头文件&源文件

2016年9月22日　　20:49

头文件

- .h文件
- 为了多个源文件可以共享一些资源，将这些共享资源放在一个头文件中，通过include指令读取
- 头文件中应该包含函数原型
- 包含头文件：

```
#ifndef BOOLEAN_H
#define BOOLEAN_H
...
#endif
```

源文件

- .c文件
- 必须包含一个名为main的函数
- 函数定义放在源文件中，在需要调用此函数的其他文件中放置声明

2016年9月9日 20:09

printf

2016年9月10日　　9:15

printf(格式串,表达式1, 表达式2, …) ;

格式串：普通字符、转换说明（以%开头）、转义序列

%d　把int型值以十进制数字输出
%i　把int型值以8,16进制数字输出
*******读写无符号数是使用u,o,x代替d*******
%u　把unsigned int型值以十进制数字输出
%o　把unsigned int型值以8进制数字输出
%x　把unsigned int型值以16进制数字输出
*******读写短整数时在d,u,o,x前面+h ********
%hd
%hu
%ho
%hx
*******读写长整数时在d,u,o,x前面+l ********
%ld
%lu
%lo
%lx
*******读写长整数时在d,u,o,x前面+ll********
%lld
%llu
%llo
%llx

%f　显示float型数
%e　以科学计数法形式显示float型数
%g　以十进制或指数形式显示float型数
******读写double类型在f, e, g前面+l********
%lf
%le
%lg
******读写long double类型在f, e, g前面+L********
%Lf
%Le
%Lg

%m.pf　显示p位小数，总长度m位的float型变量（右对齐）

%-m.pf 显示p位小数，总长度m位的float型变量（左对齐）
%c 允许对单个字符进行读取

实现表格化输出的方法：printf在指定宽度内将输出右对齐的特性！

scanf

2016年9月10日　　9:42

scanf("转换说明",&变量名1,&变量名2,…);

- 在寻找用户输入的数值型数据的起始位置时，scanf会自动忽略空白字符（空格符、水平和垂直制表符、换页符、换行符）。
- 当格式转换符是%c时，空白字符会被当做是普通字符读入。若要忽略任意个空白字符，则要写成：scanf(" %c",&x);
- 如果某一项不能成功读入，scanf不再查看格式串中的剩余部分，立即返回。

算术运算符

2016年9月10日　　10:15

运算符%

1. 要求操作数是整数
2. 右操作数不能是0
3. i%j 结果的符号和i相同

运算符/

1. 当两个操作数都是整数时，运算符/会丢掉分数部分来“截取”结果
2. 右操作数不能是0
3. -i/j　结果总是向0截取

运算符的结合性

左结合：一元算术运算符（+和-）

- -+j　等价于 -（+i）

赋值运算符

2016年9月10日　　10:37

简单赋值v=e

- e可以是变量、常量、表达式
- 如果v和e的类型不同，赋值时把e的值转换成v的类型
- 运算符=具有右结合性
    - i=j=k=9　等价于　i=(j=(k=9))

赋值运算符要求：

- 左操作数必须是左值，只有变量才能作为左值
- 不在表达式中使用赋值运算符=

复合赋值

- v+=e
- v-=e
- v*=e
- v/=e　　v除以e，结果存储到v中
- v%=e　　v除以e取余数，求余的结果存储到v中

自增&减运算符

2016年9月10日　　14:23

++i　　等价于　i=i+1
--i　　等价于　i=i-1

i++　　i值不变，下一次调用i时值为i+1
i--

逻辑表达式

2016年9月10日　　14:42

真（1），假（0）

关系运算符具有左结合性

<

 >

<=

>=

判等运算符

==

!=

逻辑运算符

!

&&

||

算术运算符--关系运算符--判等运算符（从左至右优先级递减）

if语句

2016年9月10日　　14:50

```
if(表达式)
语句

if(表达式)
  语句
else if(表达式)
  语句
…
else
  语句
```

如果表达式值为真（非零），那么执行括号后面的语句

- if(a!=0) 等价于 if(a)
- if(a==0) 等价于 if(!a)

条件表达式

2016年9月10日　　15:06

表达式1 ？表达式2 ：表达式3

如果表达式1成立（非零），那么结果等于表达式2的值，否则结果等于表达式3的值

条件表达式使用场合一般为：

- return语句
  if(i<j)
  　　return i;
  else
  　　return j;
  替换为：return (i<j ? i : j);
- printf函数
  if(i<j)
  　　printf("%d",i);
  else
  　　printf("%d",j);
  替换为：printf("%d", i<j?i:j);

switch语句

2016年9月10日　　15:19

```
switch(表达式) {
    case 常量表达式: 语句块1
    …
    case 常量表达式: 语句块n
    default: 语句块n+1
}
```

控制表达式：圆括号组成的表达式。不能用浮点数和字符串。
常量表达式：必须是整数或字符
语句块：可以包含任意数量的语句

如果没有break语句，控制将会从一个分支继续到下一分支。

while循环

2016年9月10日　　16:19

while(表达式)
　　语句块

先判定控制表达式的值为非零，再执行循环体，这个过程持续进行到控制表达式的值为零才停止

do循环

2016年9月10日　　16:44

do
　　{语句块}
while(表达式);

执行do语句，如果表达式的值是非零的，那么再次执行循环体
特点：至少需要执行一次循环

for循环

2016年9月10日　　18:46

```
for(表达式1;表达式2;表达式3)
    语句
```

等价为

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

- 只要表达式2成立循环一直进行
- 表达式3是每次循环中最后被执行的一个操作
- 可以在表达式1中声明并初始化多个相同类型的变量（但是for语句声明的变量不能再循环外部访问）。

for语句的惯用法

- 向上加：
  for(n=0;n<m;n++;)，执行m次
  for(n=1;n<=m;n++;)，执行m次
- 向下减：
  for(n=m;n>0;n--;)　，执行m次
  for(n=m-1;n>=0;n--;)，执行m次

