1.2.3

* c语言需要被编译才能运行，需要

编辑器

编译器

或者 IDE（集成开发环境）

* c语言不在乎空格 回车 缩进

1.3.2

* 程序框架， 每次写都要有（具体见7.2.4）

#in1clude

int main ()

{

return 0;

}

每句结尾分号易忽略

句内逗号可连接多个变量

2.1.2 变量

* 形式： =

e.g int price = 0, amount = 4;

* 变量有个名字---->标识符

标识符只能由数字、字母、下划线组成，数字不可以出现在第一个位置

* c语言关键字/保留字不可用做标识符

e.g auto,break,case,char,const,continue,default,do,double,else,enum,extern,float,for,goto,if,int,long, register,returrshort,signed,sizeof,static,struct,switch,typedef,union,unsigned,void,volatile,while,inline,restrict

2.1.3 赋值与初始化（同python）

* 特别注意初始化：所有变量在第一次被使用（出现在赋值运算符右侧）之前应该被赋值一次

C99任意位置定义 ANSI C 只能在开头定义

2.1.4 变量输入

int a；

int b；

scanf\_s("%d %d", &a,&b) ;用户输入，两个d间用空格，在scanf\_s 中输入要加&，而且””内部有其他符号的话在终端输入时要输入

printf("%d + %d = %d\n", a, b, a+b); 输出，%d前面加数字表示位数，比如02表示输出两位数，不足两位前面补0

2.1.5 常量与变量

int整数， const修饰符， 加在int前，给变量加上一个const（不变的）属性，一旦初始化，不可改变

强制类型转换：变量 = (类型)变量

2.1.6 浮点数

* 浮点数与整数的除法要注意
* 改法 12-->12.0
* 或者在定义是用double
* 输入输出时的初始化
  + 整数

int ;

printf("%d", ...);

scanf\_s("%d", ...);

* + 浮点数

double

printf("%f", ...)

scanf\_s("%lf", ...)

* + 字符

char

printf("%c", ...)

scanf\_s("%c", ...)

2.2.2 运算符优先等级

* 赋值也是运算
* c语言没有运算幂次的函数，需要用循环实现
* 所有的关系运算符优先等级低于算术运算符，但高于赋值运算
* 判断相等==和不等!=的优先级低

2.2.3 变量交换

插入临时变量

int a = 4 b = 6;

int t;

t = a;

a = b;

b = t;

2.2.4 复合赋值

* += -= \*= /= 均先将右侧式子算完再与左侧运算
* ++ -- 算子必须是变量，表示每次+或者- 1 前缀后缀都可以但printf结果不同
* a++打印出计算之前的a ++a打印出计算之后的a

3.0.1 PAT做题

1. 看样例的数据和题目一样重要
2. 严禁交互输出
3. 不检查输入
4. 遵守输出格式

3.1.1 if

* 格式：

if (条件成立){

操作;

}

e.g

if (im < 0){

im = 60 +im;

ih -=1;

} else {

im = im\*2;

}

* 或者(建议{}一定加否则会出错)

if (条件成立)

(一句语句)；

e.g

if (hours > standard)

pay = standard\*rate + (hours - standard)\*(rate\*1.5);

else if (hours < standard)

pay = hours \*rate;

3.1.2 判断条件

bool

true为1 false为0

3.2.1嵌套if

int a, b, c;

scanf\_s("%d %d %d", &a, &b, &c);

int max = 0;

if (a > b){

if (a > c){

max = a;

} else {

max = c;

}

} else {

if (b > c){

max = b;

} else {

max = c;

}

}

3.2.2 级联的if else

int f

if (x < 0){

f = -1;

} else if (x == 0) {

f == 0;

} else {

f = 2\*x;

}

if (exp1)

stl;

else if (exp2)

stl2;

else

st3;

3.2.3 常见错误

正确

if (age > 60){

salary = salary\*12

printf("%d", salary)

}

1. 忘记{}

if (age > 60)

salary = salary\*12

printf("%d", salary)

1. if后加;

if (age > 60);{

salary = salary\*1.2

printf("%d", salary)

}

1. 不分 == 与 =
2. 乱用else

3.2.4 多路分支

* 格式

switch(变量或表达式,只能是整数或者const) {

case 条件(常量):

语句(可以为空);

break;

case 条件(常量):

语句;

break;

default: 结尾

语句

}

* 如：

if ( type==1 )

printf("你好");

else if ( type==2)

printf("早上好");

else if ( type==3 )

printf("晚上好");

else if ( type==4 )

printf("再见");

else

printf("啊，什么啊?");

* 改为：

switch ( type ) {

case 1:

printf("你好");

break;

case 2:

printf("早上好");

break;

case 3:

printf("晚上好");

break;

case 4:

printf("再见");

break;

default:

printf("啊，什么啊?“);

}

4.1.1 while循环

先判断条件，再进入循环体

while (循环条件){

循环体语句；

}

4.1.2 do-while

先进循环体，再做判断

循环体中要有改变条件的语句(否则跑飞)

do

{

} while()； 结尾分号

int i = 1;

while (i<=n){

fact \*= i;

i++;

