一个字=2byte（字节）=8bit（位）

Ctrl+shift+b：编译

.\main.exe：运行

## 15、C数组

### 1.4 动态数组与静态数组

静态数组的特点包括：

内存分配：在程序编译时，静态数组的内存空间就被分配好了，存储在栈上或者全局数据区。

大小固定：静态数组的大小在声明时确定，并且无法在运行时改变。

生命周期：静态数组的生命周期与其作用域相关。如果在函数内部声明静态数组，其生命周期为整个函数执行期间；如果在函数外部声明静态数组，其生命周期为整个程序的执行期间。

动态数组特点如下：

内存分配：动态数组的内存空间在运行时通过动态内存分配函数手动分配，并存储在堆上。需要使用 malloc、calloc 等函数来申请内存，并使用 free 函数来释放内存。

大小可变：动态数组的大小在运行时可以根据需要进行调整。可以使用 realloc 函数来重新分配内存，并改变数组的大小。

生命周期：动态数组的生命周期由程序员控制。需要在使用完数组后手动释放内存，以避免内存泄漏。

## 16、C枚举

枚举类型通常用于为程序中的一组相关的常量取名字，功能类似于宏定义

### 1.1、枚举变量的定义

#### 1）先定义枚举类型后定义枚举变量：

enum Day

{

MON =1, TUE, WED, THU, FRI, SAI, SUN

};

(内容不用加封号)

enum Day day;

#### 2）枚举类型，枚举变量一起定义

enum Day

{

MON =1, TUE, WED, THU, FRI, SAI, SUN

} day;

#### 3）枚举省略只定义枚举变量

enum

{

MON =1, TUE, WED, THU, FRI, SAI, SUN

} day;

### 1.2、枚举遍历

枚举变量是被当做int或者unsigned int类型来处理的，所以仅仅在一些特殊情况下是可以实现遍历的，可以实现遍历的前提是枚举类型必须是连续的。

举例：

#include "stdio.h"

#include "stdlib.h"

int main()

{

enum Color

{

red=1,

blue,

green

};

enum Color favorite\_color;

printf("input your favorite color\n");

scanf("%d",&favorite\_color);

switch(favorite\_color)

{

case red:

printf("红色");

break;

case blue:

printf("蓝色");

break;

case green:

printf("绿色");

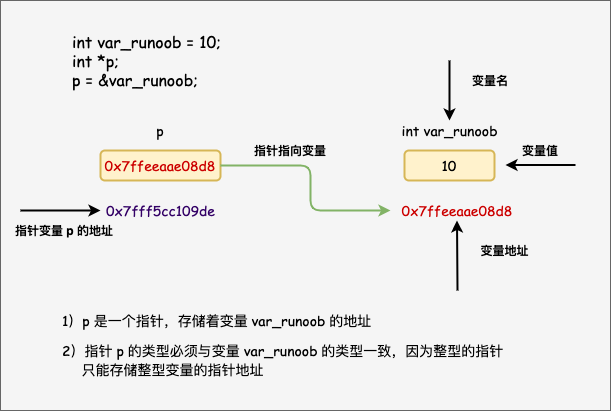
break;

}

}

输入不同的数字有不同的结果输出。

## 17、C指针



\*p表示对于p这个地址所对应的值。P=&var\_runloob表示，把后者的地址赋给p。

指针操作的步骤：定义一个指针变量、把变量地址赋值给指针、访问指针变量中可用地址的值。

指针声明时，如果没有一个确切地址可以赋值，那么就需要给与空指针（也可以不给）。

int \*p = NULL;

对于\*p，p，&p，其中，p表示指针，存储着数据的地址，它指向某个数据的地址。例如：int \*p表示声明了一个指向整数的指针p。

\*p表示对于指针p的解引用，即p处存储地址所指向的数据。

&p表示取地址，即指向指针的指针。

### 1.1、指针算数运算

例：如果一个ptr是指向地址为1000的32位整型指针，运行ptr++，则指向1004这个地址

一个字=2byte=8bit，32位即为4个字节，那么每次向后移动32位即移动4个字节。

如果ptr指向一个地址为1000的字符，那么ptr++将会指向下一个字符。因为字符通常占用一个一个字节。

总结：

（1）指针的每一次递增，它其实会指向下一个元素的存储单元。

（2）指针的每一次递减，它都会指向前一个元素的存储单元。

（3）指针在递增和递减时跳跃的字节数取决于指针所指向变量数据类型长度，比如 int 就是 4 个字节。

#### 1）递增指针：

#include "stdio.h"

const int max=3;

int main(void)

{

int var[]={100,23,56};

int \*ptr=NULL;

ptr = var;

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

{

printf("store position:var[%d] = %p\n", i , (void\*)ptr);

printf("store data:var[%d] = %d\n", i, \*ptr);

/\*指向下一个位置\*/

ptr++;

}

}

其中ptr指针获得了var的地址，进行后面的指针运算，%p是指输出指针所表示的地址16进制数值.void\*表示对于后面的指针进行类型强制转换，使得p指针的类型变为无类型指针，以便于后续对于指针地址的处理。

#### 2）递减指针：

#include "stdio.h"

const int max = 3;

int main(void)

{

int var[] = {100,56,89};

int \*ptr;

ptr = &var[max-1];

for (int i=max-1;i>=0;i--)

{

printf("store poistion:var[%d] = %p\n", i, ptr);

printf("stroe data:var[%d] = %d\n", i, \*ptr);

ptr--;

}

}

其中，对于ptr指针地址的赋值，因为要倒数所以先对ptr直接赋于最后一个数据来进行倒数。所以ptr=&var[max-1]，后续，需要进行ptr指针递减计算。

#### 3）指针的比较：

如果俩指针是相关的，例如指的是同一数组的中元素的地址，那么可以对于指针进行比较，与while循环相互配和，实现基本的循环。

#include "stdio.h"

const int max=3;

int main(void)

{

int var[] = {10,100,1000};

int \*ptr;

ptr = var;

int i = 0;

while(ptr<&var[max])

{

printf("store poistion:var[%d] = %p\n", i, ptr);

printf("store data:var[%d] = %d\n",i,\*ptr);

i++;

ptr++;

}

}

其中while(ptr<&var[max-1])即实现基本的循环结构。也可以进行倒序输出，方法相类似。

### 1.2、指针数组

即声明一个包含指针的数组，数组中每个元素都指向一个地址。（构想，通过指针数组来访问多个其他数组，即把其他数组的首地址全部存在一个指针数组中，通过指针数组来访问某一个数组）。

#include "stdio.h"

const int max=3;

int main(void)

{

int var1[]={10,100,100};

int var2[]={12,13,14};

int var3[]={5,6,4};

int \*ptr[max];

ptr[0] = var1;

ptr[1] = var2;

ptr[2] = var3;

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

{

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

{

printf("store poistion:var%d[%d]=%p\n",i+1,j,(void\*)ptr[i]+j);

printf("store data:var%d[%d]=%d\n",i+1,j,\*(ptr[i]+j));

}

}

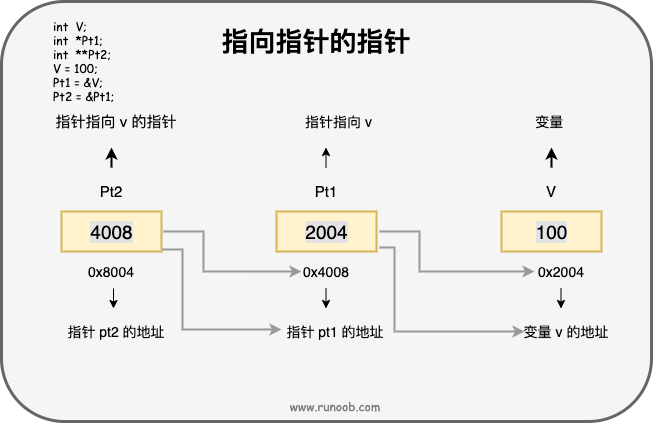
}

其中，通过指针数组来存储三个整数数组的头指针，然后通过依次访问头指针来访问整个数组。

### 1.3、指向指针的指针

指向指针的指针是一种多级间接寻址的形式，或者说是一个指针链。通常，一个指针包含一个变量的地址。当我们定义一个指向指针的指针时，第一个指针包含了第二个指针的地址，第二个指针指向包含实际值的位置。





指针Pt2指向指针Pt1的地址，指针Pt1指向实际数值的地址。

#include "stdio.h"

int main(void)

{

int v=100;

int \*ptr1=NULL;

int \*\*ptr2=NULL;

ptr1 = &v;

ptr2 = &ptr1;

printf("v position:%p\n",&v);

printf("point to v position:%p\n",(void\*)ptr1);

printf("point to ptr1 position:%p\n",(void\*)ptr2);

printf("ptr1 position:%p\n",&ptr1);

printf("v value:%d\n",v);

printf("v value:%d\n",\*ptr1);

printf("v value:%d\n",\*\*ptr2);

}

&v=ptr1

&ptr1=ptr2

v=\*ptr1=\*\*ptr2

多级指针，更多的像是开门，先开大门，再开小门。

### 1.4、传递指针给函数

#include "stdio.h"

#include "time.h"

void timeget(double \*per);

int main(void)

{

double currenttime;

timeget(&currenttime);

printf("current time:%d\n", currenttime);

}

void timeget(double \*per)

{

\*per = time(NULL);

}

通过对自定义的函数进行调用。设置一个读取当前时间的函数来读取当前时间。然后将这个时间存在自己定义的指针地址下，然后直接输出该地址下的数据。\*per = time(NULL)用于读取当前时间的秒数。

#include "stdio.h"

float average(float \*balance, int size);

int main(void)

{

float arrey[5]={10,20,15,63,20};

float averagenumber = average(arrey,5)/5;

printf("the average is:%f\n",averagenumber);

return 0;

}

float average(float \*balance, int size)

{

float sum = 0;

float \* key= balance;

float \*key;

key = balance;

while(balance < (key+size-1))

{

sum += \*balance;

balance++;

}

return sum;

}

float \* key= balance;

float \*key;

key = balance;

其中第一句float \* key= balance;类似于float \* key= NULL;第二句为常规写法。

1.5、b站视频指针

B站指针教学：指针也是作为一种数据类型

1）指针基本运算

#include "stdio.h"

int main(void)

{

    int a=10;

    int \*p = &a;//这一句类似于int \*p = NULL;p = &a;

    printf("the address of a is %p\n",&a);

    printf("a = %d\n",a);

    printf("the address pointed by p is %p\n",p);

    \*p = 12;

    int b = 15;

    \*p = b;//只对于\*p所代表的数值进行了更改，即对于a的数值进行了更改，并没有对于p所存放的指整形地址有改变。p是一个可以存放整型变量的地址的变量。\*p为指向p地址处存储的一个值，该步骤称为解引用。\*p表示p指向地址的值

    printf("a = %d\n",a);

    printf("the address of p is %p\n",&p);

    printf("the address pointed by the p is %p\n",p);

    printf("the size of the integer is %d byte",sizeof(int));//一个integer有4个字节，则p+1的地址为p地址加4。因为integer占4个字节。

}

#### 2）指针强制类型转换

#include "stdio.h"

int main(void)

{

    int a = 1025;

    int \*p = NULL;//整形指针，指针类型用于后面解引用时，判断是那种数据类型，不同数据类型不同的数据输出。

    p = &a;

    printf("the address of the a is %p\n", p);

    printf("the address of the p+1 is %p\n", p+1); //p+1的地址与p地址相差4个字节

    printf("the value of a %d\n", \*p);

    printf("the value of \*(p+1) is %d\n",\*(p+1));

    char \*k;

    k = (char\*)p;//类型强制转换，将integer类型变为char型，一个字符对应一个字节

    printf("the address of the a is %p\n", k);

    printf("the address of the p+1 is %p\n", k+1);//k+1的地址与k相差一个字节

    printf("the value of a %d\n", \*k);

    printf("the value of \*(p+1) is %d\n",\*(k+1));