逗号运算符

2016年9月11日　　9:36

表达式1,表达式2

逗号运算符用途：在for循环中第一个或第三个表达式中使用逗号运算符可以实现一次对多个变量进行自增/减操作

- 逗号运算符运算优先级最低
- 逗号表达式的左操作数在右操作数之前求值

退出循环

2016年9月11日　　9:59

break语句

- 结束break语句所在的循环体
- 如果有循环嵌套，break语句只能跳出本层循环。
- 可以使用在switch、while、do、for语句中。
- return语句后面的break不会执行。

continue语句

- 跳过循环体花括号范围内continue语句之后的剩余部分，继续下一次的循环。
- 只能用在循环语句中。

continue语句常使用在以下场合：

```
for(;;)
{
    读入数据;
    if(数据的第一条测试失败)
        continue;
    if(数据的第二条测试失败)
        continue;
    …
}
```

goto语句

标识符:语句

goto 标识符;

- 语句必须和goto在同一个函数中
- goto语句对于嵌套循环的退出很有效！

空语句

2016年9月11日　　10:58

```
;
```

- 空语句单独放置一行
- 在if,while,for语句的圆括号后面放置分号会创建空语句，会造成这些语句提前结束。

空语句常使用的场合：

- 无限循环体
- 复合语句的末尾放置标号

```
{
    …
    goto end_of_stmt;
    end_of_stmt: ;
}
```

死循环&空循环

2016年9月11日　　11:09

死循环

- **for(;;)**
- **while(1)**

空循环

- **for(i=1;i<n;i++)**
  {}
- **for(i=1;i<n;i++;)**
  continue;

整数类型

2016年9月11日　　16:15

有符号数（系统默认）
x>=0：最左边的位（符号位）为0
x<0：最左边的位（符号位）为1

无符号数（unsigned）：没有符号位

	整数类型范围	在32位系统中的字节数	
short (int)	-32768~32767	2Byte	
unsigned short (int)	0~65535	2Byte	无符号常量，结尾+U
int	-32768~32767	4Byte	
unsigned int	0~65535	4Byte	
long (int)	-2147483648~2147483647	4Byte	长整数，结尾+L
unsigned long (int)	0~4294967295	4Byte	

可以检查<limits.h>头确定整数类型范围

整数常量

十进制：0~9，不以0开头
八进制：0~7，以0开头
十六进制：0~9，a~f，以0x开头

溢出问题

1. 修改数据类型
2. 检查printf、scanf中转换说明符

浮点类型

2016年9月11日　　16:58

float　4Byte：结尾以f或F表示，精度是显示小数点后6位
double　8Byte
long double　8Byte：结尾以l或L表示

以科学计数法形式存储：符号、指数、小数

- 浮点常量
  必须包含小数和指数

字符类型

2016年9月11日　　17:14

char　1Byte

- 赋值：单字符，且用单引号括起来
- 可移植性技巧：用unsigned char, signed char

转义字符

\a　警报
\b　回退符
\n　换行符
\f　换页符
\r　回车符
\t　水平制表符
\v　垂直制表符
\\
\?
\'
\"

- 字符处理函数toupper:检测参数是否是小写字母，如果是就把小写字母转换成大写字母，否则返回参数的值。并需要写#include <ctype.h>指令
- **变量名=getchar();**// 读单个字符，等价于scanf，返回int型，读取时不会跳过空白字符
  scanf函数会遗留下它“扫视”过但未读取的字符！当在同一个程序中混合使用getchar,scanf可能会出错
  getchar函数的惯用法：
  - while(getchar() != '\n')
    ;
    // skip the rest of line
  - while((ch=getchar()) == ' ')
    ;
    // 跳过空白字符，当结束循环时遇到第一个非空字符
- **putchar('a');**// 写单个字符，等价于printf

类型转换

2016年9月11日　　20:03

常用算术转换

策略：把操作数转换成可以安全地适用于两个数值的“最狭小”的数据类型（要求的存储字节更少）

方法：整值提升（把字符或短整数转换成int型）

float-double-long double

short int-int-unsigned int-long int-unsigned long int

有符号操作数和无符号数之间的转换易出错：“comparision between signed and unsigned”

强制转换

（类型名） 表达式

类型转换函数

1. **字符串转换为数字的函数**http://c.biancheng.net/cpp/html/124.html

   stof()、atoi()、atol()、strtod()、strtol()、strtoul()

类型定义

2016年9月11日　　20:45

typedef <已有类型名> <自定义的类型名>;

例如：typedef int Dollars;

Dollars cash_in,cash_out;

//编译器会将Bool加入它所识别的类型名列表中，即把Bool看成int类型的同义词

类型定义&宏定义：

- 类型定义的移植性强
- 数组和指针是不能定义为宏的
- 类型定义的对象具有和变量相同的作用域

sizeof运算符

2016年9月11日　　21:06

sizeof(类型名)

功能：计算存储这种类型名的值所需的字节数(Byte)
返回值：无符号整数

在32位系统中：
int　4B
char　1B
指针　4B

一维数组

2016年9月13日　　8:36

声明

- 数组元素的类型、数量（推荐使用宏定义）

数组下标：从0~n-1

初始化

- int a[3]={1,2,3};
- int a[5]={1,2,3};//{1, 2, 3, 0, 0}
- int a[3]={0};
- int a[]={1,2,3};
- 指定初始化
  int a[N]={[k]=29,[p]=8};//k,p为下标指示符
  指示符：整型常量表达式。如果数组长度是省略的，系统将根据最大指示符推断数组长度。

数组常用操作：

- 清零：
  - for(i=0;i<n;i++)
    a[i]=0;
  - for(i=0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
    a[i]=0;
- 复制：
  - **memcpy(a,b,sizeof(a));**
  - for(i=0;i<n;i++)
    a[i]=b[i];

sizeof：确定数组的总字节数=n*4, n是数组长度
n=sizeof(a) / sizeof(a[0])

多维数组

2016年9月13日　　10:53

a[i][j]：第i行第j列的元素

在C语言中是按行主序存储的

初始化

- int a[2][3]={{1,2,3},{0,1,0}};
- int a[2][3]={{1},{0}};
- int a[2][2]={[0][0]=1,[1][1]=1};

