C语言

C语言的代码构成

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

一个C语言程序由以下的几部分构成：

- 头文件<stdio.h>：扩展名为 **.h** 的文件，包含了 C 函数声明和宏定义，被多个源文件中引用共享

  简单而言，我们在编程过程中会调用许多的函数（如上面最简单的printf，以及scanf等等），这些函数的定义都是在头文件中进行或者头文件提供引用。（在头文件部分会详细讲解）
- 主函数main()：程序执行的主要入口，在任何一个程序中都应该有且仅有一个主函数，没有主函数程序没有办法找到相应的程序入口，程序报错；有多个主函数（事实上不可能有多个主函数，因为函数不能重名，但是多个代码文件时可能出现）时，程序同样无法找到程序入口。
- 代码语句：用来执行相应操作的代码。
- 返回值：函数部分详细讲解。

对于一个最为简单的C语言程序而言，包括最为基础的头文件stdio.h，有一个主函数入口main即可运行。

C语言语法（要点强调）

在该部分中我们不涉及具体的基础语法，只会对语法中容易出现的问题进行简单的讲解。

if结构连比的问题

```
#include<stdio.h>
void printNumber(int number);
int main(){
    int a, b, c;
    printf("This program should output the greatest number among the three\n");
    printf("please input three numbers using black to divide:\n");
    scanf("%d %d %d", &a, &b, &c);
    if(a < b < c)
        printNumber(c);
    if(a < c < b)
        printNumber(b);
    if(b < a < c)
        printNumber(c);
    if(b < c < a)
        printNumber(a);
    if(c < b < a)
        printNumber(a);
    if(c < a < b)
        printNumber(b);
```

```
    return 0;
}

void printNumber(int number){
    printf("The greatest numbers among inputed is %d\n", number);
}
```

代码会从上至下，从左至右执行，同时在C语言中，True用1来表示，False用0来表示。所以上面的代码可能会输出多个值。

上面的代码遵从以下的逻辑：假设我们输入的三个数是30，10，20，那个执行第一个判断，30 < 10为假，此时（30 < 10）变成了0，这时候判断语句成为if（0 < 20），得出的结果为真（True），输出20。发生逻辑错误。

此时应当使用逻辑运算符 &&，||，！进行判断。

```
#include<stdio.h>
void printNumber(int number);
int main(){
    int a, b, c;
    printf("This program should output the greatest number among the three\n");
    printf("please input three numbers using black to divide:\n");
    scanf("%d %d %d", &a, &b, &c);
    if(b >= a && b >= c)
        printNumber(b);
    if(a >= b && a >= c)
        printNumber(a);
    if(c >= a && c >= b)
        printNumber(c);
    return 0;
}

void printNumber(int number){
    printf("The greatest numbers among inputed is %d\n", number);
}
```

循环的执行过程

```
#include<stdio.h>
int main(){
    //while循环
    int i = 0;
    while(i < 10)//判断语句（1）（4）
    {
        //循环体（2）
        printf("test codes for while cycle.\n");
        //自增条件（3）
        i++;
    }

    //do while循环
    int j = 0;
    do
    {
```

```
        //循环体（1）（4）
        printf("test codes for do while cycle.\n");
        //自增条件（2）
        j++;
    }
    while(j < 10);//判断语句（3）

    //for循环
    for(int i = 0/*初始化语句（1）*/;i < 10/*判断条件（2）（5）*/;i++/*自增条件（4）*/)
    {
        //循环体（3）
        printf("test codes for for cycle.\n");
    }
}
```

三种循环语句的执行根据标号进行顺次执行。

占位字符与转义字符

占位字符的存在主要是为了格式化输入输出而存在的。由于C语言的特性，其不能直接在printf中直接调用变量（其他语言基本上都可以），所以需要一个占位字符告诉系统，我需要把一个变量的值放在这个位置。同时占位符的存在可以格式化输出文本。

```
#include<stdio.h>

int main(){
    int temp = 32;
    double temp1 = 43.34343432;
    printf("%8d\n", temp);//在宽度为8的长度中右对齐输出
    printf("%-8d\n", temp);//在宽度为8的长度中左对齐输出
    printf("%08d\n", temp);//在宽度为8的长度中右对齐输出，前方用0填充
    printf("%.2lf\n", temp1);//输出两位小数
    return 0;
}
```

C语言程序的运行过程*

所有在计算机上运行的程序都必须是可执行文件（即以.exe作为后缀名的文件），CPU通过读取二进制的指令并执行相应的操作，在终端显示上则为程序的运行。在本部分主要介绍我们写的代码是如何从代码的形式变为计算机可以理解的二进制形式。由于这一部分介绍的内容不会过多涉及，故而较为简略。
#

编译器与解释器（C to 汇编）

编译器或者解释器被定义为一种翻译程序，用于将一种语言形式的代码翻译为另一种语言形式的代码。但是通常理解的编译器或者解释器指的是将高级语言（C， Java， Python等）翻译成低级语言（汇编语言）的程序。

在详细讲解C语言程序的编译过程前，先解释一下编译器与解释器的区别。

- 编译器：整体化的过程，在代码编写完成后将整个代码文件输入到编译器中，经过处理最后得到了汇编语言书写的文件。从报错的角度来说，编译器会完成对进入程序的检验后输出所有的错误，错

误未修正的情况下无法进行编译。（这一种方式主要用于C系列语言，Java等）
- 解释器：逐一进行的过程。在代码输入的过程中或者接收到整个代码文件时边翻译边检查，如果发现某一行存在语法错误，则解释过程立刻停止，必须要进行修正后才能进行下一行代码的解释。（这一种方式主要应用python， PHP，RuBy等语言）

