小甲鱼C语言《带你学C带你飞》学习笔记(1)

日常 3-15 5664阅读 239点赞 20评论

视频链接https://www.bilibili.com/video/av27744141

由于篇幅限制，本笔记分两部分：

小甲鱼C语言《带你学C带你飞》学习笔记(1)-本文

小甲鱼C语言《带你学C带你飞》学习笔记(2)-传送门

本笔记是本人根据小甲鱼视频讲课内容记录，一些程序做了小修改。根据自身补充了一部分拓展内容。如有错误，欢迎一起讨论 ——正能量的康sir

喜欢我的分享，不放点个关注、点个赞、投个币，谢谢：）

P2第一个程序

编译型：c语言——》汇编——》机器语言——》CPU执行

解释型：java——》字节码——》解释器——》CPU执行

```
#include<stdio.h>
int main(){
printf("hello world!");
return 0;
}
```

linux环境下

编辑：vi test.c

编译：gcc test.c -o test

执行：./test

windows环境下

Dev C++

CodeBlocks

P3打印

printf("要打印的内容");

printf格式化输出函数，print打印，f即format格式化

\n换行

每行写不完行尾用\，下一行为上一行的延续。函数、关键字中间也可以用，注意下行开头如果有空格、缩进也会被解释，会导致识别不了。

转义字符


正能量的康sir
粉丝: 3737 阅读: 6606
关注

查看目录 | 来自文集：C语言/C++/数据

推荐文章

本地搭建C语言环境（CodeBlocks）
10-3 阅读90

学习C语言对我以后的工作到底有没有帮助
9-24 阅读18

C语言系统化精讲 重塑你的编程思维
12-7 阅读8

深度剖析两则典故，提高作文逼格：六朝金粉地，二分明月州
12-8 阅读1502

小甲鱼Python教程课后题
11-13 阅读145

查看更多

更多

专栏投稿
前去写文章

专栏帮助
查看专栏使用

P4变量

确定目标并提供存放的空间

命名规则：

变量名只能是英文字母(A-Z,a-z)和数字(0-9)或者下划线(_)组成

第一个字母必须是字母或者下划线开头(不能是数字)

变量名区分大小写

不能使用关键字

32个关健字

1999年c99标准增加5个

2011年c11标准增加7个

数据类型

char字符型，占用1字节

int整型，通常反映了所用机器中整数的自然长度

float单精度浮点型

double双精度浮点型

声明变量语法

数据类型 变量名

如int a

P5常量和宏定义

常量：

整数常量

实型常量 小数等

字符常量 包括普通字符、转义字符

字符串常量

符号常量 使用之前必须先定义

定义符号常量 宏定义

#define标识符 常量

示例

```
#include<stdio.h>
#define NAME "小甲鱼"
#define YEAR 2010
int main (){
printf("%s成立于%d年",NAME,YEAR);
}
```

标识符：和关键字命名规则一样

字符串常量 使用null结尾，转义符是\0

P6数据类型

1.基本类型

整数型　int、short int、long int、long long int　(short int<=int<=long int<=long long int)

浮点型　float、double、long double

字符型　char

布尔型　_Bool

枚举型　enum

2.指针类型

3.构造类型

数组型

结构型

联合性

4.空类型

sizeof运算符

用于获取数据类型或表达式的长度

sizeof（变量或对象），可以不加括号，空格隔开。如sizeof(a)或sizeof a

sizeof(类型),如sizeof(int)

signed和unsigned类型限定符 限定char类型和任何整数类型的取值范围

signed带符号位，可以存放负数

unsigned不带符号位，只能存放正数和0

[signed] short [int]

unsigned short [int]

[signed] int

unsigned int

[signed] long [int]

unsigned long [int]

[signed] long long [int]

unsigned long long [int]

示例

```
#include<stdio.h>

int main(){

short i;

unsigned short j;

i=-1;

j=-1;

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

printf("%u\n",j);

}
```

输出

-1

65535

P7取值范围

比特位

CPU能够读懂的最小单位——比特位，单位bit缩写b

字节

内存机构的最小寻址单位——字节，单位Byte缩写B

1Byte=8bit

1个字节等于8比特

1字节最大:二进制11111111 十进制255 十六进制FF

n个连续的1等于2的n次方减一

```
#include<stdio.h>

#include<math.h>

int main(){

int result = pow(2,32)-1;

printf("result=%d\n",result);

return 0;

}
```

符号位

存放signed类型的存储单元中，左边第一位表示符号位。如果该位为0表示该整数是一个正数；

如果该位为1,表示该整数是一个负数。

一个32位的整数变量，除去左边第一个符号位，剩下的表示值得只有31个比特位

事实上计算机是用补码的形式来存放整数的值。

正数的补码是该数的二进制形式

负数的补码：

（1）先取得该数的绝对值的二进制形式 -7 绝对值7的二进制10000111

（2）将第一步的值按位取反 11111000

（3）最后将第二步的值加1 11111001

补码的最大值127最小值-128

01111111 127

...

