C++程序设计

郑文立

2024年 秋季学期

第一章 绪论

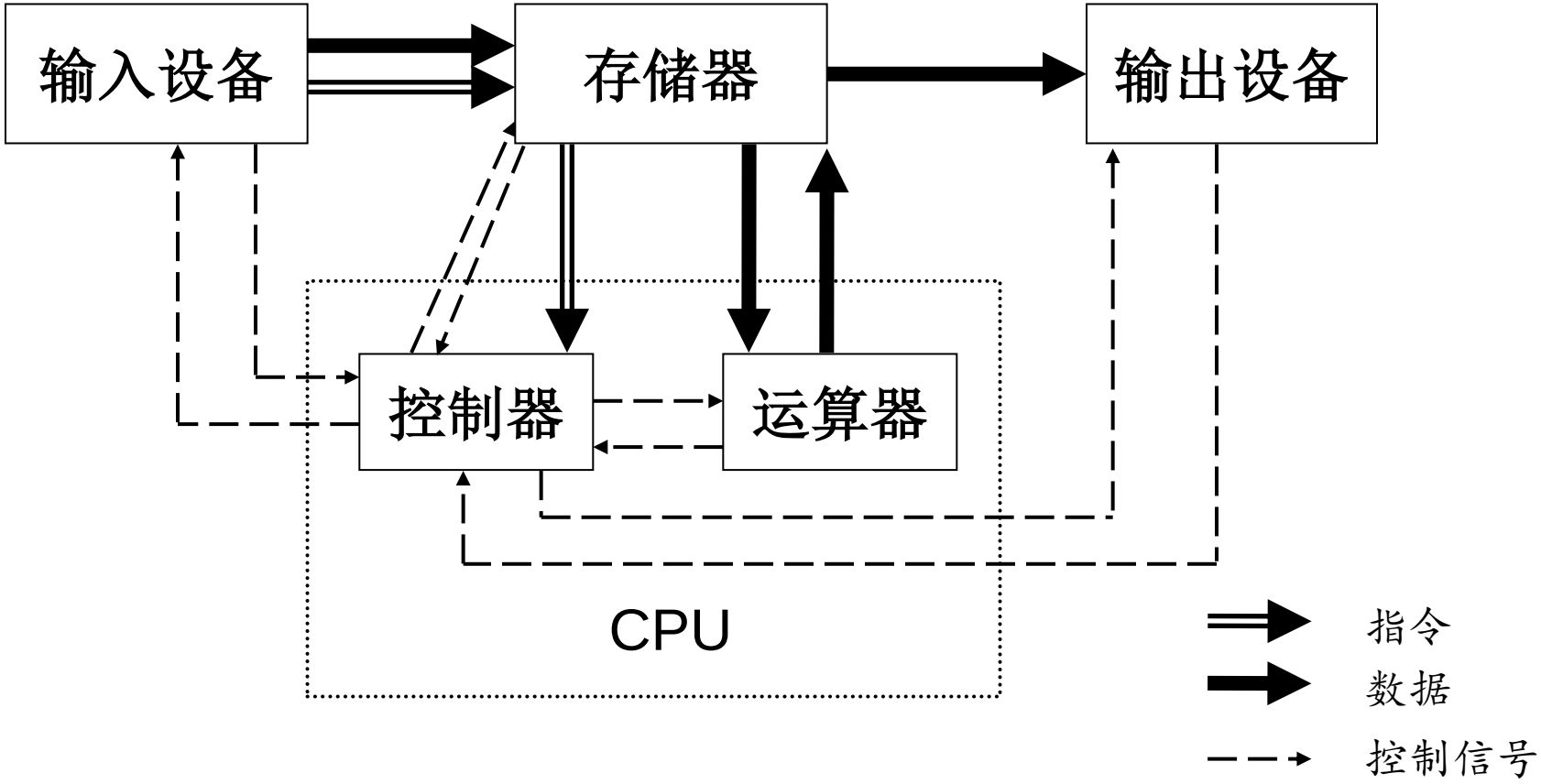
计算机的组成

- 冯诺依曼体系结构
- 内存编址
- 系统软件与应用软件

程序设计

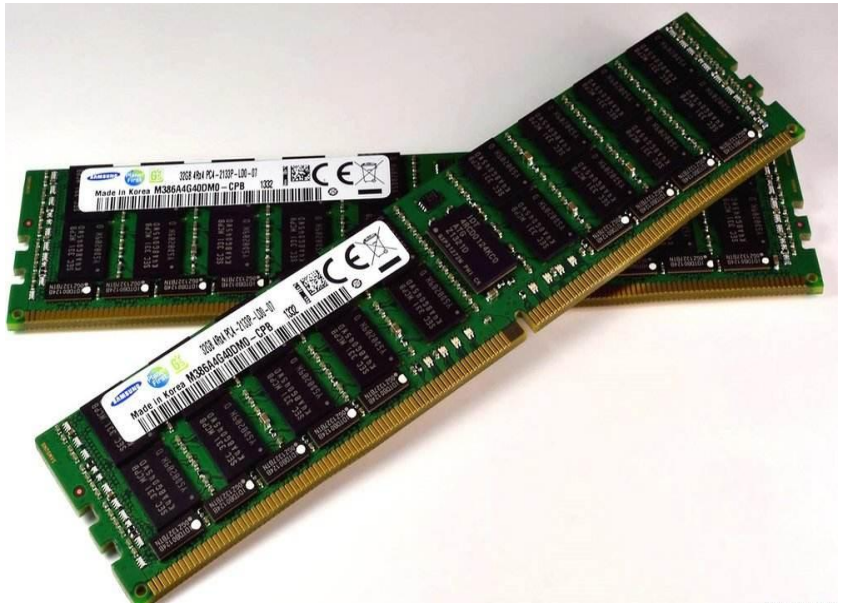
- 算法的特点与表示
- 编码（C++编程语言）
- 程序编译与执行过程

计算机硬件系统的组成



存储器

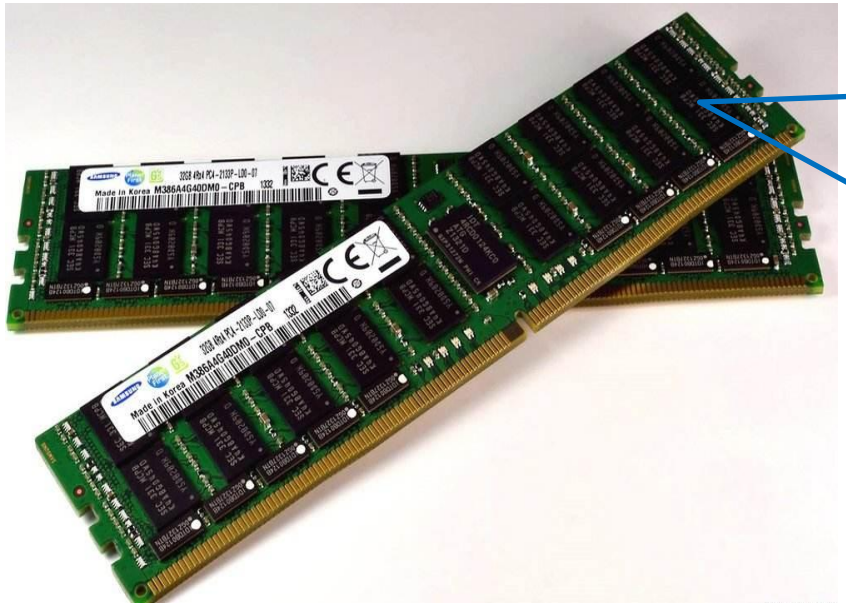
保存正在运行的程序指令和数据

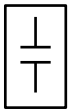


- 关机后，内存里的指令和数据全部丢失。
- 内存上的一个最小物理器件只能存储一个位（二进制），所以它的名称也叫做位（bit，缩写为b）。一般8个位组成一个字节（Byte，缩写为B），若干个字节组成一个字（word）。
- 在一般的机器中，**内存按字节编址**，内存大小也是按字节计量。

