程序设计与算法（一）

二进制与十六进制

计算机电路由逻辑门组成，状态可以为"开"（高电位）或"关"（低电位）

二进制数的一位，取值为0或1，称为一个比特（bit），简写为 b

八个二进制位称为一个字节（byte），简写为 B

1024B 称为1KB，1024KB 称为1MB（1兆），1024MB 称为1GB，1024GB 称为1TB

1B，即8个由0或1组成的串，一共有256（2^8）种不同的组合，也诞生了 ASCII 编码方案

K 进制数到十进制数的转换

假设有一个 n+1位的 K 进制数，形式如下：

$\mathrm{A_n}A_{n-1}A_{n-2}\cdots A_2A_1A_0$

则其转换为**十进制**数为：

$\mathrm{A_0}\times K^0+A_1\times K^1+A_2\times K^2+\cdots+A_{n-1}\times K^{n-1}+A_n\times K^n$

十进制数到 K 进制数的转换——短除法

有一个十进制数 N 与进制 K，则 N 可表示为：

$\mathrm{N_0}=A_0\times K^0+A_1\times K^1+A_2\times K^2+\cdots+A_{n-1}\times K^{n-1}+A_n\times K^n$

$\mathrm{N_0}=A_0+K(A_1+A_2\times K^1\ldots+A_{n-1}\times K^{n-2}+A_n\times K^{n-1})$

将N_0除以 K，得余数A_0，商$N_1=A_1+A_2\times K^1+\cdots+A_{n-1}\times K^{n-2}+A_n\times K^{n-1}$。再将商除以 K，得余数$A_1$，新的商$N_2=A_2+A_3\times K^1+\cdots+A_{n-1}\times K^{n-3}+A_n\times K^{n-2}$.

多次进行，直到除完，则所有得到的余数分别为$A_0,A_1,A_2,\ldots,A_{n-1},A_n$，那么得到的 K 进制数为$A_nA_{n-1}A_{n-2}\ldots A_2A_1A_0$.

$\mathrm{N_0}\div K=N_1\cdots\cdots A_0$ ↑

$N_1\div K=N2\cdots\cdots A_1$ ↑

$N2\div K=N_3\cdots\cdots A_2$ ↑

$\cdots$ ↑

$N_{n-1}\div K=N_n\cdots\cdots A_{n-1}$ ↑

$N_n\div K=0\cdots\cdots A_n$

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

int main(void){
  printf("Hello,World!\n");  /*"Hello,World!"是字符串,前后要用""括起来*/
  return 0;
}
```

变量

变量代表了系统分配的内存空间

变量类型决定了占用空间的大小

变量不能重复定义

命名：

大小写敏感

由字母、下划线、数字组成，不能有空格，不能以数字开头

不要为 **begin**，**end**，**next**，index，list，link 等

数据类型

signed 的最高位是符号位，1表示负数，0表示非负数

用浮点数时最好用 double，精度较高
求变量所占字节数：
sizeof(变量名/类型名)
sizeof 函数的返回值类型为 long unsigned int，需要用%lu 输出
在使用变量之前要声明变量，在声明变量同时为其赋值称为变量的初始化

有符号整数
最高位为符号位
符号位为0，则为非负数，其绝对值为除符号位以外的部分
符号位为1则为负数，其绝对值为除符号位以外的部分**取反后再加1**
将负整数表示为二进制方法：
设置符号位为1
其余位等于绝对值**取反再加1**

数据类型的自动转换
浮点型转换为整型，直接去掉小数部分
字符型转换为整型，转换为字符的 ASCII 码
整型转换为字符型，如果整型太大，则之保留最右边1字节（即二进制的第0~7位，或十六进制的第0~1位），再通过 ASCII 码转换为字符型

一个十六进制位对应四个二进制位
十六进制数以0x 或0X 开头
八进制以0开头

ASCII
0~9：48~57
A~Z：65~90
a~z：97~122

常量
转义字符
\n 换行
\r 从开头输出
\t 制表符
\b 退格
\\ 反斜杠（\）
\' 单引号（'）
\" 双引号（"）
\0 0字符（字符串结束符）
\ddd 八进制数 ddd
\xhh 十六进制数 hh
字符串常量
用双引号（""）括起来
""也是一个字符串常量，代表一个空串
字符串常量与字符不同
"a"是字符串，'a'是字符，"a"不能用 char 赋值
"123"是字符串，123是整型
符号常量
#define 常量名 常量值

输入和输出

double: %lf

long long int: %lld

用%f 输出时，默认输出小数点后 6 位

用%.nf 控制输出时，并非直接截取，而会近似输出，称为"Bankers Rounding"，这与浮点数在计算机中的存储方式有关

输入输出流

cout:

```
using namespace std;  /*在主函数之前*/

a=10; b=1.3; c='x'
cout << "a=" << a << ",b=" << b << endl;  /*endl:换行*/
cout << 123 << ",c=" << c;
```

->a=10,b=1.3

123,c=10

cin:

```
using namespace std;

int a,b; double c; char d;
cin >> a >> b >> c >> d;
cout << a << "," << b << "," << c << "," << d << endl;
```

cin,cout 速度比 scanf,printf 慢

一个程序中 cin,cout 与 scanf,printf 不能混用

赋值运算符

+=, -=, *=, /=, %=的执行速度较快

表达式有值

x=y 的值为 y 的值

表达式的值以操作数中精度高的类型为准

精度大小：double>long long>int>short>char

float 在运算时自动转化为 double

(int)a + (float)b <==> (int)a + (double)b -> (double)c

两数在加、减、乘时可能溢出，溢出的部分直接丢弃

除法运算中，若操作数都为 int，结果也为 int，直接舍去小数

通过强制转换数据类型来使在做除法时输出更高精度的数

```
c = (double)a/b;  /*<b>将 a 强制转换为 double</b>*/
```

求余运算（模运算）

除法与求余的除数都不能为 0

除以 0.0:

```
0.0/0.0 = nan
1.0/0.0 = inf
0.0/1.0 = 0.0
```

自增运算符++

++a: 将 a 的值加 1，表达式返回值为 a+1 之后的值

a++: 将 a 的值加 1，表达式返回值为 a+1 之前的值

```
  int a, b=3;
  a = ++b;  ->a==4, b==4
  a = b++;  ->a==4, b==5
```
关系运算符
```
  ==  !=  >  <  >=  <=
```
比较结果是 bool 类型，成立为 true，不成立为 false
```
  true <==> 非 0 整数值（一般为 1），false <==> 0
<b>  n3 = 0 > 10 <==> n3 = ( 0 > 10 ) </b>  ->n3==0
```
逻辑运算符
```
  &&  ||  !

01. a       b       a && b  a || b  !a
02. 0       0       0       0       1
03. 0       1       0       1       1
04. 1       0       0       1       0
05. 1       1       1       1       0
```
逻辑表达式是**短路计算**的，即只要运算到能确定表达式的真假时，就停止（即使没有算完）
```
    a && b : 如果 a 已经是假，则跳过 b（<b>b 不被计算</b>），表达式的值直接为假
    a || b : 如果 a 已经是真，则表达式的值一定为真（<b>b 不被计算</b>）

    a=0; b=1;
    bool n = a++ && b++;  -> a==1, b==1, n==0
    n= a++ && b++;  -> a==2, b==2, n==1
    n= a++ || b++;  -> a==3, b==2, n==1
```
强制类型转换运算符
```
  表达式: (运算符)变量
  只是将表达式的类型转换，没有将变量的类型转换
  double f=9.7;
  int n=(int)f <==> int n=f ;  ->n=9
  f=n/2;  ->f=4.0  /*先算 n/2 的值为(int)4，然后将 4 转换为 double 赋给 f*/
  f=double(n)/2;  ->f=4.5  /*直接将(double)n 处以 2，得到 4.5，再赋给 f*/
```
运算符的优先级
```
 ^ ++ -- !
 | * / %
 | + -
 | < > <= >= == !=
 | &&
 | ||
 | = += -= *= /= %=
  可用括号改变运算顺序
  a+++b <==> (a++)+b
```
条件分支结构
```
  if(表达式 1){
    语句组 1;
  }else if (表达式 2){
    语句组 2;
  }
```

```
  ...
  else{
    语句组 n;
  }
  if 也是短路计算——如果表达式 1 为真，则不计算之后的表达式 2 等
    编程时尽量避免表达式的二义性
  if(n%2==1) <==> if(n%2) <==> if(n&1)
  else 总与和它最近的 if 配对（与括号的配对相似）
判断闰年(假设 year>0)
  if((year%4==0 && year%100) || (year%400==0)) printf("闰年\n");
    /*<b>&&的优先级大于||</b>*/
    /*如果该年非整百年且能被 4 整除，或该年能被 400 整除，即为闰年*/
switch 语句
  switch(表达式 a){  /*表达式 a 的值必须为整数类型，如 int,char 等*/
  case 常量表达式 1:
    语句 1;
 <b>break;</b>
  case 常量表达式 2:
    语句 2:
    break;
  ...
  default:  /*default 语句可以没有*/
    语句 n;       /*如果 a 的值与常量表达式的值都不相同，则执行 default 中的语句*/
  }
  switch 开始后会一直执行，直到第一次遇到 break 语句

循环结构
for 循环（先判断，再执行）
  for(表达式 1;表达式 2;表达式 3){
    语句;
  }
  1.先执行表达式 1
  2.计算表达式 2，判断是否为真，若为真则转到 3，为假则转到 6
  3.执行语句
  4.执行表达式 3
  5.转到 2
  6.跳出循环，执行下面的语句
打印 a 到 z:
  for(int i=0;i<26;i++)
    printf("%c",'a'+i);
<b>在 for 循环中表达式 1 定义的变量只在 for 内部起作用，不影响与之同名的 for 外部变量（局部变量优先）</b>
  int i=5;
  for(int i=0;i<10;i++);
  printf("%d\n",i);  //-> 5
for 循环中表达式 1 和表达式 3 可以是多个用逗号（,）连接的表达式
  for(int i=0, int j=7; i<10; i++, j++){ ... }

while 循环（先判断，再执行）
  while(1){
```

```
  ...                              while(scanf("%d",&n)==1 && n){
  scanf("%d",&n);     <==>           ...
  if(n==0) break;                  }
}
```

用牛顿迭代法求 sqrt(n)