00000001 1

00000000 0

11111111 -1

11111110 -2

10000000 -128

基本数据类型的取值范围

P8字符和字符串

ASCII字符表

字符串：

声明字符串 char变量名[数量];

赋值 变量名[索引号]='字符';

声明+赋值 定义字符串 char name[5]={'a','b','c'};

```
#include<stdio.h>
int main(){
char a[9]={'b','i','l','i','b','i','l','i','\0'};
printf("%s\n",a);
return 0;
}
```

```
#include<stdio.h>
int main(){
//char a[8]={'b','i','l','i','b','i','l','i'};//错误 指定的数组长度不够时
//char a[]={'b','i','l','i','b','i','l','i'};//错误 不指定长度，每个字符用单引号时
//char a[9]={'b','i','l','i','b','i','l','i'};//正确 预留一个位置
//char a[]={'b','i','l','i','b','i','l','i','\0'};//正确 人工添加
char a[]={"bilibili"};//正确 字符串复制
printf("%s\n",a);
```

```
return 0;
}
```

拓展

'\0'是由C编译系统自动加上的。所以在用字符串赋初值时一般无须指定数组的长度， 而由系统自行处理。

把字符数组str1中的字符串拷贝到字符数组str2中。串结束标志'\0'也一同拷贝。

个案

1. 当数组长度不够。假设我们指定了数组长度，如：u8 str1[13]={"cxjr.21ic.org"};

由于字符组str1的长度为13,所以后面的信息会丢失，即'\0'丢失。

2. 如果在给数组赋值时，把每个字符单独用引号括起来。也会丢失'\0'。如：

u8 str1[]={'c','x','j','r','.','2','1','i','c','.','o','r','g'};

如果希望数组以'\0'结束，则可以写成以下三者之一：

u8 str1[]={"cxjr.21ic.org"}; //字符串赋值

u8 str1[]={'c','x','j','r','.','2','1','i','c','.','o','r','g', '\0'}; //人工添加

u8 str1[14]={'c','x','j','r','.','2','1','i','c','.','o','r','g'}; //故意给数组预留一个空位

P9算术运算符

求余%后面必须是整数

目

1+2 +运算符，1、2两个操作数(双目)

表达式

用运算符和括号将操作数连接起来的式子

1+1

a+b

'a'+'b'

a+'b'+pow(a,b)*3/4+5

运算符的优先性和结合性

类型转换

1+2.0转化为1.0+2.0

强制转换

(int)1.2注意不会四舍五入

正能量的康sir
粉丝: 3737 阅读: 6606

查看目录 | 来自文集：C语言/C++/数据

P10关系运算符和逻辑运算符

关系运算符 表示 比较：

优先级高：<、<=、>、>=

优先级低：==、!=

用关系运算符将两边的变量、数据、表达式连接起来，称为关系表达式。

1<2

a>b

a<1+b

'a'+'b'<='c'

(a=3)>(b=5)

结果是布尔型，c语言中正确1，错误0

逻辑运算符

非 ! 优先级高 !a

与 && 优先级中 a&&b

或 || 优先级低 a||b

短路求值

又称最小化求值，是一种逻辑运算符的求值策略。只有当第一个运算数的值无法确定逻辑运算的结果，才对第二个运算数进行求值。

C语言对于逻辑与和逻辑或采用短路求值的方式。

```
#include<stdio.h>
int main(){
int a,b=3;
(a=0)&&(b=5);  // 因为a为0，0表示假，能确定逻辑与的值，不会再对第二个运算数求值
printf("a=%d,b=%d\n",a,b);
(a=1)||(b=5);    //因为a为1，1表示真，能确定逻辑或的值，不会再对第二个运算数求值
printf("a=%d,b=%d\n",a,b);
return 0;
}
```

结果

a=0,b=3

a=1,b=3

推荐文章

本地搭建C语言环境（CodeBlocks）
10-3 阅读90

学习C语言对我以后的工作到底有没有帮助
9-24 阅读18

C语言系统化精讲 重塑你的编程思维
12-7 阅读8

深度剖析两则典故，提高作文逼格：六朝金粉地，二分明月州
12-8 阅读1502

小甲鱼Python教程课后题
11-13 阅读145

查看更多

更多

专栏投稿
前去写文章

专栏帮助
查看专栏使

P11 if语句

语句（1）：

if(表达式)

{

逻辑值为真所执行的语句、程序块

}

语句（2）：

if(表达式)

{

逻辑值为真所执行的语句、程序块

}

else

{

逻辑值为假所执行的语句、程序块

}

语句（3）：

if(表达式1) {……}

else if(表达式2){……}

else if(表达式3){……}

……

else if(表达式n){……}

else {……}

P12 switch语句和分支嵌套

switch(整型或字符型)

{

case常量表达式1：程序块1　　break;

case常量表达式2：程序块2　　break;

case常量表达式3：程序块3　　break

…….

case常量表达式n：程序块n　　break;

default:语句或程序块n+1　缺省一个break；可不写

}

悬挂else

if(a==1)

if(b==2)

printf("a等于1，b等于2");

else{lprintf("a不等于1")}

这样即使a等于1，b不等于2，也会输出a不等于1。因为else就近if

正确做法加{}

if(a==1)

{

if(b==2)

printf("a等于1，b等于2");

}

else{lprintf("a不等于1")}

等于号带来的问题

P13while语句和dowhile语句

入口条件循环

while(表达式){

循环体

}

出口条件循环

do

{

循环体

}

while(表达式);

P14for语句和循环嵌套

for(初始化表达式;循环条件表达式;循环调整表达式)

