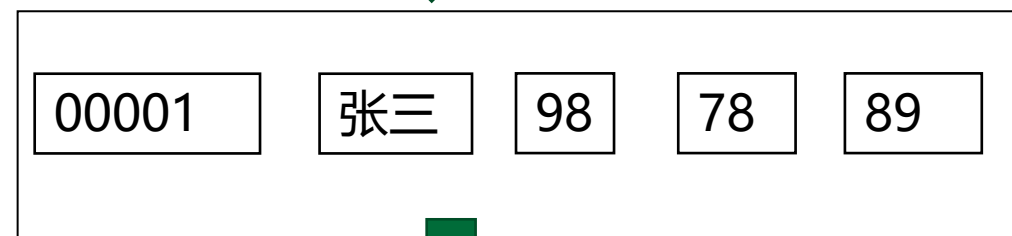
## 值返回

```
studentT GetPersonData()  
{  
    studentT person;  
    .....  
    return ( person );  
}  
  
int main()  
{  
    studentT p1;  
    p1 = GetPersonData();  
}
```

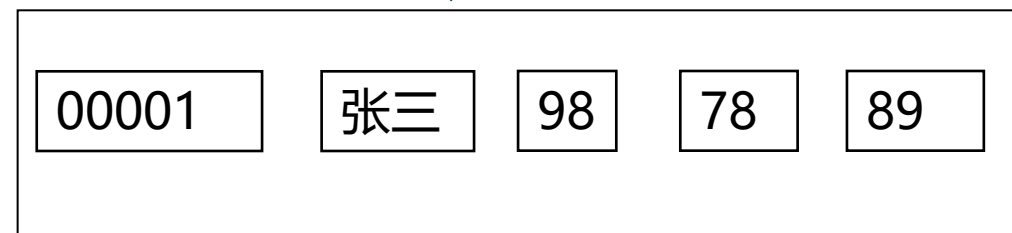
GetPersonData 中  
person



临时变量区



main函数  
p1

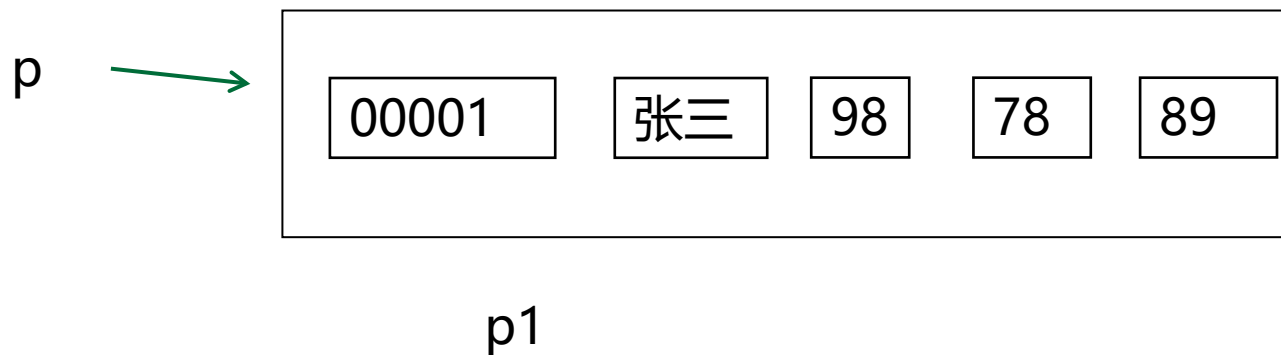


# 返回结构体引用的函数

```
studentT &GetPersonData( )
{
    studentT *p = new studentT;
    .....
    return *p;
}

int main()
{
    studentT &p1= GetPersonData();
}
```