在当前普遍使用IDE（integrated development environment， 集成开发环境）的情况下，一般不会对编译器与解释器进行区分（毕竟IDE内核会进行一定的预编译过程，这点与解释器有点像（这也是为什么Clion与codeblocks可以在未编译的情况下可以报错）），但是解释器的执行过程与编译器是完全不同的。

这一部分将接受.i文件并翻译形成一个.s文件，并且在.s形成后传入到汇编器（翻译程序，代码在格式上发生了变化），输出一个.o文件。

预处理（C to C）

在C语言进入编译器前，C源代码会进入预处理器中，这一部分将展开宏定义（#define，主要用于全局变量的定义或者头文件内容的定义等等，使用见下）部分的内容，但此时的代码仍旧为人类可以理解的高级语言代码，只不过是在项目内连接了所有的代码文件。

```
//这一段代码主要表明了引入了头文件，并且定义了全局变量（后四个），只要引用了这个文件就可以调用这些变量（VIPS_FILE）
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include "tools.h"
#define EQUIPMENTS_FILE "../data/equipments.txt"
#define HOUSES_FILE "../data/houses.txt"
#define USERS_FILE "../data/users.txt"
#define VIPS_FILE "../data/vips.txt"
```

这一部分将生成一个.i文件用于存储形成的代码文件

连接（汇编 to 机器码（二进制））

这一部分主要处理的是我们引入的#include部分的内容。连接器将读取汇编代码中使用的头文件，通过匹配将调用的函数变换为二进制格式的机器码，形成最后的.exe文件。到此。C语言的程序的编译到此结束。

C语言中的数据类型

C语言中的数据类型可以大致分为四类：基本数据类型，派生数据类型，枚举类型与void类型，但是由于后两种并不是很常用，故而在本部分内容中不进行讲解。下面简要介绍前两种数据类型。

基本数据类型

基本数据类型可以分为两类：整数型与浮点型

- 整数型：主要包括int（integer，整型），unsigned int（无符号整型，不能用于处理负数），short（短整型，存储的字节数减半，存储范围也会减半），unsigned short（同上），long（长整型，存储的字节数加倍，存储范围加倍），unsigned long（同上），同时还包括存储字符的char类型
- 浮点型，主要为处理小数，包括float（单精度浮点型，存储的字节数为4个字节），double（双精度浮点型，存储的字节数为8个字节）

上述分类的依据并不是处理的是整数还是小数，而是根据其在内存中的存储方式不同进行分类。

由于硬盘中所有数据都以二进制的形式进行存储，而为了满足现实中涉及到负数与小数的运算法则，计算机中对与带有小数的数的存储会采用另一种编码格式（反码，补码等等，计算机组成原理会讲），所以专门分出了一个类别用来处理小数。

而char属于整数型是因为char在硬盘中的存储也是以整数存储进行的。（整数与键盘上所有字符的对应关系——ASCII码表，256个字符），同时由于只有256个字符（美国标准），所以在ASCII码表中不存在汉字，故而不能定义char temp = '那';

note：65,535与32，768

派生数据类型

这一部分数据类型是基于基本数据类型产生的，主要是数组，指针等一些列数据类型以及自定义的数据类型（共用体与结构体）。

C语言中的常量

常量，指的是固定值，在程序执行期间不会改变。例如在计算圆的周长中我们可以将π定义为一个常量PI（常量的命名一般用大写）

```
#include<stdio.h>
#define PI 3.1415926    //第一种常量的定义方式，可以作为全局变量使用
int main(){
    double diameter;
    const double PI1 = 3.1415926;   //第二种常量定义的方式，使用const关键字
    printf("please input the diameter of the circle:\n");
    scanf("%lf", &diameter);
    printf("The perimeter of the circle is %.2lf\n", PI * diameter);
}
```

有两种方式可以用于常量的定义：宏定义以及const关键字。

常量既然称之为常量，表明其无法接受后来的赋值，同时需要再常量定义的时立刻进行赋值，否则系统报错。

C语言中变量的作用域

在前一部分中我们已经提到了，变量以及常量存在全局变量与局部变量的分类，下面具体解释变量的作用范围。（我定义的变量可以在哪个位置去使用它）

最基本的判断原则：大括号

```
//全局变量
#include<stdio.h>
void canculate();
#define PI 3.1415926
int main(){
    canculate();
    return 0;
}

void canculate(){
    double diameter;
```

```
    printf("please input the diameter of the circle:\n");
    scanf("%lf", &diameter);
    printf("The perimeter of the circle is %.2lf\n", PI * diameter);
}
```

全局常量定义后可以在全局进行使用，即可以在对应的.c文件中使用。同时如果是在.h文件中定义的，那么引用了该头文件的.c文件也可以使用对应的全局常量

```
//局部变量
#include<stdio.h>
void printNumber();
int main(){
    int temp = 10;
    printNumber();
    return 0;
}

void printNumber(){
    printf("The number you set is %d\n", temp);
}
```

上面的段代码会报错，因为变量temp的作用域不包括printNumber函数，此时printNumber韩式中无法调用temp

如果我们要实现以上的效果，那么可以把temp作为一个参数传入到printNumber函数中

```
//局部变量2
#include<stdio.h>
void printNumber(int temp);
int main(){
    int temp = 10;
    printNumber(temp);
    return 0;
}

void printNumber(int temp){
    printf("The number you set is %d\n", temp);
}
```