}

5.1.1 for

for中三个条件有一个即可

for (初始动作，循环继续的条件，每一轮循环要做的事){}

在C99中可在for中写（int i=1;......）即在for中定义变量

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

fact \*= i;

}

5.1.2循环计算与选择

循环头尾

三种循环for ，while ， do-while

* 如果有固定次数，用for
* 如果必须执行一次，用do\_while （有效处理0问题）
* 其他情况用while

5.2.3 从嵌套循环中跳出

两种跳出循环方式

* 连续break
* goto+指针

指针在结尾定义，定义完后加：

6.0.5 猜数游戏

6.1.1C数据类型

* C语言的变量必须
  + 在使用前定义并且
  + 确定类型
* C以后的语言两个方向
  + C++/Java更强调类型，对类型检查严格
  + Javascript/Python/PHP不强调类型，有的甚至不需要事先定义

类型安全

1. 支持强类型的观点认为明确的类型有助于尽早发现程序中的简单错误
2. 反对强类型的观点认为过于强调类型迫使程序员面对底层、实现而非事务逻辑
3. 总的来说,早期语言强调类型，面向底层的语言强调类型

（蓝色的为C99类型）

* 整数
  + char、short、int、long、long long
* 浮点数
  + float、double、long double
* 逻辑
  + bool
* 指针
* 自定义类型
* 类型名称:int、long、double
* 输入输出时的格式化: %d、%ld、%lf
* 所表达的数的范围:char < short < int < float <double
* 内存中所占据的大小:1个字节到16个字节·内存中的表达形式:二进制数（补码)、编码

sizeof(数据类型或者变量) 给出其在内存中所占用的字节数

sizeof是静态运算符，在sizeof括号内做运算是无效的，但不同类型运算可能变

char一个字节 int四个字节

6.1.3整数的运算（尤其是负数）

1. 二进制负数三种方案
2. 仿照十进制，有一个特殊的标志标识负数
3. 取中间的数为0， 如1000000表示0，比它小的是负数，比它大的是正数
4. 补码(拿补码和原码可以加出一个溢出的“零”)
   * 考虑-1，我们希望-1+1—>0。如何能做到?

0 --> 00000000 解释：在计算机中为8个字节

1 --> 00000001 11111111+00000001后变为100000000

11111111 +00000001 -->100000000 计算机自动将1省略就变成了0

* + 因为0-1-->-1，所以，-1 =

(1)00000000 - 00000001 --> 11111111

故 11111111被当作纯二进制看待时，是255，被当作补码看待时是-1

* + 同理，对于 -a，其补码就是 0-a，实际是2^n - a，n是这种类型的位数

6.1.4整数范围

计算机中的符号数有三种表示方法，即原码、反码和补码。

三种表示方法均有符号位和数值位两部分，符号位都是用0表示“正”，用1表示“负”，而数值位，三种表示方法各不相同。

1. 正整数的补码是其二进制表示，与原码相同；
2. 求负整数的补码，将其对应正数二进制表示所有位取反（包括符号位，0变1，1变0）后加1；

负数-1， （此处，假设是8位二进制表示）

对应正数的原码：0000 0001；

取反： 1111 1110；

加1： 1111 1111；

最终，-1是以1111 1111的形式进行存储的。

数的范围

对于一个字节(8位)，可以表达的是:

00000000 -11111111

其中

00000000-->0

11111111 ~10000000--> -1 ~ -128（补码）

00000001~01111111 --> 1 ~ 127（纯二进制）

* char: 1字节: -128~ 127
* short: 2字节: -32768~32767
* int: 取决于编译器(CPU)，通常的意义是"1个字" -2^32-1 ~ 2^32-1 - 1
* long:4字节
* long long:8字节

unsigned

unsigned是无符号的意思，也就是说如果你的编译系统给int分配的存储单元的长度是2个字节的话，有符号的int 取值范围是-32768（即2^15）——32767（即2^15-1），而无符号的unsigned int就是0-65535（2^16-1）

6.1.5整数的格式化

格式化输入输出整数

* 只有两种形式
  + %d：int
  + %u：unsigned
  + %ld：long long
  + %lu：unsigned long long

处理8进制， 16进制

8进制,16进制只是把数字表达为字符串，与内部如何表达数字无关

* 一个以0开始的数字字面量是 8进制 输出时 %o 或者 0%o
* 一个以0x开始的数字字面量是 16进制 输出时 %x 或者 0x%x

6.1.7浮点类型

double比float更准确，范围更大

%e为科学计数法

.(数字)%f 数字表示小数点后位数

6.1.8浮点数的范围与精度

一般直接用double

12.0/0.0为正无穷， 12/0报错

printf输出inf表示超过范围的浮点数：正负无穷

printf输出nan表示不存在的浮点数

float浮点数没有精确度

带小数点的字面量是double而不是float

float与要用f或者F后缀表明身份

float类型无法比较，只能做减法再与一个很小的数字作比较

6.1.9字符类型

char即是整数也是字符

* 用单引号表示的字符字面量：'a', '1'
* "也是一个字符
* printf和scanf里用%c来输入输出字符

6.1.10逃逸字符

用来表达无法印出来的控制字符或特殊字符，它由一个反斜杠\开头，后面跟上另一个字符，这两个字符合起来，组成了一个字符

printf("请分别输入身高的英尺和英寸，"

