

# 期末复习-PTA

## Week 5

### HW5



#### Multiple Choice 2-3

若函数调用时的实参为变量,下列关于函数形参和实参的叙述中正确的是()。

- A.函数的实参和其对应的形参共占同一存储单元
- B.形参只是形式上的存在,不占用具体存储单元
- C.同名的实参和形参占同一存储单元
- D.函数的形参和实参分别占用不同的存储单元

**解析:**

D。当函数被调用时, 实参的值会被复制到形参中, 这个过程称为**值传递**。形参和实参在内存中是独立的, 它们分别占用不同的存储单元。



#### 形参与实参

**实参:** 是在函数调用时传递给函数的值或变量。实参可以是常量、变量或表达式。

**形参:** 是在函数定义时声明的参数, 用于接收函数调用时传递的实参值。



#### Multiple Choice 2-6

以下说法正确的是:

- A.一个C语言源文件 (.c 文件) 必须包含 main 函数
- B.一个C语言源文件 (.c 文件) 可以包含两个以上 main 函数
- C.C语言头文件 (.h 文件) 和源文件(.c 文件) 都可以进行编译
- D.在一个可以正确执行的C语言程序中, 一个C语言函数的声明 (原型) 可以出现任意多次

**解析：**

答案选D。

C.C语言源文件（.c文件）可以直接编译成目标文件（.o文件）。而头文件（.h文件）通常包含函数声明、宏定义等，它们不包含实际的函数实现，因此不能直接编译。头文件通常被包含在源文件中，然后源文件被编译。所以C选项错误。

D.在C语言中，函数声明（原型）可以在程序的任何地方出现多次，只要它们的声明是一致的。函数声明的作用是告诉编译器函数的名称、返回类型和参数类型，以便编译器在调用函数时进行类型检查。因此，一个函数的声明可以出现在多个.c文件中，也可以在同一个.c文件中多次声明。所以D选项正确。

## 声明与定义

举个例子：`int x;` 是**声明** `x` 为一个 `int` 类型的变量，  
而 `int x=1;` 是**定义** `x` 为一个 `int` 类型的变量，并将其初始化为 1。

**声明**和**定义**是不同的概念。定义不仅声明了标识符的存在，还为其分配了**存储空间**，并可能包含初始化代码。例如，变量的定义会为变量分配内存空间，并可能为其赋初值；函数的定义包含了函数的具体实现代码。

在C语言中，变量和函数的声明可以重复，但定义不能重复。

**变量声明**：可以在不同的作用域中重复，但每次声明的类型和名称必须一致。

**函数声明**：可以重复，只要返回类型、函数名和参数列表一致。

**定义**：不能重复，变量或函数在整个程序中只能有一个定义。

# Week 6

## HW6



### Programming 7-7 出栈序列的合法性

给定一个最大容量为  $m$  的堆栈，将  $n$  个数字按  $1, 2, 3, \dots, n$  的顺序入栈，允许按任何顺序出栈，则哪些数字序列是不可能得到的？

例如给定  $m=5, n=7$ ，则我们有可能得到  $\{1, 2, 3, 4, 5, 6, 7\}$ ，但不可能得到  $\{3, 2, 1, 7, 5, 6, 4\}$ 。

### 输入格式：

输入第一行给出 3 个不超过 1000 的正整数： $m$ （堆栈最大容量）、 $n$ （入栈元素个数）、 $k$ （待检查的出栈序列个数）。最后  $k$  行，每行给出  $n$  个数字的出栈序列。所有同行数字以空格间隔。

### 输出格式：

对每一行出栈序列，如果其的确是有可能得到的合法序列，就在一行中输出 YES，否则输出 NO。

### 输入样例：

```
5 7 5
1 2 3 4 5 6 7
3 2 1 7 5 6 4
7 6 5 4 3 2 1
5 6 4 3 7 2 1
1 7 6 5 4 3 2
```

### 输出样例：

```
YES
NO
NO
YES
NO
```

### 代码实现

```
#include<stdio.h>
#define MAXN 10010
int a[MAXN],zhan[MAXN],top=0,f,m,n,k,f1=0,f12=0;
void ruzhan(int t){
    for(int i=f;i<=t;i++){
        zhan[++top]=i;
    }
    if(top>=m) f1=1;
}
void chuzhan(int t){
    if(t!=zhan[top]){
        f1=1;
        return;
    }
    zhan[top]=0;
    top--;
    if(top>=m) f1=1;
}
int main(){
    scanf("%d %d %d",&m,&n,&k);
    for(int i=0;i<k;i++){
        f1=0;
        f12=0;
        for(int j=0;j<n;j++){
            scanf("%d ",&a[j]);
        }
        if(a[0]>m){ //如果第一个数大于m, 那么一定不可能得到这个序列
            f1=1;
            f12=1;
            printf("NO\n");
        }
        for(int j=0;j<a[0];j++){
            zhan[j]=j+1;//把1到a[0]入栈
        }
        top=a[0]-1;//记录栈顶
        f=a[0]+1;//记录下一个要入栈的数字

        if(f1==0){
            for(int j=0;j<n;j++){
                if(f1==1){