```
double EPS=0.001; /*控制精度*/            double EPS=0.001;
double x, lastx;                         double x,lastx;
x=a/2, lastx=x+1+EPS;                    x=a/2;
while(x-lastx>EPS || lastx-x>EPS){  <==> do{
  lastx=x;                                 lastx=x;
  x=(x+n/x)/2;                             x=(x+n/x)/2;
}                                        }while(fabs(lastx-x)>EPS);
cout << x << endl;                       cout << x << endl;
```

do-while 循环（先执行，后判断）

```
do{
  语句;
}while(表达式);
```

判断两个浮点数相等不能直接用 a==b 的形式，而该用 fabs(a-b)<EPS 的形式，EPS 是很小的数，比如 1e-7（即 10^-7，为浮点型）

break 语句

在循环体中，用于跳出循环

continue 语句

在循环体中，用于结束本次循环，进行下一次循环

OJ 编程题输入数据的处理

scanf 的返回值为 int，表示成功读入变量的个数

若 scanf 返回值为 EOF(符号常量，即-1)表示输入数据已结束

```
n=scanf("%d%d",&a,&b);
```

输入： 12 56 12 a a 12 ^D([Ctrl]+d)（linux 中表示输入结束，及 EOF）

n==2 n==1 n==0 n==-1

cin 表达式的值为 bool，若成功读入所有变量则为 true，否则为 false

处理无特定结束标记，只有 EOF 的 OJ 题目的输入

输入若干个整数，输出最大值

```
scanf("%d",max);  //设第一个数为最大值（不使用数组记录）
while(scanf("%d",&n)==1){  //<==> while(scanf("%d",&n)!=EOF), EOF 值为-1
  if(n>max) max=n;
}
printf("%d\n",max);
```

用 freopen 重定向输入

将测试数据存入文件，然后用 freopen 将输入由键盘重定向为文件

```
int main(){
  freopen("test.txt","r",stdin);
  //此后所有输入都来自文件 test.txt
  ...
}
```

求 Fibonacci 数列

```
f1=1,f2=1;
while(){
  t=f1+f2;
  f1=f2;
  f2=t;
}
```

求阶乘的和

```
t=1,sum=0;
for(int i=1;i<=n;i++){
  t*=i;
  sum+=t;
}
```

求大于 2 的素数

```
//若 n 非素数，则在[2,sqrt(n)]中一定有 n 的因子
for(int i=3;i<=n;i+=2){   //枚举大于 2 的奇数 n
  int k; bool flag=1;
  sq=sqrt(n)+1;
  for(k=3;k<=sq;k+=2)     //判断 n 是否为素数
    if(k%i==0) {flag=0; break;}
  if(flag) cout << i << endl;
}
```

数组

定义数组时数组长度必须为常量

数组大小：数组长度*sizeof(数组类型)

数组 a 的大小=N*sizeof(T)=sizeof(a)

数组名代表数组的地址 &a[i]=a+i

a[i]为变量，a 为地址

防止数组越界，可在定义时增大数组长度

数组一般不定义在主函数中，尤其是大数组；大数组不能定义在主函数中

c++中，数组初始化时**不能用单个数赋值全部**，没被初始化的剩下元素自动赋 0

```
int a[100]={1}; //a[0]==1,其他都为 0
```

筛法求素数（用空间换时间，加快了计算速度）

```
cons int MAX=10000,sqMAX=sqrt(MAX)+1;
bool a[MAX+1];  //a[i]为 1 表示 i 为素数
int i,j;
for(i=0;i<=MAX;i++) a[i]=1;
for(i=2;i<=sqMAX;i++) if(a[i]==1){
    for(j=i*2;j<=MAX;j+=i) a[j]=0;
}
for(i=2;i<=MAX;i++)
  if(a[i]==1) printf("%d ",i);
```

已知给定日期为周几，求另一日期为周几

计算两日期之差，再与 7 求余

矩阵乘法

```
|-     -|   |-   -|   |-                 -|
| a b c |   | x w |   | ax+by+cz aw+au+av |
| d e f | * | y u | = | dx+ey+fz dw+eu+fv |
|-     -|   | z v |   |-                 -|
```

```
for(i=0;i<m;i++){  //m 为 a 行数
  for(j=0;j<n;j++){  //n 为 b 列数
    c[i][j]=0;
    for(k=0;k<q;k++){  //q 为 a 列数，也为 b 行数
      c[i][j]+=a[i][k]*b[k][j];
    }
  }
}
```

函数参数的传递

函数形参是实参的一个拷贝，形参的改变不影响实参

当形参是数组时（如 a[]），传递的是数组 a[0]元素的地址，所以函数中对数组的改变会影响到真实数组

```
void abc(int a[]);
abc(a);
```

声明函数的形参中多维数组的**最低维**可以省去

计算数组地址：

```
a[i][j]=a[0][0]+i*N*sizeof(a[0][0])+j*sizeof(a[0][0]);
```

递归

一个函数调用其自身

递归必须有终止条件

求 Fibonacci 数列的第 n 项

```
int Fibonacci(int n){
  if(n==1 || n==2) return 1;
  return Fibonacci(n-1) + Fibonacci(n-2);
}
```

库函数和头文件

头文件中包含函数的**声明**

编译器的库函数中含有头文件中声明的函数的可执行语句

包含头文件