GetPersonData 中



如果不允许修改返回值，可  
定义成返回常量引用

# 链表概念

存储一组数据元素的工具

存储一组数据元素

数组：元素个数必须在编程时确定

动态数组：元素个数必须在数组使用前确定

运行时元素个数还是无法确定怎么办？

使用链表



# 数组与链表

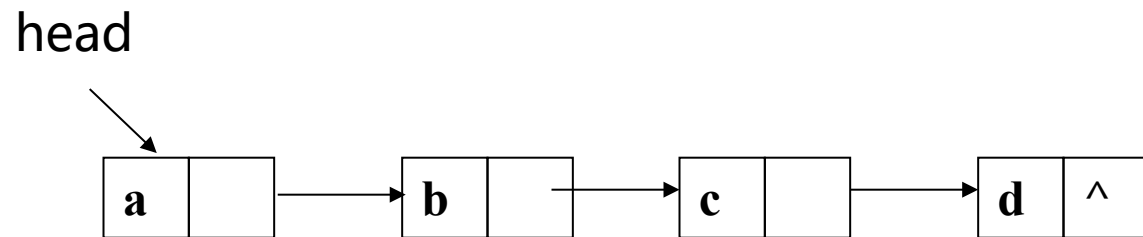
## 数组

元素存放在一块连续空间中

事先准备好空间

## 链表

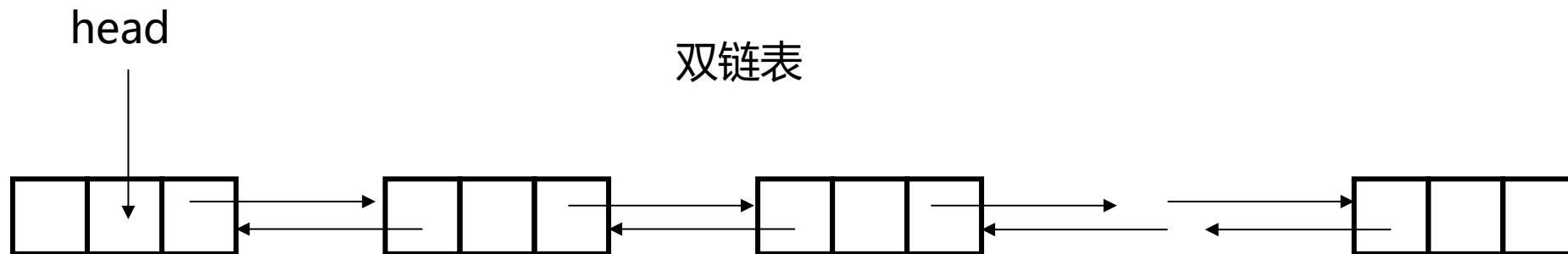
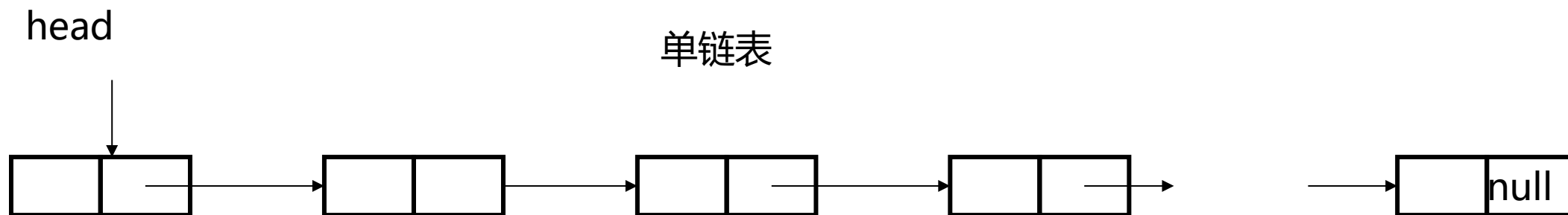
按需分配空间



# 链表

不需实现准备存储空间，需要时动态申请空间

元素分散存储在内存中，用指针记住有关系的元素的存储地址



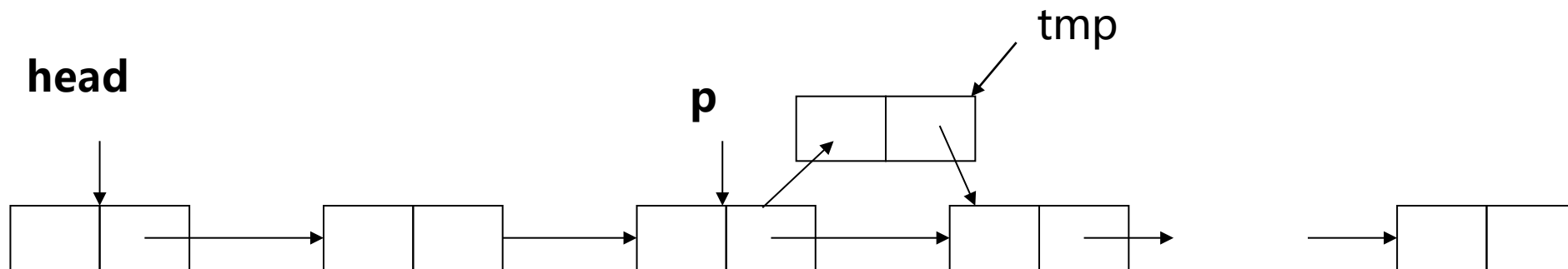
存储链表是存储链表中的一个一个结点，因此需要定义一个结点类型  
用指向第一个结点的指针表示一个链表

```
struct linkRec
{
    datatype data;
    linkRec *next;
};
```



# 单链表操作—插入

在结点p后插入一个结点



申请空间

输入数据放入申请到的空间

链入p后

```
tmp = new LinkRec;
```

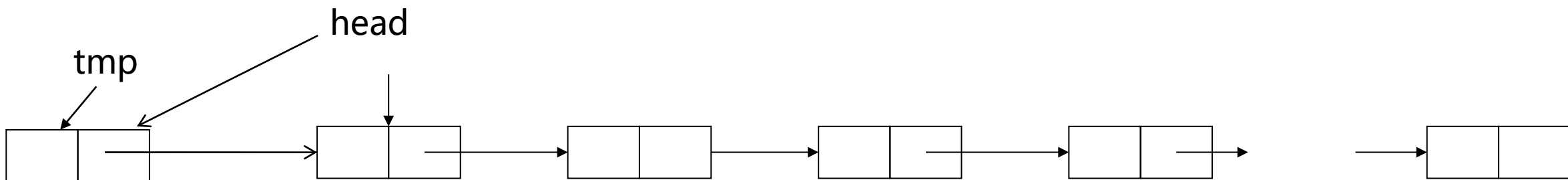
```
tmp->data = x;
```

```
tmp->next = p->next;
```

```
p->next = tmp;
```

