# 第一部分：变量和数据类型

## 初识C语言

程序：为解决特定的问题用计算机语言编写的有序指令的集合

#include <stdio.h>

int main(void)

{

// todo

return 0;

}

程序的执行过程：

* 编辑：源代码文件包括两个部分：源文件(.c)（>=1个）头文件（.h）(>=0个)。
* 编译：通过编译将源文件翻译成二进制文件目标文件(.obj)，进行语法错误的检查。
* 链接：将编译的到的所有的目标文件通过链接形成唯一的一个可执行性的文件（.exe）。
* 执行：得到程序运行的结果
* 调试(debug)：F10 F11

程序的几点说明：

* #include是预处理指令，它的作用是将指定的头文件内容放到预处理指令的所在位置。
* 头文件是一个标准的函数库，程序中使用printf库函数包含在stdio.h里（stdio表示standard input/output）。
* 头文件有**两种形式**：

<>表示系统文件，即表示预处理的文件从系统规定的位置寻找头文件

""表示当前目录下的头文件，即表示预处理的文件先从当前目录下进行寻找，若通过，继续进行下面的编译，若没有再从系统中寻找头文件。

* main函数是可执行程序的入口函数。一个C程序有且只有一个main函数。

## 变量

算法：解决问题的方法

算法的特点：有穷性、确切性、可行性、算法有0个或多个输入、算法有1个或多个输出

变量的命名规范：

* 变量名由数字，字母和下划线组成
* 数字不能打头
* 不能使用C语言的关键字
* C语言的变量名中的字母区分大小写

变量的赋值：

* 变量可以先定义再赋值，也可以在定义的同时进行赋值；在定义变量的同时赋初值称为初始化。
* 注意，在定义中不允许连续赋值，如a=b=c=5是不合法的。

## 数据类型

### 整型数据

**整型类型分类：**

* short 2个字节
* int 2/4个字节（8位和16位系统是2字节，32位系统和64位系统是4字节）
* long 4/8个字节（除了64位系统是8字节，其他都是4字节）
* long long int或long long 8个字节 （VC6.0不支持，是在C99中添加的）

整型数据可以分为符号型和无符号型 ，前缀带有unsigned的都是无符号型。在同一个编译器和系统下，有符号和无符号表示的字节都是相同的。

不同的系统必须满足sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)

**整型数据在内存中存放的方式：**在内存中数值是以补码的形式存储的

**整型数据的表示方法**

* 十进制数：十进制数没有前缀。其数码为0～9。
* 八进制数：八进制数必须以0开头，即以0作为八进制数的前缀。数码取值为0～7。八进制数通常是无符号数。
* 十六进制数：十六进制数的前缀为0X或0x。其数码取值为0~9，A~F或a~f。

### 字符类型

字符型数据是用单引号括起来的一个字符。

常见的转义字符：

\n 回车换行 \ddd 1-3位八进制数所表示的字符 \xhh 1-2位十六进制所表示的字符

字符类型说明符：char 每一个字符变量占据1个字节的内存空间

字符值是以ASCII码的形式存放在变量的内存单元之中的。

由ASCII码可以看出：

* 小写的ASCII码比大写的ASCII码大32，常用的a为97 A为65
* 后面的ASCII码比前面的ASCII码大
* '3'要变化成3需要'3'-'0'=3

### 实型类型

| 类型说明符 | 比特数（字节数） | 有效数字 | 数的范围 |
| --- | --- | --- | --- |
| float | 32(4) | 6~7 | -10^38~10^38 |
| double | 64(8) | 15~16 | -10^308~10^308 |

实型数据的表示：

* 十进制数形式：由数码0~9和小数点组成。
* 指数形式：由十进制数，加阶码标志"e"或"E"以及阶码（只能为整数，可以带符号）组成。

## 常量和字面量

| 常量 | 说明 |
| --- | --- |
| 直接常量（字面量） | 可以立即拿来用，无需任何说明的量。 |
| 符号常量 | 用标识符代表一个常量。#define 标识符 常量 |
| const常量 | const int x=3.14; |

字面值：

* 整型常量的类型：一般默认整型常量为int型。

长整型字面值 123L 23l

无符号整型字面值 123U 123u

无符号长整型字面值 123UL 123ul

long long型字面值 123ll 123LL

* 浮点型字面值：默认为double型，也可以使用后缀f，F，l，L确定类型。

## 输入和输出

### printf

格式字符串的一般形式为： % [输出最小宽度][.精度][长度]类型。

（1）最小宽度

* 当转换值的字符数（含前缀）小于最小宽度说明时，则使用填充符（空格）将数值填充到最小宽度.
* 当转换值的字符数（含前缀）大于最小宽度说明时，最小宽度说明失效。

（2）精度

精度格式符以“.”开头，后跟十进制整数。

本项的意义是：

* 如果输出数字，则表示小数的位数；
* 如果输出的是字符，则表示输出字符的个数；

（3）类型

|  |  |
| --- | --- |
| 格式字符 | 意义 |
| d | 以十进制形式输出带符号整数(正数不输出符号) |
| o | 以八进制形式输出无符号整数(不输出前缀0) |
| x,X | 以十六进制形式输出无符号整数(不输出前缀Ox) |
| u | 以十进制形式输出无符号整数 |
| f | 以小数形式输出单、双精度实数 |
| e,E | 以指数形式输出单、双精度实数 |
| g,G | 以%f或%e中较短的输出宽度输出单、双精度实数 |
| c | 输出单个字符 |
| s | 输出字符串 |