常量数组

10:59

```
const int a[]={0,0,0};
```

变长数组

2016年9月13日　　11:28

- 长度在程序执行是计算
- 没有静态存储权限
- 没有初始化
- 常见于除了main以外的函数，便于传参

函数的定义和调用

2016年9月14日　　8:46

```
<函数返回值类型>  <函数名>(<形参类型> 形参名...)
{
    声明
    语句
}
```

实际参数的转换

C语言允许在实际参数的类型与形式参数的类型不匹配的情况下进行函数调用。

- 编译器在调用前遇到原型：每个实际参数会被隐式地转换成相应形式参数的类型
- 编译器在调用前没有遇到原型：编译器执行默认的实际参数提升。float→double，char→int，short→int
- 不一定是变量，任何正确类型的表达式都可以

形参：形参没有返回类型必须写成void类型，每个形参的类型必须单独说明

函数返回值类型

- 如果函数没有返回值，可以定义void类型
- 函数不能返回数组
- 省略返回值，系统默认是int型
- 如果需要保留返回值，必须把返回值赋值给变量

定义：必须放在main函数之前

调用

- 放在需要使用这个函数返回值的地方
- 如果这个函数是void类型，则这个函数的调用必须自成一个语句。
- 如果是以下这种类型的函数，那么调用时可以这么写：

```
void  <函数名>(void)                    <函数名>();
{
    函数体（可以为空）
}
```

- 强制转换成void——“抛弃函数返回值”

声明(函数原型)

<返回类型> <函数名>(形参类型1,形参类型2…)；

- **函数必须进行声明或定义**
- 函数体内声明的变量专属于此函数，不能对这些变量进行检查和修改。

数组参数

2016年9月14日　　20:11

数组型实参

- 当形式参数是一维数组时，可以不说明数组的长度

```
int func(int a[])
{
    …
}

int main()
{
    int sum;
    int b[];
    sum=func(b);
    return 0;
}
```

- 函数可以改变数组型实际参数的元素

```
void store_zero(int a[],int n)
{
    int I;
    for(i=0;i<n;i++)
    {
        a[i]=0;
    }
}
```

数组型形参

- 如果形参是多维数组，声明参数时只能省略**第一维**的长度
- 变长数组形式参数

```
int sum_array(int n,int a[n])   //这种情况下括号中的形参顺序不能改
{…}

int sum_array(int ,int [*])
{…}

int sum_array(int m,int n,int a[m][n])
{…}
```

数组参数声明使用static

int sum_array(int a[stactic 3],int n) //保证数组a的长度至少为3

{…}

仅可用于第一维！

复合字面量

2016年9月14日　　21:01

(数组类型 [ ]){元素值}

通过指定其包含的元素而创建的无名数组

```
total=sum_array((int [ ]){1,2,3}, 5);
```

程序终止

2016年9月15日　　9:20

程序正常终止，main函数应该返回0;
程序异常终止，main函数返回非0值
——（状态码）

- return 表达式;
- exit(EXIT_SUCCESS);//退出程序
  exit(EXIT_FAILURE);//异常终止
  exit函数属于<stdlib.h>头

两种方法的差异：不管哪个函数调用exit语句都会终止程序，而return语句只有在main函数调用时才会终止程序

递归

2016年9月15日　　9:30

递归是指：函数调用本身

内联函数

2016年9月22日　　20:21

编译器把函数的每一次调用都用函数的及其指令来代替。

```
inline double average(double a,double b)
{
	return (a+b)/2;
}
```

“*”和“&”

2016年9月29日　　9:38

“*”

- 在声明中，表示声明的是指针
- 在执行语句或初始化表达式中，表示取指针所指对象的内容

“&”

- 在声明中，表示的是引用
- 在执行语句或初始化表达式中，表示取对象的地址
- 传引用：引用相当于一个别名，跟人的外号一样，指向的都是同一个人，传引用的时候虽然形式看起来跟值传递一样，但是它并不会复制一个副本进行变量保存，即只有一个空间保存变量值。

指针变量

2016年9月16日 9:46

指针就是地址，指针变量就是存储地址的变量。

声明

```
int *p; //p是指向int类型对象的指针变量
```

- 指针只能指向一种引用类型的对象

初始化

```
int *p,x;          int x,*p=&x;          int x;
p=&x;                                    int *p=&x;
```

在第二、三种初始化方式中*的作用不再是间接寻址运算符，它的作用是告知编译器p是一个指针；
而在语句中出现的*的作用就是间接寻址。

指针赋值

```
int x,y,*p,*q;
p=&x;
q=p;//p,q都指向了x
```

指针作为参数

```
int x,*p;
...
p=&x;
scanf("%d",p); // 指针会告诉scanf函数把取出的值放在哪里
```

const的用法

```
void func(const int *p)
{
    int j;
    p=&j;  //legal
    *p=0;  //wrong
}
```

用const来表明函数不会改变指针参数（*p）所指对象。因为实际参数是按值传递的，所

以通过使指针p指向其他地方的方法给p赋新值不会对函数外部产生影响。

```
void func(int * const p)
{
	int j;
	p=&j;  //wrong
	*p=0;  //legal
}
```

这种用法保护p本身

```
void func(const int * const p)
{
	int j;
	p=&j;  //wrong
	*p=0;  //wrong
}
```

指针&常量

2016年9月29日 9:49

指向常量的指针：不能通过指针来改变所指对象的值，指针可以指向另外的对象

```
const int *p=&a;
*p=2; //wrong
p=&b; //correct
```

指针类型的常量：指针本身的值无法更改

```
int *const p=&a;
p=&b; //wrong
```

void类型指针：可以存储任何类型的对象地址，任何类型的指针都可以赋值给void类型的指针变量

```
void *p;
int x=5;
p=&x;
int *p2=static_cast<int *>(p);
```

指针的算术运算

2016年9月16日　　16:11

- **指针加上整数**

  若p=&a[i]，则p+j 等价于 &a[i+j]

- **指针减去整数**

- **两个指针相减：**结果为指针之间的距离（用元素个数来度量），前提是两个指针指向同一个数组
  p=&a[0];
  q=&a[3];
  x=q-p; //x=3