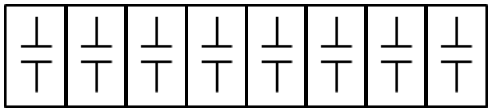
存储器

保存正在运行的程序指令和数据





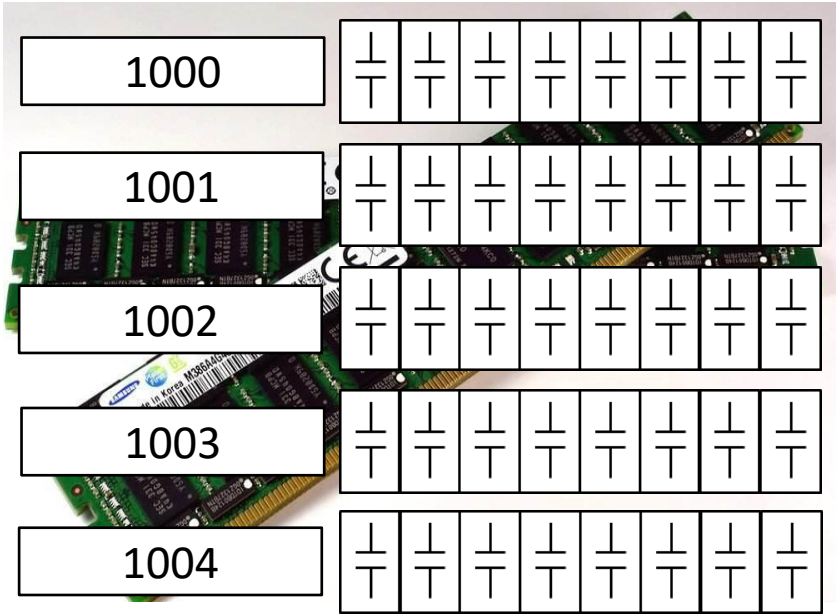
1个bit就是1个电容



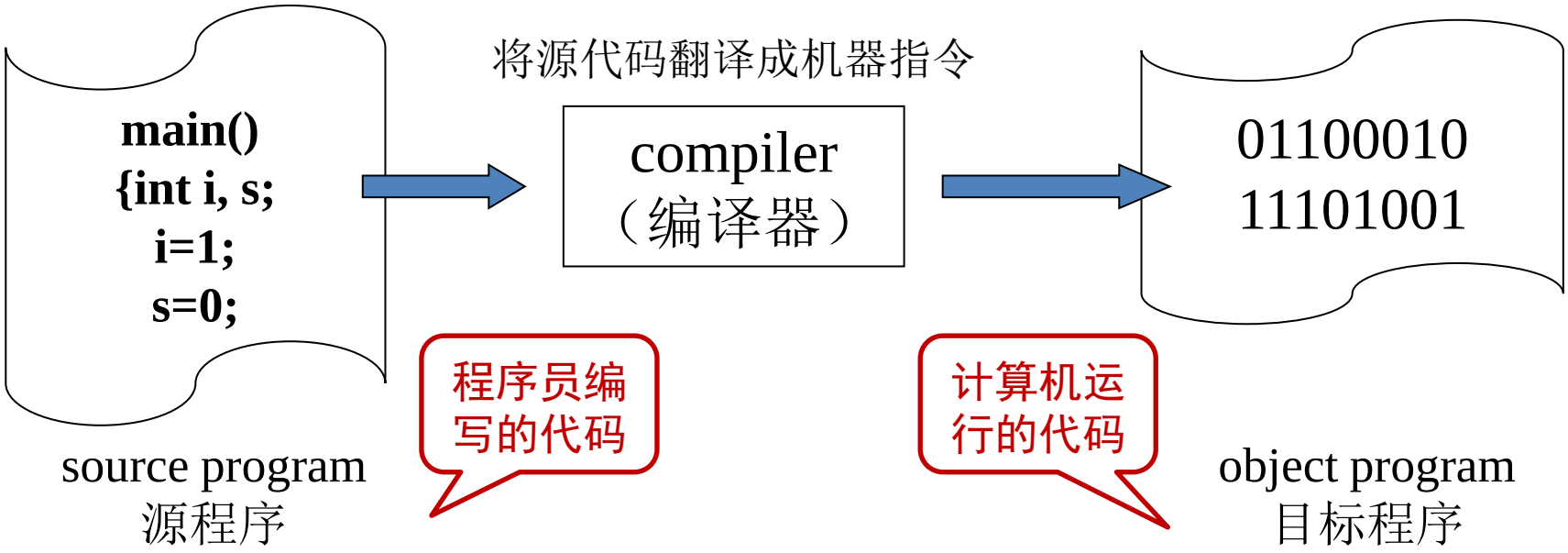
8个电容组成一个内存单元
（也就是一个字节）

存储器