同时在循环结构以及选择结构中同样遵循大括号的原则

```
//局部变量3
#include<stdio.h>
void printNumber(int temp);
int main(){
    int temp = 10;
    if(temp > 0){
        int temp1 = 20;
        printNumber(temp); //temp的定义在main函数的大括号内，同时在调用之前，故而可以调用
        printNumber(temp1); //temp1的定义在if的大括号内，同时在调用之前，故可以调用
    }
    //printf("%d\n", temp1);
    //上面这行代码会报错，因为其无法读取到在if结构里面定义的temp1
```

```
    return 0;
}

void printNumber(int temp){
    printf("The number you set is %d\n", temp);
}
```

```
//局部变量4
#include<stdio.h>
void printNumber(int temp);
int main(){
    int temp = 10;
    for(int i = 0;i < 10;i++)
        printNumber(temp);
    //printf("The variable temp is printed %d times.\n", i);
    //上面这行代码会报错，因为在for循环中定义的i是在循环开始时定义的，无法访问
    return 0;
}

void printNumber(int temp){
    printf("The number you set is %d\n", temp);
}
```

如果想访问循环的次数，应当采用下面的方法

```
//局部变量5
#include<stdio.h>
void printNumber(int temp);
int main(){
    int temp = 10, i;//外部提前进行定义
    for(i = 0;i < 10;i++)
        printNumber(temp);
    //printf("The variable temp is printed %d times.\n", i);
    //上面这行代码会报错，因为在for循环中定义的i是在循环开始时定义的，无法访问
    return 0;
}

void printNumber(int temp){
    printf("The number you set is %d\n", temp);
}
```

在代码上，下面这段代码和上面那段代码是等效的。

```
//局部变量6
#include<stdio.h>
void printNumber(int temp);
int main(){
    int temp = 10, i;
    while(i < 10){
        printNumber(temp);
        i++;
    }
    return 0;
}
```

```c
void printNumber(int temp){
    printf("The number you set is %d\n", temp);
}
```

C语言中的函数

首先明确一个内容，我们为什么需要函数？——为了偷懒。虽然函数的出现可以使得代码在封装以及调用属性上做出了众多的调整与更新，但是函数的出现主要是为了进行一段代码的多次复用（程序员不想敲代码是这样的）

```c
#include<stdio.h>
int main(){
    int var1 = 10, var2 = 20, var3 = 30, var4 = 40;
    printf("The sum of the %d and %d is %d", var1, var2, var1+var2);
    printf("The sum of the %d and %d is %d", var3, var4, var3+var4);
    return 0;
}
```

```c
#include<stdio.h>
void sum(int a, int b);
int main(){
    int var1 = 10, var2 = 20, var3 = 30, var4 = 40;
    sum(var1, var2);
    sum(var3, var4);
    return 0;
}

void sum(int a, int b){
    printf("The sum of the %d and %d is %d", a, b, a + b);
}
```

函数的结构

```c
#include<stdio.h>
void sum(int a, int b);//对函数的预定义
int main(){
    int var1 = 10, var2 = 20, var3 = 30, var4 = 40;
    sum(var1, var2);    //对函数的调用
    sum(var3, var4);    //对函数的调用
    return 0;
}

void/*返回值类型*/ sum/*函数名*/(int a, int b/*参数列表*/){
    //函数体
    printf("The sum of the %d and %d is %d", a, b, a + b);
    //返回值
    //return 0;
}
```

一个完整的函数主要由上述部分的内容构成：

- 函数名：区别于其他函数的最主要标志，在C语言中，同一个.c文件中（或者在同一个头文件下？）不允许出现同名的函数。
- 参数列表：把函数想象成一个黑盒，参数列表就是整个黑盒的输入端口规范。
    - 在参数列表中可以传递任意数据类型（任意，包括自定义的结构体以及数组等等，但是在实际操作中建议用指时可以传入指针替换）。
    - 参数列表质询要保证传递的数据类型一致即可。（我在参数列表中定义了两个int类型变量a，b，但是我在调用时传入的时var1，var2），但是请注意传入的次序问题（var1与var2的次序不能颠倒，因为变量名已经定义）
    - 值传递与地址传递的问题（指针部分会讲）
- 返回值类型：把函数想象成一个黑盒，返回值类型就是输出端口的规范。需要注意的是，此处传递的数据类型要求与参数列表中的要求类似，但是从编码规范的角度来说，我们一般会尽量减少数组或者结构体一类数据类型的传递（用指针替代）。同时函数能且仅能返回一个变量（python中可以返回多个），同时不管函数是否正常执行，只要退出函数则必须要返回一个符合返回值规范的返回值。（void函数除外）
- 函数体：函数体即为中间的处理过程
- 返回值：除了void类型的函数外，无论函数在什么时候退出，都需要存在一个返回值。
- 预定义：这一部分属于C语言的一个特点，函数必须在被调用之前定义，如果我们在main之中调用函数，而且main是第一个函数，那么就需要写一行函数的预定义用于告诉系统，我确实定义了这一个函数。

main函数

main函数本质上也是一个函数，但是程序需要一个执行的入口，所以需要一个名为main的函数，除此之外，main函数都可以进行改变。

```
#include<stdio.h>
void main(int argc, char* argv[]){
    printf("The number of parameters you pass in is %d\n", argc);
    if(argc == 2)
        printf("The parameter you pass in is %s\n", argv[1]);
}
```