```
#include<[文件名]>  //在默认头文件夹中寻找
#include"[文件名]"  //先在当前目录查找头文件，若找不到则再寻找默认文件夹
```

编译时会将头文件拷贝至当前文件中

cmath 中的函数

```
int abs(int x);
double fabs(double x);
double sin(double x); //返回 x（弧度）的正弦
double cos(double x);
<b>int ceil(double x);<\b>  //返回不小于 x 的最小整数
<b>int floor(double x);</b> //返回不大于 x 的最大整数
double sqrt(double x);
```

ctype 中的函数

```
  int isdigit(int c) //判断 c 是否为数字
  int isalpha(int c) //判断 c 是否为字母
  int isalnum(int c) //判断 c 是否为数字或字母
  int islower(int c) //判断 c 是否为小写字母
  int isupper(int c) //判断 c 是否为大写字母
  int toupper(int c) //若 c 为小写字母，则返回对应大写字母
  int tolower(int c) //若 c 为大写字母，则返回对应小写字母

const double PI = arcos(-1.0);

<b>位运算</b>
  & 按位与
    通常用来使某些位为 0，且其他位保持不变
    使一个数(2 字节)的低 8 位全为 0: n &= 0xff00;
    判断一个数的第 7 位是否为 1: if(n&0x0080==0x0080) return 1;
      0x80: 0000 0000 1000 0000
  | 按位或
    通常用来使某些位为 1，且其他位保持不变
    使一个数的低 8 位全为 1: n|=0x00ff;
      0x00ff: 0000 0000 1111 1111
  ~ 按位取反（非）
  ^ 按位异或
    相异为 1，相同为 0
    通常用来将某些位取反，且其他位保持不变
    将一个数的低 8 位取反: n^=0x00ff;
 <b>若 a^b==c ,则 c^b==a, c^a==b </b>
 <b>异或运算可以交换 a,b 的值</b>
      a=a^b; b=a^b; a=a^b;
  >> 右移
    低位舍去，高位:对于 signed，高位补符号位
                   对于 undigned，高位补 0
    右移 n 位，相当于除 2^n，并且结果<b>往小里取整</b>
      25>>4 == 1
     -25>>4 == -2
  << 左移
    高位舍去，低位补 0
    左移 n 位相当于乘 2^n

int GetBit(int n, int i){
  return (a>>i)&1;
}
void SetBit(int n, int i, int e){
  if(e==1) n|=(1<<i);
  else n&=~(1<<i);
}
void FlipBit(int n, int i){
  n^=(1<<i);
}
```

字符串

三种形式

[1]双引号中的字符串**常量**，以'\0'结尾

[2]存放于字符数组中，以'\0'结尾

[3]string 对象

字符串常量占据内存的字节数等于字符数+1，在结尾用'\0'表示字符串已结束

""为空串

scanf, cin 读取到空格为止

```
scanf("%s",str);
```

scanf,cin 可能导致数组越界

输入一行到字符数组

[1]函数: cin.getline(char str[],int strSize);

读入一行(长度不超过 bufSize-1)或 bufSize-1 个字符

```
cin.getline(str,sizeof(str));
```

[2]函数: gets(char str[]);

不使用，可能导致数组越界

[3]函数: fgets(char str[],int strSize,FILE * stream);

```
fgets(str,sizeof(str),stdin);
fputs(str,stdout);
```

字符串库函数

```
char * strcpy(char str1[],char str2[]); //将 str2 的内容拷贝到 str1
char * strcat(char str1[],char str2[]); //将 str2 的内容拼接到 str1 之后
int strcmp(char str1[],char str2[]); //若 str1==str2，返回 0；若 str1<str2，返回负数；若 str1>str2，返回正数
int strlen(char str[]);
char * strupr(char str[]);    //将 str 中的字母都转为大写
char * strlwr(char str[]);    //将 str 中的字母都转为小写

for(int i=0; str[i] ;i++);

int Strstr(char str1[],char str2[]){    //暴力运算，还可用 KMP 算法
  if(str2[0]==0) return 0;
  for(int i=0;str1[i];i++){
    if(str1[i]!=str2[0]) continue;
    int k=i,j=0;
    for(j=0;str2[j];j++){
      if(str1[k++]!=str2[j]) break;
    }
    if(str2[j]==0) return i+1;
  }
  return -1;
}
```

指针

```
T * p;
```

p 类型为 T*

*p 类型为 T

可通过指针**自由访问**内存空间

应用：底层驱动程序、病毒等
不同基类型的指针不能相互赋值

指针的运算
[]运算（递归运算）
p[i] <==> *(p+i)
p[i][j] <==> *(*(p+i)+j)

访问 int 型变量 n 的第 1 个字节：
```
char *p; p=(char*)&n;
```

空指针：指向地址 0(NULL)的指针　　NULL: 0
作为函数形参时，T *p <==> T p[]　　EOF: -1

若 int a[10], *p=a;
则 sizeof(a)==40, sizeof(p)==4