保存正在运行的程序指令和数据

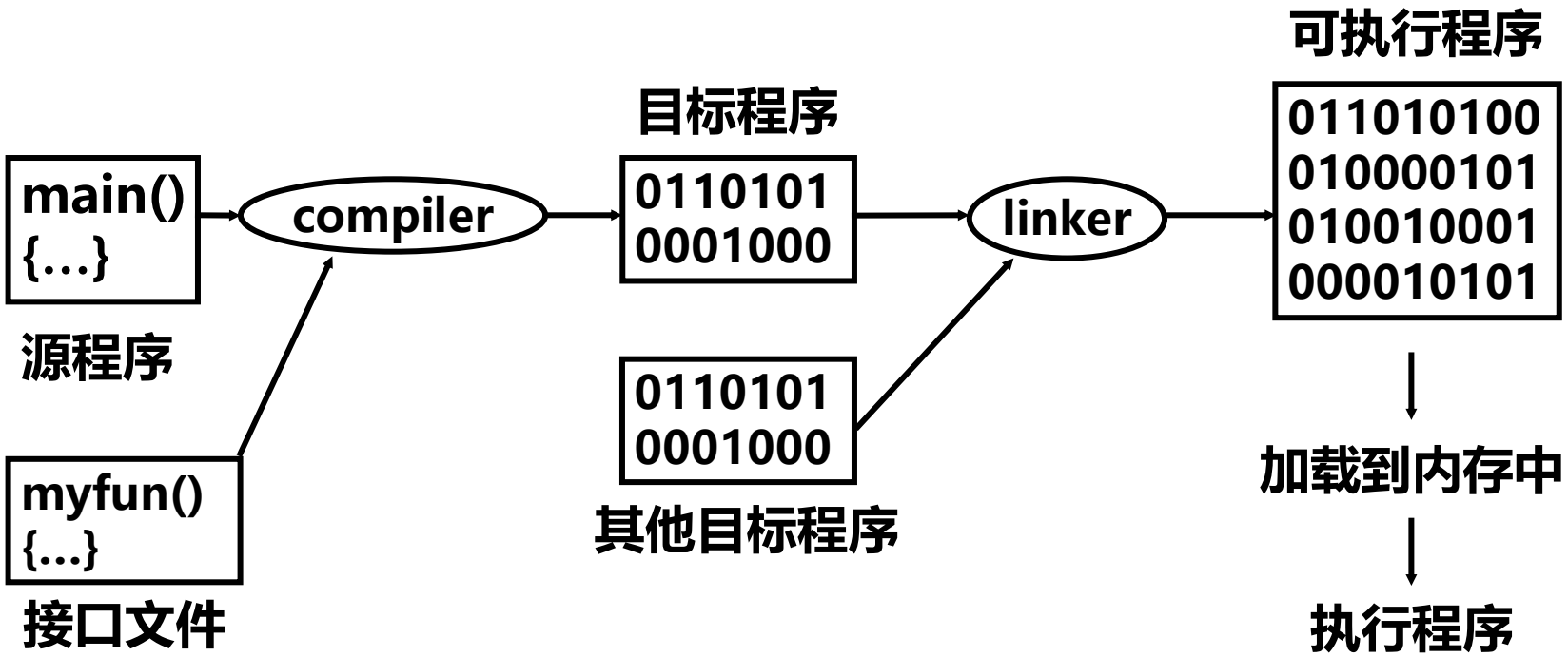


- 关机后，内存里的指令和数据全部丢失。
- 内存上的一个最小物理器件只能存储一个位（二进制），所以它的名称也叫做位（bit，缩写为b）。一般8个位组成一个字节（Byte，缩写为B），若干个字节组成一个字（word）。
- 在一般的机器中，**内存按字节编址**，内存大小也是按字节计量。