但是这个程序在执行的时候如果要传递参数，有两种方法：1.在IDE中设定传递的参数是什么。2.用命令行的形式执行这个程序，传递参数。

```
# !bin/sh
gcc helloWorld.c
#上述代码编译之后会产生一个名为a.exe的可执行文件
#可以用下面这行代码设定编译后的名字
#gcc helloWorld.c -o helloWorld
./a.exe HelloWorld
#.表示当前的目录，用于在当前目录下寻找到a.exe并且运行他
#./helloWorld HelloWorld
```

C语言中的数据结构

数组

数组是同一数据类型的数据的集合。对于所有基本的数据类型都可以有对应的数组，如int[], double[] 等等。

数组的大小根据数组的长度以及其所属的数据类型而定，例如一个长度为10的int类型数组，其大小为10 * 4 = 40bytes。

（note：在使用char[]表示字符串时长度需要加一，以包括string终止符'\0'）

数组初始化

数组支持在声明的同时进行初始化，也支持在后续逐个进行初始化。需要注意的是，如果在声明的同时进行数组的初始化，则不需要再填写数组的长度。

```
#include<stdio.h>
int main(){
    //声明并初始化（√）
    int temp[] = {0, 1, 2, 3, 4};

    //声明后初始化（√）
    int temp1[5];
    for(int i = 0; i < 5;i++)
        temp1[i] = i;
    for(int i =0;i < sizeof(temp1)/sizeof(temp1[0]);i++)
        temp1[i] = i;

    //声明并初始化（X）
    int temp2[5] = {0, 1, 2, 3, 4};
}
```

完成数组的声明后，此时数组已经存在，不同数据类型的数组对应的默认值也不同。

```
#include<stdio.h>
int main(){
    int intArr[5];
    double doubleArr[5];
    char charArr[5];

    printf("The default value of the initial array is:\n");
    printf("int: %d\n", intArr[0]);
    printf("double: %lf\n", doubleArr[0]);
    printf("char: %c\n", charArr[0]);
    return 0;
}
```

数组元素的访问以及遍历

数组元素支持直接使用下标index进行访问，但需要注意的是，数组的下标从0开始（actually speaking，计算机大部分的操作起始数均为0而不是1），所以对于一个长度为5的数组，其最大的下标为4，故而在访问时应当注意数组下标越界的问题。在实际操作中，更推荐在遍历时采用sizeof（）/ sizeof（）的格式用于获取数组的长度。

（note：char数组进行声明初始化后最后一个元素为'\0'）

数组的增删改查

增

数组的长度是固定的，所以在新增元素时如果新增元素超出了数组的上界，则会出现数组越界的问题。故而在这时想新增元素，则需要重新声明一个更大的数组，将原数组的元素拷贝到新数组上，而后进行新增操作。（以上操作并不建议，最好应该在一开始就声明一个足够大的数组）

```
#include<stdio.h>
int main(){
    int temp1[5];
    for(int i = 0; i < 5;i++)
        temp1[i] = i;
    int copied[10];
    int index = 0;
    for(index = 0; index < 5;index++)
        copied[index] = temp1[index];
    copied[index] = 5;
    return 0;
}
```

删

由于数组的长度是固定的，所以删除时不需要进行额外的操作，直接用默认值或者其他不会出现的值覆盖即可。（例如数组存的是年龄，删除时使用-1覆盖即可）

改

修改与删除类似，直接用需要更新的值覆盖原来的值即可。

查

使用for循环对数组进行遍历，而后逐个对比查看元素是否相同，相同则为找到了元素。（事实上一般不会有这样的操作，因为数组中可以存储相同的元素，逐个比对是容易找到错误的下标）

数组排序

排序并不一定基于数组进行，而且排序的方法较多，但是考试中可能会考察排序的相关算法，所以在这里以数组为例讲解一下排序的bubble算法。

变量交换

在进入排序算法前先理解一下如何交换两个变量的值。

```
#include<stdio.h>
int main(){
    int a = 10, b = 20;
    printf("a:%d  b:%d\n", a, b);
    a = b;
    b = a;
    printf("a:%d  b:%d\n", a, b);
    return 0;
}
```

以上的方法是错误的。

变量的本质更类似于一个容器，在名为a的容器里面放着一个数为10， 在名为b的容器里面放了一个数为20，所以在执行 a = b时，b容器里面的数20被放到了a容器里面，这时a容器里面是有一个数10的，但是为了放下新来的数20，所以原来的数美丢掉了。为了留住被丢掉的数，需要引入第三个变量。

```
#include<stdio.h>
int main(){
    int a = 10, b = 20, temp;
    printf("a:%d  b:%d\n", a, b);
    temp = a;
    a = b;
    b = temp;
    printf("a:%d  b:%d\n", a, b);
    return 0;
}
```

冒泡排序

想象有一个泡泡池子，泡泡里面有不同的气体，如果泡泡包含了比空气密度小的气体，那么泡泡就会浮起来；相反，如果泡泡包含了比空气密度小的气体，那么泡泡就会沉下去。放到算法之中来，我们需要让大的数与小的数按照大小排开，那就比较相邻的两个数，满足条件就交换，不满足就继续。通过一轮过程，我们一定能把最大的数放到末尾，然后经过第二轮，我们一定可以把第二大的数放到倒数第二的位置。

```
#include<stdio.h>
void printArr(int* arr);
int main(){
    int temp[] = {67, 35, 12, 45, 6, 56, 23};
    printf("initial array: \n");
    printArr(&temp);
    int count = 0;
    for(int i = 0;i < 7 - 1;i++)
        for(int j = 0;j < 7 - 1;j++){
            if(temp[j] > temp[j+1]){
                count++;
                int temp1 = temp[j];
                temp[j] = temp[j+1];
                temp[j+1] = temp1;
                printf("exchange %d times\n", count);
                printArr(&temp);
            }
        }
    printf("final result: \n");
    printArr(&temp);
    return 0;
}

void printArr(int* arr){
    for(int i = 0;i < 7; i++)
        printf("%d ", *(arr + i));
    printf("\n");
}
```