- **指针比较：**前提是两个指针指向同一个数组，比较结果依赖于数组中两个元素的相对位置

- **指向复合常量的指针**
  int *p=(int []){1,2,3}; //p指向数组的第一个元素

常见程序例子

2016年9月16日　　15:44

- 交换两个变量的值

```
#include <stdio.h>

void swap(int *p, int *q)
{
	int t;
	t = *p;
	*p = *q;
	*q = t;
}

int main()
{
	int i=1, j=2;
	printf("%d,%d\n", i, j);
	swap(&i, &j);
	printf("%d,%d\n",i,j);
	return 0;
}
```

指针 & 数组

2016年9月16日　　16:46

- **指针变量重复自增，访问数组元素**

```
#define N 100
…
int *p, a[N];
int sum=0;
for(p=&a[0]; p<&a[N]; p++)
        sum+=*p;
```

- ***运算符和++运算符的组合**

后缀++优先级高于*

表达式	含义
*p++ 或 *(p++)	自增前表达式值是*p，之后再自增p(对象地址自增)
(*p)++	自增前表达式值是*p，之后再自增*p（对象值自增）
++p 或(++p)	先自增p,自增后表达式的值是*p
++*p 或 ++(*p)	先自增*p,自增后表达式的值是*p

```
int *p,a[N];
int sum=0;
p=&a[0];
while(p<&a[N])
        sum += *p++;
```

- **指针数组**

数组中每个元素都是指针变量

数据类型 *数组名[下标表达式]

- **用数组名作为指针**

用数组名作为指向数组的第一个元素的指针

如果int a[N]，那么：

*(a+i) 等价于a[i]

a+i 等价于 &a[i]

惯用法：

```
int a[N],*p;
for(p=a; p<a+N; p++)
{ sum+=*p; }
```

- 形式参数：声明为数组和声明为指针是一样的
- 实际变量：声明为数组和声明为指针是不一样的

- 用指针作为数组名

```
int a[N],*p=a;
for(i=0; i<N; i++)
{
    sum +=p[i];
}
```

编译器把p[i]看作*(p+i)

指针 & 多维数组

2016年9月18日　　15:13

- 处理多维数组的元素

```
int a[rows][cols];
int *p;
for(p=&a[0][0];p<&a[rows][cols];p++) //对数组a清零
{
      *p=0;
}
```

- 处理多维数组的行

p=&a[i][0] 等价于 p=a[i]

表达式a[i]是指向第i行中第一个元素的指针

```
int a[rows][cols];
int *p;
for(p=a[i];p<a[i]+cols; p++) //对第i行清零
{
      *p=0;
}
```

- 处理多维数组的列

```
int a[rows][cols], (*p) [cols], j;
...
for(p=&a[0]; p<&a[rows]; p++)
      (*p) [j]=0;
```

*p选中了a的一整行，(*p) [j]选中了该行第j列的元素
(*p) [cols]表示指向长度等于cols的整形数组的指针；
*p [cols]表示元素类型为指针，元素个数为cols的数组

- 用多维数组名作为指针

若int a[rows][cols]，那么：
a是指向元素a[0][0]的指针，a的类型是指向数组的指针int (*)[cols]

a[0]是指向元素a[0][0]的指针，a[0]的类型是指向整数的指针int (*)

- 指针 & 变长数组

```
void func(int n)
{
    int a[n],*p;
    p=a;
    …
}
```

```
void func(int m, int n)
{
    int a[m][n], (*p) [n];
    p=a;
}
```

可改变类型的声明必须出现在函数体内部或者在函数原型中

动态存储分配

2016年9月20日　　16:24

内存分配函数

声明在<stdlib.h>头中

- malloc函数：分配内存块，但不对内存块进行初始化。
  void *malloc(size_t size);
  分配size个字节的内存块，并返回内存块的指针

- calloc函数：分配内存块，并对内存块清零。
  void *calloc(size_t nmemb, size_t size);
  为nmemb个元素的数组分配内存空间，每个元素长度为size个字节

- realloc函数：调整先前分配的内存块大小
  void *realloc(void *ptr, size_t size);
  ***ptr必须是来自先前malloc, calloc或realloc的调用**
  a. 如果prt==NULL, 则realloc的行为就像malloc一样
  b. 如果size==0, 等同于释放掉内存块
  c. 当扩展内存块时，不会对添加进内存块的字节进行初始化
  d. 当无法按要求扩展内存块时，返回NULL，原有内存块中的数据不会变化

  一旦realloc函数返回，必须对指向内存块的所有指针进行更新，因为realloc函数可能会使内存块移动到其他地方

动态分配字符串

```
char *p;
p=(char *) malloc(n+1); //为n个字符的字符串分配内存空间
```

在字符串函数中使用动态存储分配

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *concat(const char *s1, const char *s2)
{
    char *result;
    result = (char *) malloc(strlen(s1) + strlen(s2));
    if (result == NULL)
    {
        printf("Error: malloc failed in concat.\n");
        exit(EXIT_FAILURE);
```

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char *concat(const char *s1, const char *s2)
{
    char *result;
    result = (char *) malloc(strlen(s1) + strlen(s2));
    if (result == NULL)
    {
        printf("Error: malloc failed in concat.\n");
        exit(EXIT_FAILURE);
    }

    strcpy(result, s1);
    strcat(result, s2);
    return result;
}
```

p=concat("abc","def");

- 动态分配数组

int *a;

a=malloc(n*sizeof(int)); //为由n个整数构成的数组分配动态存储空间

释放存储空间

2016年9月21日　　9:12

- free函数原型：**void free(void *prt);**

  p=malloc(…);
  q=malloc(…);
  free(p);
  p=q;　/*必须给p指针重新指向一个内存块*/

指向指针的指针

2016年9月22日　　10:49

```c
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int value;
    struct node *next;
};
void add_to_list(struct node **list, int n)
{
    struct node *new_node;
    new_node = (struct node *) malloc(sizeof(struct node));
    if (new_node == NULL)
    {
        printf("Error:malloc failed in add_to_list.\n");
        exit(EXIT_FAILURE);
    }
    new_node->next = *list;
    new_node->value = n;
    *list = new_node;
}
```