### scanf

格式字符串的一般形式为：  %[\*][输入数据宽度][长度]类型

\* 符号

用以表示该输入项，读入后不赋予相应的变量，即跳过该输入值。

#include <stdio.h>

int main(void)

{

int a,b;

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

printf("%d %d",a,b);

return 0;

}

/\* input:2 4 5 output:2 5 \*/

# 第二部分：运算符和表达式

## 逗号表达式

逗号运算的结果是输出逗号后面的数

#include <stdio.h>

int main(void)

{

int a = 3,b = 5,z;

z = a>b,a+b;// 逗号的运算符的优先级要低于赋值运算符，因此先算赋值运算符，再算逗号运算符

printf("%d",z);

z = (a>b,a+b);// 特殊运算符的优先级高于赋值运算符，特殊运算符中，逗号运算符输出逗号后面的内容

printf("%d",z);

return 0;

}

## 赋值表达式

赋值运算符的优先级只比逗号运算符的优先级高，而比其他的要低。

#include<stdio.h>

int main(void)

{

int x = 3;

int y = 5;

printf("%d\n",y\*=x+3);// 30

printf("%d\n",y);// 30

return 0;

}

## 逻辑表达式

在此运算符中，有一个短路特性：

* （表达式1）&&（表达式2）如果表达式1为假，则表达式2不会进行运算，即表达式2“被短路”。
* （表达式1）||（表达式2）如果表达式2位真，则表达式2不会进行运算，即表达式2“被短路”。

/\*

与运算符(&&) 和或运算符(||)均为二元运算符。具有左结合性。非运算符(!)为一元运算符，具有右结合性。

“&&”和“||”优先级低于关系运算符，高于赋值运算符，且&&的优先级高于||的优先级，“!”高于算术运算符。

\*/

#include <stdio.h>

int main(void)

{

int a = 3,b = 2,c = 1,d = 5,e = 6,f;

f = a<b || b<c && c<d || d<e;

printf("%d",f);// 1

return 0;

}

## 关系判等表达式

/\*

关系运算符都是二元运算符，其结合性均从左至右。关系运算符的优先级低于算术运算符，高于赋值运算符。

在六个关系运算符中，< <= > >=的优先级相同，高于==和!=，==和!=的优先级相同。

\*/

#include <stdio.h>

int main(void)

{

int x = 3,y;

y = 2<x>=8;

printf("%d",y);

return 0;

}

## 算术表达式

#include<stdio.h>

int main(void)

{

int x = 4,y = 3,z;

printf("%d,%d,%d\n",20/7,20/-7,-20/-7);// 当两个操作数均为整型或者是char型时，除法运算实现了整除

printf("%f,%f\n",20.0/7,-20.0/7);// 只要有一个操作数是实型，则结果的类型为实型

printf("%d\n",100%3);//求余运算要求两个操作数均为整型

z = x++ + y;//++i：i自增1后再参与其它运算。--i：i自减1后再参与其它运算。i++：i参与运算后，i的值再自增1。i--：i参与运算后，i的值再自减1。

printf("%d,%d\n",x,z);

z = x++ + --y;

printf("%d,%d,%d\n",x,y,z);

return 0;

}

#include <stdio.h>

int main(void)

{

int i = 2;

printf("%d %d\n",++i,++i);// 函数的压栈顺序是从右至左进行的

return 0;

}

## 条件表达式

/\*

条件运算符为?和：，它是一个三元运算符，即有三个参与运算的操作数。由条件运算符组成条件表达式的一般形式为：

表达式1 ? 表达式2 : 表达式3

其求值规则为：如果表达式1的值为真，则以表达式2 的值作为条件表达式的值，否则以表达式3的值作为整个条件表达式的值。

在这个过程中，如果表达式1为真，那么表达式3的结果不在进行运算，即发生忽略，反之，表达2的结果进行忽略。

\*/

#include<stdio.h>

int main(void)

{

int a = 3,b = 4,c = 5,d = 6,max;

max = a>b?a:b;//结果就是为了得到a和b中较大的那个数

printf("%d\n",max);//条件运算符的运算优先级低于关系运算符和算术运算符，但高于赋值符

max = a>d?a:c>d?c:d;//条件运算符的结合方向是自右至左

printf("%d\n",max);

return 0;

}

## 位运算表达式

#include<stdio.h>

int main(void)

{

int a = 9,b = 5,c;

c = a&b;

printf("%d\n",c);//按位与运算符"&"是二元运算符。其功能是参与运算的两数各对应的二进位相与。只有对应的两个二进位均为1时，结果位才为1 ，否则为0。参与运算的数以补码方式出现

c = a|b;

printf("%d\n",c);//按位或运算符“|”是双目运算符。其功能是参与运算的两数各对应的二进位相或。只要对应的二个二进位有一个为1时，结果位就为1。参与运算的两个数均以补码出现

c = a^b;

printf("%d\n",c);//按位异或运算符“^”是双目运算符。其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为1.相同为0

c = ~a;

printf("%d\n",c);//求反运算符～为单目运算符，具有右结合性。 其功能是对参与运算的数的各二进位按位求反

c = a<<4;

printf("%d\n",c);//左移运算符“<<”是双目运算符。其功能把“<< ”左边的运算数的各二进位全部左移若干位，由“<<”右边的数指定移动的位数

c = b>>4;

printf("%d\n",c);//右移运算符“>>”是双目运算符。其功能是把“>> ”左边的运算数的各二进位全部右移若干位，“>>”右边的数指定移动的位数

// 应该说明的是，对于无符号数，在移动的时候前面补0，对于有符号数，在右移时，符号位将随同移动。当为正数时， 最高位补0，而为负数时，符号位为1

return 0;