循环体

for(;;) 都可以省略 但是分号不能省

C99、C11允许在for语句的表达式1中定义变量　　表达式1、3中可以用逗号运算符写多个语句

for(int i=0;i<100;i++)

P15break语句和continue语句

break跳出 跳出循环或switch。不能用于循环语句和switch语句之外的任何其他语句中。
break 只能跳出一层循环。当有多层循环嵌套的时候，break只能跳出“包裹”它的最里面的那一层循环，无法一次跳出所有循环。同样，在多层 switch 嵌套的程序中，break 也只能跳出其所在的距离它最近的 switch。但多层 switch 嵌套实在是少见。

continue 结束本次循环，即跳过循环体中下面尚未执行的语句，然后进行下一次是否执行循环的判定。不能用于switch

P16拾遗

赋值运算符左边必须是一个Lvalue，变量名就是Lvalue

int a;

a=5

lvalue 识别和定位存储数值的标志符

复合的赋值运算符

+=

-=

*=

/=

%=

自增自减

i++

i—

i++、++i前后的区别

j=++i先i加1，再把加1后的值给j

j=i++ 先把i的值给j，i再加1

逗号运算符

优先级最低

表达式1，表达式，。。。，表达式n

运算过程从左到右

作为一个整体，它的值为最后一个表达式（表达式n）的值

a=(b=3,(c=b+4)+5)

先将b赋值为3，c赋值为b+4的和7，c的值加5，最后赋值给a，a的值12

常见：

1. 多个变量初始化

for表达式1、3

三目运算符

表达式1？表达式1：表达式3；

表达式1正确返回表达式2，错误返回表达式3

goto历史遗留 不常用

goto A;

A:　　printf("123");

注释

//单行注释

/*多行注释

多行注释*/

P17数组

数组定义：

类型 数组名[元素个数]

int a[10];

int N=10;int a[N];

char b[24];

double c[3];

数组不能动态定义。[]中不能是一个变量，但是这个变量的值是固定的值的话可以

访问数组中的元素

数组名[下标]

第一个元素的下标是0，不是1

数组的初始化

1.所有元素赋值为0（只有0可以）

int a[10]={0};　　//只是将第一个元素赋值0，后面会自动初始化为0

2.赋值不同的值用逗号隔开

int a[10]={1,2,3,4,5,6,7,8,9,0};

3.只给一部分元素赋值，未被赋值的元素自动初始化为0.

int a[10]={1,2,3,4,5}

4.只给出各个元素的值，不指定长度，由编译器自动判断

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

5.C99新特性：指定初始化的元素。未被赋值的元素自动初始化为0

int a[10]={[3]=110,[5]=120,[8]=114};

P18啪啪啪

数组不能动态定义。[]中不能是一个变量，但是这个变量的值是固定的值的话可以

P19字符串处理函数

字符数组

char str1[10]={'F','I','S','H','\0'};

char str2[]={'F','I','S','H','\0'};不指定长度

char str3[]={"FISH"};使用字符串常量初始化字符数组，可以省略大括号

char str4[]="FISH";

字符串处理函数 在<string.h>

strcat连接字符串

strcmp比较字符串

strcpy拷贝字符串strcpy(str1,str2)把str2复制给str1.注意\0也会复制。要保证str2长度小于str1

strlen获取字符串长度 数组元素的个数

strncat连接字符串（受限）

strncmp比较字符串（受限）

strncpy拷贝字符串（受限）strncpy(str1,str2,元素个数n) 把str2的n个元素复制给str1，要自己加\0 str1[n]='\0'

P20二维数组

定义：

类型 数组名[行数][列数]

访问：

数组名[行下标][列下标]

a[0][0]第一行第一列

二维数组初始化

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

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

int a[2][3]={{1},{4}}对每行第一个赋值，其余自动为0

int a[2][3]={0}整个二维数组初始化为0

int a[][3]={1,2,3,4,5,6} 行数可不写，自动判断

C99新特性 指定初始化的元素。

P21指针

通过变量名访问变量

指针和指针变量

类型名 *指针变量名

char *pa；定义一个指向字符型的指针变量

int *pb；定义一个指向整型的指针变量

获取某个变量的地址，可以使用取地址运算符&

char *pa=&a;

int *pb=&f;

如果需要访问指针变量指向的数据，可以使用取值运算符*

printf("%c,%d",*pa,*pb)

避免访问未初始化的指针(野指针)

#include<stdio.h>

int main(){

int *a;　　//要初始化才对int *pa，a；*pa=&a；*pa=123　　但是字符串可以int *ps="haha"

*a=123;

return 0;

}

P22指针和数组

数组名是数组第一个元素的地址

指向数组的指针

char *p;

p=a;或p=&a[0];

对指针加减运算相当于指向距离指针所在位置向前或向后第n个元素

对比标准的下标访问数组元素，使用指针进行间接访问的方法叫作指针法

p+1并不是将地址加1，而是指向数组的下一个元素。（根据定义的类型长度加1个单位的int、char。。。）

P23指针数组和数组指针

int *p1[5]　指针数组 是数组 变量类型指针int *

int (*p2)[5] 数组指针 是指针 指向一个数组

定义数组指针int (*p)[3]=array

P24指针和二维数组

数组array[4][5]