调用：**add_to_list(&first , 10);**　/*其中first本身就是一个指针*/

指向函数的指针

2016年9月22日　　11:00

定义：数据类型 (* 函数指针名)(形参表);
赋值：函数指针名=函数名;

```
double integrate(doube (*f)(double), double a, double b)
{
    double y;
    y=(*f) (x);
    …
}
```

调用：result=integrate(sin, 0.0, PI/2)

返回值为指针的函数

2016年9月16日　　14:59

```
数据类型  *函数名(形参表)
{
    函数体
}
```

```
int *max(int *p , int *q)
{
    if(*p > *q)
        return p;
    else
        return q;
}
```

```
int *a, x, y;
…
a=max(&x, &y);
```

注意：函数不能 指向自动局部变量的指针

- 函数返回指针可以指向外部变量
- 函数返回指针可以指向声明为static的局部变量的指针

函数返回指针指向数组元素：

```
int *middle(int a[], int n)
{
    return &a[n/2];
}
```

字符串字面量

2016年9月18日　　16:46

字符串是以数组的形式存放的

字符串字面量：一对双引号括起来的字符序列，以转义序列\0 标志结束
延续字符串字面量：

- 行尾以“\”结束，字符串字面量必须从下一行的起始位置继续
- 用空白字符分割多条相邻的字符串字面量

字符字面量的操作：

```
char *p;
p="ab"; // 使p指向字符串的第一个字符
```

字符串字面量是不能修改的

字符串字面量 & 字符常量

"a", 用指针表示，是字符串字面量
'a'，用整数表示，是字符常量

字符串的操作

2016年9月18日　　17:14

自定义字符串处理函数

```
#define STR_LEN 80
...
char str[STR_LEN+1];
```

字符串的长度取决于空字符的位置

- 字符数组和字符指针

char str[STR_LEN+1],*p;
p=str;
p 指向了str，所以p可以当做字符串使用了

- 字符串数组

char *str[] = {"s1","s2",...};

初始化/赋值

char date1[8]="June 14";

写字符串

- printf

  printf("%s\n",str);
  printf("%m.ps\n",str);
  使字符串的前p个字符在大小为m的字段内显示（默认右对齐）
  %-m.ps，强制左对齐

- sprintf
  主要功能：把格式化的数据写入某个字符串中
  int sprintf(char *buffer,const char *format,[argument]...);

  buffer：char型指针
  format：格式化字符串
  argument：可选参数，可以是任何类型的数据
  返回值：出错返回-1，返回值被写入buffer字节数，'\0'不计入内

- puts
  puts(str);

读字符串

- **scanf("%s",str);**
  调用时scanf函数会跳过空白字符，然后读入字符并存储到str中，直到遇到下一个空白符为止，始终会在字符串末尾存储一个空字符。
  scanf函数不会读入一整行输入，在任意空白字符、换行符、制表符出现的地方停止。
- gets(str);
  gets函数不会在开始读字符串之前跳过空白字符
  gets函数会持续读入，直到找到换行符为止，并以空字符代替换行符存储到数组中

- **自定义字符串读取函数**

```
int read_line(char str[], int n)
{
        int i=0, ch;
        while ((ch = getchar()) != '\n' && ch!=EOF)
        {
                if (i<n)
                        str[i++] = ch;
        }
        str[i] = '\0';
        return i;
}
```

字符串库

2016年9月18日　　19:52

- strcpy
  char *strcpy(char *s1, const char *s2);
  功能：将s2指向的字符串复制到s1指向的数组中
  惯用法：strcpy(str1,strr2);
  strcpy(str,"abc");
  strcpy(str1,strcpy(str2,"abc"));

- strncpy
  strncpy(str1,str2,sizeof(str1)-1);
  str1[sizeof(str1)-1]='\0';

- strlen
  strlen(str);
  功能：返回字符串中第一个空字符之前的字符个数（不包括空字符）

- strcat
  char *strcat(char *str1,const char str2);
  功能：把字符串2的内容追加到字符串1的末尾，并返回字符串1

- strncat
  strcat(str1,str2, sizeof(str1)-strlen(str1)-1);

- **strcmp**
  int strcmp(const char *s1, const char *s2);
  功能：比较两个字符串的大小（ASCII表），根据s1是否<, ==,> s2返回一个小于、等于或大于0的值
  s1=s2，返回0
  s1<s2，返回负数
  s1>s2，返回正数

字符串惯用法

2016年9月19日　　13:40

搜索字符串的结尾

size_t是在标准C库中定义的，位于头文件stddef.h中，是sizeof操作符返回的结果，是一个基本的无符号整数类型（unsigned int）。

```
size_t strlen(const char *s)
{
	const char *p=s;
	while(*s)
		s++;
	return s-p;
}
```

追加字符串

版本一：

```
char *strcat(char *s1, const char *s2)
{
	char *p = s1;

	while (*p != '\0')
		p++;
	while (*s2 != '\0')
	{
		*p = *s2;
		p++;
		s2++;
	}
	*p = '\0';
	return s1;
}
```

版本二：

```
char *strcat(char *s1, const char *s2)
{
	char *p = s1;

	while (*p)
		p++;
	while ((*p++ = *s2++))  //加一对括号警告信息的出现
```

```
            ;
        return s1;
    }
```

命令行参数

2016年9月19日　　14:16

```
int main(int argc, char *argv[])
{
    …
}
```

argc是命令行参数的数量（包括程序名本身）
argv是指向命令行参数的指针数组，argv[0]指向程序名
命令行参数以字符串的方式存储

- 如果用户输入命令行：ls -l remind.c

那么argc=3，argv[1]指向-l，argv[2]指向remind.c

访问命令行参数

```
int j;
for(j=1; j<argc; j++)
    printf("%s\n", argv[i]);
```

```
char  **p;
for(p=&argv[1]; *p!=NULL; p++)
    printf("%s\n", *p);
```

结构变量

2016年9月19日　　16:18

结构：具有不同类型的成员的集合
联合：具有不同类型的成员的集合，共享同一存储空间
枚举：一种整数类型

声明

```
struct {
        int number;
        char name[NAME_LEN+1];
        int on_hand;
} part1,part2;
```