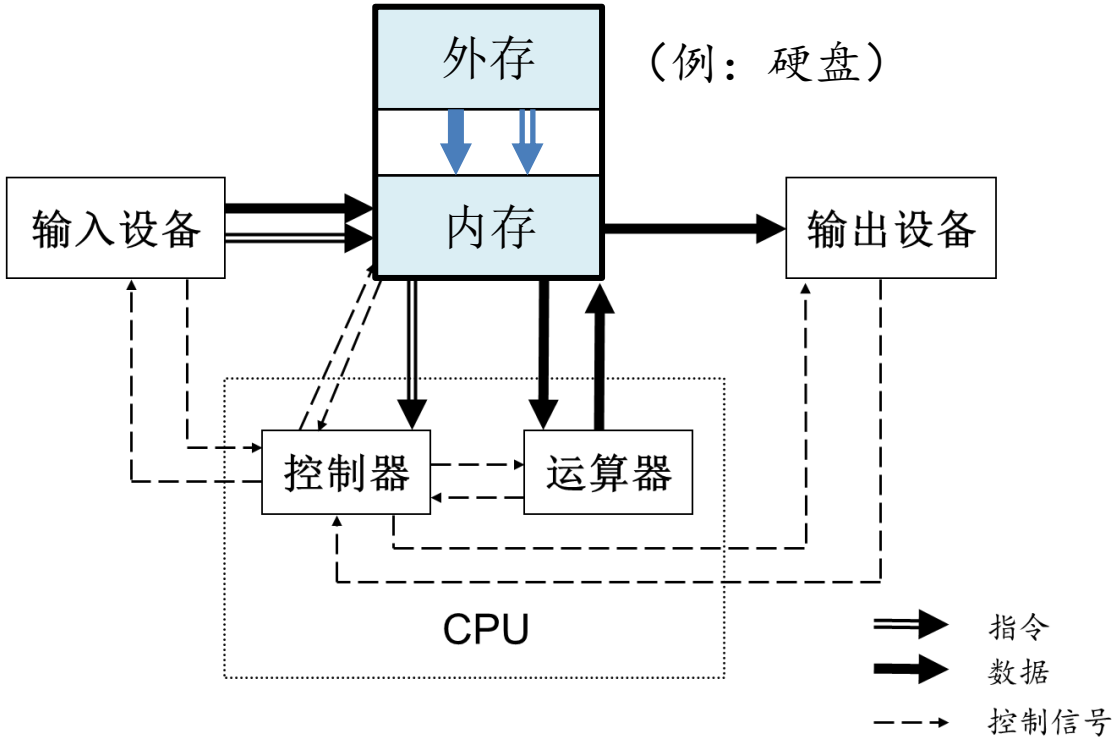
- 解释执行：开始运行后逐句翻译并执行
- 编译执行：全部翻译成机器指令后才能开始运行（例如C++）

程序编译与执行过程


main()
{…}
源程序
myfun()
{…}
接口文件
compiler
目标程序
0110101
0001000
0110101
0001000
其他目标程序
linker
可执行程序
011010100
010000101
010010001
000010101
加载到内存中
执行程序

加载

- 加载(Load)：为程序在内存中定位。
- 决定指令和数据的内存地址（在内存中的起始地址）



第二章
程序基本组成

- 第一个程序
- 第二个程序
- 变量定义
- 数据类型
- 算术表达式
- 赋值表达式
- 数据的输入输出
- 构思一个程序








变量赋初值

- 在C++中，变量定义只是给变量分配相应的空间。
- 可以在定义变量的同时给变量赋初值，格式为：

  类型名 变量名 = 初值；　或　类型名 变量名(初值)；

  如：int count = 0;　或　int count(0);

  都是定义整型变量count，并为它赋初值0。
- 如果没有赋值，则变量的值是随机的【可能不是0】。
- 数组元素同理【可能不是0】。
- 但静态变量和全局变量例外【是0】。

整型常量

- 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　　【开头加0】

  十六进制：0x123，0x3a2f　　【开头加0x】

- 一旦定义了一个整型变量，可以用一个整型常量给它赋值。

- 如 int a;

  a = 123; 或 a = 0123;　都是正确的

  但 cout << a;　输出的分别是 123 和 83（不会输出0123）。

字符类型

- 字符常量：‘A’或‘9’或‘+’等等，有单引号。常量的值是固定的，不能存储用户输入的数据。
- 字符变量：按照 char ch1, ch2; 这样的语法来定义，不加引号（所有变量名都不加引号）。变量的值是可变的，能存储用户输入的数据。
- 常量和变量比大小：　ch1 == ‘+’　或　ch2 > ‘A’
- 字符数组中的一个元素相当于一个字符变量。

字符的机内表示

- 字符参与计算时实际上是用其ASCII码的数值来计算。
- 所以，任何一个数字都可以通过 + ‘0’ 来转换为对应的数字字符。

  例如：char x = 6 + ‘0’；则 x 的值为 ‘6’。
- 反过来，‘6’ - ‘0’ 则得到 6。
- 注意只有数字（即小于10的自然数）才能这么做。
- 不存在像 ‘10’ 这样的字符。
- 类似的，char y = ‘A’ + 3；则 y 的值为 ‘D’。
- 注意， 6 + ‘0’ 表达式本身等于54， ‘A’ + 3 表达式本身等于68。

定义新的枚举类型

- 格式：enum 枚举类型名 {元素表};
- 定义一个表示一周中每天的名字的枚举类型：

  enum week {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
- 枚举类型的变量的定义：

  week weekday;
- 枚举类型变量的使用（如赋值）： weekday = Fri;

枚举类型的内部表示

- 在内部，枚举类型采用编码表示。
- 当定义enum week {Sun, Mon, Tue, Wed, Thu, Fri, Sat};时，默认用0代表Sun， 1代表Mon，…，6表示Sat
- C++语言的编译器也允许明确指出枚举类型的元素的内部表示。
  例如，希望从1而不是0开始编号，可以这样定义：
  enum week {Sun=1, Mon, Tue, Wed, Thu, Fri, Sat};
- 也可以从中间某一个开始重新指定，如：
  enum week {Sun, Mon, Tue=5, Wed, Thu, Fri, Sat};
  此时Sun默认为0，Mon默认为1，Wed为6，以此类推。

了解占用的内存量

- sizeof运算符用来了解某一类型或表达式占用多少字节的内存。
- sizeof运算符的用法：

  sizeof(类型名)　或　sizeof(表达式)
- 各种数据类型的单个变量的内存占用量（单位：字节）：

  sizeof(int)得4，sizeof(float)得4，sizeof(double)得8，

  sizeof(char)得1，sizeof(bool)得1。
- 一个数组的内存占用量（单位：字节）：

  已定义int a[5][8]；则sizeof(a)得160，sizeof(a[0])得32。

类型转换

- 当赋值运算符两边的类型不一致时，系统会将右边的值自动转换成左边的变量的类型，再赋给左边的变量。

```
int x = 1;
float y = 2.5;
x = y;
```