# 单链表操作—插入

将新结点tmp插入为第一个结点



申请空间

输入数据放入申请到的空间

链到表头

```
tmp = new LinkRec;
```

```
tmp->data = x;
```

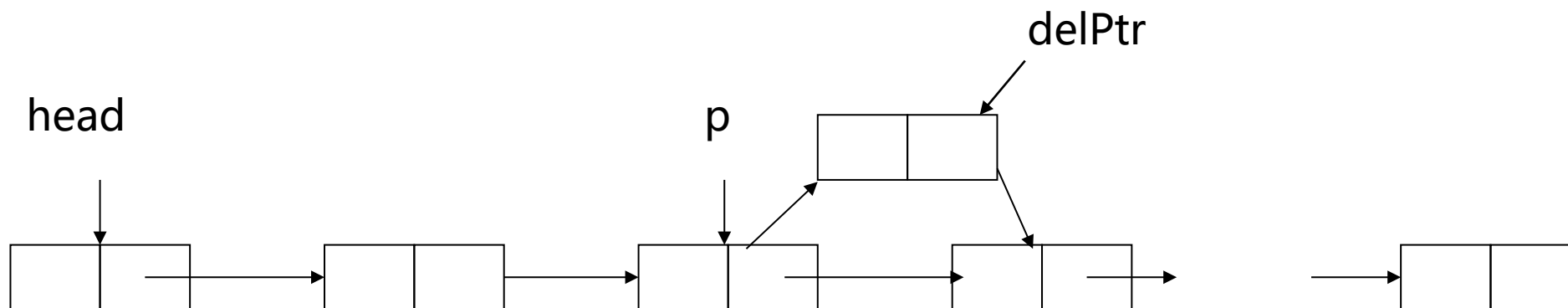
```
tmp->next = head;
```

```
head = tmp;
```



# 单链表操作—删除

把结点p后的结点删除



```
delPtr=p->next;  
p->next=delPtr->next;  
delete delPtr;
```

# 单链表操作--建立

## 定义头指针

```
linkRec *head;
```

## 建立头结点

申请空间

设为头结点

## 逐个从键盘输入数据，存入链表

接收输入

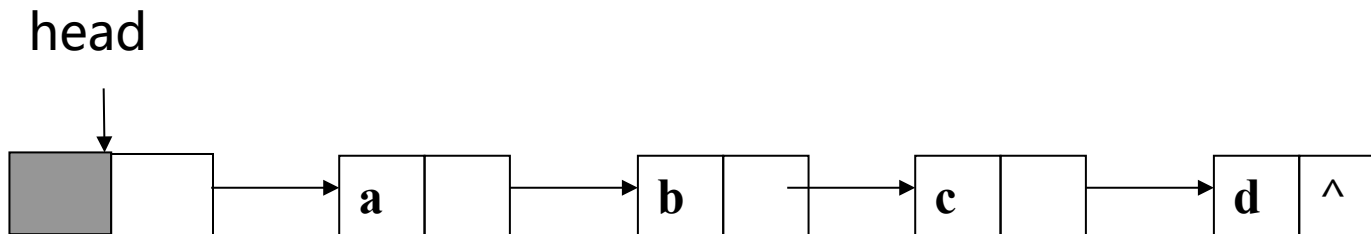
申请空间

输入数据放入申请到的空间

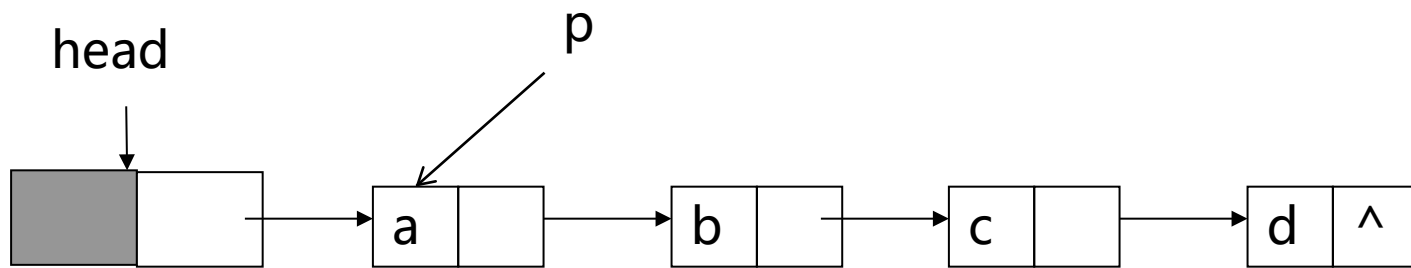
链入链表尾

## 置链表结束标志

```
head = new linkRec;
rear = head;
cin >> in_data;
while(输入未结束) {
    p = new linkRec;
    p->data = in_data;
    rear->next = p;
    rear = p;
    cin >> in_data;
}
rear->next = NULL;
```