但是我们发现，你在执行完n轮外层循环后，前n个数就会变得有序起来。

第一轮

比较temp[0] 与temp[1], temp[1]与temp[2],以此类推

第一次交换 temp[0] (67) > temp[1] (35), 交换 => {35, 67, 12, 45, 6, 56, 23};

第二次交换 temp[1] (67) > temp[2] (12), 交换 =>{35, 12, 67, 45, 6, 56, 23};

第三次交换 temp[2] (67) > temp[3] (45), 交换 =>{35, 12, 45, 67, 6, 56, 23};

第四次交换 temp[3] (67) > temp[4] (6), 交换 =>{35, 12, 45, 6, 67, 56, 23};

第五次交换 temp[4] (67) > temp[5] (56), 交换 =>{35, 12, 45, 6, 56, 67, 23};

第六次交换 temp[5] (67) > temp[6] (23), 交换 =>{35, 12, 45, 6, 56, 23, 67};

第二轮

比较temp[0] 与temp[1], temp[1]与temp[2],以此类推

第七次交换 temp[0] (35) > temp[1] (12), 交换 =>{12, 35, 45, 6, 56, 23, 67};

第八次交换 temp[2] (45) > temp[3] (6), 交换 =>{12, 35, 6, 45, 56, 23, 67};

第九次交换 temp[4] (56) > temp[5] (23), 交换 =>{12, 35, 6, 45, 23, 56, 67};

第三轮

第十次交换 temp[1] (35) > temp[2] (6), 交换 =>{12, 6, 35, 45, 23, 56, 67};

第十一次交换 temp[3] (45) > temp[4] (23), 交换 =>{12, 6, 35, 23, 45, 56, 67};

.....

冒泡排序的改进

我们发现，在比的过程中不需要再和后面的n个数去比较，因为后面的n个数必然是前几个最大的数（上面的过程可以得知），所以为了不做无用功，所以放弃和最后几个数比较。

```
#include<stdio.h>
void printArr(int* arr);
void bubble_original();
void bubble_update();
int main(){
    bubble_original();
    bubble_update();
    return 0;
}

void printArr(int* arr){
    for(int i = 0;i < 7; i++)
        printf("%d ", *(arr + i));
    printf("\n");
}

void bubble_original(){
    int temp[] = {67, 35, 12, 45, 6, 56, 23};
    printf("initial array: \n");
    printArr(&temp);
    int count = 0, compare = 0;
    for(int i = 0;i < 7 - 1;i++)
```

```
        for(int j = 0;j < 7 - 1;j++){
            compare++;
            if(temp[j] > temp[j+1]){
                count++;
                int temp1 = temp[j];
                temp[j] = temp[j+1];
                temp[j+1] = temp1;
            }
        }
    printf("final result: \n");
    printArr(&temp);
    printf("original bubble compares %d times, exchanges %d times.\n", compare,
count);
}

void bubble_update(){
    int temp[] = {67, 35, 12, 45, 6, 56, 23};
    printf("initial array: \n");
    printArr(&temp);
    int count = 0, compare = 0;
    for(int i = 0;i < 7 - 1;i++)
        for(int j = 0;j < 7 - 1 - i;j++){
            compare++;
            if(temp[j] > temp[j+1]){
                count++;
                int temp1 = temp[j];
                temp[j] = temp[j+1];
                temp[j+1] = temp1;
            }
        }
    printf("final result: \n");
    printArr(&temp);
    printf("updated bubble compares %d times, exchanges %d times.\n", compare,
count);
}
```

二维数组

二维数组乃至多维数组的本质就是数组的嵌套。在一维数组中，数组中元素的类型为int，char等基本数据类型，但是二维数组中的元素为一维的数组。故而在此可以理解为什么二维数组的下标是先行后列。一个二维数组可以写成以下形式：[[1, 2, 3], [4, 5, 6], [7, 8, 9]]。当我们想去访问元素4的时候，我们可以先找到他归属于哪个一维数组：其归属于第2个一维数组（表示为矩阵时是第二行），那么可以先访问到第二行的数组arr[1]， 在此基础上，4是一维数组里面的第一个元素，那么可以用arr[1] [0]访问到元素4.

就上面的描述，如果我想要遍历一个二维数组，那么我首先得遍历一维数组，然后遍历一维数组里面的每一个值，所以二维数组的遍历需要用循环的嵌套进行。

数组的名字

数组的名字是数组的首地址。

```c
#include<stdio.h>
int main(){
    int temp[] = {1, 2, 3, 4, 5};
    printf("The first element in the array is %d\n", temp[0]);
    printf("The address of the first element in the array is %p\n", temp);
    printf("The first element in the array is %d\n", *temp);
    return 0;
}
```

C语言中的指针

在开始讲解指针与地址之前，将先介绍计算机中的数据是如何存储的。

计算机中的数据存储

冯诺依曼体系的计算机将计算机分为了五个部分：CPU，Input，Output，内存，硬盘

一个程序是如何运行的

1. 编码过程中以及打开编码文件时，计算机将代码文件从硬盘中读取出来，放入到内存中
2. 对文件进行编译时，CPU从内存中逐行读取编译指令，并将编译形成的文件写入到内存中。
3. 执行文件时，计算机将根据程序设定读取输入，打印输出，声明变量等等。