}

## 练习

//编写一个程序，让其以二进制的形式输出

/\* 方法一：主要思路 num<<i 如果int占据32位 需要循环32次 \*/

#include <stdio.h>

int main(void)

{

int a = 20;

for(int i = 0;i<32;i++)

{

printf("%d",(a<<i)<0?1:0);

if((i+1)%4 == 0)

printf(" ");

}

printf("\n");

return 0;

}

/\* 方法二：主要思路 十进制转换为二进制的方法 \*/

#include <stdio.h>

int main(void)

{

int a = 20;

int arr[16];

for(int i = 0;i<16;i++)

{

arr[i] = a%2;

a/=2;

}

for(int j = 15;j>=0;j--)

{

printf("%d",arr[j]);

if(j%4 == 0)

printf(" ");

}

printf("\n");

return 0;

}

# 第三部分：条件、循环

## 条件语句

if语句的一般形式：

if(表达式)

{

语句;

}

其语义是：如果表达式的值为真，则执行其后的语句，否则不执行该语句。

if – else语句的一般形式

if(表达式)

{

语句1;

}

else

{

语句2;

}

注意：如果if语句后面的东西是一个赋值语句且赋值的值不是0，那么这个值一定是真的

switch语句被称为**多路分支语句** 一般形式：

switch(表达式)

{

case 整型字面值1:

语句1(集合);

break;

case 整型字面值2:

语句2(集合);

break;

case 整型字面值n:

语句n(集合);

default:

语句n+1(集合);

}

注意：

1. case的标号不能超过1023个

2. 表达式的值为整型或者字符型

3. switch语句的执行过程：

* 若表达式的值等于switch 语句中某个case 标号中的整型字面值(每个case 标号中的字面值不能相同)则程序控制转移到该case 标号表示的点，从此点开始执行,**即执行其后的语句，然后不再进行判断，继续执行后面所有case后的语句。直到遇见 break跳出switch。**
* 若表达式的值不等于任何 case 标号中的字面值，则程序控制转移到default 标号（如果有的话）表示的点，从此点开始执行。
* 若表达式的值不等于任何 case 标号中的字面值，又没有default 标号，则不执行switch 语句体中的语句。

4. 在case后，允许有多个语句，可以不用{}括起来。

5. 各case和default子句的先后顺序可以变动，而不会影响程序执行结果。

6. default子句可以省略不用。

## 循环语句

for循环语句的基本形式为：

for(初始表达式1;条件表达式2;循环表达式3)

{

语句; // 被称为for 语句体

}

执行过程：

1) 执行表达式1

2) 判断是否满足表达式2，满足进入，否则退出循环

3) 顺序执行循环体内容

4) 执行表达式3，然后转入2

while语句是另一种循环语句,其一般形式如下：

while(表达式)

{

语句；

}

执行过程：

1) 判断表达式是否为真，满足进入2)，否则退出循环

2) 顺序执行循环体内容，完毕后进入1)

do-while语句是第三种循环语句，基本形式：

do

{

语句; // 被称为do 语句体

} while(表达式);

注意：do...while循环语句至少执行一次(与前面两种语句根本区别:结构不同)

执行过程：

1) 先执行一次循环体内容（由于还要执行一次循环才能退出）

2) 判断表达式是否为真，满足进入3)，否则退出循环

3) 顺序执行循环体内容，完毕后进入2)

循环语句的执行方式可以被改变：

1) break：停止整个循环的执行，执行循环后的内容

2) continue：停止当次循环执行，执行下一次循环

## 练习

// 请输出下列的结果，一定要注意switch语句的使用情况和a—的使用情况

#include <stdio.h>

int main(void)

{

int a;

scanf("%d",&a);

switch(a--)

{

case 1:a+=3;

case 2:a+=3;

default:printf("\*");

}

printf("%d",a);

return 0;

}

// 实现计算1-2+3-4+……+99-100

#include <stdio.h>

int main(void)

{

int sum = 0,temp;

for(int i = 1;i<=100;i++)

{

temp = i;

if(i%2 == 0)

{

temp = -1\*i;

}

sum += temp;

}

printf("%d\n",sum);

return 0;

}

// 实现九九乘法表

#include <stdio.h>

int main(void)

{

for(int i = 1;i<=9;i++)

{

for(int j = 1;j<=i;j++)

{

printf("%d x %d = %d\t",j,i,j\*i);

}

printf("\n");

}

return 0;

}

# 第四部分：指针

## 指针变量的声明和含义

int x = 3;// 普通变量的定义

int \*p = &x;// 指针变量的定义

#include<stdio.h>

int main(void)

{

// 对于%p一般以十六进制整数方式输出指针的值

int x = 3;

int \*p = &x;

printf("%d\n",x);

printf("%p\n",&x);

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

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

printf("%d\n",\*p);

return 0;

}

所有的指针变量都占据相同大小的内存，在32位电脑上，指针变量占4个字节。（指针是有类型的，指针变量虽然都占4个字节，它的类型含义：该变量只能存放它所对应的类型去掉一个\*后的那个类型的变量或常量的地址，例如：int \* p，则p只能存放int的变量或常量的地址。）

**指针的运算：**

指针的加减运算中是以指针的类型的长度为单位的。

## 多重指针的声明及其等价形式

double f = 3.14;

double \*p1 = &f;

double \*\*p2 = &p1;

double \*\*\*p3 = &p2;

p2的等价表达式有：

\*p3 &p1 &(\*p2) &(\*\*p3)

\*p2的等价表达式有：

&f p1 \*\*p3 \*(&p1) &(\*p1) \*\*(&p2) &(\*\*\*p3)