"如输入\"5 7\"表示5英尺7英寸: ");

/t 每行的固定位置

一个\t使得输出从下一个制表位开始

用\t才能使得上下两行对齐

6.1.11类型转换

当运算符的两边出现不一致的类型时，会自动转换成较大的类型

大的意思是能表达的数的范围更大

char —> short—> int —> long —> long long

int —> float—> double

对于printf，任何小于int的类型会被转换成int; float会被转换成double

但是scanf不会，要输入short，需要%hd

强制类型转换

要把一个量强制转换成另一个类型(通常是较小的类型)，

需要:(类型)值

比如:(int)10.2 (short)32

注意这时候的安全性，小的变量不总能表达大的量(short)32768

强制类型转换的优先级高于四则运算

6.2.1逻辑类型

#include

之后就可以用bool和true、false

bool量输入输出时先赋值给一个变量后再输出，true为1， false为0

#include

#include

int main()

{

bool b = 6>5;

bool t = true;

return 0;

}

6.2.2 逻辑运算

结果只有0或者1

逻辑运算是关系运算或者逻辑运算的结果

短路

逻辑运算是自左向右进行的，如果左边的结果已经能够决定结果了，就不会做右边的计算

a==6 && b==1

a==6 && b+=1

对于&&，左边是false时就不做右边了

对于||，左边是true时就不做右边了

切记：不把赋值或者复合赋值组合进表达式

表达区间的方法：

正确：x>4 && x<6;

错误：4<x<6;

大小写：

c>='A' && c='Z';

取反：

!age < 20; 即为age>=20

优先级

！> && > ||

6.2.3条件运算与逗号运算

count = (count > 20) ? count -l0 : count+10;

条件 ? 条件满足时的操作 : 条件不满足时的操作

if（count>20）{

count = count - 10;

} else{

count = count + 10;

}

逗号用来连接两个表达式，并以其右边的表达式的值作为它的结果。

逗号的优先级是所有的运算符中最低的，所以它两边的表达式会先计算;

逗号的组合关系是自左向右，所以左边的表达式会先计算，而右边的表达式的值就留下来作为逗号运算的结果。

一般的用于for

for ( i=0,j=l0; i

7.1.2函数定义与调用

定义

头尾大括号必须有

函数名后的圆括号标志函数，必须有

调用

函数名（参数值）

（）起到了表示函数调用的作用

即使没有参数也需要（）

有参数，必须给出正确的数量与顺序，多个参数用，连接

这些值会被按照顺序依次用来初始化函数中的参数

7.1.3从函数中返回

return

* return停止函数的执行，并送回一个值
  + return表达式;
* 一个函数里可以出现多个return语句（虽然不符合单一出口理念）

有返回值的函数

* 可以赋值给变量
* 可以再传递给函数
* 甚至可以丢弃

没有返回值的函数

* void函数名(参数表)
* 不能使用带值的return
  + 可以没有return
* 调用的时候不能做返回值的赋值

7.2.2告诉编译器函数

直接将函数写在int main（）前

#include

void sum(int begin, int end)

{

int i;

int sum = 0;

for (i = begin; i <= end; i++) {

sum += i;

}

printf("%d到%d的和是%d\n", begin, end, sum);

}

int main()

{

sum(1, 10);

sum(20, 30);

sum(35, 45);

return 0;

}

将函数头复制到int mian（）前再加个分号

#include

void sum(int begin,int end ); 函数原型声明。此语句中可不写“begin”“end”

int main( )

{

sum(1,10);

sum(20,30);

sum (35,45);

return 0;

}

void sum(int begin, int end) 函数头（函数定义）

{

int i;

int sum =0;

for ( i=begin; i<=end; i++ ) {

sum +=i;

}

printf("%d到%d的和是%d\n", begin, end, sum) ;

}

7.2.2参数

传给函数的参数可以是表达式的结果包括

* 字面量
* 变量
* 函数的返回值
* 计算的结果

int a, b, c;

a = 5; b= 6

c = max(10, 12);

c = max(a, b);

c = max(c, 23);

c = max(max(23, 45), a);

c = max(23+45, a);

* 调用函数时给的值与参数的类型不匹配是C语言传统上最大的漏洞
* 编译器总是悄悄替你把类型转换好，但是这很可能不是你所期望的
* 后续的语言，C++/Java在这方面很严格

C语言在调用函数时，永远只能传值给函数，对main里的无影响

7.2.3本地变量：定义在函数内部的变量

函数的每次运行，就产生一个独立的变量空间，

在这个空间中的变量，是函数的这次运行所独有的，称作本地变量

定义在函数内部的变量就是本地变量

参数也是本地变量

变量的生存期和作用域

生存期:什么时候这个变量开始出现了，到什么时候它消亡了

作用域:在（代码的）什么范围内可以访问这个变量(这个变量可以起作用)