```

```
    printf("NO\n");
    f12=1;
    break;
}
if(f>a[j]) chuzhan(a[j]);
else{
    ruzhan(a[j]);
    f=a[j]+1;
    chuzhan(a[j]);
}
}
}
}
if(f1==0) printf("YES\n");
if(f1==1&&f12==0) printf("NO\n");
}
}
```

## Week 7

### Programming 7-3 hanoi 汉诺塔

汉诺塔是一个源于印度古老传说的益智玩具。据说大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着64片黄金圆盘，大梵天命令僧侣把圆盘移到另一根柱子上，并且规定：**在小圆盘上不能放大圆盘，每次只能移动一个圆盘。**当所有圆盘都移到另一根柱子上时，世界就会毁灭。

#### 输入格式

圆盘数  
起始柱 目的柱 过度柱

#### 输出格式

移动汉诺塔的步骤  
每行显示一步操作，具体格式为：  
盘片号：起始柱 -> 目的柱

其中：盘片号从 1 开始由小到大顺序编号。

## 输入样例

```
3
a c b
```

## 输出样例

```
1: a -> c
2: a -> b
1: c -> b
3: a -> c
1: b -> a
2: b -> c
1: a -> c
```

## 代码实现

```
#include<stdio.h>
void f(int n,char a,char b,char c){
    if(n==1){//递归结束条件
        printf("%d: ",n);
        printf("%c -> %c",a,c);
        printf("\n");
    }
    else{
        f(n-1,a,c,b);//将第n盘上方的n-1个盘子从a到b
        printf("%d: ",n);
        printf("%c -> %c",a,c);//将第n个盘子从a到c
        printf("\n");
        f(n-1,b,a,c);//将n-1个盘子从b到c
    }
}
int main(){
    int n;
    char a,b,c;
    scanf("%d ",&n);
    scanf("%c %c %c",&a,&c,&b);
    f(n,a,b,c);//从a到c开始递归,b是过度柱
}
```

# Week 9



Merge Sort 归并排序

```
#include<stdio.h>
#define MAX 10010
void merge(int a[],int left,int leftEnd,int right,int tmp[]){
    //合并两个有序数组
    int i,j,k;
    i=left;
    k=left;
    j=leftEnd+1;
    while(i<=leftEnd&&j<=right){
        if(a[i]<a[j]) tmp[k++]=a[i++];
        else tmp[k++]=a[j++];
    }
    while(i<=leftEnd) tmp[k++]=a[i++];
    while(j<=right) tmp[k++]=a[j++];
}
void mergeSort(int a[],int left,int right,int tmp[]){
    if(left>=right) return;//递归结束条件（别忘了）
    int mid=(left+right)/2;
    mergeSort(a,left,mid,tmp);
    mergeSort(a,mid+1,right,tmp);
    //分别对左右两个子数组进行排序
    merge(a,left,mid,right,tmp);//合并
    for(int i=left;i<=right;i++){
        a[i]=tmp[i];
    }
}
int main(){
    int n;
    int a[MAX];
    scanf("%d",&n);
    for(int i=0;i<n;i++){
        scanf("%d",&a[i]);
    }
    int tmp[MAX];
    mergeSort(a,0,n-1,tmp);
    for(int i=0;i<n;i++){
        printf("%d ",a[i]);
    }
}
```

## Quick Sort 快速排序

```
#include<stdio.h>
#define MAX 10010
void swap(int a[],int x,int y){
    int t=a[x];
    a[x]=a[y];
    a[y]=t;
}
void quickSort(int a[],int left,int right){
    if(left>=right) return;
    int mid=(left+right)/2;
    int pilot=a[mid];//选择一个基准值
    swap(a,mid,left);//将基准值放到最左边
    int last=left;//记录最后一个小于基准值的数的位置
    for(int i=left+1;i<=right;i++)//注意i从left+1开始
    {
        if(a[i]<pilot) swap(a,++last,i);//将小于基准值的数放到左边，体会一下为什么是++last
    }
    swap(a,last,left);//将基准值放到最后一个小于基准值的数的位置，这样就将数组分为了两个部分,
    quickSort(a,left,last-1);
    quickSort(a,last,right);
}
int main(){
    int n;
    int a[MAX];
    scanf("%d",&n);
    for(int i=0;i<n;i++){
        scanf("%d",&a[i]);
    }
    quickSort(a,0,n-1);
    for(int i=0;i<n;i++){
        printf("%d ",a[i]);
    }
}
```

这里推荐一个讲快速排序的blog：[快速排序](#)