在声明变量过程中，计算机会在内存中分配对应大小的空间用于存放对应的数据。（int类型4个字节，double类型8个字节等）

什么是地址

计算机把需要用到的变量初始化到内存中，所以需要一个索引告诉计算机，应该到内存中的什么位置才能找到他声明的变量，这个索引就是地址。

一般来说，在C语言中内存的地址采用7位无符号十六进制数来表述，最大表示的范围位16^7。这一段代码仅对计算机有意义（计算机通过解码找到数据的具体位置，人无法直接理解）。

两个操作符：& 与 *

在对地址进行操作的时候，一般会涉及到两个操作符，即&与*，这两个操作符表示了一种互为逆操作的关系。

&表示对后方接的变量（一般是基本的数据类型，诸如int，double一类）进行取地址的操作，我们可以用一个对应变量类型的指针承接其返回值。（记得scanf里面的&吗？scanf需要读取变量的地址用于直接写入变量而略去了根据变量名查找对应空间的过程）。

*表示对后方接的变量（一般是指针类型）进行解引用的操作。用*可以通过后方接的地址在内存中找到这个地址的内存，从而读取到其中的数据。

```c
#include<stdio.h>
int main(){
    int num = 10;
    int *p;
    p = &num;

    printf("The address of the number is %p\n", p);
    printf("The value of the number is %d\n", *p);
    return 0;
}
```

指针

指针存储的是一个十六进制的地址，这个地址指向了内存中存储对应数据类型的区域，所以我们可以通过对一个变量采用取地址的操作（&variable_name）获得指向一个变量的指针，同时可以通过对一个地址进行解引用的操作（*variable_name）获取到一个指针指向的变量存储的数据。

指针的声明

大部分人（包括我）对指针感觉到困惑问题就出在对指针的声明上，所以今天我们换一种视角来看指针的声明。

```c
#include<stdio.h>
int main(){
    int num = 10;
    int* p = num;
    return 0;
}
```

在刚开始接触指针的时候，老师会说，指针也是一种数据类型，他就像int，double一样，他有一个用于声明的关键字，是int*或者char*。但是这个说法并不是很有助于理解的。

我们之前已经说过，指针本质上存储的是一个十六进制的地址，所以在内存中我们要存一个指针，我们需要去开辟一块内存空间并且以整数的方式去存储它。所以本质上来说，指针应该属于一个整数型的变量。那为什么我们在声明一个指针的时候会需要一个*来声明？我们把*拆开来看。

```c
#include<stdio.h>
int main(){
    int num = 10;
    int *p;
    p = &num;
    printf("The value stored in %p is %d\n", p, *p);
    return 0;
}
```

我们把*p看成一个整体，那我们在执行int *p的时候，其实就是在声明一个名为*p的int类型的变量。我们之前讲过，*是解引用的意思，那么我对一个数据类型解引用之后得到了一个int类型的数据，那么这个数据类型就只能是一个地址，也就是我们所说的指针。所以在声明的时候，为什么要带上*才能声明指针就更好理解了：我解引用之后才能使一个int或者一个char，所以我声明的变量应该就是一个指针。

指针有什么用：值传递与地址传递

如果要求在一个函数中交换两个数的值然后在主函数输出，你会怎么做?

```
#include<stdio.h>
void swap(int a, int b);
int main(){
    int a = 10, b = 20;
    printf("a: %d, b: %d", a, b);
    swap(a, b);
     printf("a: %d, b: %d", a, b);
    return 0;
}

void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}
```

在这个函数下，a，b的值并没有发生交换，为什么?

在对函数进行调用时，参数传递使用的是值传递的方式。即我传入的并不是a，b两个容器，而是我把容器里面的数取出来复制了一份，然后传入到函数里面交换。那么传入到函数的时候，按照作用域的规则，在swap函数定义的a，b作用域就只有swap函数，当函数执行结束，a，b使用的内存空间被系统释放掉，那交换就是毫无意义的。

如果我们能直接修改在main函数中声明的a，b呢——地址传递。

```
#include<stdio.h>
void swap(int *a, int *b);
int main(){
    int a = 10, b = 20;
    int *p2a = &a, *p2b = &b;
    printf("a: %d, b: %d\n", a, b);
    swap(p2a, p2b);
     printf("a: %d, b: %d\n", a, b);
    return 0;
}

void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

我们把main中定义的a和b的地址传递给了函数swap，在swap中，通过对a，b的地址进行解引用然后交换他们的值，那么这就不是交换两个即将被释放的值（因为a，b在main中定义的，main仍未结束，内存不会被释放），所以交换是有效的。

通过以上的例子可以得出一个结论，当需要修改多个变量时，需要通过地址传递而不是值传递的方式进行。同时在传递过程中对于不需要修改的内容的指针，一般建议传递一个const类型的指针以防止外界对重要变量类型的修改。

多重指针

call back：数组的名字就是数组的首地址

理解这句话需要先明确一点，在C语言中，数组是存储在一段连续的内存上的。（这一点在Java，python一类的语言上不一定）

```
#include<stdio.h>
int main(){
    int arr[5];
    for(int i = 0;i < 5;i++)
        printf("%p\n", &arr[i]);
    return 0;
}
```

对于一个int数组来说，一个int占据4个字节的大小，这是固定的。所以我只需要知道一个数组的首地址在什么位置以及数组的长度，我就能访问到数组的所有元素。