对于本地变量，这两个问题的答案是统一的:大括号内又名块

本地变量的规则

* 本地变量是定义在块内的
  + 它可以是定义在函数的块内
  + 也可以定义在语句的块内
  + 甚至可以随便拉一对大括号来定义变量
* 程序运行进入这个块之前，其中的变量不存在，离开这个块，其中的变量就消失了
* 块外面定义的变量在里面仍然有效
* 块里面定义了和外面同名的变量则掩盖了外面的不能在一个块内定义同名的变量
* 本地变量不会被默认初始化
* 参数在进入函数的时候被初始化了

7.2.4

函数没有参数时：void f(void);

void f()； 在传统C中，表示f的参数表位置，并不表示没有参数

逗号对于函数

调用函数时的圆括号内的逗号时标点符号，不是运算符号

f(a, b) 两个参数a， b

f((a, b)) 一个参数b

C语言不允许函数嵌套定义，不允许在函数内定义另一个函数，但可以用另一个函数

#include

int main() 也是函数，括号内可以写void也可以不写

{

return 0; 对检查有用，返回0是正确的，返回非零是错误的

}

8.1.1数组

定义

int number[100]

类型 变量名称[元素数量]

元素数量必须是整数

赋值 从零开始

number[4] = 67

数组名[小于数组大小的数字] = 符合数组类型的值

8.1.2 数组的使用

其中所有元素类型相同

一旦创建不可改变大小

数组中的元素在内存中是连续依次排列的

对于数组的使用也必须有初始化

一个int的数组

10个单元: a[0],a[1],.....a[9]

]每个单元就是一个int类型的变量

可以出现在赋值的左边或右边: a[2] = a[1]+6;

\*在赋值左边的叫做左值

单元

数组的每个单元就是数组类型的一个变量

使用数组时放在[]中的数字叫做下表或者索引，下标从0开始计数

有效的下标范围 [0，数组的大小-1]

编译器和运行环境都不会检查数组下标是否越界，无论是对数组单元做读还是写

一旦程序运行，越界的数组访问可能造成问题，导致程序崩溃

segmentation fault

可能运气好，没造成严重的后果

8.2.1数组的运算

数组的集成初始化

直接用大括号给出数组的所有元素的初始值

不需要给出数组的大小，编译器替你数

int a[ ] = {1, 2,3 ,4 5, 6, 5, 7}；

用[n]在初始化数据中给出定位

没有定位的数据接在前面的位置后面

其他位置的值补零

也可以不给出数组大小，让编译器算

特别适合初始数据稀疏的数组

int a [10] = {[0] = 2，[2] = 3，6 ,}；

数组的大小

sizeof给出整个数组所占据的内容的大小，单位是字节

sizeof(a)/sizeof(a[0])

sizeof(a[0])给出数组中单个元素的大小，于是相除就得到了数组的单元个数

这样的代码，一旦修改数组中初始的数据，不需要修改遍历的代码

数组变量不可被赋值 ：

int a[ ] = {1,2 ,3, 4, 46, }

int b[ ] = a

要把一个数组的所有元素交给另一个数组，必须采用遍历

for ( i = 0; i<length;i++){

b[i] = a[i]

}

遍历数组

通常都是使用for循环，让循环变量i从0到<数组的长度，

这样循环体内最大的i正好是数组最大的有效下标

常见错误是︰

循环结束条件是<=数组长度，

离开循环后，继续用i的值来做数组元素的下标

数组作为函数的参数时:

往往必须用另一个参数传入数组大小

不能在[]中给出数组的大小

不能再利用sizeof来计算数组的元素个数!

8.2.3二维数组

int a[3][5]:通常历届为一个三行五列的矩阵

a[0][0] a[0][1] a[0][2] a[0][3] a[0][4

a[1][0] a[1][1] a[1][2] a[1][3] a[1][4]

a[2][0] a[2][1] a[2][2] a[2][3] a[2][41

二维数组的遍历

for ( i=0; i

for ( j=0; j

a[i][j] = i\*j;

}

}

二维数组的初始化

int a[ ] [5] = {

{1,2,3,4,3}

{2,3,5,4,6}

};

列数是必须给出的，行数可以由编译器来数

每行一个{}，内部逗号分隔

最后的逗号可以存在，有古老的传统如果省略，表示补零

也可以用定位（\*C99 ONLY)

9.1.1&取地址

scanf(“%d”，&i);里的&

获得变量的地址，它的操作数必须是变量

int i; printf("%x”",&i);

地址的大小是否与int相同取决于编译器

int i; printf(“%p",&i);

9.1.2指针

一般的在指针定义时赋值为0，便于动态分配

int i;

int\* p = &i;

int\* p,q;

int \*p, q;

仅仅把星号加给p，与中间的空格无关

q还是一个普通的int

变量的值是内存的地址

普通变量的值是实际的值

指针变量的值是具有实际值的变量的地址

void f(int \*p);

在被调用的时候得到了某个变量的地址:

int i=0; f(&i);

在函数里面可以通过这个指针访问外面的这个i

\*是一个单目运算符,用来访问指针的值所表示的地址上的变量

可以做右值也可以做左值

int k = \*p;

\*p = k+1;

9.1.3指针的应用

1. 交换两个变量的值
2. 函数返回多个值，某些值只能通过指针返回
   1. 传入的参数是实际上是需要保存待会的结果的变量
3. 函数返回运算的状态，结果通过指针返回

常用的套路是让函数返回特殊的不属于有效范围内的值来表示出错:

-1或0(在文件操作会看到大量的例子)

但是当任何数值都是有效的可能结果时，就得分开返回了

(状态用函数的return返回，值用指针参数返回)

常见错误：

定义了指针变量，还没有指向任何人变量，就开始使用指针

指针是一个地址，指针变量是存放地址的变量

指针的一般用法

1. int a = 10; int \*p; p = &a; 结果：\*p = 10，a = 10

2. int a = 10; int \*p = &a; 结果：\*p = 10，a = 10

3. int a = 10,k; int \*p; \*p = &k; \*p = i; 结果：\*p = 10，a = 10，k = 10

必须先对p赋值，否则无法使用

之后对\*p的运算就是对a的运算

有的时候会赋

空指针如 int \*p = 0;

空类型指针如 char \*p1; void \*p2; p1 = (char\*)p2;

a\_pointer->地址 \*a\_pointer->a\_pointer指向的地址中的值

i\_pointer = &i = &(\*i\_pointer)

i = \*i\_pointer = \*(&i)

9.1.4指针与数组

函数参数标中的数组实际上是指针

sizeof(a) == sizeof(int\*)

可以用数组的运算符进行运算

原因

数组变量是特殊的指针 特别的a == &a[0]

数组变量本身表达地址，所以

int a[10]; int \*p = a; 无需用&取地址

数组中的单元表达的是变量，需要用&取地址

运算符[]可以对数组做，也可以对指针做：

p[0] 即为 a[0]

运算符\*可以对指针做，也可以对数组做

\*a = 25

数组变量是const指针，不能被赋值

int a [ ] 即为 int \*const a=

9.1.5指针与const

指针是const

表示一旦得到了某个变量的地址，不能再指向其他变量

int \*constq = &i; //q是const

\*q = 25; //OK

q++; //ERROR

\*p是指针不可通过const指针改变i

const int \*p = &i;

\*p= 26; // ERROR! (\*p)是const

i = 26; //OK

p =&j; //OK

int i;

const int\* p1 = &i; //通过指针不可修改

int const\* p2 = &i;

int const p3 = &i; //指针不可修改

判断哪个被const了的标志是const在\*的前面还是后面

可以把一个非const值转换为const

void f(const int\* x);

int a = 15;

f(&a); //OK

const int b = a;

f(&b); // ok

b = a + 1; //Error !

当要传递的参数的类型比地址大的时候,这是常用的手段:既能用比较少的李节数传递值给参数，又能避免函数对外面的变量的修改

const数组

const int a[]= { l,2,3,4,5,6,};

数组变量已经是const的指针了，

这里的const表明数组的每个单元都是const int

所以必须通过初始化进行赋值

保护数组值

因为把数组传入函数时传递的是地址，所以那个函数内部可以修改数组的值

为了保护数组不被函数破坏，可以设置参数为const

int sum(const int a[], int length);

9.2.1指针运算

指针+1是加一个sizeof(类型)

给指针+1是让指针指向下一个变量

若指针不是指向一片连续分配的空间

如数组，则这种运算没有意义

指针可以+ - ++ -- += -=

不可以乘除，可以比较

两个指针可以相减：

结果是两地址差除以sizeof(类型)

\*p++

取出p所指的那个数据来，完事之后顺便把p移到下一个位置去

\*的优先级虽然高，但是没有++高

常用于数组类的连续空间操作

在某些CPU上，这可以直接被翻译成一条汇编指令

0地址

当然你的内存中有0地址，但是0地址通常是个不能随便碰的地址

所以你的指针不应该具有0值

因此可以用0地址来表示特殊的事情:

返回的指针是无效的

指针没有被真正初始化(先初始化为0)

NULL是一个预定定义的符号，表示0地址有的编译器不愿意你用0来表示0地址（NONE）

指针的类型

无论指向什么类型，所有的指针的大小都是一样的，因为都是地址

但是指向不同类型的指针是不能直接互相赋值的

这是为了避免用错指针

指针的类型转换

void表示不知道指向什么东西的指针计算时与char**相同（但不相通)指针也可以转换类型int**p = &i; void**q = (void**)p;

这并没有改变p所指的变量的类型，而是让后人用不同的眼光通过p看它所指的变量我不再当你是int啦，我认为你就是个void!

9.2.2动态内存分配

#include

void\* malloc(sizeof(size)); 借空间

向malloc申请的空间的大小是以字节为单位的

返回的结果是void\*，需要类型转换为自己需要的类型

(int\*)malloc(n\*sizeof(int))

free() 还空间，只能还申请来的空间的首地址

#include

#include // 用malloc前要引入

int main( void)

{

int number;

int\* a;

int i;

printf("输入数量:");

scanf("%d",&number) ;

// int a [number] ; C99做法

a = ( int\*)malloc( number\*sizeof(int) );

for ( i=0; i

scanf( "%d", &a[i] );

}

for ( i=number-1; i>=0; i-- ) {

printf("%d ", a[i] );

}

free(a); 前面用了malloc后面要free

return 0;