# 单链表操作—输出



```
p = head->next;  
while ( p != NULL) {  
    cout << p->data;  
    p = p->next;  
}
```

**创建并访问一个带头结点的、存储整型数据的单链表，数据从键盘输入，0为输入结束标志。**

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

```
struct linkRec {
    int data;
    linkRec *next;
};
```

```
int main()
{
    int x;    //存放输入的值
    linkRec *head, *p, *rear;

    head = rear = new linkRec;
    while (true) {
        cin >> x;
        if (x == 0) break;
        p = new linkRec;
        p->data = x;
        rear->next = p;
        rear = p;
    }
```

```
rear->next = NULL;
```

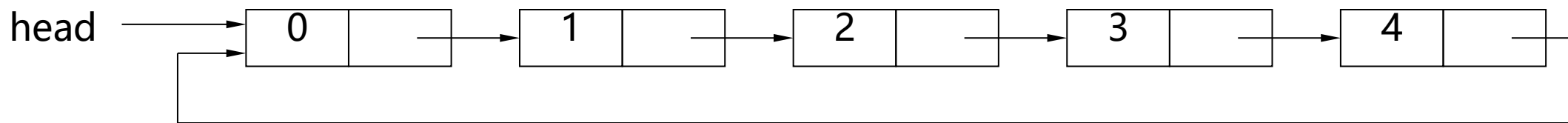
```
//读链表
cout << "链表的内容为: \n";
p = head->next;
while (p != NULL) {
    cout << p->data << '\t';
    p = p->next;
}
cout << endl;

return 0;
}
```



$n$ 个人围成一圈，从第一个人开始报数1、2、3。凡报到3者退出圈子。找出最后留在圈子中的人的序号。

解：用循环链表



当 $n = 5$ 时，其删除的节点的顺序为2, 0, 4, 1，最后剩下的节点为3。

```
struct node
{
    int data;
    node *next;
};

int main()
{
    node *head, *p, *q;
    int n, i;

    //输入n
    cout << "\ninput n:";
    cin >> n;

    //创建第一个结点
    head = p = new node;
    p->data = 0;
```

```
    for (i=1; i<n; ++i) {
        q = new node;
        q->data = i;
        p->next = q;
        p = q;
    }
    p->next = head;
    // 删除过程
    q=head;
    while (q->next != q) { //只要表中多于一个结点
        p = q->next;
        q = p->next;
        p->next = q->next; //绕过节点q
        cout << q->data << '\t'; //显示被删者的编号
        delete q; //回收被删者的空间
        q=p->next; //让q指向报1的节点
    }

    // 打印结果
    cout << "\n最后剩下: " << q->data << endl;

    return 0;
}
```