```
void Reserve(int *p,int size){
  for(int i=0;i<size/2;i++)
    swap(p[i],p[size-i-1]);
}
```

指向指针的指针
若 T a[M][N]
则 a[i]为一维数组，类型为 T*
sizeof(a[i])=sizeof(T)*N
T ** p;
p 是指向指针的指针，其指向的地方存放着一个类型为 T*的指针
p 类型为 T，*(*p)类型为 T
若 p 类型为 T*，那么&p 类型为 T**

指针和字符串
字符串常量类型为 char*
字符数组名类型为 char*
字符串操作库（cstring）函数
```
char * strchr(const char * str, int c);   //返回字符串中第一个字符'c'的地址
char * strstr(const char * str, const char * subStr);
int stricmp(const char * s1, const char * s2);    //不大小写敏感的字符串比较
int strncmp(const char * s1, const char * s2, int n); //比较前 n 个字符大小
char * strncpy(char * dest, const char * src, int n); //不添加'\0'，仅拷贝'\0'
<b>char * strtok(char * str, const char * delim);
  //从 str 中逐个抽取被 delim 中任意字符分隔开的若干个字符串</b>
```
cstdlib 函数:
```
int atoi(char * s);   //array to int: 若字符串全为数字，则返回整型，否则返回 0
                      //atoi("1234") == 1234;  atoi("a12") == 0;
long int atoll(char *s);
long long int atoll(char * s);
double atof(char * s);    //转换浮点数
char * itoa(int value, char * string, int radix);
```

```
    //int to array: 将 value 以 radix 进制表示法写入 string
```

统计单词个数

```
  #include<cstring>
  char a[100], *p;
  int main(void){
    int count=0;
    fgets(a,sizeof(a),stdin);
    a[strlen(a)-1]='\0';    //fgets 的最后一位（'\0'之前）可能是'\n'
    p=strtok(a," ,.-\"\'!");  //<b>注意 strtok 中第二个参数</b>
    while(p!=NULL){
      count++;
      printf("%s\n",p);
      p=strtok(NULL," ,.-\"\'!");  //<b>后续调用时，第一个参数必须为 NULL</b>
    }
    printf("total words: %d\n",count);
    return 0;
  }
```

void 指针

```
  void * p;
```

可用**任何基类型**的指针为 void 指针赋值

因 sizeof(void)无定义，*p 无定义，同样，p 也不能做任何运算

```
  void * memset(void * dest, int ch, int n);
    //将 dest 开使的 n 个字节都设为 ch，返回 dest
    char a[4]="";
    memset(a,'A',sizeof(a)-1);
    cout << a << endl;  //->AAA
  void * memcpy(void * dest, void * src, int n);
    //将 src 开始的 n 个字节，拷贝到 dest，返回 dest

void * MyMemcpy(void * dest, void * src, int n){
  //若原区域与目标区域有重叠，则错误
  char * pDest=(char *)dest;
  char * pSrc=(char *)src;
  for(int i=0;i<n;i++) *(pDest+i)=*(pSrc+i);
  return dest;
}
```

函数指针

程序运行时，每个函数都会占用连续的内存空间，而函数名即为函数所在内存空间的起始地址（也称"入口地址"）

可以使一个指针指向入口地址，那么通过指针就可以调用函数

类型名 (* 指针变量名)(参数类型 1, 参数类型 2, ...);

```
    int fun(int a, char b);
    int (*pf)(int, char)=fun;  //函数名即为地址
    pf(1,'a');  <==>  fun(1,'a');
/*
  void qsort(void * base, int nelem, unsigned int width, int(* pfCompare)(const void *, const void
```

```
*));
    //base 为待排序数组起始地址，nelem 为元素个数，width 为单个元素大小(字节)，pfCompare 自己编写
    //pfCompare:
    //  若 *elem1 应该在 *elem2 前面，则返回负整数
    //  若 *elem1 应该在 *elem2 后面，则返回正整数
    //  若 *elem1 与 *elem2 无次序要求，则返回 0

int Compare(const void * n1, const void * n2){
  int *p1,*p2;
  p1=(int *)n1;
  p2=(int *)n2;
  return (*p1-*p2);  //升序排列
//return (*p2-*p1);  //降序排列
}

int main(){
  int a[100];
  int count=0,i=0;

  while(scanf("%d",&a[i])==1)
    i++;
  qsort(a,i,sizeof(int),Compare);
  for(int j=0;j<i;j++){
    printf("a[%d]: %d\n",j,a[j]);
  }
  return 0;
}
*/
```

结构体

```
  struct 结构体名{
    成员类型名 成员变量名
    ...
  }[变量名];
```

结构体之间**可以相互赋值**，但不能进行比较运算

结构体的成员变量在内存中一般是连续存放的

结构体的成员变量可以是指向**相同类型**结构体的**指针**，也可以是**另一个**结构体

```
  struct Str{
    ...
    struct Str2 in;
    struct Str * next;
  };
```

访问成员变量

结构体名.成员名

结构体指针->成员名

```
    pStr <==> &Str, *(pStr) <==> Str
    Str.a <==> pStr->a <==> *(pStr).a
```

结构体变量的初始化

```
  struct Str{               struct Str2{
```

```
    int a;                    int a;
    char b[10];               char b;
    struct Str2 in;         }
  }str={10,"Hello",{20,'a'}};
```

程序结构（全局变量、局部变量、静态变量）

全局变量都为静态变量

局部变量之前使用 static 标识符，也成为静态变量

静态变量存放地址在整个程序运行期间固定不变，即不会改变也不会消失

若未初始化，则每个静态变量都为 0

静态变量只初始化一次