}

10.1.1字符串

以0(整数0）结尾的一串字符

0或'\0'是一样的，但是和'0'不同

0标志字符串的结束，但它不是字符串的一部分

计算字符串长度的时候不包含这个0

字符串以数组的形式存在，以数组或指针的形式访问更多的是以指针的形式

string.h 里有很多处理字符串的函数

C语言字符串是以字符数组的形态存在的

不能用运算符对字符串做运算

通过数组的方式可以遍历字符串

唯一特殊的地方是字符串字面量可以用来初始化字符数组，用" "

C标准库提供了一系列字符串函数

10.1.2字符串变量

" "会被编译器变成一个字符数组放在某处，这个数组的长度是内容字母数加1，结尾还有表示结束的0

char \*str = "Hello";

char word[]=“Hello";

char line[10] =“Hello";

char\* s = "Hello，world ! " ;

s是一个指针，初始化为指向一个字符串常量

由于这个常量所在的地方，所以实际上s是const char\* s ，

但是由于历史的原因，编译器接受不带const的写法

但是试图对s所指的字符串做写入会导致严重的后果

如果需要修改字符串，应该用数组

char s[ ] = "Hello, world";

指针还是数组

char \*str = "Hello";

char word[] ="Hello";

数组:这个字符串在这里

作为本地变量空间自动被回收

指针:这个字符串不知道在哪里

处理参数

动态分配空间

如果要构造一个字符串—>数组

如果要处理一个字符串—>指针

字符串可以是char\*的形式

char\*不一定是字符串

本意是指向字符的指针，可能指向的是字符的数组类似int\*

只有它所指向的祖父数组有结尾的0，才能说它所指的是字符串

10.1.3字符串输入输出

char string[8];

scanf("%s" , string);

printf("%s" , string);

scanf\_s读入一个单词(到空格、tab或回车为止)

%与s中间输入数字表示读入的字符数， 多余的字符丢掉或者交给下一个scanf

10.1.4字符串数组

char \*a[ ]; 指向另一个位置的变量

int main(int argc, char cionst \*argv[])

argv[0]是命令本身

当使用Unix的符号链接时，反映符号链接的名字

10.2.1单字符的输入输出

int putchar(int c);

向标准输出写一个字符

返回写了几个字符，EOF(-1）表示写失败

int getchar(void);

从标准输入读入一个字符

返回类型是int是为了返回EOF(-1)

Windows—>Ctrl-Z

Unix——>Ctrl-D

字符串函数#include

10.2.2strlen

size\_t strlen(const char \*s);

返回字符串的长度不包括结尾的0

还可以写函数

#include

#include

size\_t mylen(const char\* s)

{

int cnt = 0;

int idx = 0;

while (s [idx]!= '\0') {

idx++;

cnt++;

}

return cnt;

}

int main(int argc, char const \*argv[])

{

char line[] = "Hello" ;

printf( "strlen=%luln", mylen( line));

printf( "sizeof=%lu\n", sizeof ( line) ) ;

return 0;

}

10.2.3strcmp比较

int strcmp(const char \*s1, const char \*s2);

比较两个字符串，返回:

0:s1==s2

1:s1>s2

-1:s1<s2

int index = 0

while(s1[index] == s2[index] && s1[index] != '\0'){

index++

}

return s1[index] - s2[index]

wnile (\*s1 == \*s2 && \*s1 != 0){

s1++;

s2++;

}

return \*s1 -\*s2;

10.2.4strcpy拷贝

char \* strcpy(char \*restrict dst, const char \*restrict src);

把src的字符串拷贝到dst

restrict表明src和dst不重叠(C99)

返回dst

为了能链起代码来

#include

#include

//数组

char\* mycpy(chark dst,const char\* src)

{

int idx = 0;

while (src[idx]) {

dst[idx] = src[idx];idx++;

}

dst[idx] = '\0';

return dst;

}

//指针

char \*ret = dst

while (\*dst++=\*src++;)

\*dst = '\0';

return rest;

}

int main(int argc, char const \*argv [])

{

char s1[] = "abc" ;

char s2[] = "abc" ;

strcpy(s1,s2);return 0;

}

复制一个字符串

char dst = (char)malloc(strlen(src)+1);

strcpy(dst,src);

10.2.5strcat

char \* strcat(char \*restrict s1, const char \*restricts2);

把s2拷贝到s1的后面，接成一个长的字符串

返回s1

s1必须具有足够的空间

防止空间不足的安全版本

char \* strncpy(char \*restrict dst, const char \*restrictsrc,size\_t n);

char \* strncat(char \*restrict s1, const char \*restricts2,size\_t n);

int strncmp(const char \*s1, const char \*s2,size\_t n);

10.2.6字符串中搜索

char \* strchr(const char \*s, int c);

char \* strrchr(const char \*s, int c);

返回NULL表示没有找到

#include

#include

int main(int argc, char const \*argv [])

{

char s [] = "hello";

char \*p = strchr(s, 'l');

p = strchr(p+1, 'l') //从第二个l

printf("%s \n",p);

return 0;

}

字符串中找字符串（忽略大小写）

char \* strstr(const char \*s1, const char \*s2);

char \* strcasestr(const char \*s1, const char \*s2);

将l之后的字符取出

#include

#include

#include

int main(int argc, char const \*argv [])