## 几种特殊的指针

const int x;

int const x;

//两者完全等价；我们推荐第一种；

int \* const p = &x;// p不可以修改 \*p可以修改

const int \*p = &x;// p可以修改 \*p不可以修改

//两者天壤之别，第一种const修饰变量p;第二种const修饰变量\*p;

多重const指针注意：

1. const出现的位置不同意义不同

2. const后面紧跟的变量不可改变

3.大家应该尽量多的使用const

一个值为空指针字面值的指针变量被称为空指针，空指针的字面值类型为0 和 NULL

空指针注意：若声明一个指针变量时，如果不指定它所指向的变量，则指针的指向是不确定的，我们要避免

# 第五部分：数组

## 数组的定义、初识化和元素的引用

一维数组的初始化：

初始化方法一：全部初始化

int arr[5] = {11, 12, 13, 14, 15};

注意：

* 初值的个数不能大于数组的长度
* int a[3]={1,2,3,4,5};（这种初始化是不正确的）
* 每个初值的类型最好与数组的类型一致（类型不一样不能说错）

初始化方法二：部分元素赋初值

int b[5]={1,2};// b[0]=1 b[1]=2 b[2]～b[4]都为0 未赋值的均为0

初始化方法三：省略长度赋初值

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

一维数组元素的引用：

#include <stdio.h>

int main(void)

{

int a[10];

// 读入数据到数组

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

{

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

}

// 求和

int sum = 0;

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

{

sum+=a[i];

}

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

{

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

}

printf("%d",sum);

}

二维数组的初始化：

方式一：按行全部赋初值： int a[2][3] = {{1,2,3},{4,5,6}};

方式二：顺序全部赋初值： int a[2][3] = {1,2,3,4,5,6};

方式三：部分赋初值：（按存储顺序）： int a[2][3] = {1,2};

方式四：部分赋初值：（按行部分赋值）： int a[2][3] = {{1,2},{4}};

注意：要么按行赋值，要么按存储顺序赋值 int a[2][3] = {{1,2},4,5,6}

方式五：省略一维长度赋初值（不能省略二维长度）： int a[][3] = {{1},{2,3},{4}}

一维字符数组和字符串：

字符数组和字符串的区别是：字符串的末尾有一个空字符’\0’

#include <stdio.h>

int main(void)

{

char a[10];

scanf("%s",a);// get a

printf("%s",a);// get

gets(a);// get a

puts(a);// get a

return 0;

}

字符串的定义方式：

char a[10] = "123";// 定义字符串

char \*p = "123";// 定义字符串

char \*p[10] = {"123","456"};// 字符指针数组

## 指针与数组

#include<stdio.h>

int main(void)

{

int a[5] = {11,12,13,14,15};

printf("&a[0] = %p\n",&a[0]);// 数组首元素的地址

printf("&a[0]+1 = %p\n",&a[0] + 1);// 数组首元素的地址 + 元素的长度

printf("a = %p\n",a);// 数组的首地址

printf("a+1 = %p\n",a+1);// 数组的首地址 + 元素的长度

printf("&a = %p\n",&a);// 数组的首地址

printf("&a+1 = %p\n",&a+1);// 数组的首地址 + 数组的长度（5个int类型的长度）

/\*

&a[0] = 00BEFDD4

&a[0]+1 = 00BEFDD8

a = 00BEFDD4

a+1 = 00BEFDD8

&a = 00BEFDD4

&a+1 = 00BEFDE8

\*/

return 0;

}

|  |  |
| --- | --- |
| 运算方式 | 说明 |
| **p+n** | p之后第n个元素的地址 |
| **p-n** | p之前第n个元素的地址 |
| **p++** | p作为当前操作数，然后后移一个元素 |
| **++p** | p后移一个元素，然后作为当前操作数 |
| **p--** | p作为当前操作数，然后前移一个元素 |
| **--p** | p前移一个元素，然后作为当前操作数 |
| **p-q** | 表示p和q两者之间的元素个数 |

数组元素的几种等价形式： a[i] \*(a+i)

**数组指针和指针数组:**

数组指针的表示方法 类型 (\* 变量名)[hang];

指针数组的表示方法 （表示这是一个数组，只不过里面表示的都是指针）

类型 \*变量名[];

## 指针与二维数组

#include<stdio.h>

int main(void)

{

int x = 3;

int \*p = &x;

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

int a[4];

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

int (\*p)[4] = &a; // a代表的是首元素的地址 &a是数组的地址 这是数组指针 因此是&a return 0;

}

int a[3][4];

a 代表二维数组的首地址，第0行的地址，**行地址**

a + i 代表第i行的地址，但并非增加i个字节！

行地址表示的是一个指向由4列元素组成的一维数组的指针

int b[3][4];

int (\*p)[4] = b;

**a[i]**即\*(a+i)代表第i行第0列元素的地址，**行首元素地址**

\*(a+i)+j 即 a[i]+j代表第i行第j列的地址&a[i][j]

\*(\*(a+i)+j) 即 a[i][j]代表第i行第j列的内容

## 练习

//表示一下二维数组对角线元素的和

#include <stdio.h>

int main(void)

{

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

int sum1 = 0;// 表示主对角线元素的和

int sum2 = 0;// 表示辅对角线元素的和

for(int i = 0;i<3;i++)

{

for(int j = 0;j<3;j++)

{

if(i == j)

{

sum1 += a[i][j];

}

if(i+j == 2)

{

sum2 += a[i][j];

}

}

}

printf("%d %d\n",sum1,sum2);

return 0;

}

//实现数组的逆置

#include <stdio.h>

int main(void)