# Mid-term Exam

## True or False 1-8-1

已知： int x=6,y=2,z; 则执行表达式 z=x=x>y 后，变量 z 的值为 6。

- T
- F

解析：

F。

$x=x>y$  等价于  $x=(x>y)$ ， $x>y$  是一个表达式，其值为 1，因此  $z=x=x>y$  等价于  $z=x=1$ ，因此  $z=1$ 。

注意：

> 运算符优先级高于 = 运算符。

## Multiple Choice 2-3-1

以下程序的输出结果是

```
int main(){
    int x=1,a=0,b=0;
    switch(x){
        case 0:b++;
        case 1:a++;
        case 2:b++;a++;
    }
    printf("a=%d,b=%d\n",a,b);
    return 0;
}
```

- A. a=1,b=1
- B. a=2,b=1
- C. a=2,b=2
- D. a=1,b=0

解析：

C。

`x=1`，所以从 case 1 进入  
switch 语句中没有 break 语句，所以会一直执行下去。

### Multiple Choice 2-3-1

以下程序的输出结果是：

```
int a=2,b=9;  
do{  
    b-=a;  
    a++;  
}while(b--<0);  
printf("a=%d,b=%d\n",a,b);
```

- A.a=3, b=6
- B.a=2, b=8
- C.a=1, b=-1
- D.a=3, b=7

解析：

A。

首先执行 `b-=a`，`b=9-2=7`，`a++`，`a=3`

然后执行 `b--<0`，`b=7-1=6`，`6<0`，不满足条件，退出循环

因此 `a=3, b=6`

### fill in blank for programming 5-1 gcd 轮转相除法

```
#include <stdio.h>
int main()
{
    int a, b;
    scanf("%d %d", &a, &b);
    while (b>0){
        int r = a%b;
        a=b;
        b=r;
    }
    printf("%d\n", a);
}
```

## Programming 7-2-1 输出全排列

输入整数  $n$  ( $3 \leq n \leq 7$ ) ,编写程序输出  $1, 2, \dots, n$  整数的全排列，按字典序输出。

**输入格式:**

一行输入正整数  $n$ 。

**输出格式:**

按字典序输出1到  $n$  的全排列。每种排列占一行，数字间无空格。

**输入样例:**

在这里给出一组输入。例如：

3

**输出样例:**

在这里给出相应的输出。例如：

123  
132  
213  
231  
312  
321

**代码实现：(深度优先搜索 dfs )**

```
#include<stdio.h>
int n,pd[100],used[100];//pd是判断是否用过这个数
void print()//输出函数
{
    int i;
    for(i=1;i<=n;i++)
        printf("%d ",used[i]);
    printf("\n");
}
void dfs(int k)//深搜函数，当前是第k格
{
    int i;
    if(k==n) //填满了的时候
    {
        print();//输出当前解
        return;
    }
    for(i=1;i<=n;i++)//1-n循环填数
    {
        if(!pd[i])//如果当前数没有用过
        {
            pd[i]=1;//标记一下
            used[k+1]=i;//把这个数填入数组
            dfs(k+1);//填下一个
            pd[i]=0;//回溯
        }
    }
}
int main()
{
    scanf("%d",&n);
    dfs(0);//注意，这里是第0格开始的！
    return 0;
}
```

# Week 10

## HW10

### True or False 1-2

不同类型的指针变量是可以直接相互赋值的。

- T
- F

解析：

F。

会报错，一般需要强制类型转换。

但是存在例外，例如 `void*` 是可以直接赋值给其他指针的。

### Code Completion 6-4 Esc Chars

Esc characters are represented as `\x` in C string, such as `\n` and `\t`.

Function `prt_esc_chars()` gets a string which may contains esc characters, and prints the string into the standard output with all esc characters been replaced by a `\x` format combination.

For example, using `printf("%s"...)`, a string with a `\n` between the words will be printed as:

```
Hello  
World
```

But the same string will be printed by `prt_esc_chars()` as:

```
Hello\nWorld
```

Your function should be able to recognize esc characters below:

```
\n\n\r\n\t\n\b
```

And all other characters below `0x20` should be printed as:

```
\hh
```

where `hh` is the hexadecimal of the value, all letters in capital. For a value below `0x10`, a leading `0` is needed to keep two positions.

And as a C string, `0x00` will not be part of the string but the terminator.

Be aware, `printf()` is forbidden in the function.

## 函数接口定义：

```
int prt_esc_char(const char *s);
```

`s` is the string to be printed. The function returns the number of characters printed. An esc character is counted as two or three according to the characters it uses.

## 裁判测试程序样例：

```
#include <stdio.h>
int prt_esc_char(const char *s);

int main()
{
    char *line = NULL;
    size_t linecap = 0;
    getline(&line, &linecap, stdin);
    int len = prt_esc_char(line);
    printf("%d\n", len);
}

/* 请在这里填写答案 */
```

## 输入样例：

```
hello    world
```

There is a tab between hello and world.