注意：寄存器中的值发生了转换，但变量y的类型和值都没有变化！

指令 1	操作：读取	运算器R5	y的内存地址：124812	
指令 2	操作：转换	运算器R4	运算器R5	float->int
指令 3	操作：写入	运算器R4	x的内存地址：124848	

自增、自减运算符

- 自增、自减运算符：++和--，相当于+=1和-=1，
- 它有两种用法：++k与k++，或--k与k--，但含义有所不同。
- 如：i = 3

```
++i ;          // i=4
i++ ;          // i=4
j = i++ ;      // i=4    j=3
j = ++i ;      // i=4    j=4
```

用户的响应

- 当程序执行到cin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。
- cin>>用**空白字符（空格、制表符和换行符）**分隔多个输入数据。

• 如：a为整型，d为double，则对应于cin >> a >> d，用户可输入：

12 13.2↙ 或 12（tab键）13.2↙ 或 12↙13.2↙

• 当输入缓冲区以分隔符开头时，cin>>先删去它，再读取数据。

• cin>>读取数据后，会把该数据之后的分隔符留在输入缓冲区里。

第三章
逻辑思维及
分支程序设计

- 关系表达式
- 逻辑表达式
- If 语句
- Switch语句

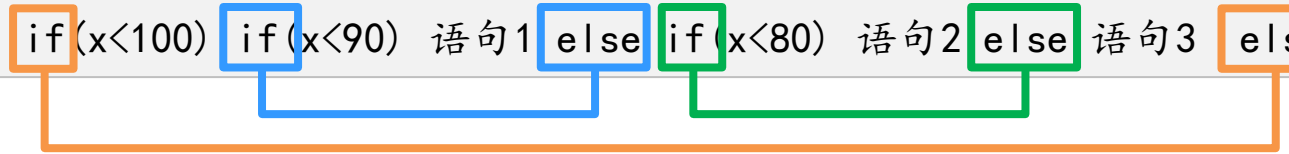
运算符的优先级

- 优先级： ! > 算术运算符 > 关系运算符 > && > || > = > ,
  - 逻辑运算符： ! && ||
  - 算术运算符： + - * / % （后三个优先级较高）
  - 关系运算符： > >= <= < == != （前四个优先级较高）
  - 赋值运算符： =
  - 逗号运算符： ,
- **遇事不决加括号（）**

If 语句的嵌套

- 当if语句的then子句和else子句也是if语句，称为if语句的嵌套。
- 因为if 语句可以没有else子句，if语句的嵌套可能产生歧义，如

```
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。

问号冒号表达式

- “？：”运算符：问号冒号运算符
- 作用：更加简练的用来表达条件执行的机制
- 形式：**(条件) ？ 表达式1 ： 表达式2**
- 例如语句：

```
max = (x > y) ? x : y;
cout << ( a ? “true” : “flase” ) << endl;
```

- 冒号两边的数据类型必须是相同或可以互相转化的。

switch语句

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
}
```

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。

符合哪一个，就从哪一个开始**执行到底**

break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。

有break就只执行一个

switch语句

case后的值必须是常量！

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

只能是整型、字符、枚举，不能是浮点型

执行过程：

当表达式值为常量1时，执行语句1；

当表达式值为常量2时，执行语句2；

⋮

当表达式值为常量n时，执行语句n；

否则，执行语句n+1。

第4章
循环控制

For循环

While循环

Do ... While循环

循环的中途退出

枚举法与贪婪法

Do ... While循环语句

- 格式： do 语句 while（表达式）
- 如将若干个输入数相加，直到输入0为止。

```
total = 0;
do
{   cout << “ ? ”;
    cin >> value ;
    total += value;
} while (value != 0); //注意末尾的分号
```

中途退出

- break语句：跳出循环。

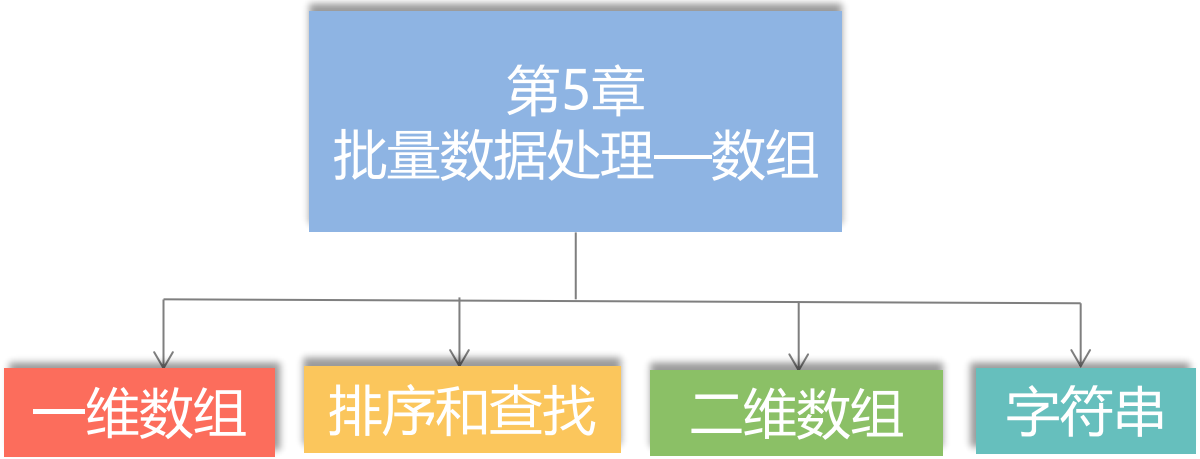
```
while (true) {
    提示用户并读入数据
    if (value==标志) break;
    根据数据作出处理
}
```