{

int a[6] = {1,2,3,4,5,6};

int i,j;

for(i = 0;i<6;i++) printf("%d\t",a[i]);

i = 0;

j = 5;

while(i < j)

{

int temp = a[i];

a[i] = a[j];

a[j] = temp;

i++;

j--;

}

for(i = 0;i<6;i++) printf("%d\t",a[i]);

return 0;

}

//实现数组的冒泡排序

#include <stdio.h>

int main(void)

{

int a[6] = {2,4,3,1,6,5};

int i,j;

for(int i = 0;i<6;i++)

{

for(int j = 0;j<5;j++)

{

if(a[j] > a[j+1])

{

int temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

for(int i = 0;i<6;i++) printf("%d\t",a[i]);

return 0;

}

//实现数组的冒泡排序

#include <stdio.h>

int main(void)

{

int a[6] = {2,4,3,1,6,5};

for(int i = 0;i<6;i++)

{

for(int j = 0;j<6-i-1;j++)

{

if(a[j] > a[j+1])

{

int temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

}

}

}

for(int i = 0;i<6;i++) printf("%d\t",a[i]);

return 0;

}

//实现数组的折半查找和顺序查找

#include <stdio.h>

int main(void)

{

// 顺序查找

int a[5]={5,4,12,7,345},i;

for(i = 0;i<5;i++)

{

if(a[i] == 17)

{

break;

}

}

if(i < 5) printf("true");

else printf("false");

// 折半查找

int a[5]={4,5,7,12,345},i = 0,j = 4;

int count = 345;

while(i<=j)

{

int mid = (i+j)/2;

if(a[mid] > count)

{

j = mid - 1;

} else if(a[mid] < count){

i = mid + 1;

} else {

break;

}

}

if(i <= j) printf("true");// 注意 不能忘记等于 可能是最后一个

else printf("false");

return 0;

}

# 第六部分：函数

## 函数的定义与声明

int \* fun(void); // 函数的声明

int \* fun(void)

{

// 函数的定义 返回类型 函数名 形参列表

int a = 0;

return &a;// 注意函数返回值不能是局部变量的地址！

}

函数的定义包括：返回值的类型，函数名，形参列表。

返回值

1. 如果函数没有返回值一定要注意写void类型.不需return.

2. 一个函数可能有零个或者多个参数，最多只能由一个返回值.

3. 函数返回值不能是局部变量的地址！

## 函数的调用

* 主调程序保护现场
* 给形式参数（如果有）分配内存，将实参传给形参，将程序的控制权交给被调函数
* 执行被调函数语句
* 保存返回值（如果有）到某处，这个值所有是可见的，释放被调函数在栈区分配的空间
* 将程序控制权交给主调函数，主调函数取得返回值

一维数组作形参的过程：

一维数组做形参时，实际参数为&x[0],或者是x，此时的形式参数是\*a，a[]。

二维数组做形参时，此时的形式参数是a[][4],(\*a)[4]。

#include <stdio.h>

void fun(int \*p,const int len)// void fun(int p[],const int len) 这是因为p就是数组第一个元素的地址 int p[] = {} 此时的p也是第一个元素

{

for(int i = 0;i<len;i++)

{

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

}

}

int main(void)

{

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

fun(a,3);// a和&a[0]是一样的 可以写成fun(&a,3);

return 0;

}

## 递归函数

// 编程实现n的阶乘

int fun(int n)

{

if(n == 1)

return n;

else

return n\*fun(n-1);

}

## 生命周期与声明作用域

**静态生存周期**

若某对象在程序开始执行之前即分配到存储空间，而且保持到程序终止，则称该对象具有静态生存期。（所有函数、在顶层声明的变量、静态变量。）

**本地生存期**

如果对象在进入块或函数时分配到存储空间，且在退出块或函数时删除，则称该对象具有本地生存期。C 语言把具有本地生存期的变量称为自动变量。

**声明作用域**是声明变量有效的区域(空间)

* 顶层标示符（从声明点到源程序结束之间的文本区域）
* 函数定义中的形式参数（从声明点到函数体结束之间的文本区域）
* 块内标示符（从块中的声明点到块结束之间的文本区域）
* 标号标示符（整个函数体）