输出样例：

```
hello\tworld\n14
```

代码实现：

```
int prt_esc_char(const char *a){
    int count=0,flag,d,g;
    for(int i=0;a[i]!='\0';i++){
        flag=0;
        switch(a[i]){
            case '\n': putchar('\\');putchar('n');flag=1;count+=2;break;
            case '\r': putchar('\\');putchar('r');flag=1;count+=2;break;
            case '\t': putchar('\\');putchar('t');flag=1;count+=2;break;
            case '\b': putchar('\\');putchar('b');flag=1;count+=2;break;
        }
        if(a[i]<0x20&&flag==0){
            putchar('\\');
            d=a[i]/16+'0';
            g=a[i]%16+'0';
            putchar(d);
            putchar(g);
            count+=3;
        }
        else{
            if(flag==0){
                putchar(a[i]);
                count++;
            }
        }
    }
    return count;
}
```

# Week 12

## Lab12

### Code Completion 6-3/6-4 D字符串的创建函数与连接函数

D字符串是动态分配内存的字符串，它也采用 char 数组来保存字符串中的字符，但是这个数组是在堆中动态分配得到的。

**函数接口定义如下：**

```
char *dstr_create(const char *s);
char *dstr_add(char *s, char c);
char *dstr_concat(char *this, const char *that);
```

dstr\_create 用输入的字符串 s 的内容创建一个新的字符串。

dstr\_add 在 s 的后面加上一个字符 c，返回新的字符串。

dstr\_concat 在 this 后面加上字符串 that，返回新的字符串。

这两个函数的第一个参数都必须是D字符串，不能是静态数组。

**裁判测试程序样例：**

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

//该函数由系统提供
char *dstr_readword();

char *dstr_create(const char *s);
char *dstr_add(char *s, char c);
char *dstr_concat(char *this, const char *that);

int main()
{
    char *s = dstr_create("hello");
    s = dstr_add(s, '!');
    printf("%lu-%s\n", strlen(s), s);
    char *t = dstr_readword();
    s = dstr_concat(s, t);
    free(t);
    printf("%lu-%s\n", strlen(s), s);
    free(s);
}

/* 请在这里填写答案 */
```

### Sample Input:

123A

### Sample Output:

6-hello!  
10-hello!123A

### 代码实现:

```

char *dstr_create(const char *s){
    int len=strlen(s);
    char *s1=(char *)malloc(len*sizeof(char)+1);
    //体会一下为什么是len*sizeof(char)+1
    strcpy(s1,s);
    return s1;
}

char *dstr_add(char *s, char c){
    s=(char *)realloc(s,strlen(s)+2); //realloc函数的使用
    //体会一下为什么是strlen(s)+2
    *(s+strlen(s)+1)='\0';
    *(s+strlen(s))=c;
    return s;
}

char *dstr_concat(char *this, const char *that){
    this=(char *)realloc(this,strlen(that)+strlen(this)+1);
    char *t=this+strlen(this);
    strcat(t,that); //strcat不用加'\0'
    return this;
}

```

## HW12

### Code Completion 6-1 奇数值节点链表

本题要求实现两个函数，分别将读入的数据存储为单链表、将链表中奇数值的结点重新组成一个新的链表。

**链表节点定义如下：**

```

struct ListNode {
    int data;
    ListNode *next;
};

```

**函数接口定义如下：**

```
struct ListNode *readlist();
struct ListNode *getodd( struct ListNode **L );
```

函数 `readlist` 从标准输入读入一系列正整数，按照读入顺序建立单链表。当读到`-1`时表示输入结束，函数应返回指向单链表头结点的指针。

函数 `getodd` 将单链表 `L` 中奇数值的结点分离出来，重新组成一个新的链表。返回指向新链表头结点的指针，同时将 `L` 中存储的地址改为删除了奇数值结点后的链表的头结点地址（所以要传入 `L` 的指针）。

**裁判测试程序样例：**

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

struct ListNode {
    int data;
    struct ListNode *next;
};

struct ListNode *readlist();
struct ListNode *getodd( struct ListNode **L );
void printlist( struct ListNode *L )
{
    struct ListNode *p = L;
    while (p) {
        printf("%d ", p->data);
        p = p->next;
    }
    printf("\n");
}

int main()
{
    struct ListNode *L, *Odd;
    L = readlist();
    Odd = getodd(&L);
    printlist(Odd);
    printlist(L);

    return 0;
}

/* 你的代码将被嵌在这里 */
```