- continue语句：跳出当前循环周期，进入下一循环周期。
- 不要在循环体的末尾写continue（或else continue）。

For循环的进一步讨论

- For循环的三个表达式可以是任意表达式。
- 表达式1：在整个循环语句开始之前执行一次。
- 表达式2：在每个循环周期开始之前执行一次并判断结果的真假。
- 表达式3：在每个循环周期结束时执行一次。

- 如果在循环体中有continue，不会跳过表达式3
- 如果在循环体中有break，会跳过表达式3


第5章
批量数据处理—数组
一维数组
排序和查找
二维数组
字符串

一维数组

数组

- 数组是一组同类元素，它有两个特征：
  - 数组元素是有序的（内存地址有序且连续）
  - 数组元素是同类的
- 定义数组要定义三个基本内容：
  - 数组名字
  - 数组元素的类型
  - 数组的大小（即**元素个数，必须是常量**）

int n = 10;
n不是常量！

初始化

初始化表只能在定义语句中使用

- 定义数组的同时可以对数组初始化

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

- 如果初始化表的长度短于要被初始化的数组元素数目，那么剩余元素被初始化为0。
- 把数组的所有元素初始化为0是个好习惯。
- `char dst[10]; strcpy(dst, "ABC"); //剩余元素可能不为0`
- 带有初始化的数组可以不定义长度

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

数组在内存中

- 定义数组就是定义了一块连续的内存空间，空间的大小等于元素数乘以每个元素所占的空间大小。
- 数组元素依次存放在这块空间中。
- 不带[]的数组名（如int a[5];中的a）可看做存储该数组起始地址的常量。它可以被输出（如cout << a;），但不可被赋值。

N、(N+1)/2和$\log_2 N$的值

	顺序查找	二分查找
N	(N+1)/2	$\log_2 N$
10	5	3
100	50	7
1000	500	10
1,000,000	500,000	20
1,000,000,000	500,000,000	30

注：二分查找的前提是该组元素已经按顺序排列。

选择排序法

- 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1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;
     交换第lh个元素和第rh个元素的值;}
输出排好序的元素;
```

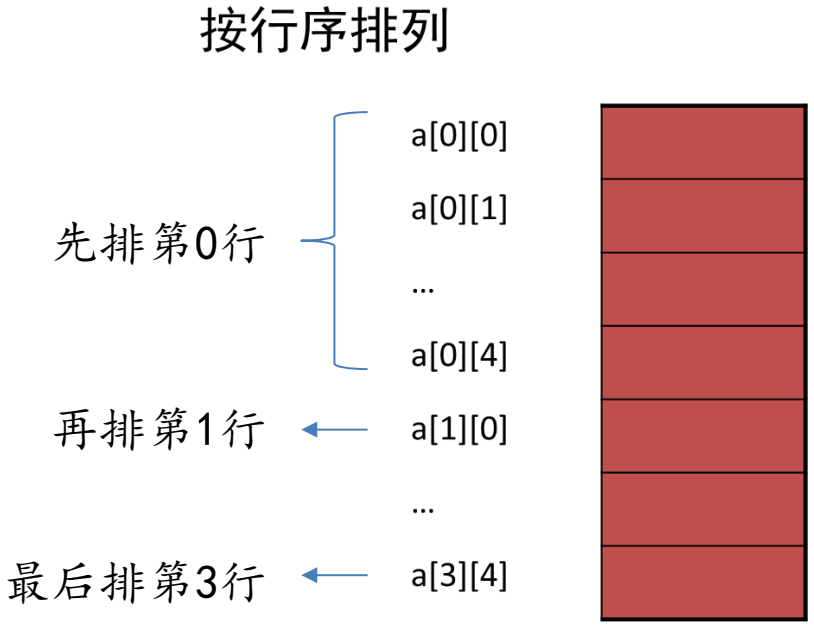
冒泡排序法

- 对数组元素进行扫描。第一遍扫描冒出一个最大的气泡，放入最后一个位置。然后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进行起泡，最大的泡放入元素n-i；
    if （没有发生过数据交换） break;
}
```

二维数组内存排列

- 二维数组：当只指定其中一维的下标x时（如对于int b[4][5];使用b[x]），则表示第x行在内存中的起始地址。同样可被输出，不可赋值。
- 二维数组的 b 和 b[0] 是相等的，但意义不同。



二维数组初始化

- 格式：

  类型说明 数组名[常量表达式1][常量表达式2] = {.....};

- 为每一行的部分元素赋初值

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

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 5 & 0 & 0 & 0 \end{bmatrix}$$

```
char name[3][10] = {“Tom”, “Kevin”, “Steven”};
```

字符串的储存

- 字符串的本质是一系列的有序字符，因此可以用一个字符数组来保存这组字符，用数组名表示这个字符串。
- 由于数组名是数组的起始地址，因此该字符串从该地址开始存储。但到哪里为止？C++用‘\0’表示字符串的结束。
- 字符串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（10个字母 + 1个逗号 +‘\0’）。

空字符串

- 不包含任何字符的字符串称为空字符串。
- 空字符串占用的空间为1个字节，存储‘\0’。
- 注意‘a’和“a”的区别
  - 前者是单个字符常量，后者是包含2个字符的字符串。
- 注意空白字符、空字符、空字符串的区别。

字符概念区分

	代码	ASCII码	含义
空字符	‘\0’	0	可表示字符串结尾
空格字符	‘ ’	32	空格这个字符，可简称为空格符
空白字符	N/A	N/A	空格符、换行符和制表符的统称，不包括空字符
字符零	‘0’	48	数字零这个字符

字符串的输入和输出

- 逐个字符的输入输出：这种做法和普通的数组操作一样。
  for（i=0; i<10; i++）cout << ch[i];
- 将整个字符串一次性地用cin和cout输入或输出。
- 要输出ch的内容，可直接用cout << ch;（输出的是字符串，不是这个字符数组在内存中的起始地址，输出到空字符为止）
- 通过函数cin.getline输入一整行，包含空白字符在内（但不包含getline的结束符）。