**全局变量和局部变量**

称在顶层声明的变量为全局变量，与全局变量相对的是局部变量（如：函数体内的变量、形式参数、块内的变量）。

**总结：**

1. C语言程序中，**在同一作用域内不允许有同名的变量、函数等**，**但在不同的作用域内，可能会出现同名的变量。**

2. 若使用的变量是在不同作用域内的同名变量，则**作用域小的变量优先。**

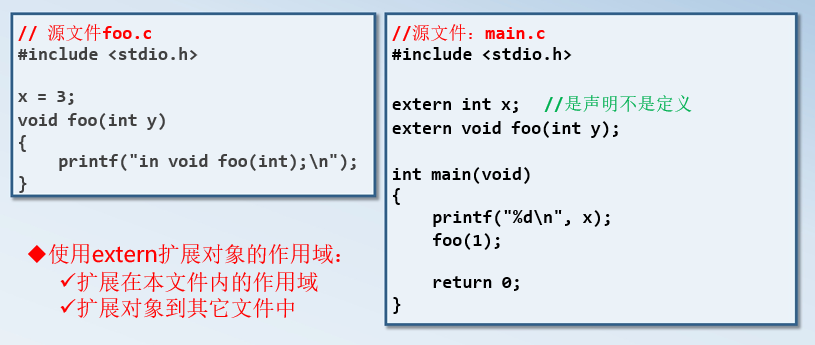
3. 局部变量的作用域为定义它的块儿

4. 全局变量的作用域为从定义它开始到程序结束

## 存储类型说明符

存储类说明符用于确定声明对象的生存期（typedef 除外），存储类说明符出现在声明说明符中，一个声明说明符中最多允许一个存储类说明符。

|  |  |  |
| --- | --- | --- |
| **存储类型** | **说明** |  |
| **auto** | 自动变量 | 局部变量在缺省存储类型的情况下归为自动变量。 |
| **register** | 寄存器变量 | 存放在**CPU**的寄存器中。对于循环次数较多的循环控制变量及循环体内反复使用的变量均可定义为寄存器变量。 |
| **static** | 静态变量 | 在程序执行时存在，并且只要整个程序在运行，就可以继续访问该变量。作用在本文件内。 |
| **extern** | 外部变量 | 作用域是整个程序，包含该程序的各个文件。生存期非常长，它在该程序运行结束后，才释放内存。 |



Static相关：

1. 静态局部变量生存期为整个源程序

2. 静态局部变量作用域与自动变量相同

3. static只是初始化一次，并且如果没有初始化，默认是0

4. 函数被调用时，其中的静态局部变量的值将保留前次被调用的结果

5. 静态全局变量与普通全局变量在存储方式上完全相同

* 非静态全局变量的作用域是整个源程序中的所有文件
* 静态全局变量的作用域只是定义它的文件中

6. 在函数定义中，说明此函数不可在其他文件中使用。

7. 在变量声明中，说明此变量不可在其他文件中使用。

Typedef相关：

**typedef** **int** **\*** **intPointer;** *//* 使*intPointer* 成为了*int\**的别名

**intPointer** **x;** *//* 在编译阶段，该语句将被处理成*int\** *x;*

## 程序在内存中的分布区域

库函数中有一类函数可以让我们自己分配内存：malloc calloc free

这些库函数应该包含的头文件有：**malloc.h或stdlib.h**



* 使用malloc、calloc分配的内存一定要自己用free释放。
* 用malloc、calloc进行内存分配，一定要检测成功与否。
* malloc、calloc函数的返回值是void类型，一定要根据需要进行强制类型转换。

**void\*** **malloc(size\_t** **size);**

**void\*** **calloc(** **size\_t** **num,** **size\_t** **size);**

**void free(void** **\*** **\_Memory);**

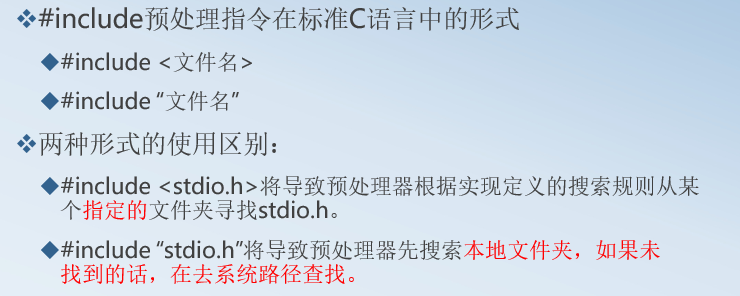
## 指针函数和函数指针

指针函数：类型说明符 \* 函数名(参数)

函数指针：类型说明符 (\*函数名)(参数)

# 第七部分：预处理和宏

## 文件包含

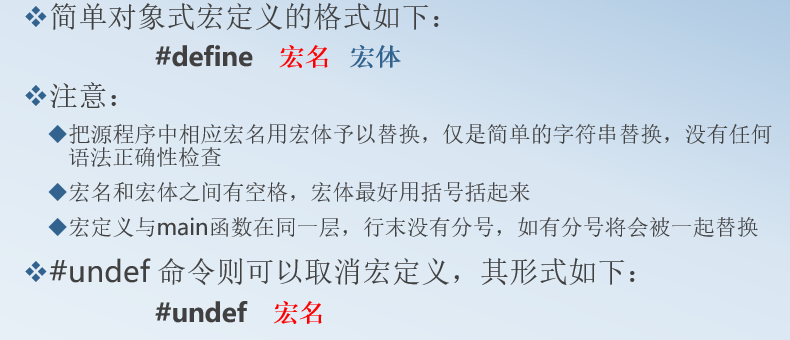


注意：在<>或“”中，有时会出现 .. 或 .

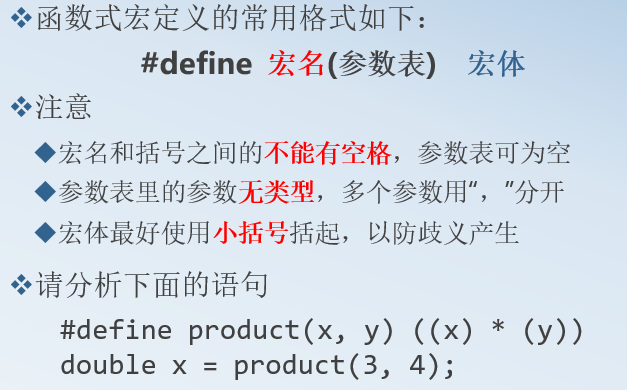
.. 表示当前文件所在目录的上一级目录；

. 则表示当前文件所在的目录.