### Sample Input:

```
1 2 2 3 4 5 6 7 -1
```

### Sample Output:

```
1 3 5 7
2 2 4 6
```

代码实现：

```
struct ListNode *readlist(){
    struct ListNode *left=NULL,*right=NULL,*p;
    int temp=0;
    int t=scanf("%d ",&temp);
    while(temp!=-1){
        p=(struct ListNode *)malloc(sizeof(struct ListNode));
        p->data=temp;
        if(left==NULL) right=p, left=p;//注意left和right的初始化
        else{
            right->next=p;
            right=p;
        }
        t=scanf("%d",&temp);
        t++;
    }
    right->next=NULL;
    return left;
}

struct ListNode *getodd( struct ListNode **L ){//注意L是一个二级指针，所以要传指针的地址
    if(*L==NULL) return NULL;
    struct ListNode *EvenHead=NULL,*OddHead=NULL,*p,*EvenTail,*OddTail;
    int temp=0;
    while(*L){
        struct ListNode *temp=L;//保存L的地址，便于后续free
        temp=(*L)->data;
        p=(struct ListNode *)malloc(sizeof(struct ListNode));//新建一个p节点，便于奇偶节点的
        p->data=temp;
        p->next=NULL;
        if(temp%2!=0){
            if(OddHead==NULL) OddHead=p,OddTail=p;
            else{
                OddTail->next=p;
                OddTail=p;
            }
        }
        else{
            if(EvenHead==NULL) EvenHead=p,EvenTail=p;
            else{
                EvenTail->next=p;
                EvenTail=p;
            }
        }
        *L=(*L)->next;
    }
}
```

```
    free(temp); //释放原先的L链表节点  
}  
*L=EvenHead;  
return OddHead;  
}
```

## Week 13

### HW13

#### True or False 1-1

文件指针和位置指针都是随着文件的读写操作在不断改变。

- T  
 F

解析：

F。

- 文件指针是指向文件的指针，用于定位文件中的位置。
- 位置指针是用于在文件中移动的指针，用于读取、写入和定位文件中的位置。
- 文件指针和位置指针是两个独立的概念，它们的作用和用法是不同的。

相关链接：

- [C语言中文件指针，文件位置指针，详细解析](#)

#### True or False 1-2

随机操作只适用于文本文件

- T  
 F

解析：

F。

### True or False 1-3

随机操作只适用于二进制文件

- T
- F

解析：

F。

### True or False 1-6

fseek 函数一般用于文本文件

- T
- F

解析：

F。

## 随机操作(Random Access)

定义：

- **随机操作(Random Access)** 是指在文件中可以直接访问任意位置的数据，而不需要按照顺序依次读取或写入。这种操作方式适用于**随机访问文件 (Random Access File)**，例如数据库文件、二进制文件等。
- 在**随机访问文件**中，每个数据项都有一个唯一的地址，可以通过该地址直接访问数据。这使得在文件中查找、修改或删除数据变得更加高效，因为不需要像顺序访问文件那样逐个读取或写入数据。

- 然而，随机操作并不适用于所有类型的文件。对于文本文件，由于其结构是基于字符的，而不是基于字节的，因此在进行随机操作时需要考虑字符的编码方式，以确保操作的正确性。此外，文本文件的结构也可能会影响随机操作的效率，因为在文本文件中，字符的位置可能不是固定的，因此在进行随机操作时，需要遍历整个文件来查找目标字符。
- 因此，在处理文件时，需要根据文件的类型和结构选择合适的操作方式。对于随机访问文件，可以使用**随机操作**来提高操作效率；对于文本文件，建议使用**顺序操作**来处理。

## 适用文件：

- **二进制文件**
- **少部分文本文件**

对于一些简单的文本文件，如**配置文件**或**日志文件**，其中的数据结构相对简单，并且数据的长度是固定的或可预测的，那么在这种情况下，随机操作可能是可行的。例如，如果一个配置文件中的每个配置项都占用固定的行数，那么可以通过计算行号来直接访问特定的配置项。

### Multiple Choice 2-3

若读文件还未读到文件末尾，`feof()` 函数的返回值是（）。

- A. -1
- B. 0
- C. 1
- D. 非0

### 解析：

B。

`feof()` 函数用于检查文件指针是否已经到达文件末尾。如果文件指针已经到达文件末尾，它将返回非零值（通常是1）；如果文件指针未到达文件末尾，它将返回0。

### Multiple Choice 2-4

若 `fopen()` 函数打开文件失败，其返回值是（）。

- A. 1

- B. -1
- C. NULL
- D. ERROR

解析：

C。

`fopen()` 函数用于打开文件并返回一个文件指针。如果文件打开成功，它将返回一个指向文件的指针；如果文件打开失败，它将返回 `NULL`。

### Multiple Choice 2-5

`fputc(ch, fp)` 把一个字符ch写到fp所指示的磁盘文件中，若写文件失败则函数的返回值为（）。

- A. 0
- B. 1
- C. EOF
- D. 非零

解析：

C。

`fputc()` 函数用于将一个字符写入文件。如果写入成功，它将返回写入的字符；如果写入失败，它将返回 EOF (End of File)。

### Code Completion 6-7 单链表分段逆转

给定一个带头结点的单链表和一个整数K，要求你将链表中的每K个结点做一次逆转。

例如给定单链表  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$  和  $K = 3$ ，你需要将链表改造成  $3 \rightarrow 2 \rightarrow 1 \rightarrow 6 \rightarrow 5 \rightarrow 4$ ；