*(array+1)==array[1]指向数组第二行第一个元素

*(array+1)+3==&array[1][3] 指向数组第二行第4个元素

结论

*(array+i)==array[i]

((array+i)+j)==array[i][j]

((*(array+i)+j)+k)==array[i][j][k]

P25void指针和NULL指针

void指针 称为通用指针，可以指向任意类型的数据。也就是说任何类型的指针都可以赋值给void指针。

int a=1024;

int *pi=&a;

void *pv;

pv=pi;

NULL指针 用于指针和对象

#define NULL ((void *)0)

好习惯 不清楚将指针初始化为什么地址时，将她初始化为NULL

NULL不是NUL，NUL是空字符\0

P26指向指针的指针

int num=520;

int *p=#

int **pp=&p;两次解引用

好处:

避免重复分配内存

只需进行一次修改

P27常量和指针

常量：1，'a'，2.5

或者

#define N 10

或者使用const修饰

const int price = 520

指向常量的指针

指针可以修改为指向不同的变量

指针可以修改为指向不同的变量

可以通过解引用来读取指针指向的数据

不可以通过解引用来修改指针指向的数据

常量指针

指向非常量的常量指针:

指针自身不可被修饰

指针指向的值可以被修改

指向常量的常量指针:

指针自身不可以被修改

指针指向的值也不可以被修改

指向"指向常量的常量指针"的指针

P28函数

函数的定义;

类型 函数名(参数列表)

{

函数体

}

函数的声明

写在主函数mian前面

或者

在mian方法后写函数，在前面写函数声明

```
void f1(int);
int main(){
int x=1;
f1(x);
}
void f1(int x){
...
}
```

P29参数和指针

形参 形式参数

实参 实际参数

传值和传址

可变参数

P30指针函数和函数指针

指针函数int *f()

char *f1(char c){

return "字符串"

}

字符数组只要指名开头地址。结尾是\0自动判断

不要 返回局部变量的指针

函数指针int (*p)()

int square(int num){

return num*num;

}

int main(){

int num=5;

int (*fp)(int);

fp=square;函数名相当于地址 等价于fp=&square

printf("%d",(*fp)(num))

}

P31局部变量和全局变量

不同函数的变量无法相互访问

在函数里面定义的 局部变量

在函数外边定义的 全局变量

如果不对全局变量进行初始化，它会自动初始化为0

如果在函数内部存在一个与全局变量同名的局部变量，并不会报错，而是在函数中屏蔽全局变量（全局变量不起作用）

extern关键字 告诉程序变量在后面定义了，不要报错

不要大量使用全局变量，因为

1、全局变量从被定义开始，直到程序退出才能被释放，会占用更多的内存

2、污染命名空间，虽然局部变量会屏蔽全局变量，但这样一来也会降低程序的可读性，人们往往很难一下子判断出每个变量的含义和作用范围

P32作用域和链接属性

变量的作用范围 作用域

c语言编译器可以确认4种不同类型的作用域：

代码块作用域block scope

文件作用域file scope

原型作用域prototype scoope

函数作用域function scope

代码块作用域

在代码块中定义的变量，具有代码块作用域。从变量定义的位置开始，到标志该代码结束的右大括号}处

尽管函数的形参不在大括号内定义，但其同样具有代码块作用域，隶属于包含函数体的代码块

文件作用域

任何在代码块之外声明的标识符都具有文件作用域。从声明的位置开始，到文件的结尾处

函数名也具有文件作用域，因为函数名本身也在代码块之外

原型作用域

原型作用域只适用于那些在函数原型中声明的参数名。函数在声明的时候可以不写参数的名字（但参数的类型是必须要写上的），其实函数原型的参数名还可以随便写一个名字，不必与形式参数相匹配(当然，这样做毫无意义)。

void func(int a,int b,int c);

void func(int d,int e,int f){

...

}

函数作用域

函数作用域只适用于goto语句的标签，作用将goto语句的标签限制在同一个函数内部，以及防止出现重名标签。

定义和声明

当一个变量被定义的时候，编译器为变量申请内存空间并填充一些值

当一个变量被声明的时候，编译器就知道该变量被定义在其他地方

声明通知编译器该变量名及相关的类型已存在，不需要再为此申请内存空间。

局部变量既是定义又是声明

定义只能来一次，否则就叫做重复定义某个同名变量；而声明可以有很多次

链接属性

.c编译.o，.o链接lib库文件

external外部的 多个文件中声明的同名标识符表示同一个实体

internal内部的 单个文件中声明的同名标识符表示同一个实体static

none无 声明的同名标识符被当做独立不同的实体

只有具有文件作用域的标识符才能拥有external或internal的链接属性，其它作用域的标识符都是none属性

默认情况下，具备文件作用域的标识符拥有external属性。也就是说该标识符允许跨文件访问。对于external属性的标识符，无论在不同文件中声明多少次，表示的都是同一个实体

P33生存期和存储类型

c语言的变量拥有两种生存期

静态存储期

自动存储期

具有文件作用域的变量属于静态存储期，函数也属于静态存储期，属于静态存储期的变量在程序执行期间将一直占据存储空间，直到程序关闭才释放。

具有代码块作用域的变量一般情况下属于自动存储期。属于自动存储期的变量在代码块结束时将自动释放存储空间。

生存期

存储类型

指存储变量值的内存类型

auto

register

static

extern

typedef

自动变量auto

在代码块中声明的变量默认的存储类型就是自动变量，使用关键字auto来描述,可省略

```
#include<stdio.h>
int main(){
auto int l,j,k;
return 0;
}
```