}

注意指针所指向的地址

#### 3）void\*指针的应用

通用型指针

#include "stdio.h"

int main(void)

{

    int a = 1025;

    int \*p = NULL;

    p = &a;

    printf("the address of the a is %p\n", p);

    printf("the address of the p+1 is %p\n", p+1);

    printf("the value of a %d\n", \*p);

    printf("the value of \*(p+1) is %d\n",\*(p+1));

    void \*k;

    k = p;

    printf("the address of the a is %p\n", k);

}

在其中运用void \*来声明指针不需要在后续的地址赋值中进行类型强转，但是不能解应用，也不可以进行指针的算术运算。void\*指针主要运用于当不知道所用的指针类型时运用void\*指针，也叫通用型指针。

#### 4）指针的指针

#include "stdio.h"

int main(void)

{

    int a = 6;

    int \*p = NULL;

    p = &a;

    \*p = 5;

    printf("the address of a is %p\n",p);

    printf("the value of a is %d\n",\*p);

    int \*\*q;

    q = &p;

    printf("the address of p is %p\n", &p);

    printf("the address of p is %p\n", \*q);

    printf("the value of \*p is %d\n", \*\*q);

    int \*\*\*k = &q;

    printf("the address of q is %p\n", &q);

    printf("the address of q is %d\n", \*\*\*k);

    printf("%p\n",\*k);

    \*\*q = \*p + 5;

    printf("the value of the \*\*q is %d", \*\*q);

}

注意解引用时，指针的地址。即k解引用的数值为p的地址，再次解引用为a的地址，再次解引用为所对应的数值。

#### 5）指针传值&传引用

#include "stdio.h"

#include "main.h"

int main(void)

{

    int a = 10;

    int \*p = NULL;

    p = &a;

    add(p);//p为实参地址

    printf("the value of a+1 is %d",\*p);

    return 0;

}

void add(int \*b)//\*b为形式参数

{

    \*b = \*b + 1;

}

其中，a是一个局部变量，生命周期仅仅在这个一个函数中，因为程序运行时会给main函数在栈（stack）中开辟一块内存即栈帧，当main函数中调用其他函数时，程序会停止去执行被调用的函数，然后再在栈中开辟一块内存一个新的栈帧来存储调用函数的变量，但他只能访问自己栈帧中的变量，当函数调用结束以后，程序控制流再一次回到当时停止的位置，而计算机此时会清理结束运行结束的函数的栈帧，进而被调用函数的栈帧被清除。因此需要用指针来调用其他函数来在a的地址上操作。

函数在调用时会被分配栈来存储函数的变量，当main函数中调用到其他函数时就会停止main函数，去执行其他函数，这种结构被称为函数调用栈，栈顶的函数最先被调用，并且栈的大小时固定的，超出栈的大小程序会崩溃。

库函数编写格式：

#ifndef MAIN\_H

#define MAIN\_H

void add(int \*b);

#endif

以上的例子时通过传递地址，在调用函数中进行解引用，因此称为传引用，如果传递数值则称为传数值。

#### 6）指针和数组

左值：即为即将存储的内容提供一个空间（存储单元）；

右值：即为所要存储的常量，这个常量可以为数值，也可以为地址。因此有些左值可以做右值而，右值不可以做左值。

int a[]={};

int \*p = a;

p = &a;

其中，\*p为一个指针可以做左值也可以做右值，因此可以进行地址的自增p++，而a为&a[0]的地址，为一个地址常量，不可以地址自增a++；他可以在原有的&a[0]上进行赋值，此时可以做左值。

#include "stdio.h"

int main(void)

{

    int a[] = {10,0,100,20,56};

    int \*p = a;

    p++;//可以写p++，不可以写a++，因为左值需要提供一个存储空间

    printf("the address of a is %p\n", a);

    printf("the address of a is %p\n", &a[0]);

    printf("the address of a is %p\n", p);

    printf("the value of a is %d\n", \*p);

    printf("the value of a is %d\n", a[0]);

}

++p先是自增再引用，p++先是引用后自增

#### 7）数组作为函数参数

#include "stdio.h"

#include "main.h"

int main(void)

{

    int a[] = {1,2,3,4,5,6};

    int size = sizeof(a)/sizeof(a[0]);

    int sum = sumallarray(a,size);

    printf("the sum of array is %d", sum);

}

int sumallarray(int a[], int size)//其中int a[]相当于int \*a

{

    int sum = 0;

    for(int i = 0;i<size;i++)

    {

        sum+= a[i]; //a[i]对于计算机来说相当于\*(a+i)，即直接对于a+i这个地址处的数值进行操作

    }

    return sum;

}

在其中sizeof(a)/sizeof(a[0])，在主函数和在被调用函数中不一样。首先程序运行到mian函数时首先对于main函数，在栈stack中开辟一块栈帧，运来存储main函数中的变量，其中就有对于a[]这个数组一共有6数，则给他开辟24个字节，再给size和sum各4个字节。然后，运行到sunmallarray函数时，调用函数，再在栈中给这个子函数开辟一块栈帧，但此时在这个栈帧中给子函数的变量存储空间时，因为传递给子函数的为int a[]和int size，其中size为传值给子函数说明数组的大小，而int a[]对于子函数来说并没有把所有的a[]数组内容进行传递，而是传递了数组的首地址，因此int a[]对于计算机来说相当于int \*a，这一步不是传值，而是传引用。而对于子函数中后面的sum+= a[i]，相当于\*(a+i)。

#include "stdio.h"

#include "main.h"

int main(void)

{

    int a[] = {1,2,3,4,5,6};

    int size = sizeof(a)/sizeof(a[0]);

    doublearray(a,size);

    // for(int c = 0;c<size;c++)

    // {

    //  printf("the sum of array is %d\n", a[c]);

    // }

}

void doublearray(int a[],int size)

{

    for(int i = 0;i<size;i++)

    {

        a[i] = 2\*a[i];

        printf("the sum of array is %d\n", a[i]);

    }

}

在这段代码中，就体现了计算对于a[i] = 2\*a[i];是当作\*(a+i)的，即直接对于main函数中的数组进行解引用。因此在子函数中对于数组a[]仅分配了四个字节，即用这四个字节来逐个解引用对对应地址的数值进行操作。

#### 8）指针和字符数组（上）

#include "stdio.h"

int main(void)

{

    char str[6];

    str[0] = 'h';

    str[1] = 'e';

    str[2] = 'l';

    str[3] = 'l';

    str[4] = 'o';

    str[5] = '\0';

    printf("%s",str);

}

对于字符串需要在后面加\0，原因首先对字符串，一个字符串占一个字节。字符串中\0相当于结束符，告诉计算机这是结束符，不再向后读取数据。结束符一般在数组最后面加\0，也可以省略，但必须要空一个数组地址。如果没有，则会在现有数据的最后一个地址之后，继续寻址读取数据，以至于出现乱码。对于string.h库的函数均需要在后面跟着\0做结束符。

#include "stdio.h"

#include "string.h"

int main(void)

{

    char c[] = "hello";//后面的NULL是隐式的，默认加上NULL，但如果写成{'h','e','l','l','o','\0'}这个NULL就是显式的必须手动的加上

    printf("the byte of c is %d\n",sizeof(c));

    printf("%s\n",c);

    int length = strlen(c);

    printf("length = %d",length);

}

其中输出结果，c[]有6个字节，长度为5。表明在运用字符串字面值赋值时，计算机会默认的给字符串后面加一个NULL即\0，这对于字符串字面值来说是隐式的。但长度计算时只是把\0当作结束符号，不计入长度中。但会占用字节。

对于指针中指针和数组首地址，在传递时都是进行解引用。

#include "stdio.h"

#include "string.h"

void print(char c[])

{

    int i = 0;

    while(c[i] != '\0')

    {

        printf("%c",c[i]);

        i++;

    }

    printf("\n");

}

int main(void)

{

    char c[] = "hello!";

    print(c);

}

Printf函数的程序实现，NULL的作用就在这个代码中显现。因为数组的传递是采用传引用的格式，Printf函数并不知道要打印的数组的长度，因此需要给他结束符。

对于print函数也可以改进改为：

#include "stdio.h"

#include "string.h"

void print(char c[])

{

    while(\*c != '\0')

    {

        printf("%c",\*c);

        c++;

    }

    printf("\n");

}

int main(void)

{

    char c[] = "hello!";

    print(c);

}

对于去掉去其中的i是因为，可以用c++来访问数组的内容。是因为对于数组名不可以用c++，这样是违法的，因为左值必须能够提供一个空间来存储右值，但是当c数组传引用到子函数时，他就可以进行c++，因为此时的c为一个指针，指向的是c这个数组的首地址。

#### 8）指针和字符数组（下）

#include "stdio.h"

#include "string.h"

void print(char \*c)

{

    printf("%c",\*(c+1));

    while(\*c != '\0')

    {

        printf("%c",\*c);

        c++;

    }

    printf("\n");

}

int main(void)

{

    char \*c = "hello!";

    print(c);

}

其中，c是指向hello的一个指针，这也意味着可以用c[]来访问字符串中的字符，例如可以用c[0]来访问字符串中的h。但由于char \*c=“hello“，字符串的字面值具有常量的属性，因此在分配空间时并不是在栈帧中分配，而是在code程序中。

字符串字面值通常被视为常量，所以使用 char \*c 声明的指针 c 具有指向常量字符的属性，不能用于修改字符串中的字符。如果您尝试修改 c 指向的字符，会导致未定义的行为。如果需要一个可修改的字符串，应该使用字符数组来存储字符串。（字符串字面值是由字符组成的序列，可以包含字母、数字、标点符号等字符。字符串字面值可以包括单引号（'）或双引号（"）括起来。并且字符串：char \*a = ‘hello world’;实际上是char a[] = “hello world”,一个字符一个字节，所以hello world占12个字节）

注意，加结束符的意义在于给计算机一个指令用来确定字符串的具体大小，例如：char a[] = “hello world”在计算机不知道有多少字符时用“\0”来确定。而字符数组如：char a[3] = {‘a’, ‘b’, ‘c’}，已经明确了字符数的大小，即3个字节，所以不需要加“\0”，所以他是合法的。

#### 9）指针与二维数组

#include "stdio.h"

#include "string.h"

int main(void)

{

    int b[2][2]={1,2,3,6};

    int (\*p)[2] = b;

    printf("%d\n",\*(\*p+1));

    printf("address is %p\n",b);

    printf("address is %p\n",p);

    printf("address is %p\n",\*p);

    printf("address is %p\n",b[0]);

    printf("address is %p\n",\*p+1);

    printf("address is %p\n",&b[0][1]);

}

对于二维数组可以把它看作两个一维数组，例如b[0]，b[1]，其中每个一维数组又有2个整形数据。对于这个二维数组要定义指针，则需要像(\*P)[2]来进行定义，否则会报错。定义好指针以后p处即为b数组的地址。对于数组内容的访问，首先\*p表示访问b[0]和b[1]，那么可以很好理解，\*p+1即表示对于对于b[0]数组中的b[0][0]的地址，所以\*（\*p+1）就可以理解对于\*p+1地址的解引用，进而你访问到相应的地址处的数值。

#### 10）指针与多维数组

#include "stdio.h"

#include "string.h"

int main(void)