```
char * MyStrtok(char * p, char * sep){
  static char * start;
  if(p!=NULL) start=p;
  while(*start!=0 && strchr(sep,*start)) start++; //跳过分隔符号
  if(*start==0) return NULL;
  char * q=start;
  while(*start!=0 && strchr(sep,*start)==0) start++; //寻找下一个分隔符号
  if(*start!=0){
    *start=0;
    start++;
  }
  return q;
}
```

标识符的作用域

变量名、函数名、类型名统称为标识符

一个标识符的作用范围称标识符的作用域

函数的形参作用域为整个函数

局部变量的作用域为它所在的整个语句组（即大括号括出的）

for 循环体内定义的循环控制变量的作用域是整个 for 循环

局部变量优先

变量的生存期

在一个变量的生存期内，其占有的内存只能自己使用，不能由其他变量使用

选择排序 O(N^2)

```
  void SelectSort(int a[], int size){
    for(int i=0;i<size-1;i++){
      int min=i;
      for(int j=i+1;j<size;j++) if(a[j]<a[min]) min=j;
      SWAP(a[i],a[min]);
    }
  }
```

插入排序 O(N^2)

```
  void InsertSort(int a[], int size){
    for(int i=1;i<size;i++){  //将 a[i]放到合适的位置
      for(int j=0;j<i;j++){  //寻找比 a[i]大的元素
        if(a[j]>a[i]){  //将 a[i]插入 a[j]之前，a[j]之后的元素都后移
```

```
          int temp=a[i];
          for(int k=i;k>j;k--) a[k]=a[k-1];
          a[j]=temp; break;
        }
      }
    }
  }
```

冒泡排序 O(N^2)

```
  void BubbleSort(int a[], int size){
    for(int i=size-1;i>0;i--){
      for(int j=0;j<i;j++){
        if(a[j]>a[j+1]) SWAP(a[j],a[j+1]);
      }
    }
  }
```

时间复杂度

用大写 O 和小写 n 表示，n 表示问题的规模

用算法运行过程中某种时间**恒定的**，也是被执行次数最多的操作被执行次数与 n 的关系来衡量

平均复杂度与最坏复杂度可能相同，也可能不同

主要表现的是 n 的**增长**对复杂度的影响

常见复杂度：

常数级 O(1)，对数级 O(logn)，线性级 O(n)，多项式级 O(n^k)，指数级 O(a^n)，阶乘级 O(n!)

常见算法复杂度：

在无序数列中查找某数（顺序查找） O(n)

平面上有 n 个点，求出所有任意两点间的距离 O(n^2)

简单排序 O(n^2)

快速排序 O(nlogn)

二分查找 O(logn)

二分查找

每通过一次计算，将问题规模缩小到原来的一半

条件：范围内的内容有序

```
int BinarySearch(int a[], int size, int p){
  int L=0, R=size-1;    //在全闭区间[L,R]中寻找
  while(L<=R){  //如果查找空间不为空就继续查找(L 要<b>小于等于</b>R)
 <b>int M=(R-L)/2+L;</b>  //防止溢出
    if(p==a[M]) return M;
    else if(p>a[M]) L=M+1;
    else R=M-1;
  }
  return -1;
}
```

在给定的从小到大排序的区间内查找比 p 小的，且下标最大的元素

```
int lower_bound(int a[], int size, int p){
  int L=0, R=size-1;
  int lastPos=-1;
```

```
  while(L<=R){
    int M=L+(R-L)/2;
    if(a[M]>=p) R=M-1;
    else{lastPos=M; L=M+1;}
  }
  return lastPos;
}
```

二分法求 f(x)=x^3-5x^2+10x-80 的根(精确到 10^-6)

f(x)在[0,100]上单调递增且有一根

[1]

```
  double lastx=0, x=100;
  while(fabs(lastx-x)>=EPS){
    double M=x+(lastx-x)/2;
    if(f(M)>EPS) x=M;
    else  lastx=M;
  }
```

[2]

```
  double x1=0, x2=100, y;
  double root=x1+(x2-x1)/2;
  y=f(root);
  while(fabs(y)>EPS){
    if(y>0) x2=root;
    else x1=root;
    root=x1+(x2-x1)/2;
    y=f(root);
  }
  printf("%lf\n",root);
```

输入 n 个整数(n<=100,000，整数在 int 范围内)，找出其中两个数，使它们的和等于 m

[1]O(nlogn)

1.将数组排序

2.对数组中每个元素 a[i]，在数组中二分查找 m-a[i]，看能否找到

[2]O(nlogn)

1.将数组排序

2.令 i=0,j=n-1，看 a[i]+a[j]，如果大于 m 则 j--，小于 m 则 i++，直至 a[i]+a[j]==m

STL 概述

Standard Template Libraty

头文件: algorithm

STL 中的排序

[1]对基本类型数组从小到大排序

```
  sort(数组名+n1, 数组名+n2);
```

将数组中[n1,n2)的元素从小到大排序，下标为 n2 的元素不在区间内

```
  int a[6]={3,2,4,1,5,9};
  sort(a+1,a+4)  //-> a:{3,1,2,4,5,9}
```

[2]对元素类型为 T 的基本类型数组从大到小排序