初始化

```
struct {
        int number;
        char name[NAME_LEN+1];
        int on_hand;
} part1={528,"abc",2},
  part2={222,"ccdd",3};
```

```
struct {
        int number;
        char name[NAME_LEN+1];
        int on_hand;
} part1={.number=528, .name="abc", .on_hand=2},
  part2={.number=222, .name="ccdd", .on_hand=3};
```

对结构的操作

- 访问结构成员：**part2.name**
- 赋值：**part2=part1;**　仅适用在类型兼容的结构之间

对于整个结构体的操作只有赋值！

结构类型

2016年9月19日　　17:11

- **命名结构的方法：**

（1）结构标记

（2）typedef类型定义

结构标记

```
struct part
{
        int number;
        char name[NAME_LEN+1];
        int on_hand;
};
```

用这种结构声明变量：**struct part part1,part2;**

类型定义

```
typedef struct
{
        int number;
        char name[NAME_LEN+1];
        int on_hand;
}Part;
```

用这种结构声明变量：**Part part1,part2;**

- **结构作为参数和返回值**

```
//给函数传递part结构
void print_part(struct part p)
{
        printf("Part number: %d\n", p.number);
        printf("Part name: %s\n", p.name);
}
```

函数调用：**print_part(part1);**

```
//返回part结构
struct part build_part(int number, const char* name)
{
```

```
    struct part p;
    p.number=number;
    strcpy(p.name, name);
    return p;
}
```

调用方法：**part1=build_part(528,"Disk drive",10);**

复合字面量

2016年9月19日　　19:22

```
print_part((struct part) {528,"luoxin"});
print_part((struct part) {.number=528, .name="luoxin"});

part1=(struct part) {528,"luoxin"};
```

嵌套的数组和结构

2016年9月19日　　19:27

嵌套的结构

```
struct person_name
{
	char first[FIRST_NAME_LEN+1];
	char middle_initial;
};

struct student
{
	char sex;
	int age;
	struct person_name name;
} student1, student2;

strcpy(student1.name.first, "Fred");
```

结构数组

定义：**struct part inventory[100];**

访问元素：**print_part(inventory[i]);**

元素赋值：**inventory[i].number=18;**

初始化：

```
struct dialing_code
{
    char *country;
    int code;
};

const struct dialing_code country_codes[] =
{ { "China", 86 }, {"America",01},
   {"Brazil",55} , {"Italy",39}   };
```

联合

2016年9月20日　　14:45

编译器只为联合中最大的成员分配足够的内存空间。所以改变一个成员就会使之前存储在任何其他成员中的值发生改变。

定义：

```
union {
    int x;
    double y;
}u;
```

赋值：

```
u.x=89;
```

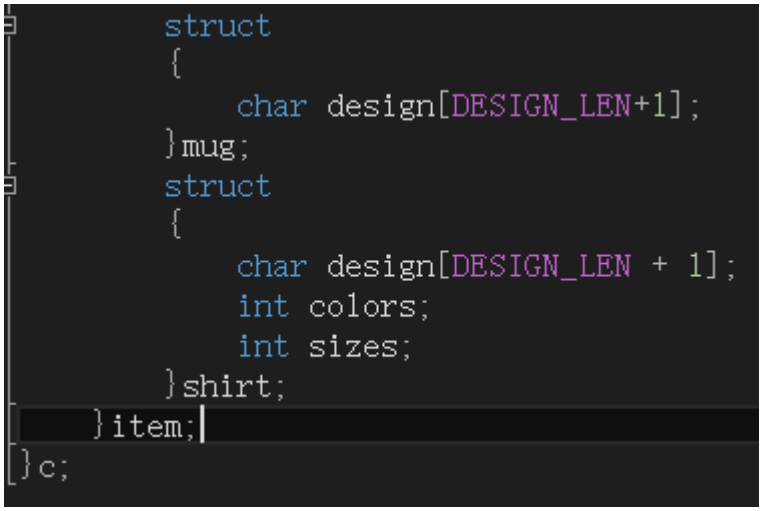
初始化：

```
union {
    int x;
    double y;
}u={0};  //默认是第一个成员可以获得初值
```

```
union {
    int x;
    double y;
}u={.y=0.5};
```

联合&结构的相互嵌套：

```
#include <stdio.h>
#define  TITLE_LEN 20
#define DESIGN_LEN 20
struct catalog_item
{
    int stock_number;
    float price;
    int item_type;
    union
    {
        struct
        {
            char title[TITLE_LEN + 1];
            char AUTHOR;
            int num_pages;
        }book;
```



联合的两个或多个成员是结构体，而这些结构最初的一个或多个成员是相匹配的（顺序相同，类型相同，名字可以不同），那么当前某个结构有效，则其他结构中的匹配成员也有效。

```
strcpy(c.item.mug.design, "Cats");
printf("%s", c.item.shirt.design);      /*prints "Cats"*/
```

- 用联合来构造混合的数据结构

```
typedef union
{
      int x;
      double y;
} Number;

Number number_array[100];
number_array[0].x=78;
number_array[1].y=78.4;
```

枚举

2016年9月20日　　15:30

- 枚举类型是一种值由程序员列出的类型。
- 枚举常量在其作用域范围内必须不同于已声明的其他标识符

enum 枚举类型名 {变量值列表};

标记

```
enum suit {RED, YELLOW, GREEN};
enum suit s1,s2;
```

```
typedef enum {RED, YELLOW, GREEN}  Suit;
Suit s1,s2;
```

枚举作为整数

```
enum suit {RED=1, YELLOW=3, GREEN=0};
```

判定素数

2016年9月14日　　19:36

- 2到n(或n的平方根)之间有没有n的约数
- 打素数表：如求出n之内的所有素数，其思路是从1开始遇到一个素数就标记一下，并去掉n之内的大于它的所有倍数，直循环到n:

```
#include<stdio.h>
int n,i,j,a[1000001],p[100000],t=0;
void main()
{
    scanf("%d",&n);
    a[1]=0;
    for(i=2;i<=n;i++)a[i]=1;
    for(i=2;i<=n;i++)
        if(a[i]){
            p[t++]=i;
            for(j=i+i;j<=n;j+=i)a[j]=0;
        }
    for(i=0;i<t;i++)
        printf("%d%c",p[i],i<t-1?' ':'/n');
}
```