{

char s[] = "hello";

char \*p = strchr(s, 'l');

char t = ( char)malloc(strlen(p)+1);

strcpy(t, p);

printf("9%sin", t);

free(t);

return 0;

}

将l之前的字符取出

#include

#include

#include

int main(int argc, char const \*argv [])

{

char s[] = "hello";

char \*p = strchr(s, 'l');

char c =\*p;

\*p = 'l0';

char t = (char)malloc(strlen(s)+1);

strcpy(t, s);

printf ( "%sin" , t);

free(t);

return 0;

}

11.1.1枚举

用枚举而不是定义独立的const int变量

常用于switch case

enum （枚举类型名字）{name1， name2，......};

枚举类型名字不常用，要用的是大括号里的名i，因为它们就是常量符号，类型为int， 值依次从0到n

枚举量可以作为值

枚举类型可以跟上enmu作为类型

但是实际上是以整数来作内部计算和外部输入输出的

enum {color red , yellow, green};

void f(enmu color c)

套路:自动计数的枚举

#include

enum COLOR {RED，YELLOW，GREEN，NumCOLORS};

int main(int argc, char const \*argv [])

{

int color = -1;

char \*ColorNames [NumCOLORS]={"red" , "yellow" , "green",};

char \*colorName = NULL;

printf("输入你喜欢的颜色的代码:");

scanf(""%d"" ,&color) ;

if ( color>=0 && color < NumCOLoRS ) {

colorName = ColorNames [color];

}else {

colorName= "unknown" ;

}

printf("你喜欢的颜色是%s\n"，colorName);

return 0;

)

定义时可以

enum COLOR {RED=1,YELLOw，GREEN=5，NumCOLORS};

虽然枚举类型可以当作类型使用，但是实际上很少(不好)用

如果有意义上排比的名字，用枚举比const int方便

枚举比宏(macro）好，因为枚举有int类型

11.2.1结构类型

对于第一和第三种形式，都声明了结构point。但是第二种形式没有声明point，只是定义了两个变量

struct point {

int x;

int y;

};

struct point pl,p2;

pl和p2都是point里面有x和y的值

struct {

int x;

int y;

}pl, p2;

pl和p2都是一种无名结构，里面有x和y

struct point {

int x;

int y;

} pl,p2;

pl和p2都是point里面有x和y的值t

结构成员

* 结构和数组有点像
* 数组用运算符和下标访问其成员
  + a[0]=l0;
* 结构用.运算符和名字访问其成员
  + today.day
  + student.firstName
  + pl.x
  + pl.y

结构运算

要访问整个结构，直接用结构变量的名字对于整个结构,可以做赋值、取地址，也可以传递给函数参数

pl = (struct point){5,l0}; //相当于pl.x = 5;

pl.y = l0;

pl= p2； //相当于pl.x = p2.x; pl.y = p2.y;

#include

struct date {int month;int day;int year;}; 结构类型//忘记分号报错

一般在函数外部声明结构这样可被多个函数使用

int main(int argc, char const \*argv [])

{ 结构变量

struct date today;

赋值三种方式

today.month = 07; today.day = 31; today.year = 2014;

struct date today = {07,31,2014};

struct date thismonth = {.month=7, .year=2014}; 未赋值的day为0

printf( ""Today 's date is %i-%i-%i. \n",

today.year,today.month,today.day);

return 0;

}

11.2.2结构与函数

整个结构可以作为参数传给函数

这时候是在函数内新建一个结构变量，并复制调用者的结构的值

也可以返回一个结构

没有直接的方式可以依次scanf\_s一个结构

在函数外结构改变值函数内部的不变

之前的方案,把一个结构传入了函数，然后在函数中操作，但是没有返回回去

问题在于传入函数的是外面那个结构的克隆体，而不是指针

传入结构和传入数组是不同的

在这个输入函数中，完全可以创建一个临时的结构变量，然后把这个结构返回给调用者

用->表示指针所指的结构变量中的成员

11.2.3结构中的结构

结构中的变量可以是基础变量也可以是结构

嵌套结构

struct point {

int x;

int y;

};

struct rectangle {

struct point ptl;

struct point pt2;};

如果有变量

struct rectangle r;

就可以有:

r.ptl.x、 r.ptl.y,

r.pt2.x和r.pt2.y

如果有变量定义:

struct rectangle r,\*rp;rp = &r; ,

那么下面的四种形式是等价的:

r.ptl.x

rp->ptl.x

(r.ptl).x

(rp->ptl).x

但是没有rp->ptl->x(因为ptl不是指针)

11.2.3类型定义

自定义数据类型(typedef)

C语言提供了一个叫做typedef 的功能来声明一个已有的数据类型的新名字。比如:

typedef int Length;

使得Length 成为 int类型的别名。

这样，Length这个名字就可以代替int出现在变量定义和参数声明的地方了:

Length a, b, len ;

Length numbers[l0];

声明新的类型的名字

新的名字是某种类型的别名

改善了程序的可读性

typedef long int64\_t; //重载已有的类型名字，新名字的含义更清晰具有可移植性

typedef struct ADate {

int month;

int day;

int year;

}Date; //简化了复杂的名字

int64\_t i = 100000000000;

Date d = { 9, 1, 2005};