实现较复杂

插入、删除效率高，但查找第 $i$ 个元素效率低

无表满的问题

适合于动态表

## 设计阶段

自顶向下的分解

每个小问题设计为一个函数

公共小问题可以设计成一个库

## 实现阶段

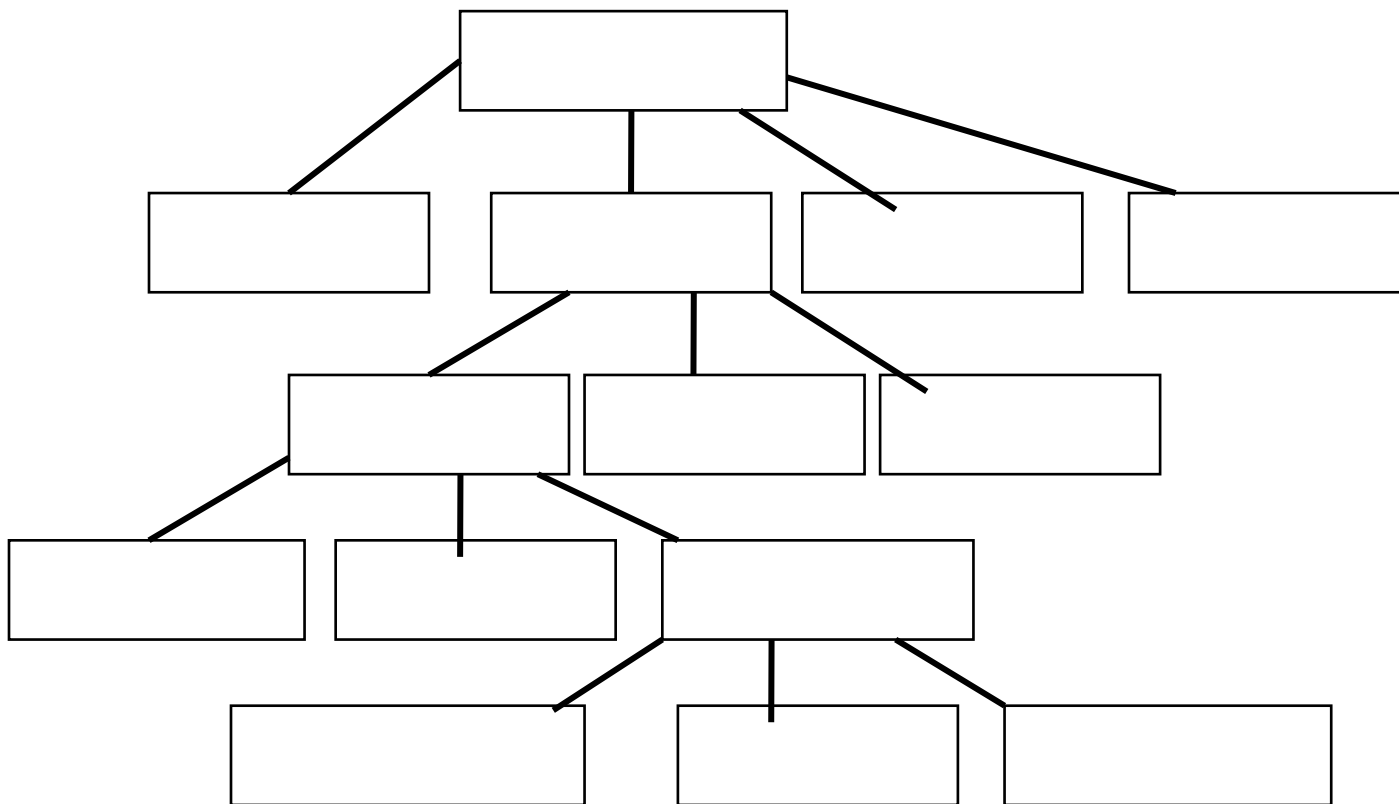
自底向上的实现

## 程序由3种基本结构组成

顺序

分支

循环



# 猜硬币的游戏

## 功能

提供游戏指南

计算机随机产生正反面，让用户猜，报告对错结果

重复玩游戏过程，直到用户不想玩了为止

程序要做两件事：显示程序指南；模拟玩游戏的过程。

```
main()  
{  
    显示游戏介绍;  
    玩游戏;  
}
```

主程序的两个步骤是相互独立的两个，没有什么联系，因此可设计成两个函数：

```
void prn_instruction()  
void play()
```

```
int main()  
{  
    prn_instruction();  
    play();  
  
    return 0;  
}
```

# prn\_instruction的实现

prn\_instruction函数的实现非常简单，只要一系列的输出语句把游戏指南显示一下

```
void prn_instruction()
{
    cout << "这是一个猜硬币正反面的游戏.\n";
    cout << "我会扔一个硬币，你来猜.\n";
    cout << "如果猜对了，你赢，否则我赢。 \n";
}
```

Play函数随机产生正反面，让用户猜，报告对错结果，然后询问是否要继续玩

```
void play()
{
    char flag = 'y' ;
    while ( flag == 'Y' || flag == 'y' ) {
        coin = 生成正反面;
        输入用户的猜测;
        if (用户猜测 == coin)
            报告本次猜测结果正确;
        else
            报告本次猜测结果错误;
        询问是否继续游戏
    }
}
```

## 生成正反面

正反面表示：用0表示正面，1表示反面

生成正反面：生成0和1两个随机数

## 输入用户的猜测

直接用一个输入语句

但想让程序做得好一点，就必须考虑得全面一些

比如，用户既不输入0也不输入1

解决方案：抽象成一个函数get\_call\_from\_user。



```
void play()
{
    int coin ;
    char flag = 'Y';

    srand(time(NULL));           //设置随机数种子
    while (flag == 'Y' || flag == 'y') {
        coin = rand() * 2 / RAND_MAX;           //生成扔硬币的结果
        if (get_call_from_user() == coin)
            cout << "你赢了";
        else
            cout << "我赢了";
        cout << "\n继续玩吗 (Y或y) ? ";
        cin >> flag;
    }
}
```

# get\_call\_from\_user的实现

该函数接收用户输入的一个整型数。如果输入的数不是0或1，则重新输入，否则返回输入的值

```
int get_call_from_user()
{
    int guess;          // 0 = head, 1 = tail
    do {
        cout << "\n输入你的选择 (0表示正面, 1表示反面) :";
        cin >> guess;
    } while (guess !=0 && guess !=1);

    return guess;
}
```

这是一个猜硬币正反面的游戏.

我会扔一个硬币, 你来猜.