{

    int b[2][3]={1,2,3,6,8,9};

    int (\*p)[3] = b;//p是一个指向包含三个整形的一维数组的指针

    printf("%d\n",\*(\*p+1));

    printf("address is %p\n",b);

    printf("address is %p\n",p);

    printf("address is %p\n",\*p);

    printf("address is %p\n",b[0]);

    printf("address is %p\n",&b[0][0]);

    printf("address is %p\n",\*p+1);

    printf("address is %p\n",&b[0][1]);

}

其中，b，p，\*p，b[0]，&b[0][0]均为相同的地址。b和\*b均为指针，不过b指针指向的是一维数组b[]，而\*b指针指向的是整形的地址。b[i][j]类似于\*(\*b[i]+j)，也类似于\*(\*(b+i)+j)，因为对于数组来说a[i]等价于\*(a+i)。则：

b[i][j]=\*(b[i]+j)=\*(\*(b+i)+j)

#include "stdio.h"

int main(void)

{

    int a[3][2][2] = {1,2,3,4,5,6,7,8,9,0,10,11};

    int (\*p)[2][2] = a;

    printf("%p\n",a);

    printf("%p\n",p);

    printf("%p\n",a[0]);

    printf("%p\n",a[0][0]);

    printf("%p\n",&a[0][0][0]);

    printf("%d\n",\*\*\*p);

    printf("%d\n",a[0][0][0]);

    printf("%d\n",a[0][0][1]);

    printf("%d\n",\*((a[0][0])+1));

    printf("%d\n",\*(\*((a[0]))+1));

    printf("%d\n",\*(\*(\*(a)))+1);

}

其中,要明白a[i]=\*(a+i)，其中a,a[0],a[0][0],p,&a[0][0][0]，在只输出地址的时候都是一样的结果，但意义不一样，首先a是一个指向二维数组的指针，a[]是一个指向一维数组的指针，a[][]是指向整型的指针。对于多维数组的指针指向都可以用指针解引用的概念来理解。

多维数组的传引用

#include "stdio.h"

extern int function(int (\*a)[2][2],int size);

int main(void)

{

    int a[][2][2] = {1,2,3,4,5,6,7,8,9,0,10,11};

    int (\*p)[2][2] = a;

    printf("%p\n",a);

    printf("%p\n",p);

    printf("%p\n",a[0]);

    printf("%p\n",a[0][0]);

    printf("%p\n",&a[0][0][0]);

    printf("%d\n",\*\*\*p);

    printf("%d\n",a[0][0][0]);

    printf("%d\n",a[0][0][1]);

    printf("%d\n",\*((a[0][0])+1));

    printf("%d\n",\*(\*((a[0]))+1));

    printf("%d\n",\*(\*(\*(a)))+1);

    int size = sizeof(a)/sizeof(a[0][0][0])/2/2;

    printf("the byte of a is %d\n",size);

    int sum = function(a,size);

    printf("the sum of a is %d\n",sum);

}

int function(int (\*a)[2][2],int size)

{

    int sum = 0;

    for (int i = 0;i<size;i++)

    {

        for (int j = 0;j<2;j++)

        {

            for (int k = 0;k<2;k++)

            {

                sum+= a[i][j][k];

            }

        }

    }

    return sum;

}

对于多维数组的行不需要给出具体数，而除了行以外的都需要在数组声明时就写明具体数值。除此以外，当三维数组即多维数组传递时，例如int function(int (\*a)[2][2],int size)，可以写为a[][2][2]也可以写为(\*a)[2][2]，数组与在传递时与指针解引用是一致的。

#### 11）指针和动态内存——栈和堆

指针中的堆与数据结构中的堆有区别，而指针中的栈与数据结构的栈类似，都是一种先进后出的存储结构。

Excemple1

#include "stdio.h"

#include "stdlib.h"

int main(void)

{

    int a[2][2] = {{1,2},{5,6}};

    int \*\*p = (int\*\*)malloc(2\*sizeof(int\*));//申请了一块地址，用于存储指向指针的指针，即将二维数组看做两个一维数组，每个数组存储两个整形,申请了可以存放两个指针的指针的内存

    for(int i = 0;i<2;i++)

    {

        p[i] = (int\*)malloc(2\*sizeof(int));//列

    }

    for(int k=0;k<2;k++)

    {

        for(int j=0;j<2;j++)

        {

            p[k][j] = a[k][j];

        }

    }

    for(int i = 0;i<2;i++)

    {

        int c = 0;

        while(c<2)

        {

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

            c++;

        }

    }

    printf("%d",sizeof(int\*));

    for (int i = 0;i<2;i++)

    {

        free(p[i]);//先释放指向整数的指针，再释放指向数组的指针

    }

    free(p);

}

运用malloc函数申请动态内存，存储二维数组，首先申请指向指针的指针，申请时根据要存储的数组大小（行或者列），例子中的申请存放2个指针的指针的内存，然后再给二级指针解引用后的地址申请地址，每个解引用的地址又申请两个内存用于存储整形数值。然后再把数值存在申请好的地址上。之后内存释放，内存释放需要显示的释放，申请的内存会一直存在在整个程序的运行过程中，对于这个例子的的内存释放，从内往外依次释放，即先释放指向整型的地址，再释放指向指针的指针。对于malloc函数，其申请的地址在heap层，并且其返回值为一个void\*类型指针，即通用性指针，为了能够运用到整型数组，则需要给这个指针进行强制类型转换为int\*类型。

Excemple2

#include "stdio.h"

#include "stdlib.h"

int main(void)

{

    int a[2][3] = {{1,2,5},{3,4,6}};

    int \*\*p = (int\*\*)malloc(2\*sizeof(int\*));

    for(int i = 0;i<2;i++)

    {

        p[i] = (int\*)malloc(3\*sizeof(int));

    }

    for(int i = 0;i<2;i++)

    {

        for(int j = 0;j<3;j++)

        {

            p[i][j] = a[i][j];

        }

    }

    for(int i = 0;i<2;i++)

    {

        int j = 0;

        while(j<3)

        {

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

            j++;

        }

    }

    for(int i=0;i<2;i++)

    {

        free(p[i]);

    }

    free(p);

}

#### 12）指针和动态内存malloc,calloc,realloc,free

其中，malloc函数返回一个void\*类型指针，void\* malloc(size\_t size)，其中，size\_t表示unsigned int，即无符号整数，size可以表示为n\*sizeof即单元个数与单元大小的乘积，不将返回的指针初始化。

Calloc也是返回一个void\*类型指针，void\* calloc(int n,size\_t size)，calloc函数只是将malloc函数中一起输入的size分为了两个参数n和单个单元的size，并且将返回的指针进行了初始化。

Realloc函数也是返回一个void\*类型指针，他是对进行分配后的动态内存大小修改，即在已经用malloc后者calloc分配好的内存的基础上进行扩展或者缩小，如int \*a = (int\*)malloc(n\*sizeof(int)),将这个内存进行改变为int \*b = realloc(a,m\*sizeof(int))，其中m\*sizeof(int)为新的想要变化的内存大小。然后会将之前在a地址下的东西拷贝过去，并且地址分配如果是缩减即在原a的地址上进行缩减，要是扩展即如果在原a地址的基础上可以扩展到所需要的地址即在原地址上扩展，如果不够则需要另外寻找来寻找的地址，同时会释放之前的地址（但是对于部分编译器计算机，内存管理系统通常不会主动将超出新范围的部分初始化，而是会保留原内存块的数据，但这种方法不可取）。如果新的内存大小为0，则该语句相当于free(a)，即int \*b = (int\*)realloc(a,0);相当于free(a)，而当int \*a = realloc(NULL,m\*sizeof(int))等价于int \*a = (int\*)malloc(m\*sizeof(int))。

#include "stdio.h"

#include "stdlib.h"

int main(void)

{

    int n;

    printf("input your wanner array\n");

    scanf("%d",&n);

    int \*a = (int\*)malloc(n\*sizeof(int));  //int \*a = (int\*)calloc(n,sizeof(int));二者一样不过会初始化指针

    for (int i = 0;i<n;i++)

    {

        a[i] = i+1;

    }

    // free(a);

    int \*b = (int\*)realloc(a,(n/2)\*sizeof(int));

    //int \*b = (int\*)realloc(NULL,(n/2)\*sizeof(int));//相当于\*a = (int\*)malloc(n\*sizeof(int));

    //int \*b = (int\*)realloc(a,(n/2)\*sizeof(int));//相当于free(a)

    for (int j = 0;j<n;j++)

    {

        printf("%d ",b[j]);

    }

}

二维数组，列不变的情况下改行数：

#include "stdio.h"

#include "stdlib.h"

int main(void)

{

    int a[2][3] = {{1,2,3},{4,5,6}};

    int \*\*p = (int\*\*)malloc(2\*sizeof(int\*));

    for(int i=0;i<2;i++)

    {

         p[i] = (int\*)malloc(3\*sizeof(int));

    }

    for(int i=0;i<2;i++)

    {

        for(int j=0;j<3;j++)

        {

            p[i][j] = a[i][j];

        }

    }

    for(int i=0;i<2;i++)

    {

        for(int j=0;j<3;j++)

        {

            printf("%d ",p[i][j]);

        }

    }

    printf("\n");

    int \*\*c = (int\*\*)realloc(p,3\*sizeof(int\*));

    if(c==NULL)

    {

        printf("application of memory defeat");//需要加内存申请成功的标志，因为有可能申请不成功

    }

    else

    {

        p=c;

        p[2] = (int\*)malloc(3\*sizeof(int));//二级指针有两个存储空间，为其中的二级指针再申请所指向的空间。

    }

    for(int i=0;i<3;i++)

    {

        for(int j=0;j<3;j++)

        {

            printf("%d ",p[i][j]);

        }

    }

    for(int i =0;i<3;i++)

    {

        free(p[i]);释放内存由内向外释放内存，首先释放最内部的内存，即一级指针指向的地址，再释放二级指针所指向的地址。

    }

free(p);

}

行不变的情况下对列的扩充：

#include "stdio.h"

#include "stdlib.h"

int main(void)

{

    int a[2][3] = {{1,2,6},{3,4,5}};

    int \*\*p = (int\*\*)malloc(2\*sizeof(int\*));

    for(int i=0;i<2;i++)

    {

        p[i] = (int\*)malloc(3\*sizeof(int));

    }

    for(int i=0;i<2;i++)

    {

        for(int j=0;j<3;j++)

        {

            p[i][j]=a[i][j];

        }

    }

    for(int i=0;i<2;i++)

    {

        for(int j=0;j<3;j++)

        {

            printf("%d ",p[i][j]);

        }

    }

    printf("\n");

    for(int i=0;i<2;i++)

    {

        p[i] = (int\*)realloc(p[i],4\*sizeof(int));

    }

    if(\*p==NULL)

    {

        printf("application of memory defeat");

    }

    // else

    // {

    //  p = c;

    // }

    for(int i=0;i<2;i++)

    {

        for(int j=0;j<4;j++)

        {

            printf("%d ",p[i][j]);

        }

    }

    for(int i=0;i<4;i++)

    {

        free(p[i]);

    }

    free(p);

}

对行变更好记得对于新申请的指向一维数组的指针进行地址分配，及对于新增的列进行分配地址，对于列变行不变，只需要重新对指向整形指针的内存块分配地址即可。

#### 13）指针和动态内存-内存泄漏

内存泄漏是由于heap堆中未使用或者未引用的内存块引起的，即每次通过malloc申请的的内存块未得到释放导致的。而栈中最多会发生栈溢出，因为栈的大小都是固定的，并且在栈中函数调用完以后会自动的清除之前的内存。所以不会发生内存泄漏。

Example

#include "stdio.h"

#include "stdlib.h"

#include "time.h"

extern void play(int bet);

int cash = 100;

int main(void)