寄存器变量register

将一个变量声明为寄存器变量，那么该变量就有可能被存放于CPU的寄存器中。

寄存器变量和自动变量在很多方面都是一样的，都拥有代码块作用域、自动存储期和空连接属性

不过将变量声明为寄存器变量，那么你就没有办法通过取址运算符获得该变量的地址。

静态局部变量static

static使得局部变量具有静态存储期，所以它的生存期与全局变量一样，直到程序结束才释放

作用于文件作用域的static和extern，static关键字使得默认具有external链接属性的标识符变成internal链接属性，而extern关键字是用于告诉编译器这个变量或函数在别的地方已经定义过，先去别的地方找找，不要急着报错。

typedef

P34递归

汉诺塔

谢尔宾斯基三角形

目录树的索引

女神的自拍

函数调用本身

递归必须有结束条件，否则程序将崩溃

```
#include<stdio.h>
void f1(){
static int count =10;
printf("HI\n");
if(--count) f1();
```

```
}
int main(){
f1();
return 0;
}
```

递归求阶乘

```
#include<stdio.h>
long fact(int num)                  //循环方法
{
                                    long result;
                                    for(result=1;num>1;num--)
                                    {
                                    result*=num;
                                    }
                                    return result;
}
long fact1(int num)                 //递归方法
{
                                    long result;
                                    if (num>0)
                                    {
                                        result=num*fact1(num-1);
                                    }
                                    else
                                    {
                                        result=1;
                                    }
                                    return result;
}
int main(void)
{
int num;
printf("请输入一个整数：");
scanf("%d",&num);
printf("%d的阶乘是：%d",num,fact1(num));
}
```

实现递归满足两个条件

调用函数本身

设置了正确的结束条件

普通程序员用迭代，天才程序员用递归，但我们宁可做普通程序员

P35汉诺塔

递归求解汉诺塔

简单分为三个步奏：

1、将前63个盘子从X移动到Y上

2、将最底下的第64块盘子从X移动到Z上

3、将Y上的63个盘子移动到Z上

而1、3都能在此分解为类似的三步

1分为

1.1、将前62个盘子从X移动到Z上

1.2、将最底下的第63块盘子从X移动到Y上

1.3、将Z上的62个盘子移动到Y上

3分为

3.1、将前62个盘子从Y移动到X上

3.2、将最底下的第64块盘子从Y移动到Z上

3.3、将X上的62个盘子移动到Y上

。。。

```
#include<stdio.h>
void hanoi(int n,char x,char y,char z) //将x上的n个盘子借助y移动到z上
{
    if(n==1)
    {
        printf("%c-->%c\n",x,z);
    }
    else
    {
        hanoi(n-1,x,z,y);
        printf("%c-->%c\n",x,z);
        hanoi(n-1,y,x,z);
    }
}
int main()
```

```
{
    int n;
    printf("请输入汉诺塔的层数：");
    scanf("%d",&n);
    hanoi(n,'X','Y','Z');
    return 0;
}
```

P36快速排序

分治法

大事化小，小事化了

快速排序

基本思想:通过一趟排序将待排序数据分割成独立的两部分，其中一部分所有元素均比另一部分的元素小，然后分别对两部分继续进行排序，重复上述步骤直到排序完成。

P37动态内存管理1

4个库函数stdlib.h

malloc申请动态内存空间

free释放动态内存空间

calloc申请并初始化一系列内存空间

realloc重新分配内存空间

malloc

函数原型void *malloc(size_t size);

向系统中申请分配size个字节的内存空间，并返回一个指向这块空间的指针。

函数调用成功，返回一个指向申请的内存空间的指针，由于返回的类型是void指针，所以它可以被转换成任何类型的数据；如果函数调用失败，返回值是NULL。另外，如果size参数设置为0，返回值也可能为NULL，但并不意味着函数调用失败。

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *ptr;
    ptr=(int *)malloc(sizeof(int));
    if(ptr==NULL)
    {
        printf("分配内存失败！\n");
```

```
        exit(1);
    }
    printf("请输入一个整数:\n");
    scanf("%d",ptr);
    printf("请输入的整数是:%d\n",*ptr);
    return 0;
}
```

free

函数原型：void free(void *ptr);

free函数释放ptr参数指向的内存空间。该内存空间必须是由malloc、calloc或realloc函数申请的。否则，该函数将导致未定义行为。如果ptr参数是NULL，则不执行任何操作。注意：该函数并不会修改ptr的值，所以调用后仍然指向原来的地方(变为非法空间)。

在以上程序中添加

free(ptr);

printf("请输入的整数是:%d\n",*ptr);//此时输出的值是无意义的

一个吃满内存的死循环

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    while(1)
    {
        malloc(1024);
    }
    return 0;
}
```

内存泄露

如果申请的动态内存没有及时释放会怎么样?

c语言没有垃圾回收机制

导致内存泄露主要两种情况：

隐式内存泄露(用完内存块没有及时使用free函数释放)