  此方法也有局限性，数据量中等时才不会超时，数据量过大时也会超时，而且只能用素数打表，不能对单个数进行判定！
- 若正整数a>1,且a不能被不超过a的平方根的任一素数整除，则a是素数

排序算法

2016年9月15日　　9:40

快速排序

假设要排序的数组下标从1~n

1. 选择数组元素e作为“分割元素”，然后重新排列数组，使得元素从1~i-1都是<=e，元素i包含e，而元素从i+1~n都是>=e
2. 通过递归地采用快速排序方法，对从1~i-1的元素进行排序
3. 通过递归地采用快速排序方法，对从1~i+1的元素进行排序

步骤一采用分割算法

原理：开始low指向数组中第第一个元素，high指向末尾元素。首先把第一个元素（分割元素）复制给其他地方的一个临时存储单元，从而在数组中留下一个“空位”。接着，从右至左移动high，直到high指向小于分割元素的数时停止，把这个数复制给low指向的“空位”，这将产生一个新的high指向的空位。从左向右移动low，寻找大于分割元素的数，并把这个数复制给high指向的空位。重复执行此过程。直到low和high在数组中间的某处相遇时停止，此时二者指向同一个空位，把分割元素复制给该空位。

改进分割算法

取第一个、中间、最后一个元素的中间值作为分割元素

局部变量

2016年9月15日　　16:11

在函数体内声明的变量称为该函数的局部变量

性质：

- 自动存储期限：在包含该变量的函数被调用时“自动”分配的，函数返回时收回分配。当再次调用该函数时，无法保证局部变量拥有原先的值。
- 块作用域：从局部变量声明的点开始一直到所在函数体的末尾。

静态局部变量

拥有永久的存储单元，在整个程序执行期间都会保留变量的值，在函数返回时，这种变量不会丢失值。

```
void f(void)
{
    static int I;
    …
}
```

形式参数：具有自动存储期限、块作用域

外部变量（全局变量）

2016年9月15日　　16:21

- 静态存储期限
- 文件作用域

使用外部变量时要确保它们有实际意义的名字

程序结构说明

2016年9月15日　　17:03

1. 预处理命令(#include, #define)
2. 类型定义（tydefine）
3. 外部变量声明
4. 函数声明（函数原型）
5. main函数定义
6. 其他函数定义

链表

2016年9月21日　　9:24

声明结点类型

```
struct node
{
	int value;
	struct node *next;
};
struct node *first = NULL;
```

创建结点类型

```
struct node *new_node;  //这个变量临时指向待创建的新结点
new_node=malloc(sizeof(struct node));
(*new_node).value=10;   或   new_node->value=10;
```

->运算符产生左值，可以在任何运行普通变量的地方使用

```
scanf("%d", &new_node->value);
```

插入结点

（1）在开始处

```
next_node->next=first;
first=next_node;
```

```c
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int value;
    struct node *next;
};

struct node *add_to_list(struct node *list, int n)
{
    struct node *new_node;
    new_node = (struct node *) malloc(sizeof(struct node));
    if (new_node == NULL)
    {
        printf("Error: malloc failed in add_to_list.\n");
        exit(EXIT_FAILURE);
    }
    new_node->next = list;
    new_node->value = n;
    return new_node;
}
```

搜索链表

for(p=first; p!=NULL; p=p->next)

{…}

```c
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int value;
    struct node *next;
};

struct node *search_list(struct node *list, int n)
{
    struct node *p;
    for (p = list; p != NULL; p = p->next)
    {
        if (p->value == n)
            return p;
    }
    return NULL;
}
```

删除结点

*prev指向前一个结点的指针， *cur指向当前结点的指针

1） 定位要删除的结点

```
for(prev=NULL, cur=list;
    cur!=NULL && cur->value!=n;
    prev=cur, cur=cur->next)
    ;
```

2） 改变前一个结点的位置，从而使它“绕过”删除结点

```
prev->next = cur->next;
```

3） 调用free函数收回删除结点占用的内存空间

```
free(cur);
```

```c
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int value;
    struct node *next;
};

struct node *delete_list(struct node *list, int n)
{
    struct node *prev, *cur;
    for (prev = NULL, cur = list;
        cur != NULL && cur->value != n;
        prev = cur, cur = cur->next)
        ;
    if (cur == NULL)//n was not found
        return list;
    if (prev == NULL)//n is in the first node
        list = list->next;
    else
        prev->next = cur->next;
    free(cur);
    return list;

}
```

声明的语法

2016年9月22日　　16:49

！！变量在程序中可以有多次声明，但只能有一次定义

<声明说明符>　<声明符>;

声明说明符：

- 存储类型: auto, static, extern, register
- 类型限定符: const（变量为“只读”）, volatile
- 类型说明符: void, int, struct, typedef…
- 函数说明符: inline

const和宏的区别：

- const对象遵循和变量相同的作用域规则，#define创建的常量不受这些规则的限制
- const对象不可以用于常量表达式 (const int n=5;/*wrong*/)
- const对象用于&符号合法

声明符

- **始终从内往外读声明符**
- **在做选择时，始终使[]和()优先于***

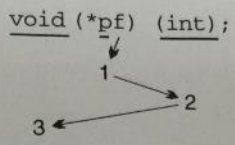
```
int *a[10];  //指针数组
float *fp(float);  //  返回指针的函数
void (*pf)(int);   // 指向函数的指针
int * (*x[10]) (void);
```

声明是一个小陷阱：

```
void (*pf)(int);
```

由于*pf包含在圆括号内，所以pf一定是指针。但是(*pf)后边跟着(int)，所以pf必须指向函数，且此函数带有int型的实际参数。单词void表明了此函数的返回类型。