{

    printf("welcome to virtual game\n");

    printf("total cash is %d\n",cash);

    int bet;

    while(cash>0)

    {

        printf("what is your bet? how many cash$?\n");

        scanf("%d",&bet);

        if(bet==0 || bet > cash)

        break;

        play(bet);

        printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    }

}

void play(int bet)

{

char c[3] = {'j','q','k'};

//char c\* = (char\*)malloc(3\*sizeof(char));

//c[0] = 'j';

//c[0] = 'q';

//c[0] = 'k';

    printf("shuffling...\n");

    srand(time(NULL));//generate a seeding number

    for(int i = 0;i<3;i++)

    {

        int x = rand() % 3;//rand函数产生随机数，对3取余，保证0，1，2为所取值

        int y = rand() % 3;

        char temp = c[x];

        c[x] = c[y];

        c[y] = temp;

    }

    int playerguess;

    printf("where is the queen -1,2 or 3?\n");

    scanf("%d",&playerguess);

    if(c[playerguess - 1] == 'q')

    {

        cash += 3\*bet;

        printf("you win! the result is %c,%c,%c\n,total cash = %d\n",c[0],c[1],c[2],cash);

    }

    else

    {

        cash -= 3\*bet;

        printf("you lose! the result is %c,%c,%c\n,total cash = %d\n",c[0],c[1],c[2],cash);

}

//free(c);

}

其中对于子函数的调用，运用栈内存每次调用后会自动的释放内存，而运用动态内存即在堆上进行内存分配需要主动的去释放内存。对于%s用于输出字符串，而运用%c输出字符。

对于其中的随机数函数scand(time(NULL))和rand()，而这需要配合使用，因为对于电脑来说，计算机并不能产生随机数，而是一些写好的无规则数，将这些无规则的数字n等份，然后给着n等分的数进行编号，通过scand()函数来取得这些编号，但是对于scand()如果不加以参数，他将会一直读取同一组数，为了使他更加随机，那么就需要给他加种子数，即用time(NULL)来实现种子数的变化进而一直变化。

（记忆版本）

#include "stdio.h"

#include "stdlib.h"

#include "time.h"

extern void play(int bet);

int cash = 100;

int main(void)

{

    int bet;

    printf("welcome to virual game!\n");

    while(cash>0)

    {

        printf("your current total cash is %d\n",cash);

        printf("please input your bet cash?\n");

        scanf("%d",&bet);

        if(bet==0||bet>cash)

        {

            printf("please input right number\n");

        }

        play(bet);

        printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    }

}

void play(int bet)

{

    char \*a = (char\*)malloc(3\*sizeof(char));

    a[0] = 'j';

    a[1] = 'q';

    a[2] = 'k';

    printf("having three roles are distributed three position\n");

    printf("shuffling------\n");

    srand(time(NULL));

    char temp;

    for (int i = 0; i < 5; i++)

    {

        int x = rand() % 3;

        int y = rand() % 3;

        temp = a[x];

        a[x] = a[y];

        a[y] = temp;

    }

    printf("plase guess which position is the queen?-1,2 or 3\n");

    int position;

    scanf("%d", &position);

    if(a[position-1]=='q')

    {

        cash += 3\*bet;

        printf("you win! the position is %c,%c,%c\n",a[0],a[1],a[2]);

    }

    else

    {

        cash -= 3\*bet;

        printf("you lose! the position is %c,%c,%c\n",a[0],a[1],a[2]);

    }

    free(a);

}

#### 14）函数返回指针

#include "stdio.h"

#include "stdlib.h"

extern int \*add(int \*a, int \*b);//函数指针的运用，是因为返回值是一个指针

int main(void)

{

    int a = 5, b = 6;

    int \*ptr = add(&a,&b);

printf("the sum is %d",\*ptr);

free(ptr);

}

int \*add(int \*a,int \*b)

{

    int \*c = (int\*)malloc(sizeof(int));//运用动态内存是因为对于栈来说，局部地址或者局部变量可以从下往上传，而不能从上往下传

    \*c = \*a+\*b;

    return c;

}

对于其中的函数，首先，extern int \*add(int \*a, int \*b);表示声明了一个外部函数，使得其他函数可以调用它，其中，int\* add(int\* a, int\* b)表示申明了一个函数指针，该函数，接受两个整型指针作为输入后，并返回一个指向整数的指针。

对于int \*add(int \*a, int \*b)子函数内部，首先，要明白栈是一个先进后出的运行结构，首先，当main函数被调用时，栈在最底部为main函数分配地址空间，该空间不可以改变大小，然后，面函数调用add函数，该函数在栈中位于main函数的上方，add函数可以访问到main函数中的局部变量，而当add函数运行完以后，计算机会释放add函数所使用过的空间，但计算机可能没有重写或清除，add函数的局部变量使用过的地址所存储的数据。因此如果对返回的子函数中使用过的栈中的地址进行解引用也有可能得到想要的数据。

对于栈中，局部变量或者传引用只可以从下往上传，而不可以从上往下传，因此，为了使得栈中的被调用子函数的数值可以被运用，要注意返回值的运用条件，首先，对于返回值不会随着值函数的结束而被清除，因为，返回值通常存储在寄存器或者一些其他的特殊位置，如，堆栈，堆内存（对于数据特别的结构体），因此子函数的生命周期结束以后还是可以被访问到，但也仅限于调用函数可以访问，其他函数不可以访问。所以，对于返回值为具体数值时可以不用管内存的情况，但对于返回值为一个地址时就需要注意数据存储内存了，因为，在子函数中运行结束以后，返回了一个计算得到的数据的指针，如果这个指针是指向了子函数中的局部变量，那么他就有可能在主函数中解引用时得不到想要的值，因为返回了地址，党对这个地址解引用时，子函数对应的栈帧已经被释放了，能解引用想要的值，也只是运气好。所以，对于子函数中数据，如果要选择返回指针，一般要选择存储在heap或者global中。

#### 15）函数指针

对于函数指针理解，首先，计算机指令都是一些二进制数，通过执行一些二进制数字来实现一些功能。对于函数也是，计入有一个.c文件，将它给编译器，编译器首先会将这个.c文件，翻译为计算机所能看懂的二级制可执行文件，也就是.exe文件。计算机会将这这可执行文件存在它的磁盘上，当程序开始执行时，计算机会为他分配一些内存，也就是code区域，然后将可执行文件的机器指令拷贝过来，指令的执行是不是在磁盘上执行，而是在主存上执行的，每条指令都有一个内存地址，而一个函数就相当于连续的几个指令，而想要通过指针调用该函数，只需要访问到该函数的第一条指令或者指令的入口，即可对该函数进行调用。

Int\* function(int a,int b)这是声明了一个函数，其返回值是一个指向整型的指针。

Int (\*P)(int,int)这是声明了一个函数指针，其中int表明所指向的函数的返回值是int类型的，（int，int）表明所指向函数的两个参数都是整数。对于函数的调用，取地址阶段可以写&add，也可以直接写函数名，意义一样。

#include "stdio.h"

#include "stdlib.h"

int add(int a,int b)

{

    return a+b;

}

int main(void)

{

    int a=5,b=6;

    int (\*p)(int, int);

    //p = &add;

    p = add;

    //int c = (\*p)(a,b);

    int c = (\*p)(a,b);

    printf("%d",c);

}

对于这种函数的调用是直接对内存操作，不需要进行函数的外部声明。并且对于上述程序，首先对于，申请的函数指针要与所指向的函数的返回值类型一致，其次对于指向函数的参数类型也要一致。另外，对于函数的取地址，可以直接用函数名，因为一个函数的函数名也代表着它的首地址，并且函数的实现也可以直接用p(a,b)。而不用(\*p)(a,b)，其具体意义与用函数名直接代替其地址一致。

#include "stdio.h"

void printfhello(char\* name)

{

    printf("hello %s\n",name);

}

int main(void)

{

    void (\*ptr)(char\*);

    ptr = printfhello;

    ptr("tom");

}

对于这段代码，首先，通过void类型的指针ptr访问printfhello函数，再传入一个char类型的指针，因为输入为字符串，所以用双引号，并用char\*进行传引用，用传入的是一个字符串。字符串只能这样。

#### 16）函数指针使用案例（回调函数）

#include "stdio.h"

extern void calling(void (\*ptr)());

extern void print();

int main(void)

{

    calling(print);

}

void print()

{

    printf("hello world");

}

void calling(void (\*ptr)())

{

    ptr();//指针指向被调用的函数，也叫回调

    //(\*ptr)();

}

其中，运用了指针函数，但也需要进行函数声明，如果called函数在main函数之前，则不用声明了，它会隐式的声明，除此以外需要显示的声明。

排序：

#include "stdio.h"

#include "stdlib.h"

int compare(int a,int b)

{

    if(a>b)

    {

        return -1;

    }

    else if(a<b)

    {

        return 1;

    }

    else

    {

        return 0;

    }

}

void buddlesort(int \*a,int size,int //冒泡算法(\*compare)(int,int))//(\*compare)(int,int)代表函数指针

{

    for(int i=0;i<size;i++)

    {

        for (int j=0;j<size-1;j++)

        {

            if(compare(a[j],a[j+1])>0)//回调函数

            {

                int temp = a[j];

                a[j] = a[j+1];

                a[j+1] = temp;

            }

        }

    }

}

int main(void)

{

    int a[] = {1,3,2,6,4,5};

    int size = sizeof(a)/sizeof(a[0]);

    buddlesort(a,size,compare);//此处分别为传引用，传数值，传引用，最后一可以写为&compare意味着取函数的地址，函数名也有地址的意思

    for(int i=0;i<size;i++)

    {

        printf("%d ",a[i]);

    }

}

通过写一个比较函数，compare实现两个数值的比较，然后将比较的结果返回一个标志位，来进行换位。

改良快速排序：

#include "stdio.h"

#include "stdlib.h"

#include "math.h"

int compare(const void\* a,const void\* b)

{

    int A = \*((int\*)a);

    int B = \*((int\*)b);

    return -A+B;//传入的第一个数与第二个数比较，即为从小到大，反之从大到小

}

int main(void)

{

    int a[] = {1,3,2,6,4,5};

    int size = sizeof(a)/sizeof(a[0]);

    // buddlesort(a,size,&compare);

    qsort(a,size,sizeof(int),&compare);

    for(int i=0;i<size;i++)

    {

        printf("%d ",a[i]);

    }

}

对于qsort表示，quicksort，即快速排序，时stdlib.h这个头文件中的，其中qsort(a,size,sizeof(int),&compare);这个函数中的参数，第一项为传入的数组，字符串，字符，结构体等，均可，第二项为前一项包含元素的个数，第三项为每个元素的大小，第四项为一个比较函数的地址。

#### 17）指针及其应用（基于ARM contex—M）

对于int \*ptr；ptr++意味着ptr =ptr+1\*sizeof(int)，并且对于指针运算来说，\*解引用符（&引用符）优先级高于+。

x=\*ptr++:为两句语句：先取值再移动指针

x=\*ptr；

ptr++；

一个字根据GBK来说通常占2个字节，但对于UTF8来说有时候占3个有时候占4个字节。

### 1.6、左值与右值

左值（L-value）：

左值是表达式的标识符，它代表可以被赋值的内存位置。

左值通常是变量、数组元素或类对象的成员。

左值可以出现在赋值语句的左边，例如 x = 5; 中的 x 是左值。

左值具有持久性，可以在程序的不同部分引用和修改。

右值（R-value）：

右值是表达式的值，它代表一个具体的数据值或临时结果。

右值通常是常量、字面量、表达式的计算结果或函数返回值。

右值不能被赋值，因为它们没有关联的内存位置。

右值具有短暂性，通常在表达式计算后就不再存在。

### 1.7、const的应用

全局 const 变量 GLOBAL\_CONST 存储在静态存储区，可以在程序的任何地方访问。

局部 const 变量 LOCAL\_CONST 存储在栈上，在函数调用时分配，在函数返回时释放。

const 修饰的字符串常量 STR\_CONST 存储在常量存储区，在程序运行期间保持不变。

 总之，const 修饰的变量可以存储在静态存储区、栈或常量存储区，具体取决于变量的声明位置和用途

## 18、字符串

### 1.1、字符串

#include "stdio.h"