丢失内存快地址

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *ptr;
    int num=123;
```

```
    ptr=(int *)malloc(sizeof(int));
    if(ptr==NULL)
    {
        printf("分配内存失败！\n");
        exit(1);
    }
    printf("请输入一个整数:\n");
    scanf("%d",ptr);
    printf("请输入的整数是:%d\n",*ptr);
    ptr=&num;//ptr又指向了别处，原来malloc申请的内存块丢失，free不能释放原来的动态内存。
    free(ptr);//另一个错误，free不能释放这里不是动态内存的ptr
    return 0;
}
```

P38动态内存管理2

malloc还可以申请一块任意尺寸的内存空间

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *ptr=NULL;
    int num,i;
    printf("请输入待录入的整数的个数：");
    scanf("%d",&num);
    ptr=(int *)malloc(num*sizeof(int));
    for(i=0;i<num;i++)
    {
        printf("请录入第%d个数：\n",i+1);
        scanf("%d",&ptr[i]);
    }
    printf("你录入的整数是：\n");
    for(i=0;i<num;i++)
    {
        printf("%d ",ptr[i]);
    }
    free(ptr);
    return 0;
}
```

初始化内存空间

以mem开头的函数被编入字符串标准库，函数的声明包含在string.h这个头文件中

memset使用一个常量字节填充内存空间

memcpy 拷贝内存空间

memmove 拷贝内存空间

memcmp比较内存空间

memset示例

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define N 10//没分号
int main()
{
	int *ptr=NULL;
	int i;
	ptr = (int *)malloc(N*sizeof(int));
	if(ptr==NULL)
	{
		exit(1);
	}
	memset(ptr,0,N*sizeof(int));
	for(i=0;i<N;i++)
	{
		printf("%d ",ptr[i]);
	}
	free(ptr);
	return 0;
}
```

calloc

函数原型void *calloc(size_t nmemb,size_t size);

calloc函数在内存中动态地申请nmemb个长度为size的连续内存空间(即申请的总空间尺寸为nmemb *size)，这些内存空间全部被初始化为0。

calloc函数与malloc函数的一个重要区别是：

calloc函数在申请完内存后，自动初始化该内存空间为零；malloc函数不进行初始化操作，里面数据是随机的。

所以下面两种写法是等价的：

calloc（）分配内存空间并初始化

```
int *ptr=(int *)calloc(8,sizeof(int));
```

malloc()分配内存空间并用memset()初始化

```
 int *ptr=(int *)malloc(8*sizeof()int);
memset(ptr,0,8*sizeof(int));
```

memcpy示例

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main(){
        int *ptr1=NULL;
        int *ptr2=NULL;
        ptr1=(int *)malloc(10 *sizeof(int));
        ptr2=(int *)malloc(20 *sizeof(int));
        memcpy(ptr2,ptr1,10);
        free(ptr1);
        //略...对ptr2进行若干操作
        free(ptr2);
        return 0;
}
```

realloc

函数原型void *realloc(void *ptr,size_t size);

以下几点需要注意；

realloc函数修改ptr指向的内存空间大小为size字节

如果新分配的内存空间比原来的大，则旧内存块的数据不会发生改变；如果新的内存空间大小小于旧的内存空间，可能会导致数据丢失，慎用!

如果ptr参数为NULL，那么调用该函数就相当于调用malloc(size)

如果size参数为0，并且ptr参数不为NULL，那么调用该函数就相当于调用free(ptr)

除非ptr参数为NULL，否则ptr的值必须由先前调用malloc、calloc或realloc函数返回

P39C语言的内存布局

C语言的内存布局规律

根据内存地址从低到高分别划分为：

代码段（Text segment）

数据段（Initialized data segment）

BSS段（Uninitialized data segment）

栈（Stack）

堆（Heap）

代码段

代码段（Text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

数据段

数据段（Initialized data segment）通常用来存放已经初始化的全局变量和局部静态变量。

BSS段

BSS段（Bss segment/Uninitialized data segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。BSS是英文Block Started by Symbol的简称，这个区段中的数据在程序运行前将被自动初始化为数字0。

堆

堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩展或缩小。当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上；当利用free等函数释放内存时，被释放的内存从堆中被剔除。

栈

大家平时可能经常听到堆栈这个词，一般指的就是这个栈。栈是函数执行的内存区域，通常和堆共享同一片区域。

堆和栈的区别

申请方式：

堆由程序员手动申请

栈由系统自动分配

释放方式：

堆由程序员手动释放

栈由系统自动释放

生存周期：

堆的生存周期由动态申请到程序员主动释放为止，不同函数之间均可自由访问

栈的生存周期由函数调用开始到函数返回时结束，函数之间的局部变量不能互相访问

发展方向：

堆和其它区段一样，都是从低地址向高地址发展

栈则相反，是由高地址向低地址发展

P40高级宏定义

宏定义

文件包含

条件编译

不带参数的宏定义

#define PI 3.14

为了和普通的变量进行区分，宏的名字通常我们约定是全部由大写字母组成

宏定义只是简单的进行替换，并且由于预处理是在编译之前进行，而编译工作的任务之一就是语法检查，所以编译器不会对宏定义进行语法检查

宏定义不是说明或语句，在末尾不必加分号

宏定义的作用域是从定义的位置开始到整个程序结束

可以用#undef来终止宏定义的作用域

```
#include<stdio.h>

#define PI 3.14

int main(){
    int r;
    float s;
    printf("请输入圆的半径：");
    scanf("%d",&r);
    //#undef PI
    s=PI*r*r;
    printf("圆的面积是：%.2f\n",s);
    return 0;
}
```