```
#include<stdio.h>
int main(){
    int arr[] = {1, 2, 3, 4, 5};
    int *p = arr;
    for(int i = 0;i < 5;i++)
        printf("%d\n", *(p + i));
    return 0;
}
```

在这里需要注意一点，尽管我们知道他地址之间会相差4个字节，但是C语言为了简便操作同时也为了防止误操作，在进行遍历的时候会默认加上对应长度*i的长度（如上面加上的是4*i）。这一点对于不同的数据类型都是一样的。

```
#include<stdio.h>
int main(){
    double arr[] = {1, 2, 3, 4, 5};
    double *p = arr;
    for(int i = 0;i < 5;i++)
        printf("the address of %lf is %p\n", *(p + i), &arr[i]);
    return 0;
}
```

call back：二维数组与指针数组

此前说过，二维数组本质上是一维数组作为元素的一维数组，而根据数组是在内存空间上是连续分布的，我们可以使用二维数组的首地址以及单层循环遍历整个二维数组。

```c
#include<stdio.h>
int main(){
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}};
    for(int i = 0;i < 2;i++){
        printf("The address of the first element in arr[%d] is %p\n", i,
&arr[i]);
        for(int j = 0;j < 3;j++){
            printf("the address of number %d is %p\n", arr[i][j], &arr[i][j]);
        }
    }
    return 0;
}
```

```c
#include<stdio.h>
int main(){
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}};
    int *p = arr;
    for(int i = 0;i < 6;i++){
        printf("%d\n", *(p + i));
    }
    return 0;
}
```

对于数组而言，其也可以存储多个指针作为指针数组。

C语言中的结构体

结构体的主要用处为同时存储多个变量作为一个对象的属性。例如当我想存储一个人的信息时，里面包括姓名，年龄，电话等等属性，但是在调用时如果过于零散，不利于调用的进行，所以有了结构体的概念。

结构体的声明与访问

```c
#include<stdio.h>
#include<string.h>
typedef struct Student{
    char name[64];
    char studentID[64];
    int age;
}Student;
int main(){
    Student s;
    strcpy(s.name,"pancake");
    s.age = 8;
    strcpy(s.studentID,"2019010101");
    printf("%s %s %d\n", s.name, s.studentID, s.age);
    return 0;
}
```

可以使用struct 关键字声明一个结构体。同时结构体的声明一般采用宏定义的方式进行。在进行结构体定义时可以使用typedef关键字重命名定义的结构体。同时可以通过.访问结构体中的属性。

结构体数组

与基本的变量类型类似，结构体也可以形成一个结构体数组用来存储多个结构体。

```
#include<stdio.h>
#include<string.h>
typedef struct Student{
    char name[64];
    char studentID[64];
    int age;
}Student;
int main(){
    Student s[5];
    char name1[] = "pancake";
    char ID[] = "2019";
    for(int i = 0;i < 5;i++){
        strcpy(s[i].name, name1);
        strcpy(s[i].studentID, ID);
        s[i].age = 8;
    }
    for(int i = 0;i < 5;i++)
       printf("%s %s %d\n", s[i].name, s[i].studentID, s[i].age);
    return 0;
}
```

对于结构体数组，同样可以采用循环遍历的方式访问其中某一个特定的结构体或者通过特定的下标访问到对应的结构体。

但受限于结构体数组长度固定以及内存的连续分配，在存储大量的结构体时，结构体数组不会作为主要的存储方式。（可能不会有这么大的连续内存），故而我们引入了链表的概念。

C语言中的链表

与数组不同的是，链表在内存分配策略上采用了不连续内存分配的机制，所以如果可以在内存中找到相应大小的地方，我就可以声明一个结构体用来存储对象。

链表的结构

链表的基础单元为结构体。由于内存上的不连续，所以无法通过索引与首地址寻找到对应位置的数据，所以在此引入了指针。

一个简单的链表节点

```
#include<stdio.h>
typedef struct Node{
    int data;
    struct Node *next;
}Node;
```

链表的一个节点中包含了需要存储的数据data（可以是任意类型的，包括结构体类型）以及指向下一个节点存储位置的指针。所以当我们维护一个链表时，我们需要只需要保存一个头节点（头节点中包括了指向下一个节点指针）。同时为了方便操作，一般头结点的data不存储内容，其主要任务是存储指向第一个有内容节点的指针。（这样设计可以在访问时更为方便）

链表的初始化

由于我们需要知道保存有第一个节点，同时后面有节点进入时我们需要知道新建的节点的地址，但是在C语言中直接声明变量时是无法得到这个变量的地址的，所以我们需要自己维护内存中的地址——malloc函数.