#include "stdlib.h"

int main(void)

{

    char \*p = (char\*)malloc(4\*sizeof(char));

    p[0] = 'j';

    p[1] = 'k';

    p[2] = 'l';

    p[3] = '\0';

    char c[4] = {'j','k','l','\0'};

    printf("%c%c%c\n",c[0],c[1],c[2]);

    printf("%s\n",c);

    printf("%s\n",p);

    printf("%c%c%c\n",p[0],p[1],p[2]);

}

字符串输入，要用双引号括起来，而字符输入用单引号。并且，对于有已知长度的的字符数组，如果不用字符串输出（即%s）用字符输出（即%c）是可以不加结束符\0的，其他的都需要加\0，并且对于动态内存空间申请的指针数组也需要考虑字符串输出还时字符输出，要求与之前的一样。

即只要是字符串输出必须后面加“\0”,字符输出可以不加。

基本的字符串函数：

#include "stdio.h"

#include "string.h"

int main(void)

{

    char a[20] = "hello world!";

    char b[20] = "ello worl!";

    char c[20];

    printf("%d\n",strlen(a));

    printf("%d\n",strlen(b));

    printf("%d\n",strlen(c));

    strcat(a,b);

    printf("strcat(a,b) = %s\n",a);

    printf("strlen(strcat(a,b)) = %d\n",strlen(a));

}

### 1.2、字符串变量

在字符串中运用，char\* s = “hello world”，其中不能对字符串的内容进行修改了，因为，char\* s声明了一个指针s，这个指针指向hello world一个字符串常量，这个字符串常量位于程序的code区域，在编译时就已经生成了，所以他是只读的，所以实际上这段代码为：const char \*s = “hello world”。所以对于字符串的内容是不能修改的。

如果要写可以对字符串修改的代码，那么就需要写：char a[] = “hello world”数组来修改其内容。并且对于字符串来说，在printf函数中，若果两段字符串只用双引号，双引号中间没有其他东西，计算机会默认的把他们连接起来。效果和反斜杠一样。

#include "stdio.h"

#include "string.h"

int main(void)

{

    printf("please aribitury input\

input your favorite thing");

\\printf("please aribitury input""input your favorite thing");

char\* v = "hello";

类似于const char v[]="hello";

    char a = 'h';

    char d[] = "hello";

    printf("%s\n",v);

    printf("%c\n",a);

    printf("%s",d);

}

\*v所指向的地址和a[]的地址相差很远。所以他们位于不同的区域，其中可以验证，d[]位于本地栈，\*v指向的地址在code中。

#include <stdio.h>

int main()

{

    int i = 0;//本地变量

    char\* s = "hello world";//位于代码段code段，不可修改

    // s[0] = 'b';

    char\* s2 = "hello world";//s和s2为相同的

    char s3[] = "hello world";//s3和i很近

    printf("%p\n",s3);

    printf("%p\n",&i);

    printf("%p\n",s);

    printf("%p\n",s2);

    s[0] = 'b';

    printf("Here!s[0]=%c\n",s3[0]);

}

数组：作为本地变量被自动回收。->构造字符串

指针：位于code段不可以被修改，所以通常指针用于处理参数和动态内存分配。->处理字符串

char\*不一定表示字符串，其本意是指向字符的指针，可能是字符数组，只有字符数组的结尾有“\0”时，才能说他是字符串。

### 1.3、字符串的输入输出

#include <stdio.h>

int main()

{

    char word[8];

    char word2[8];

    scanf("%s",word);

scanf("%s",word2);

    //scanf("%7s",word);

//scanf("%7s",word2);

    printf("%s##%s##\n",word,word2);

    return 0;

}

通过输入hello world来验证对于，scanf函数来说：

1、他不知道字符串有多大。就容易导致如果输入超出数组极限的字符串，就会导致部分数据丢失，或者程序崩溃。对此需要给%s这里加上限值，即最大字符数是8，那么就要写%7s，起到的作用是告诉scanf函数去读几个数据，剩下的给下一部分读。

2、每次读取一个完整的单词（到空格，tab或者回车为止）。

字符串常见错误：

char\* string；对于定义的指针变量不能直接使用，要对其进行一定的初始化万一指针初始是指向一个了错值，那么就会对程序有影响。所以指针一定要初始化。

### 1.4、字符串数组

#include <stdio.h>

int main()

{

    char \*a[] = {"january","february","march","april","may",

    "june","july","august","september","october","november","december"};

    printf("please input your wanner month\n");

    int month = 0;

    scanf("%d\n",&month);

    printf("%s\n",a[month - 1]);

}

其中\*a[]代表有个指针数组，其中每个指针都指向一个不定大小的字符串。输出时采用。其中printf函数中printf("%s\n",a[month - 1]);计算机会逐个地址开始解引用来读取这个指针数组中指针a[month-1]指向的字符串。

程序参数：

int main(int argc,char const \*argv[])对于main函数是由参数传递的，一个是整数一个是字符串数组，其中整数用来说明后面的字符串数组有多大。

#include <stdio.h>

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

{

    for(int i = 0;i<arge;i++)

    {

        printf("%d %s\n",i,argv[i]);

    }

    return 0;

}

对于这个程序运行时，可以在他的名字后面跟上一些其他任意的数据。哪些数据都会被main函数的argc和argv所读到。但不论怎样他的第一个参数，永远是执行命令的名字（或者可执行的程序的名字）。

### 1.5、单字符输入输出

getchar和putchar

### 1.6、字符串函数

1、strlen函数：size\_t strlen(const char\*s)，返回s字符串的长度。

#include <stdio.h>

#include <string.h>

int main(int argc,const char \*argv[])//char const\*一样

{

   char a[] = "hello";

   printf("length = %d\n",mylen(a));

   printf("size = %d\n",sizeof(a));

   return 0;

}

其中的mylen实现了strlen函数的作用。

2、strcmp函数：int strcmp(const char\* s1,const char\* s2)，比较两个字符串，返回数值：

s1=s2:0

s1>s2:1

s1<s2:-1

其中对于，字符串的比较，主要为不也一样的哪一位的ascll码比较。小写大于大写，大即返回1，小返回-1。

#include <stdio.h>

#include <string.h>

int mycmp(const char\*s,const char\* p)

{

    int i = 0;

    while(s[i]==p[i]&&s[i]!='\0')

    {

        i++;

    }

    return s[i]-p[i];

}

int main(int argc,const char \*argv[])//char const\*一样

{

    char a[] = "abkkjjkj";

    char b[] = "abkkjjkj";

    printf("%d\n",mycmp(a,b));

    printf("%d\n",strcmp(a,b));

    return 0;

}

其中mycmp函数即实现了strcmp函数的作用。通过return字符的差值直接进行判断。但他返回的是两个不一样字符的ascll码的差值。

改进对于mycmp函数，其中原本使用数组进行比较，现在可以改为用指针进行操作。

int mycmp(const char\*s,const char\* p)

{

    while(\*s==\*p && \*s!='\0')

    {

        s++;

        p++;

    }

    return \*s - \*p;

}

对于字符串进行比较时，常用的方法，用数组进行或者用指针进行。

3、strcpy函数：char\* strcpy(char\* restrict dst,const char\* restrict src)，把src的字符串拷贝到dst，其中restrict表明src和dst不重叠（C99），返回dst指向的字符串。不重叠意味着src的字符串所占的字节所处的地址，和即将拷贝到的地址在分配字节时两段内存不能有重叠的。因为对于计算机来说，多核操作系统在进行copy时，会每个核各做一部分，防止重叠就是为了更好的进行后续的工作。

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

char\* mycopyarray(char\* dst,const char\* src)//数组版本

{

    int i = 0;

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

//while(src[i])//和上面意义一样

    {

        dst[i] = src[i];

        i++;

    }

    dst[i] = '\0';//要为最后面加上\0，保证是一个字符串

    return dst;

}

char\* mycopypointer(char\* dst,const char\* src)//指针版本

{

char\* rest =dst;//由于指针被一直移动那么就需要刚开始也定义一块指向同一地址的指针。

//while((\*dst++ = \*src++)!='\0')//while循环整体可以写成这样

while(\*src!='\0')

//while(\*src)//意义一样

    {

        \*dst = \*src;

        dst++;

        src++;

//上面三句可以和为一句\*dst++ = \*src++;

    }

    \*dst = '\0';// //要为最后面加上\0，保证是一个字符串

    return rest;//此时dst指向字符串的最后一位

}

int main(int argc,const char \*argv[])//char const\*一样

{

    char s1[] = "hello world";

    char\* dst =(char\*)malloc(strlen(s1)+1);

    char \*p = mycopyarray(dst,s1);

    char \*q = mycopypointer(dst,s1);

    printf("array = %s\n",p);

    printf("pointer = %s\n",q);

    return 0;

}

while((\*dst++ = \*src++)!='\0')可以写成这样//while(\*dst++ = \*src++)，因为while函数具有判断0和1的功能，其执行流程为：引用和解引用优先级最高，然后是++，再进行判断。也就是，先赋值，再移位，最后进行判断是否等于\0。其实分成多句的while循环代码被编译器编译时也会被优化为一句的while循环代码。

4、字符串搜索函数：

char\* strchr(const char\* s,int c)表示字符c第一次在字符串中出现的位置（是一段字符串），从左往右数。

char\* strrchr(const char\* s,int c)表示字符c第一次出现的位置（是一段字符串），从右往左数。以上两个函数都返回指针，找不到就返回去NULL。

Exemple1找到了第一个再找第二个

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

int main(int argc,const char \*argv[])//char const\*一样

{

    char s[] = "hello";

    char \*p = strchr(s,'l');

    p = strchr(p+1,'l');

    printf("%s",p);

    return 0;

}

Exemple2把获得的字符串进行了复制

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

int main(int argc,const char \*argv[])//char const\*一样

{

    char s[] = "hello";

    char \*p = strchr(s,'l');

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

    strcpy(t,p);

    printf("%s\n",t);

    printf("%s\n",p);

    return 0;

    free(t);

}

Exemple3复制除了某个字符第一出现之前的

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

int main(int argc,const char \*argv[])//char const\*一样

{

    char s[] = "hello";

    char \*p = strchr(s,'l');

    char c = \*p;//先将\*p所代表的字符串

    \*p = '\0';

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

    strcpy(t,s);

    \*p = c;

    printf("total = %s\n",s);

    printf("copy = %s\n",t);

    printf("p = %s\n",p);

    return 0;

    free(t);

}

把p指针所指的那一块变为\0，其他的不是\0仅仅p指针指的哪一个字母是\0，因为对于strcpy来说是在找\0，来作为截至符，检测到\0以后就结束了拷贝，再拷贝s字符串。然后将p所指的字符改回来。

5、字符串中找字符串：

char\* strstr(const char\* s1,const char\* s2)，会注意到大小写

char\* strcasestr(const char\* s1,const char\* s2)，会忽略大小写

### 1.7、while循环

while（！k）  
 如果k是bool类型，！k等价于k==false（当k等于false时，！k为true，否则！k为false），while(!k)表示直到k等于false时停止循环。  
 如果k是整型，！k等价于k==0（当k等于0时，！k为true，否则！k为false），while(!k)表示直到k等于0时停止循环。  
 如果k是字符型，！k等价于k=='\0'（当k等于'\0'时，！k为true，否则！k为false），while(!k)表示直到k等于'\0'时停止循环。