## 对象宏（无参宏）

****

## 函数式宏（有参宏）



#include<stdio.h>

#include<stdlib.h>

#define F(x) x-2

#define D(x) x\*F(x)

int main(void)

{

printf("%d,%d",D(3),D(D(3)));

system("pause");

return 0;

}

要求功能简单，代码短小，则可以使用函数宏；如果要求效率较高，可以采用函数宏，函数宏没有函数调用的代价。

## 条件编译

预处理器条件指令（#if、#else、#elif、#endif 等）允许预处理器根据计算条件处理和替换宏。

**#ifdef、#ifndef用于测试一个名称是否被定义为预处理宏。**

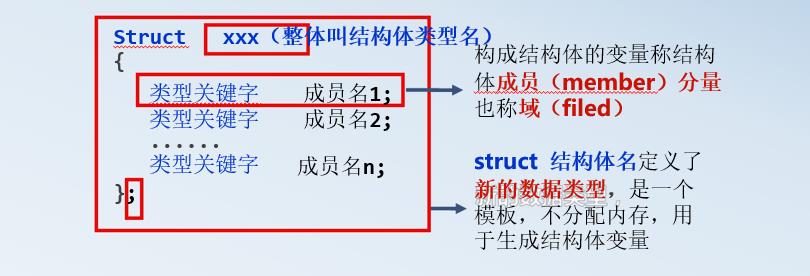
**主要用来解决头文件重复包含的问题。**

# 第八部分：结构体

## 结构体

结构体是一种自定义数据类型，结构体变量代表的结构体类型的数据对象作为一个整体存储在内存中。

结构体类型定义的基本语法形式为：



成员变量的类型不能和本结构体类型相同，即不能递归定义。

结构体的成员变量可以是本结构体类型**的指针变量**。这种结构也叫自引用结构。

**什么不允许出现在结构体成员中？**

* 本身结构体类型的成员变量
* 函数
* 静态变量

typedef struct Complex Complex;使用typedef简化结构体的名称

结构体成员的赋值：

#include <stdio.h>

typedef struct Complex

{

double real;

double imag;

}Complex;

int main(void)

{

// 在定义结构体变量的时候可以按照成员的顺序和类型对成员整体赋初值

Complex c1 = {1.4,1.5};

// 在定义结构体变量后，单独给成员变量赋值

Complex c1;

c1.real = 1.4;

c1.imag = 2.5;

// 同类型结构体变量可相互赋值(不涉及到指针成员)

Complex c1,c2;

c1.real = 1.4;

c1.imag = 2.5;

c2 = c1;

return 0;

}

访问成员：

若是普通结构体变量，则访问其成员的方式为**结构体变量名.成员名**

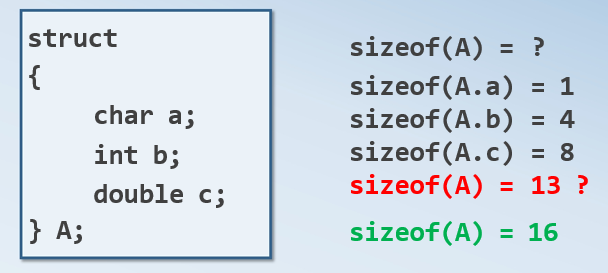
若是通过结构体指针访问结构体对象，则形式为：

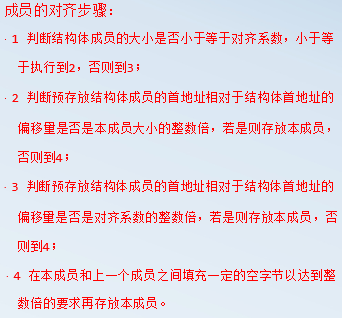
**结构体指针名->成员名**

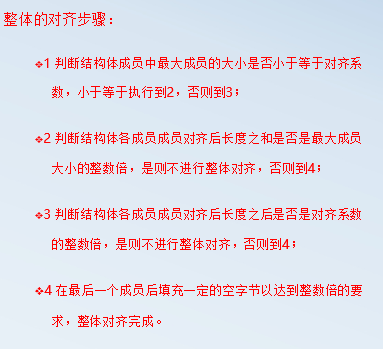
**（\*结构体指针名）.成员名**

## 结构体的内存分配

结构所占的实际空间一般是按照编译器默认对齐方式（对齐系数）来进行内存分配的。







当结构体中某成员为数组时，则在处理结构体的对齐时，该成员数组在求整数倍关系时使用数组类型的大小。

当结构体中某成员为结构体时，则在处理结构体的对齐时，该成员结构体在求整数倍关系时使用该成员结构体中最大成员的大小。

## 链表相关程序

//创建一个链表并对它进行遍历

#include <stdio.h>

#include <stdlib.h>

typedef struct Node

{

int data;

struct Node \* next;

}Node;

// 创建一个链表 头插法

Node \*createhead(int n)

{

Node \*head = NULL;

int i =0;

while(i<n)

{

Node \*p = (Node \*)malloc(sizeof(Node));

printf("请输入第%d个元素",n-i);

scanf("%d",&(p->data));

p->next = head;

head = p;

i++;

}

return head;

}

// 创建一个链表 尾插法

Node \* createfoot(int n)

{

Node \*head = NULL;

Node \*p = NULL;

int i = 0;

while(i<n)

{

Node \*q = (Node \*)malloc(sizeof(Node));

if(i == 0)

{

p = q;

head = q;

}

printf("请输入第%d个元素",i+1);

scanf("%d",&(q->data));

q->next = NULL;

p->next = q;

p = q;

i++;

}

return head;

}

void print(Node \*head)

{

Node \*p = head;

while(p !=NULL)

{

printf("%d",p->data);

p = p->next;

}

printf("\n");

}

// 链表逆置

Node \*inverse(Node \*head)

{

Node \*p = NULL;

Node \*q;

while(head!=NULL)

{

q = head;

head = head->next;// 一定要注意head->next的位置

q->next = p;

p = q;

}

return p;

}

int main(void)