宏定义允许嵌套

```
#include<stdio.h>

#define PI 3.14

#define R 6371

#define V PI*R*R*R*4/3

int main(){
    printf("地球的体积是：%.2f\n",V);
    return 0;
}
```

带参数的宏定义

```
#define MAX(x,y) (((x)>(y))?(x):(y))
```

注意MAX没有空格(x,y)

```
#include<stdio.h>

#define MAX(x,y) (((x)>(y))?(x):(y))

int main(){
    int a,b;
```

```
    printf("请输入两个数：");
    scanf("%d%d",&a,&b);
    printf("较大的数是：%d\n",MAX(a,b));
    return 0;
}
```

括号不能省

```
#include<stdio.h>
#define SQUARE(x) x*x
int main(){
    int a;
    printf("请输入一个数：");
    scanf("%d",&a);
    printf("%d的平方是：%d\n",a,SQUARE(a));
    printf("%d的平方是：%d\n",a+1,SQUARE(a+1));//如果算x+1的平方会x+1*x+1，宏很傻，直接替换，不会帮你加括号 。保险做法#define SQUARE(x) (x)*(x)，也不完美例如计算a++则(a++)*(a++)或造成a加两次
    return 0;
}
```

P41内联函数和一些鲜为人知的技巧

内联函数 解决程序中函数调用的效率问题。(但会增加编译时间)

定义函数前加上inline关键字

内联函数执行过程是在主函数中展开，而不是主函数-子函数-返回主函数。

现在的编译器很聪明，不写inline，也会自动将一些函数优化成内连函数

```
#include<stdio.h>
inline int square(int x)
{
    return x*x;
 }
int main(){
    int i=1;
    while(i<=100){
        printf("%d的平方是：%d\n",i-1,square(i++)); //提高编译效率，也可以避免想宏定义出现两次加的错误
    }
    return 0;
}
```

#和##

两个预处理运算符

在带参数的宏定义中，#运算符后面应该跟一个参数，预处理会把这个参数转化为一个字符串。

```
#include<stdio.h>
#define STR(s) # s
int main(){
    printf("%s\n",STR(FISHC));
    return 0;
}
```

会把多个空格转化为一个空格

```
#include<stdio.h>
#define STR(s) # s
int main(){
    printf(STR(Hello   %s num=%d\n),STR(FISHC),520);
    return 0;
}
```

##运算符被称为记号连接运算符，比如我们可以使用

##运算符连接两个参数

```
#include<stdio.h>
#define TOGETHER(x,y) x ## y
int main(){
    printf("%d\n",TOGETHER(2,50));
    return 0;
}
```

可变参数

带参数的宏定义也可以使用可变参数

#define SHOWLIST(...) printf(#__VA_ARGS__)

其中...表示可变参数，__VA_ARGS__在预处理中被实际的参数集所替换(就像参数列表)（两边是两个下划线哦）。

```
#include<stdio.h>
#define SHOWLIST(...) printf(#__VA_ARGS__)
int main(){
    SHOWLIST(FishC,520,3.14\n);
    return 0;
}
```

可变参数允许空参数

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

```
#define PRINT(format,...) printf(#format,##__VA_ARGS__)
int main(){
    PRINT(num=%d\n,520);
    PRINT(Hello FishC!\n);//这个里面可变参数是空的
    return 0;
}
```

P42结构体

结构体声明：

struct 结构体名 //英语单词structure结构

{

结构体成员1;

结构体成员2;

结构体成员3;

...

};//这里有一个分号

示例

struct Book

{

char title[128];

char author[40];

float price;

unsigned int date;

char publisher[40];

};

定义结构体类型变量

struct结构体名称 结构体变量名;

或者

在声明结构体时定义

struct结构体名{

。。。

} 变量名;//不过这时是全局变量

注意：如果

typedef struct结构体名称{

。。。

}简称；或typedef struct结构体名 简称;

使用typedef给结构体定义了一个简称，并不是变量

结构体可以放在函数外(全局)，可以放在函数内(局部)

```
#include<stdio.h>
struct Book
{
char title[128];
char author[40];
float price;
unsigned int date;
char publisher[40];
} book;
int main(){
        //struct Book book;
        printf("请输入书名：");
        scanf("%s",book.title); //字符数组名指向开头元素地址 不用&
        printf("请输入作者：");
        scanf("%s",book.author);
        printf("请输入售价：");
        scanf("%f",&book.price);
        printf("请输入出版日期：");
        scanf("%d",&book.date);
        printf("请输入出版社：");
        scanf("%s",book.publisher);
        printf("\n===========数据录入完毕========\n");
        printf("书名：%s\n作者：%s\n售价：%.2f\n出版日期：%d\n出版社：%s\n",book.title,book.author,book.price,book.date,book.publisher);
}
```

初始化一个结构体变量

```
book={
"数学",
"小明",
21.5,
20200311,
"家里蹲大学出版社"};
```

c99新特性:初始结构体的指定成员值

结构体指定初始化成员使用点号运算符和成员名。

比如我们可以让程序只初始化Book的price成员；

struct Book book={.price=21.5};