```
  sort(数组名+n1, 数组名+n2, greater<T>());
```

```
  sort(a+1,a+4,greater<int>());
[3]对任意类型为 T 的数组排序
  sort(数组名+n1, 数组名+n2, 排序规则结构体名());
  排序规则结构:
    struct 结构名{
      bool operator()(const T & a1, const T & a2){
        //若 a1 应该在 a2 前面，则返回 true，否则返回 false
          return a1>a2;  //从大到小排序
      }
    };

struct Str{
  int a;
  char b[10];
  double c;
}a[7];
struct Rule{
  bool operator()(const Str & s1, const Str & s2){
    if(strcmp(s1.b,s2.b)<0) return true;  //按 b 中字典序排序
    else return false;
  }
};
sort(a, a+sizeof(a)/sizeof(Str), Rule());

STL 中的二分查找
  binary_search
  lower_bound
  upper_bound
  对<b>已经有序</b>的数组进行二分查找

bool binary_search(...)
[1]在升序数组上二分查找
  binary_search(数组名+n1, 数组名+n2, 查找值);
  区间同样为[n1,n2)
  若找到则返回 true，找不到则返回 false
<b>等于(查找值与数组值): a 必须在 b 前面 和 b 必须在 a 前面 都不成立，而非 a==b </b>
[2]在类型为 T 的有序数组内二分查找
  binary_search(数组名+n1, 数组名+n2, 查找值, 排序规则结构体名());
  查找时的规则与排序时的一致
  等于: a 必须在 b 前面 和 b 必须在 a 前面 都不成立，而非 a==b

int a[6]={12,45,3,98,21,7};
sort(a,a+6);  //-> a:{3,7,12,21,45,98}
binary_search(a,a+6,7);  //返回 true
binary_search(a,a+6,77); //返回 false
sort(a,a+6,Rule());  //按个位数从小到大排序
                     //-> a:{21,12,3,45,7,98}
binary_search(a,a+6,17);         //返回 false
binary_search(a,a+6,17,Rule());  //返回 true
```

T* lower_bound(...) O(logn)
  二分查找**下界**，返回类型为 T*的指针
  如果找不到，则返回下标为 n2 的元素地址
[1]在升序数组中查找下界
  lower_bound(数组名+n1，数组名+n2，查找值);
  查找下标最小，且**大于等于**查找值的元素
[2]自定义规则查找下界
  lower_bound(数组名+n1，数组名+n2，查找值，排序规则结构体名());
  查找下标最小，且按照自定义排序规则**可以**排在查找值后面的元素

T* upper_bound(...) O(logn)
  二分查找**上界**，返回类型为 T*的指针
  如果找不到，则返回下标为 n2 的元素地址
[1]在升序数组中查找上界
  查找下标最小，且**大于**查找值的元素
[2]自定义规则查找上界
  查找下标最小，且按照自定义排序规则**必须**排在查找值后面的元素

STL 中的平衡二叉树数据结构
  在大量增加、删除数据的同时，进行大量数据的查找
  O(logn)
  排序容器:
    multiset
    set
    multimap
    map
    可自动维护数组的有序

multiset
  头文件: set
  multiset<T> st;
  可以有重复元素
  排序规则: a<b 为 true，则 a 在 b 前面
  st.begin()   //返回值为 multiset<T>::iterator，指向第一个元素
  st.end()   //返回值为 multiset<T>::iterator，指向最后一个元素**之后**，
  st.size()
  st.insert(a); //添加元素 a O(logn)
  st.find(a);   //查找元素 a O(logn)，找不到则返回值为 this->end()
  st.erase(it); //删除迭代器 it 所指向的元素 O(logn)
multiset 上的迭代器
  multiset<T>::iterator it;
  it 为迭代器，相当于指针，可用于指向 multiset 中的元素
  访问 multiset 中的元素要通过迭代器
  访问时使用间接访问符号(*)
  与指针的不同:
    multiset 中的迭代器可以自增、自减，可以用==、!=比较，但不能比大小，不能加减整数，不能相减

#include<set>

```
int main(){
  int a[10]={1,14,12,13,7,13,21,19,8,8};
  multiset<int> st;
  for(int i=0;i<10;i++) st.insert(a[i]);
  multiset<int>::iterator it;  //迭代器，类似于指针
  for(it=st.begin();it!=st.end();it++)  //注意，用 it!=st.end() 进行比较
    printf("%d ",*it);  //-> 1 7 8 8 12 13 13 14 19 21
  it=st.find(22);  //*it==st.end()
  st.insert(22);
  it=st.find(22);  //*i==22
  it=st.lower_bound(13);
    //返回最靠后的迭代器 it，使[begin(),it)中的元素都在 13 之前，即下界
    //*it == 13(第一个)
  it=st.upper_bound(8);
    //返回最靠前的迭代器 it，使[it,end())中的元素都在 8 的后面，即上界
    //*it == 12
  return 0;
}

<b>删除 multiset 中的元素 x </b>
  for(i=st.lower_bound(x) ; i!=st.upper_bound(x) ;i++)
    st.erase(i);