{

Node \* head = createfoot(5);

print(head);

head = inverse(head);

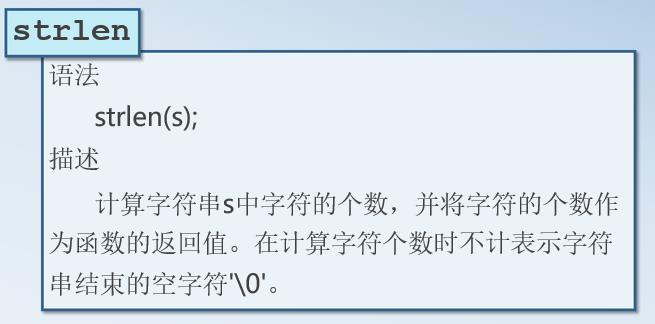
print(head);

return 0;

}

# 第九部分：字符串函数

## strlen



#include <stdio.h>

#include <assert.h>

int strlen(const char \*str)

{

assert(str!=NULL);

int i = 0;

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

i++;

return i;

}

int main(void)

{

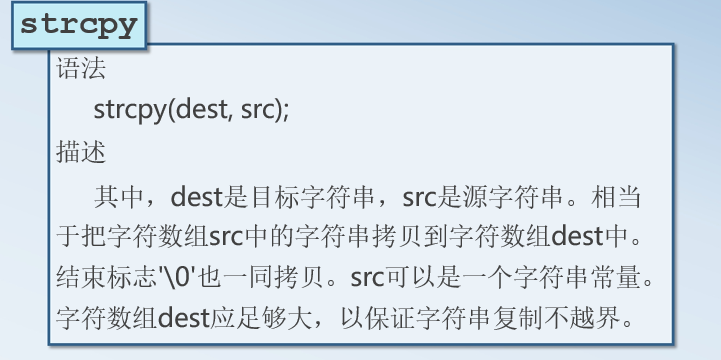
char \*str = "hello";

printf("%d",strlen(str));

return 0;

}

## strcpy



#include <stdio.h>

#include <assert.h>

char \* strcpy(char \*dest,const char\* str)// str本身是常量

{

assert(dest != NULL && str != NULL);// 加上断言 注重空指针的代码健壮性

int i = 0;

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

{

dest[i] = str[i];

i++;

}

dest[i] = '\0';

return dest;// 返回dst的原始值使函数能够支持链式表达式。 int l=strlen(strcpy(strA,strB));

}

int main(void)

{

char dest[20];

char \*str = "hello";

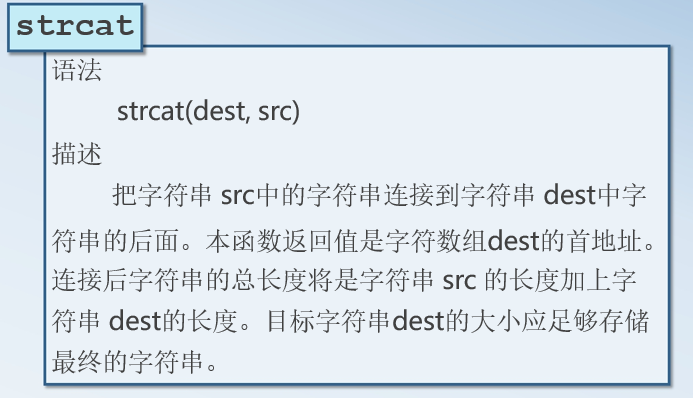
strcpy(dest,str);

printf("%s",dest);

return 0;

}

## strcat



#include <stdio.h>

#include <assert.h>

int strlen(const char \*str)

{

assert(str!=NULL);

int i = 0;

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

i++;

return i;

}

char \*strcat(char \* dest,const char \* str)

{

assert(dest != NULL && str != NULL);

int i = 0;

int y = strlen(dest);

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

{

dest[y] = str[i];

y++;

i++;

}

dest[y] = '\0';

return dest;

}

int main(void)

{

char dest[100] = "hello";

char str[] = "world";

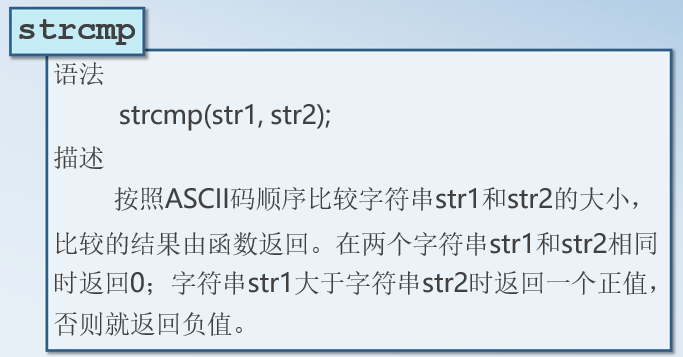
strcat(dest,str);

printf("%s",dest);

return 0;

}

## strcmp



#include <stdio.h>

#include <assert.h>

int strlen(const char \*str)

{

assert(str!=NULL);

int i = 0;

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

i++;

return i;

}

int strcmp(const char \*dest,const char \* src)

{

assert(dest !=NULL && src !=NULL);

int x = strlen(dest);

int y = strlen(src);

if(x > y)

{

return 1;

}

else if(x < y)

{

return -1;

}

else

{

for(int i = 0;i<x;i++)

{

if(dest[i] > src[i])

{

return 1;

} else if(dest[i] < src[i])

{

return -1;

}

}

return 0;

}

}

int main(void)

{

char dest[] = "hello";

char str[] = "hello";

printf("%d\n",strcmp(dest,str));

return 0;

}