如果  $K = 4$ ，则应该得到  $4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 5 \rightarrow 6$ 。

裁判测试程序样例：

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

typedef int ElementType;

typedef struct Node *PtrToNode;
struct Node {
    ElementType Data; /* 存储结点数据 */
    PtrToNode Next; /* 指向下一个结点的指针 */
};
typedef PtrToNode List; /* 定义单链表类型 */

List ReadInput(); /* 裁判实现，细节不表 */
void PrintList( List L ); /* 裁判实现，细节不表 */
void K_Reverse( List L, int K );

int main()
{
    List L;
    int K;

    L = ReadInput();
    scanf("%d", &K);
    K_Reverse( L, K );
    PrintList( L );

    return 0;
}

/* 你的代码将被嵌在这里 */
```

`L` 是给定的带头结点的单链表，`K` 是每段的长度。函数 `K_Reverse` 应将 `L` 中的结点按要求分段逆转。

**函数接口定义如下：**

```
void K_Reverse( List L, int K );
```

**代码实现：**

```
void K_Reverse( List L, int K ){
    int cnt=0,i=0;
    List p=NULL,q,l=L->Next;//注意链表是带头节点的
    while(l!=NULL){
        i++;
        if(i%K==0) cnt++;//计算有多少段需要逆转
        l=l->Next;
    }
    if(i>=K){//至少有一段需要逆转
        l=L->Next;//l的初始化
        List temptail=L->Next,head;
        int flag=0;
        while(cnt>0){
            List temp=l;
            i=0;
            while(i<K){
                q=l->Next;
                l->Next=p;
                p=l;
                l=q;
                i++;
            }
            //逆转链表的过程
        }
        if(flag){
            temptail->Next=p;//上一段的尾巴接上这一段的头
            temptail=temp;//更新尾巴
        }
        else flag=1,head=p;//第一次逆转时，head头节点的位置
        cnt--;
    }
    L->Next=head;//链表头节点更新
    tail->Next=l;//尾巴的更新
}
}
```

# Week 14

## HW14

### True or False 1-5

#define PI 3.1415926 是一条C语句。

- T
- F

解析：

F。

### True or False 1-6

宏定义不存在类型问题，宏名无类型，它的参数也无类型。

- T
- F

解析：

T。

### Multiple Choice 2-2

凡是函数中未指定存储类别的局部变量，其隐含的存储类型为( )

- A. auto 自动
- B. static 静态
- C. extern 外部
- D. register 寄存器

解析：

A。

## 存储变量类型分类

### 1. 自动变量 auto

函数中所有的非静态局部变量、程序中默认的变量都是auto自动变量。

例如: `int num = 100;`

`num`就是一个自动变量,auto是可以省略的

但只限于C语言中,C++中不能加auto.

C语言中, `int num = 100;` 和 `auto int num = 100;` 是相同的

C++中 `auto int num = 100` 会报错。

### 2. 静态变量 static

- 局部静态变量

静态变量在整个程序生命周期中只拷贝一份, 如果某函数内的静态变量被访问并且值发生了改变, 那么他就会保存新的值。

- 全局静态变量

定义在最前面的 `static int num = 100;`

在代码中任何地方都可以访问到, 而局部静态变量只能在定义他的函数或者块中才能访问到.

### 3. 外部变量 extern

把全局变量在其他源文件中声明成extern 变量, 可以扩展该全局变量的作用域至声明的那个文件, 其本质作用就是对全局变量作用域的扩展。

假设有文件aa.c

```
#include<stdio.h>
int num = 100;
```

有文件bb.c

```
#include<stdio.h>
#include<stdlib.h>
extern int num;
int main(){
    printf("%d\n", num);
}
```

输出: 100

将变量num声明成外部变量extern，可以将num的作用域拓展至bbb.c文件中。

#### 4. 寄存器变量 register

寄存器是CPU上面的一个挂件，运行速度很快，一般用在某些常被访问的变量上，将该变量设置为寄存器变量存储在寄存器中方便CPU使用。即某个变量如果一直要被CPU使用的话，存储在寄存器中要比存储在内存中读取起来快的多。

注意点1:

例如：register int num = 100;

num被声明成一个寄存器变量，其保存在寄存器中是打印不出地址的&num，但是如果使用 printf("ox%p",&num); 打印其地址的，就会变成普通的auto变量。

注意点2:

寄存器变量不能定义到全局变量位置

#### Multiple Choice 2-8

以下是一个C语言程序的除标准库之外的全部源代码，则说法正确的是：

```
#include <stdio.h>
extern int k;
int main() {
    k = 2223;
    printf("%d\n", k);
    return 0;
}
```

- A. 这段程序编译错误。
- B. 这段程序编译正确，但是链接（link）错误。
- C. 这段程序编译、链接正确，但是运行时错误。
- D. 程序无错，可正常运行。

解析：

B。

extern关键字表示变量k是一个外部声明，程序假设变量k已在其他地方定义（通常在另一个源文件中）。

但在当前代码中，并没有提供变量k的定义，因此在编译阶段不会报错，但在链接（linking）阶段会出错，因为链接器找不到k的实际定义。

## Multiple Choice 2-11

有如下多文件组织：

### **header.h**