理解，如果有while(k!=0)，那么就相当于说k=0时结束，k=0，为结束符，那么就可以写为while(k)，也就是说不是0，就为true。

## 19、枚举

（主要被用于当做符号量，很少用作类型）

在程序中用符号来代表数字而不是具体的数字在程序中运行。

枚举定义：

enmu 枚举类型名字{名字0，名字1，，，}；

对于枚举类型来说枚举类型的名字不常用，用的是大括号内部的名字。而且可以当做常量使用，用于整数的输入输出计算，他们的类型就是int类型，数值依次为0到n，如：

enmu colors{yellow,red,green};

对于枚举类型来说，枚举量可以用做值，枚举类型可以跟上enmu作为类型使用。

#include <stdio.h>

enum color{red,green,yellow};

int callculate(enum color f)

{

    printf("%d",f);//输出

}

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

{

    enum color t = red;

    scanf("%d",&t);//输入

    callculate(t);

    return 0;

}

C语言中枚举可以用做整数的输入输出，当做int来使用。但要注意前面加上枚举符enmu。（结构体在c语言也需要加上struct，c++可以不加）

自动计时的枚举：

#include <stdio.h>

enum color{red,green,yellow,black,numcolors};

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

{

    int color = 0;

    char \*colornames[numcolors] = {"red","green","yellow","black"};

    char\* colorname = NULL;

    printf("please input your favorite color number\n");

    scanf("%d",&color);

    if(color>=0 && color<numcolors)

    {

        colorname = colornames[color - 1];

    }

    else

    {

        colorname = "unknown";

    }

    printf("your favorite color is %s\n",colorname);

    return 0;

}

枚举声明也可以指定值，不指定就是从0开始。不一定连续也可以离散。

枚举应用场景：

1、如果有很有意义的排比的名字，用枚举比用const int方便

2、枚举比用宏好，因为枚举有int类型

## 20、结构体

### 1.1、结构类型

struct 类型名字{内部元素}结构体变量；结构体变量结构体名字和内部成员列表至少需要有两个。

#include <stdio.h>

struct date

{

    int month;

    int day;

    int year;

};//声明一种结构体类型date

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

{

    struct date today;//用新声明的类型定义变量，这个变量本身没有意义，有意义的是能够访问到的结构体的内容

    today.day = 8;

    today.month = 11;

    today.year = 2023;

    printf("today date is %d-%d-%d",today.year,today.month,today.day);

    return 0;

}

对于结构体来说，一般处于和函数相同的地位，那么就应该放在函数外面。在后续声明struct类型变量时需要带上struct，c++可以不带，c语言需要带上。

### 1.2、结构体初始化

#include <stdio.h>

struct date

{

    int day;

    int month;

    int year;

};

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

{

    struct date today;

    today.day = 8;

    today.month = 11;

    today.year = 2023;

    struct date thisday = {8,11,2023};

    struct date thismonth = {.month=7,.year=2023};

    printf("today date is %d-%d-%d\n",today.year,today.month,today.day);

    printf("today date is %d-%d-%d\n",thisday.year,thisday.month,thisday.day);

    printf("today date is %d-%d-%d\n",thismonth.year,thismonth.month,thismonth.day);

    return 0;

}

对于结构体的初始化赋值，以上三种方法其本质都是一样的。注意对于输入8和08有区别，对于计算机来说08指的是八进制。赋值和数组很想给的数值就有数值，没有给值就默认填0。

### 1.3、结构成员

结构和数组很像：

其中数组通过[]运算符和下标访问其成员：

a[] = 10;

而对于结构体通过.运算符和名字访问其内部成员:

toady.day，student.firstname

### 1.4、结构体运算

1、对于结构体赋值可以使用定义的结构体类型变量来访问结构体的内容来赋值。

2、对于struct point{int x,int y};这个结构体来说可以直接写p1=(struct point){5,10}，相当于p1.x=5;p1.y=10。

3、p1=p2，相当于p1.x=p2.x; p2.y=p2.y;

对于结构体变量来说结构变量并不是结构体的地址，必须用&取地址。

#include <stdio.h>

struct date

{

    int day;

    int month;

    int year;

};

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

{

    struct date day;

    struct date h;

    day = (struct date){7,11,2023};

    //h = day;

    struct date\* today = &day;

    printf("today date is %d-%d-%d\n",day.year,day.month,day.day);

    printf("today date is %d-%d-%d\n",h.year,h.month,h.day);

    printf("%p\n",today);

    printf("%p\n",&h);

    return 0;

}

对于结构体类型变量day和h本身都是一个有相同结构的局部变量，都有各自的空间，占据不同的内存空间。

### 1.5、结构与参数

结构体用作参数时可以将整个结构体传入函数，传入的结构体变量实际是在函数内部新建一个结构体变量，然后将传入的结构体变量的值进行拷贝，在调用完以后会返回一个结构体。这个与数组完全不同。

#include <stdio.h>

#include <stdbool.h>

// #include "main.h"

struct date

{

    int day;

    int month;

    int year;

};

// extern struct date day,today;

int lastday(struct date today);

bool ifleap(struct date today);

// struct date tommorrow,today;

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

{

    struct date today,tommorrow;

    printf("please input the date of today, \

    following year,month and day\n");

    scanf("%d %d %d",&today.year,&today.month,&today.day);

    if(today.day != lastday(today))

    {

        tommorrow.day = today.day + 1;

        tommorrow.month = today.month;

        tommorrow.year = today.year;

    }

    else if(today.month == 12)

    {

        tommorrow.day = 1;

        tommorrow.month = 1;

        tommorrow.year = today.year+1;

    }

    else

    {

        tommorrow.day = 1;

        tommorrow.month = today.month+1;

        tommorrow.year = today.year;

    }

    printf("today date is %d-%d-%d\n",today.year,today.month,today.day);

    printf("tomorrow date is %d-%d-%d\n",tommorrow.year,tommorrow.month,tommorrow.day);

    return 0;

}

int lastday(struct date today)

{

    int days=0;

    const int daypermonth[] = {31,28,31,30,31,30,31,31,30,31,30,31};

    if(today.month==2 && ifleap(today))

    {

        days = 29;

    }

    else

    {

        days = daypermonth[today.month - 1];

    }

    return days;

}

bool ifleap(struct date today)

{

    bool leap = false;

    if((today.year%4==0 && today.year%100!=0) || today.year%400==0)

    {

        leap = true;

    }

    return leap;

}

从上面这段代码可以看出，取成员优先级高于取地址。并且bool值也可以有返回值。对于一个scanf函数或者printf函数来说，没有直接的方式可以一次输出一个结构的内容或者对一个结构内容进行赋值。输入输出不可以一次性进行，但是赋值可以一次性赋值。

模拟一个一次性对于一个结构体输出和输入：

#include <stdio.h>

#include "main.h"

typedef struct

{

    int x;

    int y;

}input;

input getstruct(void);

void output(input p);

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

{

    input p = {0,0};

    p = getstruct();

    output(p);

    return 0;

}

input getstruct(void)

{

    input p;

    printf("you want to value\n");

    scanf("%d",&p.x);

    scanf("%d",&p.y);

    return p;

}

void output(input p)

{

    printf("%d,%d",p.x,p.y);

}

对于其中的结构体部分可以用

typedef struct

{

    int x;

    int y;

}input;

即把一个结构体重定义为input。变量定义或者函数声明时用input就可以了

或者：

struct input

{

    int x;

    int y;

};

这样在定义变量或者函数申明时就要带上struct。

上面这种传值的方式，既费空间又费时间。

### 1.6、结构指针作为参数

对于结构体用指针访问结构体成员时，用->来进行访问，表示指针所指的结构体变量中的成员。也可以用（\*p）.来访问。

#include <stdio.h>

#include "main.h"

typedef struct

{

    int x;

    int y;

}input;

input\* getstruct(input\* p);

void output(input p);

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

{

    input p = {0,0};

    // getstruct(&p);

    output(\*getstruct(&p));

    return 0;

}

input\* getstruct(input\* p)

{

    printf("you want to value\n");

    scanf("%d",&p->x);

    scanf("%d",&p->y);

    return p;

}

void output(input p)

{

    printf("%d,%d",p.x,p.y);

}

可以将上一段代码改为对于指针的引用，注意其中对于指针函数的应用，input\* getstruct(input\* p)的返回值为一个指针，所以要加上\*，也可以直接对\*getstruct(&p)进行赋值语句为：\*getstruct(&p)=(input){4,5}其中input为定义的结构体类型对后面的数进行类型强制转换。(output(\*getstruct(&p)=(input){4,5};不管输入是几输出都是4和5)

### 1.7、结构体中的结构

a++和++a的区别是：a++ 是后递增操作符。它会先返回变量a的当前值，然后再将a的值加1，会先使用a的当前值，然后再递增a的值；++a 是前递增操作符。它会先将变量a的值加1，然后再返回a的新值，会先递增a的值，然后再使用新的a值。

#include <stdio.h>

#include "main.h"

typedef struct

{

    int hour;

    int minute;

    int second;

}time;

time update(time now);

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

{

    time next[5]={0};

    time current[5] = {{11,59,59},{12,0,0},{1,29,59},{23,59,59},\

    {19,12,27}};

    for(int i=0;i<5;i++)

    {

        printf("time is %d:%d:%d\n",current[i].hour,current[i].minute,\

        current[i].second);

        printf("one second later....\n");

        next[i] = update(current[i]);

        printf("nexttime is %d:%d:%d\n",next[i].hour,next[i].minute,next[i].second);

    }

    return 0;

}

time update(time now)

{

    ++now.second;

    if (now.second==60)

    {

        now.second = 0;

        now.minute++;

        if(now.minute==60)

        {

            now.minute=0;

            now.hour++;

            if(now.hour==24)

            {

                now.hour = 0;

            }

        }

    }

    return now;

}

结构体中的类型，可以是int，float，double也可以是结构体变量。

例如：

struct dateandtime

{

    struct date sdate;

    struct time stime;

};

嵌套的结构，从一个结构索引另外一个结构，逐级索引。

#include <stdio.h>

#include "main.h"

struct point

{

    int y;

    int x;

};

struct rectangle

{

    struct point p1;

    struct point p2;

};

void print(struct rectangle points);

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

{

    struct rectangle rects[] =

    {

        {{0,0},{1,1}},

        {{3,4},{5,6}}

    };//逐级嵌套

    for(int i=0;i<2;i++)

    {

        print(rects[i]);

    }

    return 0;

}

void print(struct rectangle points)

{

    printf("<%d,%d>,<%d,%d>\n",points.p1.x,points.p1.y,\

    points.p2.x,points.p2.y);

}

### 1.8、类型定义

C语言提供了一个typedef的功能来声明一个已有的数据类型：

Typedef int length；

使得length成为int类型的别名。

typedef struct rectangle

{

    struct point p1;

    struct point p2;

}date;

对于这个类型定义，typedef之后到date之前全部都是原有的类型，date是新的类型。

typedef char\* string[10];

对于这个类型定义，定义string为一个指向字符串的指针数组，并且用它定义的变量最多只能有10个元素。即，用string类型声明变量时，这个变量实际上一个包含10个字符串指针的数组。

#include <stdio.h>

#include "main.h"

typedef char\* string[2];

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

{

    string charpoint;

    charpoint[0] = "hello";

    charpoint[1] = "world";