字符串的输入和输出

用cin>>输入

- 如定义了一个字符数组ch。
- 要输入一个字符串放在ch中，可直接用cin >> ch;
- 用cin>>输入时是以空格、换行或制表符作为结束符。因此无法输入包含空白字符的字符串。
- 在用cin>>输入时，要注意输入的字符串的长度不能超过数组的长度。注意，存入数组后，字符串的末尾有‘\0’。

字符串的三种输入方式

	每次读取的内容	对字符串的处理	适用场景
cin>>	空白字符（空格符、换行符、制表符）前的内容，**不包括空白字符**	把读取的字符都存储在字符数组里，并在读取的最后一个字符之后**添加一个空字符**，表示字符串结尾	读取不含空白字符的字符串（如一个单词）；读取其它类型的数据（如整数或实数、单个字符）
cin.get	一个任何字符（包括空白与非空白字符）	可通过每次读取一个字符、循环读取的方式来存储一串字符，但用户输入的回车依然是换行符，并**不会被处理成空字符**，因此严格意义上还不是一个字符串	适用于需要读取并存储空白字符的场景；适用于根据每个字符来依次处理的场景
cin.getline	结束符（默认是换行符）前的内容（例如一行），**包括结束符**	把读取的字符都存储在字符数组里，末尾的**结束符**（默认是换行符）会被**替换为空字符**，表示字符串结尾	适用于一次读取一句话（包含空格符的多个单词）的场景

cstring

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 （从头数到空字符为止，不算空字符）
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

cstring

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 （从头数到空字符为止，不算空字符）
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

cstring

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 （从头数到空字符为止，不算空字符）
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

cstring

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 （从头数到空字符为止，不算空字符）
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


第6章
过程封装—函数
函数定义
带默认值的函数
函数的使用
内联函数
变量的作用域
重载函数
变量的存储类别
函数模板
数组作为函数参数
递归函数

函数的声明

- 所有函数**在使用前必须被声明**。
- 函数的声明展示了函数的原型，它的形式为：

  返回类型　函数名（参数表）；

参数表中的参数说明之间用“，”分开，每个参数说明可以是类型，也可以是类型后面再接一个参数名。

如：int max(int, int);　　int max(int a, int b);

函数原型声明

```
#include <iostream>
int max(int a, int b); // 函数声明处要加分号
int main()
{
    int x, y;
    cin >> x >> y;
    cout << max(x + 5, y - 3);
    return 0;
}
int max(int a, int b)
{
    if (a > b) return(a); else return(b);
}
```

函数调用

函数定义

实际参数与形式参数

```
#include <iostream>

int difference(int, int);

int main() {
    int x, y;
    cin >> x >> y; // 输入 5    3
    cout << difference(x, y) << endl;
    cout << x << endl; // x的值仍是5，因为是值传递
    return 0;
}

int difference(int a, int b) {
   a = a - b; // a的值从5变成2
   return(a); // 返回2
}
```

数组参数的传递机制

- 数组作为参数传递时，传递的是数组的首地址，即地址传递。因此，在被调函数中对形参数组的修改，就是对主调函数中实参数组的修改（**改变主调环境**）。
- 数组作为参数时只能传递数组的起始地址，因此形式参数中不需要像定义数组那样说明数组的大小，真正的元素个数要通过另一个参数来表示。

```
void PrintIntegerArray(int arr[ ], int n);
void ReverseIntegerArray(int arr[ ], int n);
```

数组作为函数的参数

数组形参与实参

- 数组作为形参：

```
void myfun (int arr[ ])        // 正确的写法
void myfun (int arr[10])       // 不规范的写法，但不产生错误
void myfun (arr[ ])            // 错误的写法
void myfun (int arr)           // 这是单个变量，不是数组
void myfun (int [ ])           // 在声明中正确，在定义中不合理
```

- 数组作为实参：

```
myfun (arr)                    // 正确的写法
myfun (int arr[ ])             // 错误的写法
myfun (arr[10])                // 这是单个元素，且下标越界
```

数组作为函数的参数

数组形参与实参

数组的定义	用于实参的方式	对应的形参
T a[5];	a	T arr[]
	a[i]	T x
T b[5][4];	b	T arr[][4]
	b[i]	T arr[]
	b[i][j]	T x
T c[5][4][3];	c	T arr[][4][3]
	c[i]	T arr[][3]
	c[i][j]	T arr[]
	c[i][j][k]	T x

数组形参与实参

- 数组作为函数的参数时，
- 如果实参是常量，那么形参也应该是常量

  例：已知定义const int arr[5]，并调用myfun (arr)，

  那么void myfun (const int list[])中的const不可少。
- 如果形参是常量，实参可以是常量或变量

  例：已知声明void myfun (const int list[ ])，

  那么调用myfun (arr)中的arr是常量数组或变量数组皆可。

带默认值的函数 – 错误用法

```
#include <iostream>

int difference(int a, int b);

int main()
{
    int x;
    cin >> x;
    cout << difference(x) << endl;
    return 0;
}

int difference(int a, int b = 10)
{
    return( a - b );
}
```

检查函数声明，发现参数表不匹配！

带默认值的函数 – 正确用法

```
#include <iostream>

int difference(int a, int b = 10);

int main()
{
    int x;
    cin >> x; // 输入 16
    cout << difference(x) << endl; // 输出 6
    cout << difference(x, 7) << endl; // 输出 9
    return 0;
}

int difference(int a, int b)
{
    return( a - b );
}
```

内联函数

- 目的：减少函数调用的开销（但代码本身会膨胀）
- 作用：函数代码复制到程序中

```
#include<iostream>
inline float cube(float s) { return s*s*s; } //定义必须前置
int main()
{       float side;
        cin >> side;
        cout << cube(side) << endl;
        return 0;
}
```