如果猜对了, 你赢, 否则我赢。

输入你的选择 (0表示正面, 1表示反面) :1

我赢了

继续玩吗 (Y或y) ? y

输入你的选择 (0表示正面, 1表示反面) :6

输入你的选择 (0表示正面, 1表示反面) :1

你赢了

继续玩吗 (Y或y) ? n

Press any key to continue

**当程序变得更长的时候，要在一个单独的源文件中处理如此众多的函数会变得困难**

## 解决方案

把程序再分成几个小的源文件。每个源文件都包含一组相关的函数

一个源文件被称为一个模块

## 模块划分标准

同一模块中的函数比较类似

块内联系尽可能大，块间联系尽可能小

# 石头、剪刀、布游戏

## 游戏规则

布覆盖石头

石头砸坏剪刀

剪刀剪碎布

## 游戏的过程

游戏者选择出石头、剪子或布

计算机也随机选择一个

输出输赢结果

继续游戏，直到游戏者选择结束为止

在此过程中，游戏者也可以阅读游戏指南或看看当前战况。

# 第一层的分解

```
while (用户输入 != quit) {  
    switch (用户的选择) {  
        case paper, rock, scissor: 机器选择;  
                                   评判结果;  
                                   报告结果;  
        case game: 显示目前的战况;  
        case help: 显示帮助信息;  
    }  
}  
显示战况;
```

用户输入数据类型???

评判结果是什么类型???

## 函数抽取

获取用户输入    selection\_by\_player  
获取机器输入    selection\_by\_machine  
评判结果        compare  
报告结果并记录结果信息    report  
显示目前战况    prn\_game\_status  
显示帮助信息    prn\_help

## 方案一

用整型编码

缺点：可读性、安全性差

## 方案二

用符号常量

## 类型定义格式

enum 枚举类型名 {元素表};

## 石头、剪子、布中的枚举类型

用户输入值的类型: enum p\_r\_s { paper, rock, scissor, game, help, quit } ;

比较结果类型: enum outcome { win, lose, tie } ;

## 枚举类型变量的定义

p\_r\_s select;

## 枚举类型变量的使用

赋值: select = paper;

比较: paper < rock 比较这两个值的内部表示

枚举类型不能直接输入输出

# 枚举类型的内部表示

## 内部采用编码表示

当定义

```
enum p_r_s { paper, rock, scissor, game, help, quit } ;
```

默认用0代表pape, 1代表rock , ..., 5 表示quit

## 指定编码值

希望从1而不是0开始编号, 可以这样定义

```
enum p_r_s { paper = 1, rock, scissor, game, help, quit } ;
```

可以从中间某一个开始重新指定, 如

```
enum p_r_s { paper, rock = 5, scissor, game, help, quit } ;
```



# 模块划分

## 主模块

main函数

## 获取选择的模块

selection\_by\_player  
selection\_by\_machine

## 比较模块

compare

## 输出模块

report  
prn\_game\_status  
prn\_help

## **selection\_by\_player**

功能：从键盘接收用户的输入并返回此输入值

原型： `p_r_s selection_by_player () ;`

## **selection\_by\_machine**

功能：由机器产生一个石头、剪子、布的值，并返回

原型： `p_r_s selection_by_machine () ;`

# Compare模块的设计

## 功能

compare函数比较用户输入的值和机器产生的值，确定输赢

## 参数

用户输入的值和机器产生的值，都是p\_r\_s类型的

## 返回值

判断的结果，是outcome类型

## 原型

```
outcome compare( p_r_s, p_r_s );
```

## **prn\_help**

功能：显示一个用户输入的指南，告诉用户如何输入他的选择。它没有参数也没有返回值

原型： `void prn_help( );`

## **Report**

功能：函数报告输赢结果，并记录输赢的次数

参数：输赢结果、输的次数、赢的次数和平局的次数

返回值：无

## **prn\_game\_status**

功能：报告至今为止的战况

参数：输的次数、赢的次数和平的次数

返回值：无

# print模块的进一步考虑

## 输的次数、赢的次数、平局的次数

在Report和prn\_game\_status两个函数中都出现

Report函数修改这些变量的值

prn\_game\_status函数显示这些变量的值

## 三个参数与其他模块的函数无任何关系

## 内部状态可以作为该模块专用的全局变量

## 函数原型

```
void prn_game_status();
```

```
void report(outcome result);
```

# 头文件的设计

## 头文件

包含所有的符号常量定义、类型定义和函数原型声明

## 每个模块都include这个头文件

每个模块就不必要再写那些符号常量定义、类型定义和函数的原型声明

## 问题

链接时，编译器会发现这些类型定义、符号常量和函数原型的声明在程序中反复出现多次

## 解决方法

需要用到一个新的编译预处理命令：

```
#ifndef 标识符
```

```
...
```

```
#endif
```

# 头文件的格式

```
#ifndef _name_h
```

```
#define _name_h
```

头文件真正需要写的内容

```
#endif
```

# 石头、剪子、布游戏的头文件

```
// 文件: p_r_s.h
// 本文件定义了两个枚举类型, 声明了本程序包括的所有函数原型

#ifndef P_R_S
#define P_R_S
    #include <iostream>
    #include <cstdlib>
    #include <ctime>
    using namespace std;

    enum p_r_s { paper, rock, scissor, game, help, quit };
    enum outcome { win , lose, tie };

    outcome compare( p_r_s player_choice, p_r_s machine_choice );
    void prn_game_status( );
    void prn_help();
    void report (outcome result );
    p_r_s selection_by_machine( );
    p_r_s selection_by_player( );
#endif
```