//    charpoint[2] = "tomorrow";//string定义的变量只能有两个变量，这一个就属于越界了，能够输出是因为这个指针所指的那一块地址没有被其他函数运用，所以可以输出

    for(int i =0;i<3;i++)

    {

        printf("%s\n",charpoint[i]);

    }

    return 0;

}

### 1.9、结构体中的字节对齐

对于计算机来说，字节分配需要满足计算机的规则：

1）结构体变量的起始地址能够被其最宽的成员大小整除

2）结构体每个成员相对于起始地址的偏移能够被其自身大小整除，如果不能则在前一个成员后面补充字节

3）结构体总体大小能够被最宽的成员的大小整除，如不能则在后面补充字节。

#include <stdio.h>

#include "main.h"

typedef struct node

{

char character;

double length;

    int size;

}  \_\_attribute\_\_((packed))Size;

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

{

    int x = sizeof(Size);

    int y = sizeof(double);

    int z = sizeof(int);

    int a = sizeof(char);

    printf("%d\n",y);

    printf("%d\n",z);

    printf("%d\n",a);

    printf("%d\n",x);

}

上面这段代码就会导致最后分配字节达到24个字节。

#include <stdio.h>

#include "main.h"

typedef struct node

{

    char character;

    int size;

    double length;

}  \_\_attribute\_\_((packed))Size;

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

{

    int x = sizeof(Size);

    int y = sizeof(double);

    int z = sizeof(int);

    int a = sizeof(char);

    printf("%d\n",y);

    printf("%d\n",z);

    printf("%d\n",a);

    printf("%d\n",x);

}

改写为这样就只会到16个字节。

typedef struct node

{

    char character;

    int size;

    double length;

}  \_\_attribute\_\_((packed))Size;

添加\_\_attribute\_\_((packed))语句则会使得其自动补齐字节被关闭。

## 21、联合

通常情况下，sizeof(int) 的返回值是4字节（32位系统）或8字节（64位系统）。在绝大多数现代计算机体系结构中，int 类型的大小是由编译器和操作系统的规范决定的。因此，为确保代码的可移植性，最好不要依赖于特定系统或编译器的 int 大小，而是使用 sizeof 运算符来获取。

1、所有的成员共享一个空间。

2、同一时间只有一个成员有效。

3、只对第一个成员做初始化。

4、union的大小是其最大的成员。

## 22、S&G变量（尽量避免使用）

### 1.1、全局变量

#include <stdio.h>

#include "main.h"

typedef char\* string[2];

int gall = 12;

int f();

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

{

    printf("in %s gall = %d\n",\_\_func\_\_,gall);

    f();

    printf("in %s gall = %d\n",\_\_func\_\_,gall);

    return 0;

}

int f()

{

    printf("in %s gall = %d\n",\_\_func\_\_,gall);

    gall+=2;

    printf("in %s gall = %d\n",\_\_func\_\_,gall);

}

全局变量作用于整个程序，但进入子函数以后局部变量的优先级是要高于全局变量的。

对于上段代码中的\_\_func\_\_是一个字符串，主要意思是输出，当前所运行的函数。

全局变量的初始化：

1、对于没有进行初始化的全局变量会得到0值，指针会得到NULL。

2、只能用于编译时刻已知的值来初始化全局变量，即全局变量不能在程序编译时用未知数进行赋值。

3、全局变量的初始化在main函数之前进行。

### 1.2、静态本地变量

1、在本地变量定义时加上static修饰符就成为静态本地变量。

2、档函数结束时，静态本地变量会继续存在并保持其数值。

3、静态变量初始化只会在第一次进入这个函数的时候创建，以后进入只会保持上次的数值，不会再执行操作。

#include <stdio.h>

#include "main.h"

int f();

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

{

    f();

    f();

    f();

    return 0;

}

int f()

{

    static int all = 1;

    printf("in %s all = %d\n",\_\_func\_\_,all);

    all+=2;

    printf("in %s all = %d\n",\_\_func\_\_,all);

}

static int all=1只执行一次，以后不会再对all进行赋值。

1、对于静态本地变量实际上是特殊的全局变量，他们位于相同的内存区域。

2、静态本地变量具有全局的生存期，但作用于仅在函数局部域。

3、static意味着局部作用域。即静态全局变量和全局变量位于相同的地方，但有这着和局部变量一致的作用域。

#include <stdio.h>

#include "main.h"

int f();

int gall = 12;

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

{

    f();

    return 0;

}

int f()

{

    int i = 0;

    static int all = 1;

    printf("i %p\n",&i);

    printf("gall %p\n",&gall);

    printf("all %p\n",&all);

    printf("in %s all = %d\n",\_\_func\_\_,all);

    all+=2;

    printf("in %s all = %d\n",\_\_func\_\_,all);

}

### 1.3、S&G返回指针的函数

1、返回本地变量的地址是危险的。

2、返会全局变量或者静态变量的地址是安全的，因为他们与函数无关。

3、返回函数内的malloc也是安全的，但容易有问题。记得用完内存后对他进行释放，释放对应的内存即可。

4、最好是返回传入的指针。

## 23、编译预处理指令

#开头是编译预处理指令，他不是c语言的成分，但c语言程序离不开它。

### 1.1、宏的定义

#include：引用头文件

：#define fromat “%f\n”

#ifndef，#define，#endif

#### 1）程序运行分解

在终端输入：gcc mian.c --save-temp，来保存.c文件在编译到执行之间的临时文件，对于临时文件有.i结尾，.o结尾以及.s结尾，采取这样的排序：.c.h->.i->.s->.o->.exe，其中.i文件即预编译后的文件，将预编译的文件进行编译得到.s文件，.s是个汇编代码文件，对于汇编代码文件进行汇编得到了目标代码文件.o，将目标代码文件进行链接得到了可执行文件。其中预编译即把.c文件中需要预编译的文件斤西进行预编译如下：

这是.c文件

#include <stdio.h>

#define pi 3.14159

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

{

    printf("%f\n",pi);

    return 0;

}

对上面的.c文件夹进行预编译这是.i文件的最后几行，输入（tail.main.i）

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

{

    printf("%f\n",3.14159);

    return 0;

}

#### 2）#define

#define：宏定义，在预编译阶段将宏定义的参数进行原始的文本替换。

#define 名字 数值（操作指令甚至是函数空着也可以，仅仅为了定义宏）

#include <stdio.h>

#define pi 3.14159

#define format "%f\n"

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

{

    printf(format,pi);

    return 0;

}

宏替换：

1. 如果一个宏的值有其它宏的名字，也是会被替换的
2. 如果一个宏的值超过了一行，最后一行的行末要加上“\”
3. 宏的值后面的注释不会被当做宏的一部分。
4. #include <stdio.h>
5. #define pi 3.14159
6. #define format "%f\n"
7. #define pi2 2\*pi//pi\*2
8. #define prt printf(format,pi);\
9. printf(format,pi2)
10. int main(int argv, const char\* argr[])
11. {
12. prt;
13. return 0;
14. }

多行的宏定义需要再处最后一句以外的都要加上“；”

条件编译宏：

#define \_debug，这类宏没有值，特是用来条件编译的，后面有其他的编译预处理指令来检查这个宏是否已经被定义过了，在库函数的定义中常用。

#### 3）定义好的宏

\_\_LINE\_\_：这个文件原代码当前所在行的行号，只有这个是整数

\_\_FILE\_\_：文件的文件名

\_\_DATE\_\_：编译时的日期

\_\_TIME\_\_：编译时的时间

\_\_STDC\_\_

#include <stdio.h>

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

{

    printf("%s,%d\n",\_\_FILE\_\_,\_\_LINE\_\_);

    printf("%s,%s\n",\_\_DATE\_\_,\_\_TIME\_\_);

    return 0;

}

### 1.2、含参宏

#### 1）宏定义是可以带参数的：

#define cube(x) ((x)\*(x)\*(x))，对于这个宏定义其中x是没有类型的，后面会有参数传入，去替代那个x的。

.c文件

#include <stdio.h>

#define cude(x) ((x)\*(x)\*(x))

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

{

    printf("%d\n",cude(5));

    printf("%s,%d\n",\_\_FILE\_\_,\_\_LINE\_\_);

    printf("%s,%s\n",\_\_DATE\_\_,\_\_TIME\_\_);

    return 0;

}

.i文件

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

{

    printf("%d\n",((5)\*(5)\*(5)));

    printf("%s,%d\n","main.c",6);

    printf("%s,%s\n","Nov 11 2023","21:05:16");

    return 0;

}

可以看到预编译以后的.i文件对于cude(x)进行了替换。

#### 2）带参数的宏的原则：

1、一切都要带括号

2、整个值要带括号

3、参数出现的每个地方都要有括号

举例（错误）：.c文件

#include <stdio.h>

#define ci1(x) (x\*57.29578)

#define ci2(x) (x)\*57.29578

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

{

    printf("%f\n",ci1(5+2));

    printf("%f\n",180/ci2(1));

    return 0;

}

预编译.i文件：

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

{

    printf("%f\n",(5+2\*57.29578));

    printf("%f\n",180/(1)\*57.29578);

    return 0;

}

可见括号不全预编译后有问题。

宏没有类型检查，部分宏可以用incline来替代。

## 24、头文件

对于#inlcude有两种形式来指出要插入的文件：

1、“”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。

2、< >让编译器只在指定的目录寻找。

**误区：**

#include不是引入库函数的，只是把函数原封不动的，放在前面，其原理和函数声明一样。如：stdio.h里面只有printf函数的原型，其具体代码在别的地方。现在的c语言编译器默认会引入所有的标准库。写#include只是为了让编译器知道所写的函数的正确类型。

写入库函数：

#include <stdio.h>

#include "main.h"

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

{

    int a = 5;

    int b = 6;

    printf("%f\n",max(a,b));

    return 0;

}

写入的库函数实际：

# 3 "main.h"

double max(double a,double b);

# 3 "main.c" 2

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

{

    int a = 5;

    int b = 6;

    printf("%f\n",max(a,b));

    return 0;

}

**头文件：**

1、在使用和定义这个函数的地方都应该#include这个头文件。

2、一般的做法就是任何.c文件都有对应的同名的.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：**

1、在函数前面加上static就是的它成为只能在躲在的编译单元中被使用的函数。

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

## 25、声明

1、全局变量声明：

先是在库函数中声明一个外部变量：extern int gall;

main.h

#ifndef MAIN\_H

#define MAIN\_H

extern int gall;

double max(double a,double b);

#endif

然后在一个.c文件中对于这个变量进行定义。

max.c

#include <math.h>

#include "main.h"

int gall = 12;

double max(double a,double b)

{

    return a+b;

}

main.c

#include <stdio.h>

#include "main.h"

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

{

    int a = 5;

    int b = 6;

    printf("%f\n",max(a,gall));

    return 0;

}

声明是不产生代码的，定义才会产生代码（只有函数和全局变量会产生代码）。

2、只有声明可以被放在头文件中：是规则不是法律，如果头文件里放了定义，就会出现多重定义。

标准头文件结构（为了防止多重定义结构）

#ifndef MAIN\_H

#define MAIN\_H

#endif

运用条件编译和宏，保证这个头文件在一个编译单元中只会被#include一次，防止多个头文件中对同一个变量进行声明。

## 26、输入输出

### **1.1、printf函数**

%[flag][width][.prec][hIL]type

|  |  |
| --- | --- |
| Flag | 含义 |
| - | 左对齐 |
| + | 在前面放+或- |
| （space） | 正数留空 |
| 0 | 0填充 |

#include <stdio.h>

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

{

    printf("%9d\n",123);

    printf("%-9d\n",123);

    printf("%+9d\n",123);

    printf("% d\n",123);

    printf("%09d\n",123);

    return 0;