11.3.2联合union

union AnElt{

int i;

char c;

}elt1, elt2;

elt1.i =4;

elt2.c = 'a' ;

elt2.i = 0xDEADBEEF;

存储

所有的成员共享一个空间

同一时间只有一个成员是有效的

union的大小是其最大的成员

初始化

对第一个成员做初始化

12.1.1全局变量

* 定义在函数外面的变量
* 全局变量具有全局的生存期和作用域（生存期，作用域长）
  + 与任何函数无关
  + 在任何函数中都可以使用

12.2.2静态本地变量

* 在本地变量定义时加上static修饰符就成为静态本地变量
* 当函数离开的时候，静态本地变量会继续存在并保持其值
* 静态本地变量的初始化只会在第一次进入这个函数时做，以后进入函数时会保持上次离开时的值
* 实际上是特殊的全局变量
* 他们位于相同的内存区域
* 具有全局的生存期，函数内的局部作用域
* static是指局部作用域（本地可访问）

12.1.3返回值指针的函数

返回本地变量的地址是危险的

返回全局变量或者静态本地变量的地址是安全的

返回再函数内malloc（动态内存变量）的内存是安全的，但容易产生问题

最好的做法是返回传入的指针

tips

不要使用全局变量来在函数间传递参数和结果

尽量避免使用全局变量

丰田汽车的案子

（使用全局变量和静态本地变量的函数是线程不安全的）

12.2.1宏定义

宏没有类型检查

#编译预处理

#define

注意没有结尾的分号，因为不是C的语句

名字必须是一个单词，值可以是各种东西

在C语言的编译器开始编译之前，编译预处理程序(cpp）会把程序中的名字换成值

完全的文本替换

gcc -save-temps

宏

如果一个宏的值中有其他的宏的名字，也是会被替换的

如果一个宏的值超过一行，最后一行之前的行末需要加 \

宏的值后面出现的注释不会被当作宏的值的一部分

12.2.2带参数的宏

牺牲空间以换取效率

宏可以带参数

带参数的宏的原则

一切都要括号

整个值要括号

参数出现的每个地方都要括号

#define RADTODEG(x)((x)\*57.29578)

可以带多个参数

#define MIN(a,b) ((a)>(b)?(b):(a))

也可以组合（嵌套）使用其他宏

在大型程序的代码中使用非常普遍

可以非常复杂，如“产生"函数

在#和##这两个运算符的帮助下

存在中西方文化差异

部分宏会被inline函数替代

12.3.1多个源代码文件

main()内部代码过长适合分为几个函数

一个源代码文件太长适合分成几个文件

两个读了的源

代码文件不能编译形成可执行文件

在DEV C++中新建一个项目，让后把几个源代码问价加入

对于项目，DVE C++的便宜会把一个项目中的所有源代码文件都变以后链接起来

有的IDE有分开的编译和构建两个按钮，前者是对单个源代码文件编译，后者是对整个项目做链接

一个.c文件是一个编译单元

编译器每次编译只处理一个编译单元

12.3.2头文件

那函数原型放到一个头文件（.h）中，在需要调用这个函数的源代码文件（.c）中#include这个头文件，就能让编译器在编译的时候知道函数的原型

#include

#include是一个编译预处理指令，和宏一样，在编译之前就处理了

它把那个文件的全部文本内容原封不动地插入到它所在的地方

所以也不是一定要在.c文件的最前面#include

“”还是<>（一般的自己的.h文件用“”，系统的用<>）

#include有两种形式来指出要插入的文件

“"要求编译器首先在当前目录(.c文件所在的目录)寻找这个文件，如果没有，到编译器指定的目录去找

<>让编译器只在指定的目录去找

编译器自己知道自己的标准库的头文件在哪里

环境变量和编译器命令行参数也可以指定寻找头文件的目录

#include的误区

#include不是用来引入库的

stdio.h里只有printf的原型，printf的代码在另外的地方，某个.lib(Windows)或.a(Unix)中

现在的C语言编译器默认会引入所有的标准库

#include 只是为了让编译器知道printf函数的原型，保证你调用时给出的参数值是正确的类型

在使用和定义这个函数的地方dou'ying'gadouyinggai#include此头文件

一般的做法是任何.c都有对应的同名的.h，把所有的对外公开的函数的原型和全局变量的声明都放进去

不对外公开的函数

在函数前面加上static就使得它成为只能在所在的编译单元中被使用的函数

在全局变量前面加上static就使得它成为只能在所在的编译单元中被使用的全局变量

12.3.3声明

extern int i;是变量的声明

int i;是变量的定义

声明和定义

声明是不产生代码的东西（编译器将变量记住）

* 函数原型
* 变量声明
* 结构声明
* 宏声明
* 枚举声明
* 类型声明
* inline函数

定义是产生代码的东西

只有声明可以被放在头文件中

是规则不是法律

否则会造成一个项目中多个编译单元里有重名的实体

\*某些编译器允许几个编译单元中存在同名的函数，

或者用weak修饰符来强调这种存在

同一个编译单元里，同名的结构不能被重复声明

如果你的头文件里有结构的声明，很难这个头文件不会在一个编译单元里被#include多次

所以需要“标准头文件结构”