# 主模块的实现

```
//文件: main.cpp  
// 石头、剪子、布游戏的主模块
```

```
#include "p_r_s.h "
```

```
int main()
```

```
{
```

```
    outcome result;
```

```
    p_r_s player_choice, machine_choice;
```

```
    // seed the random number generator
```

```
    srand(time(NULL));
```

```
while((player_choice = selection_by_player()) != quit)
    switch(player_choice) {
        case paper: case rock: case scissor:
            machine_choice = selection_by_machine();
            result = compare(player_choice, machine_choice);
            report(result);
            break;
        case game:
            prn_game_status();
            break;
        case help: prn_help();
    }
    prn_game_status();

    return 0;
}
```

# select模块的实现

//文件: select.cpp  
//包括机器选择selection\_by\_machine和  
//玩家选择selection\_by\_player函数的实现

```
#include "p_r_s.h"
```

```
p_r_s selection_by_machine()  
{
```

```
    int select = (rand() * 3 / (RAND_MAX + 1));
```

```
    cout << " I am ";
```

```
    switch(select) {
```

```
        case 0: cout << "paper. ";  
                break;
```

```
        case 1: cout << "rock. ";  
                break;
```

```
        case 2: cout << "scissor. ";  
                break;
```

```
    }
```

```
    return ((p_r_s) select);
```

```
}
```

```
enum p_r_s { paper, rock, scissor, game, help, quit };
```



```
p_r_s selection_by_player()
{
    char c;
    p_r_s player_choice;

    prn_help();
    cout << "please select: "; cin >> c;
    switch(c) {
        case 'p': player_choice = paper;
                  cout << "you are paper. "; break;
        case 'r': player_choice = rock;
                  cout << "you are rock. "; break;
        case 's': player_choice = scissor;
                  cout << "you are scissor. ";break;
        case 'g': player_choice = game; break;
        case 'q': player_choice = quit; break;
        default : player_choice = help; break;
    }
    return player_choice;
}
```

# Compare模块的实现

//文件: compare.cpp

//包括compare函数的实现

```
#include "p_r_s.h"
```

```
outcome compare(p_r_s player_choice, p_r_s machine_choice)
```

```
{
```

```
    outcome result;
```

```
    if (player_choice == machine_choice) return tie;
```

```
    switch(player_choice) {
```

```
        case paper: result = (machine_choice == rock) ? win : lose;
```

```
            break;
```

```
        case rock: result = (machine_choice == scissor) ? win : lose;
```

```
            break;
```

```
        case scissor: result = (machine_choice == paper) ? win : lose;
```

```
    }
```

```
    return result;
```

```
}
```

# Print模块的实现

```
//文件: print.cpp  
//包括所有与输出有关的模块。  
//有prn_game_status, prn_help和report函数
```

```
#include "p_r_s.h "  
static int win_cnt = 0, lose_cnt = 0, tie_cnt = 0;           //模块的内部状态  
  
void report(outcome result)  
{  
    switch(result) {  
        case win: ++win_cnt;  
                   cout << "You win. \n"; break;  
        case lose: ++lose_cnt;  
                   cout << "You lose.\n"; break;  
        case tie:  ++tie_cnt;  
                   cout << "A tie.\n"; break;  
    }  
}
```



```
void prn_game_status()
{
    cout << endl;
    cout << "GAME STATUS:" << endl;
    cout << "win: " << win_cnt << endl;
    cout << "Lose: " << lose_cnt << endl;
    cout << "tie:  " << tie_cnt << endl;
    cout << "Total: " << win_cnt + lose_cnt + tie_cnt << endl;
}
```

```
void prn_help()
{
    cout << endl
        << "The following characters can be used:\n"
        << "  p  for paper\n"
        << "  r  for rock\n"
        << "  s  for scissors\n"
        << "  g  print the game status\n"
        << "  h  help, print this list\n"
        << "  q  quit the game\n";
}
```

## 库

常用的工具

## 库的主题

同一个库中的函数都应该是处理同一类问题

如标准库iostream包含输入输出功能，cmath包含数学运算函数

自己设计的库也要有一个主题

## 库的通用性

库中的功能应来源于某一应用，但不局限于该应用，而且要高于该应用

在某一应用程序中提取库内容时应尽量考虑到兼容更多的应用，使其他应用程序也能共享这个库



## 设计库的接口

库的用户必须了解的内容，包括库中函数的原型、这些函数用到的符号常量和自定义类型

接口表现为一个头文件

## 设计库中的函数的实现

表现为一个源文件

**库的这种实现方法称为信息隐藏**

## 库功能提取

在9.1中，用到了随机生成0和1

在9.2中，用到了随机生成0和2

在自动出题中，用到了随机生成0和3及随机生成0到9

## 库功能确定

生成low到high之间的随机数 `int RandomInteger(int low, int high)`

初始化函数 `RandomInit()`实现设置随机数种子的功能

## 头文件的格式

#ifndef

#define

.....