多个可以不按结构体顺序进行初始化

struct Book book={.price=21.5,.title="数学"};

结构体的长度与内存对齐

```
#include<stdio.h>
int main(){
    struct A
    {
        char a;
        int b;
        char c;
    } a={'x',520,'o'};
    printf("size of a=%d\n",sizeof(a));//结果结构体a长度为12 ,因为内存对齐(让CPU更快处理数据 )
    // 如果顺序为char a;char c;int b;则长度为8
    return 0;

}
```

拓展：扫码阅读 如何手工打包c结构体声明，减少内存空间占用

P43结构体数组和结构体指针

结构体嵌套

```
struct Date
{
int year;
int month;
int day;
};
struct Book
{
char title[128];
char author[40];
float price;
struct Date date;
char publisher[40];
} book={
"数学",
```

"小明",

21.5,

{2020,3,11},

"家里蹲大学出版社"};

.。。。

printf("日期：%d-%d-%d",book.Date.year,book.Date.month,book.Date.day);

。。。

结构体数组

声明方法

第一种

struct结构名称

{

结构体成员;

...

}数组名[长度];

第二钟

struct结构体名称 数组名[长度];

结构体数组初始化struct Book book[3]={{。。。},{。。。},{。。。}}

结构体指针

struct Book *pt;

pt=&book;

通过结构体指针访问成员

(*结构体).成员名 //*优先级低于点. (*pt).title

结构体指针->成员名 pt->title

P44传递结构体变量和结构体指针

传递结构体变量

两个相同结构体类型的结构体变量可以赋值。book1=book2;

```
#include<stdio.h>
struct Date {
    int year;
    int month;
```

```
    int day;
};
struct Book {
    char title[128];
    char author[40];
    float price;
    struct Date date;
    char publisher[40];
};
struct Book getInput(struct Book book){
    printf("请输入书名：");
    scanf("%s",book.title);
    printf("请输入作者：");
    scanf("%s",book.author);
    printf("请输入售价：");
    scanf("%f",&book.price);
    printf("请输入出版日期：");
    scanf("%d-%d-%d",&book.date.year,&book.date.month,&book.date.day);
    printf("请输入出版社：");
    scanf("%s",book.publisher);
    return book;

}
void printBook(struct Book book){
    printf("书名：%s\n作者：%s\n售价：%.2f\n出版日期：%d-%d-%d\n出版社：%s\n",book.title,book.author,book.price,book.date.year,book.date.month,book.date.day,book.publisher);
}
int main(){
    struct Book book1,book2;
    printf("请输入第一本书的信息");
    book1=getInput(book1);
    printf("请输入第二本书的信息");
    book2=getInput(book2);
    printf("显示第一本书的信息");
    printBook(book1);
    printf("显示第二本书的信息");
    printBook(book2);
    return 0;
}
```

提高执行效率，函数可以不用结构体传址，而是传递指向结构体变量的指针。

```
#include<stdio.h>
struct Date {
	int year;
	int month;
	int day;
};
struct Book {
	char title[128];
	char author[40];
	float price;
	struct Date date;
	char publisher[40];
};
void getInput(struct Book *book){
	printf("请输入书名：");
	scanf("%s",book->title);
	printf("请输入作者：");
	scanf("%s",book->author);
	printf("请输入售价：");
	scanf("%f",&book->price);
	printf("请输入出版日期：");
	scanf("%d-%d-%d",&book->date.year,&book->date.month,&book->date.day);
	printf("请输入出版社：");
	scanf("%s",book->publisher);
}
void printBook(struct Book *book){
	printf("书名：%s\n作者：%s\n售价：%.2f\n出版日期：%d-%d-%d\n出版社：%s\n",book->title,book->author,book->price,book->date.year,book->date.month,book->date.day,book->publisher);
}
int main(){
	struct Book book1,book2;
	printf("请输入第一本书的信息");
	getInput(&book1);
	printf("请输入第二本书的信息");
	getInput(&book2);
	printf("显示第一本书的信息");
	printBook(&book1);
```

```
    printf("显示第二本书的信息");
    printBook(&book2);
    return 0;
}
```

动态申请结构体

使用malloc函数为结构体分配存储空间

修改

```
int main(){
    struct Book *book1,*book2;
    book1=(struct Book *)malloc(sizeof(struct Book));
    book2=(struct Book *)malloc(sizeof(struct Book));
    if(book1==NULL||book2==NULL){
        printf("内存分配失败！\n");
        exit(1);//需要stdlib.h
    }
    printf("请输入第一本书的信息");
    getInput(book1);
    printf("请输入第二本书的信息");
    getInput(book2);
    printf("显示第一本书的信息");
    printBook(book1);
    printf("显示第二本书的信息");
    printBook(book2);
    free(book1);
    free(book2);
    return 0;
}
```

篇幅限制，后面部分在这里——传送门

C语言 小甲鱼 带你学C带你飞

来自文集：C语言/C++/数据结构/算法

< 上一篇 小甲鱼C语言《带你学C带你飞》学习笔... | 查看目录 2/2 | 下一篇 > 到底啦～看看其他的吧

20 评论

loading...

正能量的康sir

粉丝: 3737 阅读: 6606

关注

查看目录 | 来自文集：C语言/C++/数据

更多

专栏投稿
前去写文章

专栏帮助
查看专栏使用

20