}

其中9d表示占个字符，-9d表示左对齐，+9d表示强制加符号，09d表示除了输出的字符外其他都填充0。

|  |  |
| --- | --- |
| Width或prec | 含义 |
| number | 最小字符数 |
| \* | 下一个参数是所占的字符数 |
| .number | 小数点后的位数 |
| .\* | 下一个参数是小数点后的位数 |

#include <stdio.h>

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

{

    printf("%9.2f\n",123.2);

    printf("%\*d\n",6,123);

    printf("%-+9d\n",123);

    printf("%\*.\*f\n",6,2,123.0);

    printf("%.\*f\n",2,123.0);

    return 0;

}

printf("%\*.\*f\n",6,2,123.0);对于第一个\*来说是表示宽度，第二个表示精度，最后一个为浮点数，注意对于输出123，需要将它写为浮点数123.0，不然%\*.\*f在试图解释后面的数据输出时，会报错解释不了。

|  |  |
| --- | --- |
| 类型修饰 | 含义 |
| hh | 输出占单个字节长度的数 |
| h | short（指两个字节） |
| I | long（4个字节） |
| II | long long（8个字节） |
| L | long double（16个字节） |

对于使用%hhd一般指单个字节部分编译器可能会输出两个字节，但还是要按规则写。

|  |  |  |  |
| --- | --- | --- | --- |
| tyoe | 用于 | type | 用于 |
| i或d | int | g | float |
| u | unsigned int | G | float |
| o | 八进制 | A或a | 十六进制浮点 |
| x | 十六进制 | c | char |
| X | 字母大写的十六进制 | s | 字符串 |
| F或f | float，6 | p | 指针 |
| E或e | 指数 | n | 读入/写出的个数 |

%n 格式说明符，该说明符用于将前面打印的字符数保存到相应的参数中，但一般别用，编译器会有问题。对于其中的i来说，可用于输入时更加灵活，即可输入十进制也可以是十六进制以及八进制都可以。

1.2、scanf函数

%[flag]type

|  |  |  |  |
| --- | --- | --- | --- |
| flag | 含义 | flag | 含义 |
| \* | 跳过 | I | long,double |
| 数字 | 最大字符数 | II | long,long |
| hh | char | L | long double |
| h | short |  |  |

#include <stdio.h>

int main()

{

    int num;

    scanf("%\*d%d",&num);

    printf("%d\n",num);

    return 0;

}

对于其中的%\*d是输入两个时，跳过第一个数输出第二个。其中对于scanf函数生成的可执行函数可以用gcc mian.c -o main，即运行这个.c文件然后运用-o把生成的可执行文件命名为main。

printf函数和scanf函数都有返回值:

scanf函数，返回输入的item

printf函数，返回输出的字符数

#include <stdio.h>

int main()

{

    int num;

    int i1 = scanf("%d",&num);

    int i2 = printf("%d\n",num);

    printf("%d\n",i1);

    printf("%d\n",i2);

    return 0;

}

输入123，i1为1，而i2为4，因为有123和一个回车。

## 27、文件输入输出

### 1.1、powershall命令输入输出

文件重定向：用<指向要输入的文件名，>指向要输出的文件名（对于powershall可能对于部分编译器不适用）

1、more：打开文件

2、.\main.exe > 12.out：创建一个文件12.out将main.exe的输出存进去，会覆盖之前的。

3、cat > 12.in：创建一个文件12.in放入内容，编译器部分可能不适用，需要写：

"123456" | set-content -path "C:\Users\zxc\Desktop\c\_language\structure\12.in"对于最开始为要写入的文本，后面需要加绝对路径。

4、读取12.in中的数据：.\main.exe <12.in > 12.out，同样对于部分编译器不支持，需要写：

get-content 12.in | .\main.exe

5、删除12.in或者删除12.out输入命令 rm 12.out和rm 12.in即可。也可以使用del 12.out或者remove-item 12.out。

### 1.2、FILE文本文件

FILE\* fopen(const char\* restrict path,const char\* restrict mode)

int fclose(FILE\* stream)

fscanf(FILE\*,….)

fprintf(FILE\*,….)

FILE\*是一个结构指针。

#### 1）fopen

|  |  |
| --- | --- |
| r | 打开只读 |
| r+ | 打开读写，从文件头开始 |
| w | 打开只写。如果不存在则新建，如果存在则清空 |
| w+ | 打开读写，如果不存在则新建，如果存在则清空 |
| a | 打开追加。如果不存在则新建，如果存在则从文件尾开始 |
| ..x（主要用于w和a） | 只新建，如果文件已存在则不能打开 |

### 1.3、二进制文件（难）

### 1.4、逻辑运算和按位运算

对于逻辑运算只有0和1，可以认为逻辑运算相当于把所有非零值变成1，然后按位运算：

5&4->4而5&&4->1&1->1

5|4->5而5||4->1|1->1

~4->3而!4->!1->0（~表示按位取反）

|  |  |
| --- | --- |
| & | 按位相与 |
| | | 按位相或 |
| ~ | 取反 |
| ^ | 按位异或 |
| ！ | 非 |

### 1.5、移位运算

|  |  |
| --- | --- |
| << | 左移 |
| >> | 右移 |

i<<j：

i中所有的位向左移动j个位置，而右边填入0

所有小于int的类型，移位以int的方式来做，结果是int

x <<= 1 等价于 x \*= 2

x <<= n 等价于 x \*= 2n

i>>j

i中所有的位右移j位

所有小于int的类型，移位以int的方式来做，结果是int

对于unsigned的类型左边填入0

所有singed的类型，左边填入原来的最高位（保持符号位不变） 即原来是1000，有符号数，右移一位后就变成1100。往左移就不管符号位了。

x >>= 1 等价于 x /= 2

x >>= n 等价于 x /= 2n

#include <stdio.h>

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

{

    int a = 0x80000000;

    unsigned int b = 0x80000000;

    printf("%d\n",a);

    printf("%u\n",b);

    printf("%d\n",a>>1);

    printf("%d\n",b>>1);

    return 0;

}

举例：

#include <stdio.h>

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

{

    puts("please input one number");

    int number;

    scanf("%d",&number);

    // unsigned int mask = 1u<<31;

    unsigned mask = 1u<<31;

    for( ; mask!=0 ; mask>>=1)

    {

        printf("%d",number & mask? 1:0);

    }

    return 0;

}

其中，unsigned int mask可以用unsigned mask来替代，是因为对于编译器来说，他们会来猜这是一个整型的无符号数。对于for循环中，for( ; mask ; mask>>=1)，其中，具体意思为：判断是否mask!=0，则进行后续的运算mask>>=1，意味着mask = mask>>1。下面的输出函数，number&mask即按位与，将按位与数值为真则输出1，否则输出0。（condition ? expression\_if\_true : expression\_if\_false; 在这里，condition 是一个条件，如果为真，表达式的值为 expression\_if\_true，否则为 expression\_if\_false）

1.6、位段

把一个int的若干位组合成一个结构

typedef struct

{

    unsigned int leading : 3;

    unsigned int FLAG1 : 1;

    unsigned int FLAG2 : 1;

int trailing : 27;

}uo;

其中每个结构体的内容后面都是其所占的位数，一共是32位对应4个字节。如果超过32位，那么他就需要用两个int来表示也即是8个字节。

#include <stdio.h>

#include "main.h"

typedef struct

{

    unsigned int leading : 3;

    unsigned int FLAG1 : 1;

    unsigned int FLAG2 : 1;

    int trailing : 27;

}uo;

void print(unsigned int number);

uo uu;

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

{

    uu.leading = 7;

    uu.FLAG1 = 1;

    uu.FLAG2 = 0;

    uu.trailing = 0;

    printf("%d\n",sizeof(uu));

    print(\*(int\*)&uu);

    return 0;

}

void print(unsigned int number)

{

    unsigned int mask = 1u << 31;

    for ( ; mask!=0 ; mask>>=1)

    {

        printf("%d",number & mask? 1:0);

}

}

对于其中main函数中调用的print函数来说，\*(int\*)&uu，进行了一个强制类型转换，&uu本身表示一个结构体类型的指针，运用（int\*）转换为int类型的指针，在运用\*解引用这个指针。

位段：

1、可以直接用位段成员名称来访问，比移位、与、或还方便

2、编译器会安排其中的位的排列，不具有可移植性（有可能不同编译器对应不同的排列次序）

3、当所需的位超过一个int时会采用多个int。

对于结构体声明:

typedef struct

{

    int\* array;

    int size;

}\* Array;

Array lo;

对于上面的结构体类型可以这样理解，把它看作int\* p来理解，Array前面的所有都是一个结构体类型，定义Array是一个指向结构体的指针。按一般那不这样写，程序可读性太低。

28、可变数组

main.c

#include <stdio.h>

#include <stdlib.h>

#include "main.h"

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

{

    array\_int n = 0;

    printf("please input init size of array\n");

    scanf("%d",&n);

    Array a = array\_create(n);

    printf("array size is %d\n",array\_size(&a));

    for (array\_int j=0;j<n;j++)

    {

        printf("array\_pointer(&a,%d) = %d\n",j,\*array\_pointer(&a,j));

    }

    printf("array\_size is %d\n",array\_size(&a));

    array\_int key = 0;

    while(key!=100)

    {

        printf("please input your want to see poistion 1 start or 0 end\n");

        scanf("%d",&key);

        switch (key)

        {

            case 1:

            {

                printf("please input number\n");

                array\_int number = 0;

                scanf("%d",&number);

                printf("array\_pointer(&a,%d) = %d\n",number,\*array\_pointer(&a,number));

            }

            // case 2:

            // {

            //     break;

            // }

        }

    }

    array\_free(&a);

    return 0;

}

//create a array

Array array\_create(array\_int init\_size)

{

    Array b;

    b.size = init\_size;

    b.array = (array\_int\*)calloc(init\_size,sizeof(array\_int));

    array\_initvalue(&b);

    return b;

}

//read size of array

array\_int array\_size(const Array\* a)

{

    return a->size;

}

//read special poistion of value

array\_int array\_get(const Array \*a,array\_int index)

{

    return a->array[index];

}

//modify special poistion of value

void array\_set(Array\* a,array\_int index,array\_int value)

{

    a->array[index] = value;

}

//release memony of array pointer

void array\_free(Array\* p)

{

    free(p->array);

    p->array = NULL;

    p->size = 0;

}

//modify the size of array

void array\_enlarge(Array\* a,const array\_int more\_size)

{

    array\_int i = 0;

    array\_int \*p = (array\_int\*)realloc(a->array,sizeof(array\_int)\*(a->size + more\_size));

    a->array = p;

    a->size += more\_size;

}

//get back the specific value of poistion

int\* array\_pointer(Array\* a,array\_int index)

{

    if(index<1)

    {

        ;

    }

    if(index>a->size)

    {

        array\_enlarge(a,(index/block\_size+1)\*block\_size);

    }

    return &(a->array[index]);

}

//array init value

Array array\_initvalue(Array\* a)

{

    for(array\_int i=0;i<a->size;i++)

    {

        a->array[i] = i;

    }

}

main.h

#ifndef \_\_MAIN\_H\_\_

#define \_\_MAIN\_H\_\_

#define block\_size 20

typedef int array\_int;

typedef struct code

{

    array\_int size;

    array\_int\* array;

}Array;

extern Array a;

Array array\_create(array\_int init\_size);

array\_int array\_size(const Array\* a);

array\_int array\_get(const Array \*a,array\_int index);

void array\_set(Array\* a,array\_int index,array\_int value);

void array\_free(Array\* p);

void array\_enlarge(Array\* a,const array\_int more\_size);

int\* array\_pointer(Array\* a,array\_int index);

Array array\_initvalue(Array\* a);

#endif