函数重载

- 给参数个数不同、参数类型不同或两者兼而有之的两个以上的函数取相同的函数名。
- 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);
```

函数重载的实现

- 由编译器确定某一次函数调用到底是调用了哪一个具体的函数。这个过程称之为绑定（binding，又称为联编或捆绑）。
- 编译器首先会为这一组重载函数中的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**确定具体调用的是哪个函数，将这个函数的内部函数名取代重载的函数名。

函数模板的定义

- 一般的定义形式

```
template<类型形式参数表>
返回类型  函数名（形式参数表）
{
   //函数定义语句
}
```

- 类型形式参数表中，数据类型参数（此处为T）需加前缀“class”

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

```
template<class T>
void func(T arr[], int n);
```

函数模板的声明

- 例：

```
template<class T>
bool fun1 (T a, int b); // 声明和定义的函数头保持一致

// 在其它函数中调用了 fun1

template<class T>
bool fun1 (T a, int b) { // 声明和定义的开头都要加template那行
   return a>b ? true : false;
}
```

复合语句

- 用{ }括起来的一组语句称为复合语句。
- 在复合语句中可以定义变量，例如：

```
int x = 1;
{
    int y = 2;
    x = x + y;
}
cout << x << endl; // 不能 cout << y << endl;
```

- 复合语句之内定义的变量在该复合语句之外没有意义。
- 在for循环控制行中定义的变量，也只在循环内部有意义。

局部变量和全局变量

- 局部变量：在程序块内定义的变量称为局部变量，即使是main函数中定义的变量也是局部的。
  - 作用范围：从定义的位置到程序块结束。

- 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：从定义位置到文件结束。

变量的存储类别

存储类型

- 在C语言和C++中，每个变量有两个属性：
  - 数据类型：变量所存储的数据类型
  - 存储类别：变量所存储的位置
- 标准的变量定义：

  存储类别　数据类型　变量名;
- 存储类别：
  - 自动变量：auto
  - 外部变量：extern
  - 寄存器变量：register
  - 静态变量：static

变量的存储类别

extern

- 声明一个不在本变量作用范围内的全局变量。
- 如： extern int num; //num为某个全局变量。
- extern的用途:
  - 在某函数中引用了一个定义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。
- 调用另一个文件中定义的函数不需要extern，只需要先声明。

外部变量示例

```
//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;
}
```

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

int x = 1; //全局变量的定义

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

变量的存储类别

static

- 在整个程序的运行期间都存在的变量。
- 两类静态变量：
  - 静态的局部变量
  - 静态的全局变量

静态变量的使用

- 未被程序员初始化的**静态变量**都由系统初始化为0。
- 局部静态变量是在编译时分配内存空间的，在第一次执行初始化语句时赋初值。当运行时重复调用此函数时，不重复赋初始值。
- 虽然局部静态变量在函数调用结束后仍然存在，但其他函数不能引用它。

静态的局部变量

- 允许局部变量保存它的原有值，以便再次进入该程序块时还可以使用此值。

```
int f (int a) {
    static int b = 3;
    b = b+1;
    return (a+b);
}

main() {
    int a = 2, i;
    for (i=0; i<3; ++i)
        cout << f(a);
}
```

再次进入该函数时，跳过该语句。

运行结果为：

6 7 8

静态的全局变量

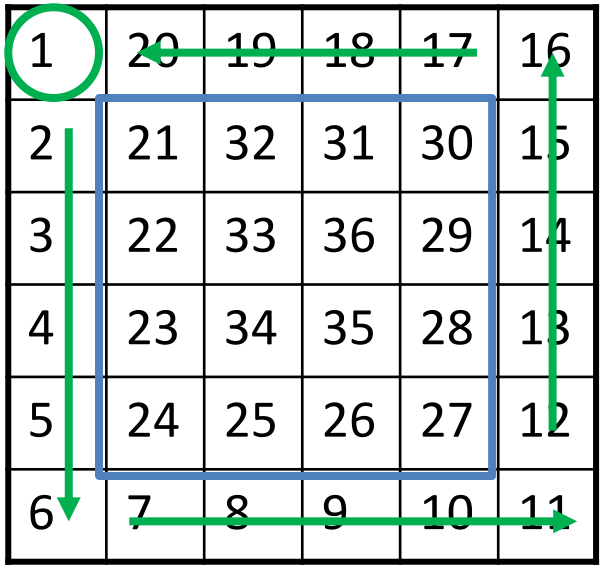
- 用static定义的全局变量不能被其它源文件用extern引用。
- static还可以用在函数定义或声明中，使该函数只能被用于本源文件中，其它源文件不能调用此函数。
- 用途：不同源文件中的同名同参函数不会冲突。

递归条件

- 递归调用是有条件的，否则会死循环：
  - 必须有递归终止的条件；
  - 如果在递归终止时找到了整个问题的解，那么有时需要在回溯过程中将该解逐层返回到最顶层。
- 对递归函数的每次调用都需要新的内存空间。
  - 由于很多调用的活动都是同时进行的，操作系统可能会耗尽可用的内存，在处理过大的n时产生溢出问题。

递归函数 – 蛇阵

```
void fill( int number, int begin, int size) {
    int i, row = begin, col = begin;
    if (size == 0) return;
    if (size == 1) { p[begin][begin] = number; return; } //终止条件
    p[row][col] = number; ++number;
    for (i=0; i<size-1; ++i)
    { ++row; p[row][col] = number; ++number; }
    for (i=0; i<size-1; ++i)
    { ++col; p[row][col] = number; ++number; }
    for (i=0; i<size-1; ++i)
    { --row; p[row][col] = number; ++number; }
    for (i=0; i<size-2; ++i)
    { --col; p[row][col] = number; ++number; }
    fill( number, begin+1, size-2 ); // 递归调用
}
```



第7章
间接访问—指针

- 指针的概念
- 字符串再讨论
- 指针与函数
- 指针运算与数组
- 引用类型与函数
- 动态内存分配
- 指针数组与多级指针