正如最后那个例子所示，理解复杂的声明符经常需要从标识符的一边折返到另一边：

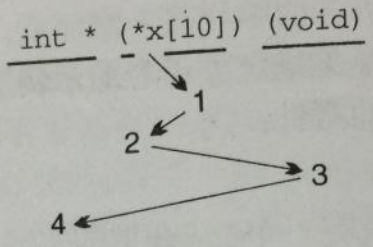


pf的类型：
1. 指针指向
2. 具有int型实际参数的函数
3. 返回void型值

下面用这种折返方法来解释早前给出的声明：

```
int *(*x[10])(void);
```

首先，定位声明的标识符（x）。在x前有*，而后边又跟着[]。因为[]优先级高于*，所以取右侧（x是数组）。接下来，从左侧找到数组中元素的类型（指针）。再接下来，到右侧找到指针所指向的数据类型（不带实际参数的函数）。最后，回到左侧看每个函数返回的内容（指向int型的指针）。图示过程如下：



x的类型：
1. 数组
2. 指针指向
3. 不带实际参数的函数
4. 返回指向int型的指针

要想熟练掌握C语言的声明需要花些时间并且要多练习。唯一的好消息是

声明的特定内容。函数不能返回数组：

```
int f(int)[];       /*** WRONG ***/
```

函数不能返回函数：

```
int g(int)(int);    /*** WRONG ***/
```

函数型的数组也是不可能的：

```
int a[10](int);     /*** WRONG ***/
```

在上述情形中，我们可以用指针来获得所需的效果。函数不能返回数组，但

```
typedef int *Fcn(void);
typedef Fcn  *Fcn_prt;
typedef Fcn_prt  Fcn_prt_array[10];
 Fcn_prt_array  x
```

存储类型

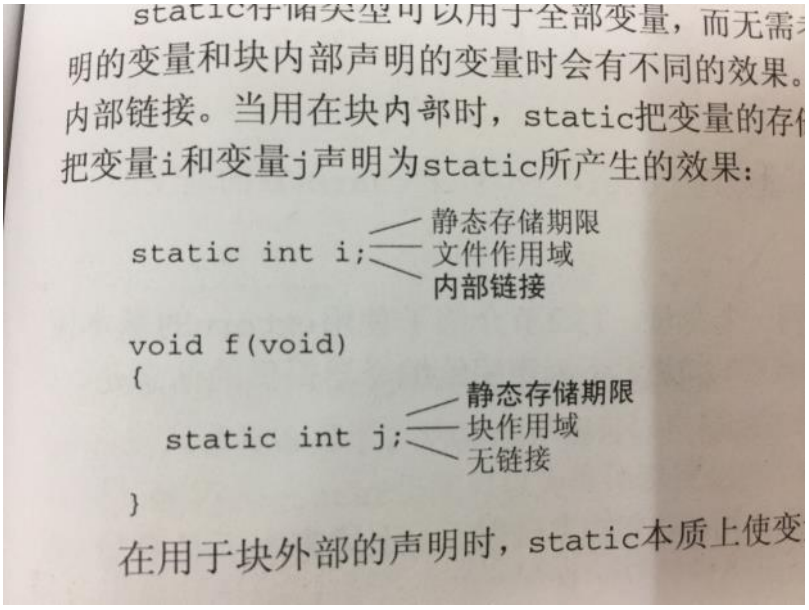
2016年9月22日　　17:23

变量的默认存储期限、作用域和链接都依赖于变量声明的位置

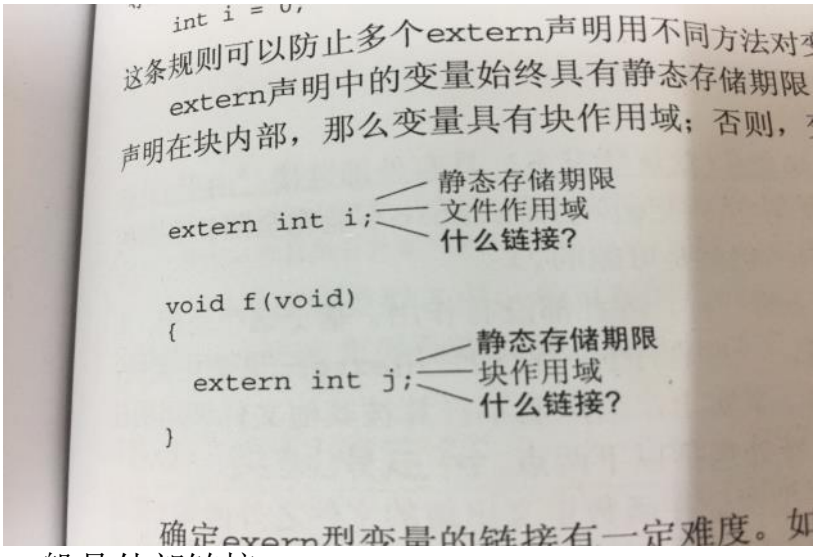
- 在块内部：自动存储期限、块作用域、无链接
- 在程序外层：静态存储期限（变量可以无限期保留值）、文件作用域、外部链接

存储类型

- auto
- static：



- extern：



一般是外部链接

- register：只对声明在块内的变量有效，和auto唯一的不同是不能对register变量用&符号。适合用于需要频繁进行访问或更新的变量。

预处理指令

2016年9月22日　　17:49

- 文件包含
  #include <文件名>　/*C语言库的头文件*/
  #include "文件名"　/*用户自定义的头文件*/

- 宏定义

#define定义一个宏
#undef删除一个宏定义

- **宏定义**在编译前会进行宏替换，INT_PTR a,b;被替换为int* a,b;所以a是指针，b是int型
- typedef int* int_ptr;是**类型定义**，把int*取别名为int_ptr,所以int_ptr c,d;定义的c,d变量都是整型指针

- 条件编译

#if 常量表达式
#endif
计算常量表达式的值，如果为0，那么二者之间的行在预处理过程中从程序中删除

#ifdef 标识符
　　程序段1
#else
　　程序段2
#endif
如果“标识符”经#define定义过，且未经undef删除，则编译程序段1，否则编译程序段2

#ifndef 标识符 测试一个标识符是否没有定义为宏

#if 常量表达式 1
当表达式1非零时包含的代码
#elif 常量表达式2
当表达式1等于零，表达式2非零时包含的代码
#else
其他情况下需要包含的代码

```
#endif
```

- defined操作符
  defined(标识符)