自定义规则的 multiset
  multiset<int,greater<int>_> st;  //排从大到小排序

struct Rule{
  bool operator()(const int & a, const int & b){
    return a%10 < b%10;
  }
};
int main(){
  multiset<int,greater<int>> st;
  int a[10]={1,14,12,13,7,13,21,19,8,8};
  for(int i=0;i<10;i++) st.insert(a[i]);
  multiset<int,greater<int>>::iterator i;
  for(i=st.begin();i!=st.end();i++) printf("%d ",*i);
    //-> 21 19 14 13 13 12 8 8 7 1
  mulsiset<int,Rule> st2;
    for(int i=0;i<10;i++) st2.insert(a[i]);
  multiset<int,Rule>::iterator p;
  for(p=st2.begin();i!=st2.end();i++) printf("%d ",*p);
    //-> 1 21 12 13 13 14 7 8 8 19
  p=st2.fine(133);  //*p==13

  return 0;
}

set
```

不能有重复元素
a,b 重复 <==> a 可以排在 b 前面，b 也可以排在 a 前面
set 插入元素可能失败，失败时返回值指向重复元素
判断是否插入成功：

```
pair<set<int>::iterator, bool> result=st.insert(10);
/*  pair<set<int>::iterator, bool>
 *   <==>
 *  struct {
 *    set<int>::iterator first;
 *    bool second;
 *  }
*/
  if(!result.second)  //如果插入失败
    printf("%d already exists\n",*result.first);
  else
    printf("%d inserted\n",*result.first);
```

```
pair<T1,T2>
  <==>
struct {
  T1 first;
  T2 second;
}
```

multimap
头文件：map
一个 key 可以对应多个 value
元素为 pair 形式
若 multimap<T1,T2> mp;
则 mp 中元素为：

```
struct {
  T1 first;  //关键字（key）
  T2 second; //值（vaule）
};
```

元素按 first 排序，并且由于(multi)map 重载[]运算符，可用[]按 first 查找
默认排序规则：若 a.first<b.first 为 true，则 a 在 b 前
插入：mp.insert(make_pair(a,b));

map
关键字不能重复（重复：两个关键字谁在谁前面都可以）
一个 key 只能对应一个 value
插入元素可能失败
重载了[]运算符，下标为关键字，返回值为含有该关键字的元素的值
类似于数组下标
若 map 中没有所查找的关键字，则插入，**并将值初始化为 0**
插入：mp.insert(make_pair(a,b));

```
#include<iostream>
#include<cstdio>
```

```
#include<cstring>
#include<map>
using namespace std;
struct StudentInfo{         struct Student{
  int id;                     int score;
  char name[20];              StudentInfo info;
};                          };
typedef multimap<int,StudentInfo> MAP_STD;  //multimap 按照 int 排序
int main(void){
  MAP_STD mp; Student st;
  char cmd[20];
  while(cin >> cmd){
    if(cmd[0]=='A'){
      cin >> st.info.name >> st.info.id >> st.score;
      mp.insert(make_pair(st.score,st.info));
      //make_pair 生成一个 pair<int,StudentInfo>变量
      //其中 first == st.score, second == st.info
    }else if(cmd[0]=='Q'){
      int score; cin >> score;
      MAP_STD::iterator p = mp.lower_bound(score);
      //mp.lower_bound(score)返回 p，使[mp.begin(),p)中的元素都小于等于 score
      if(p!=mp.begin()){
        p--;
        score=p->first;
        MAP_STD::iterator maxp=p;
        int maxId=p->second.id;
        for(;p!=mp.begin() && p->first==score; p--){
        //<b>倒序遍历</b>所有与 score 相等的学生
          if(p->second.id>maxId){
            maxp=p; maxId=p->second.id;
          }
        }
        if(p->first==score){
        //当 for 循环因 p==mp.begin()结束时可检查 mp.begin()
          if(p->second.id>maxId){
            maxp=p; maxId=p->second.id;
          }
        }
        cout << maxp->second.name << "␣"
             << maxp->second.id << "␣"
             << maxp->first << endl;
      }else{  //当 p==mp.begin()时
        cout << "Nobody" << endl;
      }
    }
  }
  return 0;
}
```

```
#include<iostream>
#include<cstdio>
#include<string>  //字符串处理
#include<map>
using namespace std;

struct Student {
  string name;
  int score;
};
Student st[5]={
  {"Jack",89},{"Tom",87},{"Alysa",87},
  {"Cindy",87},{"Micheal",98}
};
typedef map<string,int> MP;
int main(void){
  MP mp;
  for(int i=0;i<5;i++)
    mp.insert(make_pair(st[i].name,st[i].score));
  cout << mp["Jack"] << endl;  //->89
  mp["Jack"]=60;  //修改关键字为"Jack"的元素 second 为 60
  for(MP::iterator it=mp.begin();it!=mp.end();it++)
    cout << "(" << it->first << "," << it->second << ")";
  cout << endl;
  Student st;
  st.name="Jack"; st.score=99;
  pair<MP::iterator,bool> p=mp.insert(make_pair(st.name,st.score));
  if(p.second==1)  //如果插入成功
    cout << "(" << p.first->first << ","
    << p.first->second << ") inserted" << endl;
  else
    cout << "insertion failed" << endl;
  mp["Harry"]=78;  //mp 中原本不存在"Harry"，因此添加"Harry"与 78
  MP::iterator q=mp.find("Harry");
  cout << "(" << q->first << "," << q->second << ")" << endl;
  return 0;
}
```