```
#ifndef _HEADER_H
#define _HEADER_H
char school[] = "Sanben";
void fun(char* s);
#endif
```

### **File1.c**

```
#include "header.h"
int main()
{
    fun(school);
    printf("%s", school);
}
```

### **File2.c**

```
#include "header.h"
#include <string.h>
void fun(char* s)
{
    strcpy(s, "Yiben");
    return;
}
```

程序输出结果为：

- A. Sanben
- B. Yiben
- C. 编译错误
- D. 链接错误

**解析：**

D。

每个包含 `header.h` 的源文件都会认为 `school` 是一个新的变量定义。

因此，当 `File1.c` 和 `File2.c` 同时包含 `header.h` 后，两个目标文件（`File1.o` 和 `File2.o`）中会各自定义一个 `school` 变量。

在链接阶段，链接器会发现 `school` 被多次定义，报出链接错误。

### Multiple Choice 2-12

C语言的全局变量的初始化是在以下哪个阶段完成的：

- A. `main()` 函数开始后
- B. 编译链接的时候
- C. `main()` 函数开始前
- D. 第一次用到的时候

#### 解析

答案选C。

C. 全局变量在程序加载到内存时（即在 `main()` 函数执行之前）就会被分配存储空间，并初始化。初始化发生在程序的启动代码（由运行时库完成）执行时，`main()` 函数被调用之前。

而B. 编译链接的时候：编译和链接阶段只是分配变量的符号表和地址信息，但不会进行初始化。



### 链接错误 (Link Error)

连接错误是指程序在**编译成功后**，但在链接阶段（linking）出现的问题。链接错误通常发生在编译器试图将各个目标文件（object files）和库文件结合在一起，生成可执行文件的过程中。此时，如果有未定义或无法解析的符号（如变量、函数等），就会导致链接失败。

## 链接的作用

编译过程通常分为以下几步：

1. **预处理 (Preprocessing)**：处理宏定义、头文件等，生成纯C代码。
2. **编译 (Compilation)**：将源代码（C文件）编译为目标文件（.o 或 .obj）。
3. **链接 (Linking)**：将多个目标文件和所需的库文件链接在一起，生成最终的可执行文件。

链接阶段的作用是将各个独立编译的模块（目标文件）及库文件中的符号进行解析，使它们彼此关联，完成程序的整体组装。

## 链接错误的常见原因

链接错误通常与以下情况有关：

### 1. 未定义的符号（Undefined Reference）：

- 变量或函数在某个文件中被声明，但没有实际定义。
- 示例：

```
extern int x; // 声明外部变量 x
int main() {
    x = 10; // 使用变量 x
    return 0;
}
```

**错误原因：** x 被声明为外部变量，但未在任何地方定义。

### 2. 多重定义（Multiple Definition）：

- 同一个符号（变量或函数）在多个文件中重复定义。
- 示例：

文件 file1.c：

```
int x = 10;
```

文件 file2.c：

```
int x = 20;
```

编译时不会报错，但在链接阶段会出现冲突。

### 3. 函数缺失：

- 调用了某个函数，但没有提供该函数的实现（定义）。
- 示例：

```
void func(); // 声明
int main() {
    func(); // 调用
    return 0;
}
```

如果没有提供 `func` 的定义，链接器会报错。

#### 4. 库文件丢失：

- 程序依赖的库文件没有包含在链接过程中。
- 示例：使用数学库函数 `sin`，但未链接数学库（需要 `-lm`）。

#### 5. 链接顺序错误：

- 在使用静态库时，库的链接顺序不正确会导致符号未解析。

## 编译的四个过程

C语言程序从源代码到可执行文件，通常需要经过**四个主要阶段**：**预处理 (Preprocessing)**、**编译 (Compilation)**、**汇编 (Assembly)** 和 **链接 (Linking)**。以下是各阶段的详细解释：

## 1. 预处理 (Preprocessing)

### • 主要功能：

- 处理源代码中的预处理指令（如 `#include`、`#define` 等）。
- 删除注释，展开宏，将头文件的内容插入到源代码中。
- 生成一个**预处理后的源文件**（通常后缀为 `.i` 或 `.ii`）。

### • 关键操作：

- **宏替换：** 替换 `#define` 定义的宏，例如：