#endif

## 注释

头文件头上有段注释，说明库的主题、功能

每个函数声明前有一段注释，告诉用户如何使用这些函数

# 随机函数库接口文件

```
//文件: Random.h  
//随机函数库的头文件
```

```
#ifndef _random_h  
#define _random_h
```

```
//函数: RandomInit  
//用法: RandomInit()  
//作用: 此函数初始化随机数种子  
void RandomInit();
```

```
//函数: RandomInteger  
//用法: n = RandomInteger(low, high)  
//作用: 此函数返回一个 low 到 high 之间的随机数, 包括 low 和 high  
int RandomInteger(int low, int high);
```

```
#endif
```

## 实现文件名

与头文件的名字是相同

如头文件为Random.h，则实现文件为Random.cpp

## 实现文件的格式

注释：这一部分简单介绍库的功能

include此cpp文件所需的头文件

每个实现要包含自己的头文件，以便编译器能检查函数定义和函数原型声明的一致性

每个函数的实现代码

在每个函数实现的前面也必须有一段注释

```
//文件: Random.cpp  
//该文件实现了Random库  
#include <cstdlib>  
#include <ctime>  
#include "Random.h"
```

```
//函数: RandomInit  
//该函数取当前系统时间作为随机数发生器的种子  
void RandomInit()  
{    srand(time(NULL));    }
```

```
// 函数: RandomInteger  
// 该函数将0到RAND_MAX的区间的划分成high - low + 1 个 子区间。当产生的随机数落在第一个  
// 子区间时, 则映射成low。 当落在最后一个子区间时, 映射成high。当落在第 i 个子区间时  
// (i 从 0 到 high-low) , 则映射到low + i  
int RandomInteger(int low, int high)  
{    return (low + (high - low + 1) * rand() / (RAND_MAX + 1));    }
```

# 库的应用 -- 龟兔赛跑

动物	跑动类型	占用时间	跑动量
乌龟	快走	50%	向前走3点
	后滑	20%	向后退6点
	慢走	30%	向前走一点
兔子	睡觉	20%	不动
	大后滑	20%	向后退9点
	快走	10%	向前走14点
	小步跳	30%	向前走3点
	慢后滑	20%	向后退2点

# 龟兔赛跑解题思路

分别用变量tortoise和hare代表乌龟和兔子的当前位置

时间用秒计算

用随机数来决定乌龟和兔子在每一秒的动作

根据动作决定乌龟和兔子的位置的移动

跑道的长度设为70个点



# 第一层分解

```
main()
{
    int hare = 0, tortoise = 0, timer = 0;    //timer是计时器, 从0开始计时

    while ( hare < RACE_END && tortoise < RACE_END ) {
        tortoise += 乌龟根据他这一时刻的行为移动的距离;    int move_tortoise();
        hare += 兔子根据他这一时刻的行为移动的距离;    int move_hare();
        输出当前计时和兔子乌龟的位置;    void print_position(int timer, int tortoise, int hare);
        ++timer;
    }
    if (hare > tortoise)
        cout << "\n hare wins!";
    else
        cout << "\n tortoise wins!";
}
```

# 模块划分

## 主模块

main

## 移动模块

move\_tortoise

move\_hare

## 输出模块

print\_position

```
#include "Random.h"    //包含随机数库
#include <iostream>
using namespace std;

const int RACE_END = 70;    //设置跑道的长度

int move_tortoise();
int move_hare();
void print_position(int, int, int);
```

```
int main()
{
    int hare = 0, tortoise = 0, timer = 0;

    RandomInit(); //随机数初始化
    cout << "timer tortoise hare\n"; //输出表头
    while (hare < RACE_END && tortoise < RACE_END) {
        tortoise += move_tortoise(); //乌龟移动
        hare += move_hare(); //兔子移动
        print_position(timer, tortoise, hare);
        ++timer;
    }
    if (hare > tortoise)
        cout << "\n hare wins!\n";
    else
        cout << "\n tortoise wins!\n";

    return 0;
}
```

## 乌龟的行为

快走	50%
后滑	20%
慢走	30%

## 用户行为生成方法

利用随机数的等概率

生成0-9之间的随机数

0-4是第一种情况

5-6是第二种情况

7-9是第三种情况

# Move模块

```
// 文件名: move.cpp
#include "Random.h"          //本模块用到了随机函数库

int move_tortoise()
{
    int probability = RandomInteger(0,9);    //产生0到9之间的随机数

    if (probability < 5) return 3;           //快走
    else if (probability < 7) return -6;     //后滑
    else return 1;                          //慢走
}
```

```
int move_hare()
{
    int probability = RandomInteger(0,9);

    if (probability < 2) return 0;           //睡觉
    else if (probability < 4) return -9;      //大后滑
        else if (probability < 5) return 14;    //快走
            else if (probability < 8) return 3;  //小步跳
                else return -2;                 //慢后滑
}
```

# Print模块

// 文件名: print.cpp

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

```
void print_position(int timer, int t, int h)
{
    if (timer % 6 == 0)           //每隔6秒空一行
        cout << endl;
    cout << timer << '\t' << t << '\t' << h << '\n';
}
```



如何利用结构化程序设计的技术来设计解决一个大问题的程序

自顶向下分解

模块划分

如何在模块中保存内部状态

如何从程序中抽取出库以及如何设计和使用库