```
#define PI 3.14
printf("%f", PI); // 替换为 printf("%f", 3.14);
```

- **头文件展开：** 将 `#include` 的头文件内容插入到源文件中。
- **条件编译：** 根据宏条件判断是否保留某段代码（如 `#ifdef`）。
- **工具和命令：**

- 常用的 GCC 命令: `gcc -E main.c -o main.i`

## 2. 编译 (Compilation)

- 主要功能:**
  - 将预处理后的源代码（.i 文件）翻译为汇编代码（.s 文件）。
  - 检查语法和语义错误。
- 关键操作:**
  - 语法分析:** 确保代码符合 C 语言语法规则。
  - 语义分析:** 检查变量是否定义、类型是否匹配、函数调用是否正确等。
  - 中间代码生成:** 将代码转换为编译器内部的中间表示 (IR)。
  - 优化:** 对代码进行优化（如循环展开、常量折叠等）。
  - 生成汇编代码:** 输出汇编代码文件（.s 文件）。
- 工具和命令:**
  - 常用的 GCC 命令: `gcc -S main.i -o main.s`

## 3. 汇编 (Assembly)

- 主要功能:**
  - 将汇编代码（.s 文件）翻译为机器代码，生成目标文件（.o 文件）。
- 关键操作:**
  - 汇编器根据目标平台的指令集，将汇编代码翻译为对应的机器指令。
  - 生成目标文件，目标文件中包含机器指令和二进制数据，但并非完整的可执行程序。
- 工具和命令:**
  - 常用的 GCC 命令: `gcc -c main.s -o main.o`

## 4. 链接 (Linking)

- 主要功能:**
  - 将多个目标文件（.o 文件）和依赖的库文件组合在一起，生成最终的可执行文件。
  - 解析外部符号（如跨文件的函数调用或全局变量引用）。
- 关键操作:**
  - 符号解析:** 找到所有未定义的符号（如函数或变量的定义）。
  - 地址分配:** 为每个目标文件中的代码和数据分配内存地址。

- **库文件链接：** 将静态库或动态库的内容与目标文件结合。
- **工具和命令：**
- 常用的 GCC 命令： `gcc main.o -o main`

## 四个阶段的总结

以下是每个阶段的输入和输出文件的对应关系：

| 阶段  | 输入文件      | 输出文件      | 命令示例                                 |
|-----|-----------|-----------|--------------------------------------|
| 预处理 | 源文件（.c）   | 预处理文件（.i） | <code>gcc -E main.c -o main.i</code> |
| 编译  | 预处理文件（.i） | 汇编文件（.s）  | <code>gcc -S main.i -o main.s</code> |
| 汇编  | 汇编文件（.s）  | 目标文件（.o）  | <code>gcc -c main.s -o main.o</code> |
| 链接  | 目标文件（.o）  | 可执行文件     | <code>gcc main.o -o main</code>      |

## 示例代码的完整编译流程

假设源文件是 `main.c`：

```
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

### 1. 预处理 (`gcc -E main.c -o main.i`)

- 输出文件 `main.i`，内容为展开后的代码（头文件和宏替换后）。

### 2. 编译 (`gcc -S main.i -o main.s`)

- 输出文件 `main.s`，内容为汇编代码。

### 3. 汇编 (`gcc -c main.s -o main.o`)

- 输出文件 `main.o`，内容为机器代码的目标文件。

#### 4. 链接 ( gcc main.o -o main )

- 输出文件 `main`，这是最终的可执行文件。

## 注意

- 错误检查：
- 预处理阶段：宏错误或头文件找不到会报错。
- 编译阶段：语法错误、语义错误会在此阶段报错。
- 汇编阶段：汇编代码生成问题会报错（很少见）。
- 链接阶段：外部符号未定义、多重定义等问题会在链接阶段报错。

### Code Completion 6-1 快速幂的实现

#### Quick Power

```
int Power(int N, int k){  
    int temp=N%MOD,ans=1;  
    while(k>0){  
        if(k&1) ans=ans*temp%MOD;  
        temp=temp*temp%MOD;  
        k>>=1;  
    }  
    return ans%MOD;  
}
```

### Programming 7-2 研究：双精度实数的机内码\*

请编写程序，输入十进制双精度实数，输出其 64 位机内码。

要求：以十六进制形式输出机内码

Sample Input

3.6

Sample Output

400CCCCCCCCCCCCCD

## Explanation:

要求：十六进制机内码中的字母均为大写。

例如：实数 3.6 转换成二进制是 11.1001100110011001...，科学计数法记为：

$1.11001100110011001... \times 2^1$

因此 64 位的机内码为：

01000000000011001100110011001100110011001100110011001100110011001101

用十六进制书写则为：400CCCCCCCCCCCCCD

```
#include<stdio.h>
int main()
{
    double num;
    scanf("%lf",&num);
    unsigned char* p=&num;
    for(i=sizeof(num)-1;i>=0;i--)
        printf("%02X",*(p+i));//%02X 表示输出两位十六进制数，不足两位用0填充
}
```

## 相关资源

- 惭愧！直到今天才真正明白为什么 int 型的取值范围是  $-2^{31} \sim 2^{31}-1$
- int、unsigned int、float、double 和 char 在内存中存储方式

## Markdown 语法大全