```
#include<stdio.h>
typedef struct Node{
    int data;
    struct Node *next;
}Node;

int main(){
    Node *head = (Node*)malloc(sizeof(Node));
    free(head);
}
```

malloc函数用于向系统申请一片固定大小的内存，所以需要接受申请的内存大小，并且返回在申请的内存空间的地址。但是由于系统也不知道申请的空间主要用于存储什么，所以其只能返回一个指向空数据类型的指针（void*），所以我们需要强制转换成我们需要的结构体的指针。对于此后每一个新进入的节点，我们都需要使用malloc申请一片空间并且获取对应的指针。

需要特别强调的时，由于我们在这里直接操作了电脑的内存，系统并不知道你什么时候不需要这块内存了，所以不会在程序结束后自动释放这片空间。所以需要人为在使用结束后使用free函数释放这块内存空间。如果不释放并且长期运行，可能会造成系统内存的溢出。

链表元素的访问

由于链表是不连续的，所以无法利用index直接访问到链表中的元素。所以无论是访问还是遍历，都需要从头结点开始才能找到对应的元素。

```

#include<stdio.h>
typedef struct Node{
    int data;
    struct Node *next;
}Node;
int add_front(Node* head, int num);
int tranves(Node* head);
int search(Node* head, int num);
int main(){
    Node *head = (Node*)malloc(sizeof(Node));
        head->next = NULL;
    int arr[] = {1, 2, 3, 4, 5};
    printf("add_front\n");
    for(int i = 0;i < 5;i++)
        add_front(head, arr[i]);
    tranves(head);
    free(head);
}

int tranves(Node* head){
    Node *temp = head;
    while(temp->next != NULL){
```

```
        temp = temp->next;
        printf("%d ", temp->data);
    }
    printf("\n");
}

int add_front(Node* head, int num){
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = num;
    temp->next = head->next;
    head->next = temp;
    return 0;
}

int search(Node* head, int num){
    Node *current = head->next, *pre = head;
    while(current != NULL){
        if(num == current->data){
            return current->data;
        }
        pre = pre->next;
        current = current->next;
    }
    return 1;
}
```

链表的增删改查

```
#include<stdio.h>
#include<stdlib.h>
typedef struct Node{
    int data;
    struct Node *next;
}Node;
int add_end(Node* head, int num);
int add_front(Node* head, int num);
int remove1(Node* head, int num);
int search(Node* head, int num);
int modify(Node* head, int num1, int num2);
int tranves(Node* head);
int main(){
    Node *head = (Node*)malloc(sizeof(Node));
        head->next = NULL;
    int arr[] = {1, 2, 3, 4, 5};
    printf("add_front\n");
    for(int i = 0;i < 5;i++)
        add_front(head, arr[i]);
    tranves(head);
    printf("add_end\n");
    for(int i = 0;i < 5;i++)
        add_end(head, arr[i]+6);
    tranves(head);
    printf("modify\n");
    modify(head, 1, 31);
    tranves(head);
```

```
    printf("delete\n");
    remove1(head, 4);
    tranves(head);
    printf("search\n");
    printf("%d\n", search(head, 5));
    free(head);
    return 0;
}

int tranvers(Node* head){
    Node *temp = head;//声明一个指向Node节点的指针，用于复制指向头结点的指针
    while(temp->next != NULL){//判断当前的节点是否有后续节点
        temp = temp->next;//由于头结点为空，所以可以直接跳过第一个节点，访问第二个节点（第一个数据节点）
        printf("%d ", temp->data);//输出访问的结果（或者其他的处理）
    }
    printf("\n");
}

int add_front(Node* head, int num){
    Node* temp = (Node*)malloc(sizeof(Node));//申请一块内存空间用于存放需要插入的节点
    temp->data = num;//对新节点进行赋值（将要存的数据放到节点中）
    temp->next = head->next;//将原来的第一个数据节点接到新插入的节点后面
    head->next = temp;//将头结点的next指针指向新插入的节点，新插入的节点成为第一个数据节点
    return 0;
}

int add_end(Node* head, int num){
    Node *temp = head;//复制指向头结点的指针以防丢失
    Node* temp1 = (Node*)malloc(sizeof(Node));//申请一块内存空间用于存放需要插入的节点
    //对新节点进行赋值（将要存的数据放到节点中）
    temp1->data = num;
    temp1->next = NULL;  // 确保新节点的next为NULL（部分情况下可能由于初始化错误导致默认值为野指针）

    //透过遍历找到最后一个数据节点
    while(temp->next != NULL){
        temp = temp->next;
    }

    //将最后一个数据节点的next指针指向需要插入的节点
    temp->next = temp1;
    return 0;
}

int remove1(Node* head, int num){
    Node *current = head->next, *pre = head;//声明需要维护的两个指针（当前指针与前一指针）
    while(current != NULL){//当前指针不为空时，其前置指针必定不为空
        if(num == current->data){//判断当前指针指向的节点是否为要删除的节点
            pre->next = current->next;//前一节点的next直接指向当前节点的next，完成这一节点在链表中的删除
            free(current);//由于C语言直接操作内存，所以需要手动释放申请的内存空间，否则内存空间将被占据
```

```
            return 0;//删除完成后直接退出，但是如果需要删除链表中的多个满足条件的值时，应该删除这一行代码
        }
        //如果当前节点不是要删除的节点，继续遍历直到找到节点或者找不到退出
        pre = pre->next;//理论上这一行代码也可以写作 pre = current;但是在逻辑上不是很好理解
        current = current->next;
    }
    return 1;
}

//修改与查找的逻辑与删除实际上是类似的，都是在链表中找到对应的目标节点（然后进行删除，修改或者返回找到的值），所以在代码注释中不再赘述，仅对一些细节性内容进行讲解
int modify(Node *head, int num1, int num2){
    Node *temp = head;
    while(temp->next != NULL){
        if(num1 == temp->data){
            temp->data = num2;
            return 0;//在C语言中，我们一般默认返回值为0为程序正常结束运行，如果返回非0则程序是异常推出的
        }
        temp = temp->next;
    }
    return 1;
}

int search(Node* head, int num){
    Node *current = head->next;
    while(current != NULL){
        if(num == current->data){
            return current->data;
        }
        current = current->next;
    }
    return 1;
}
```

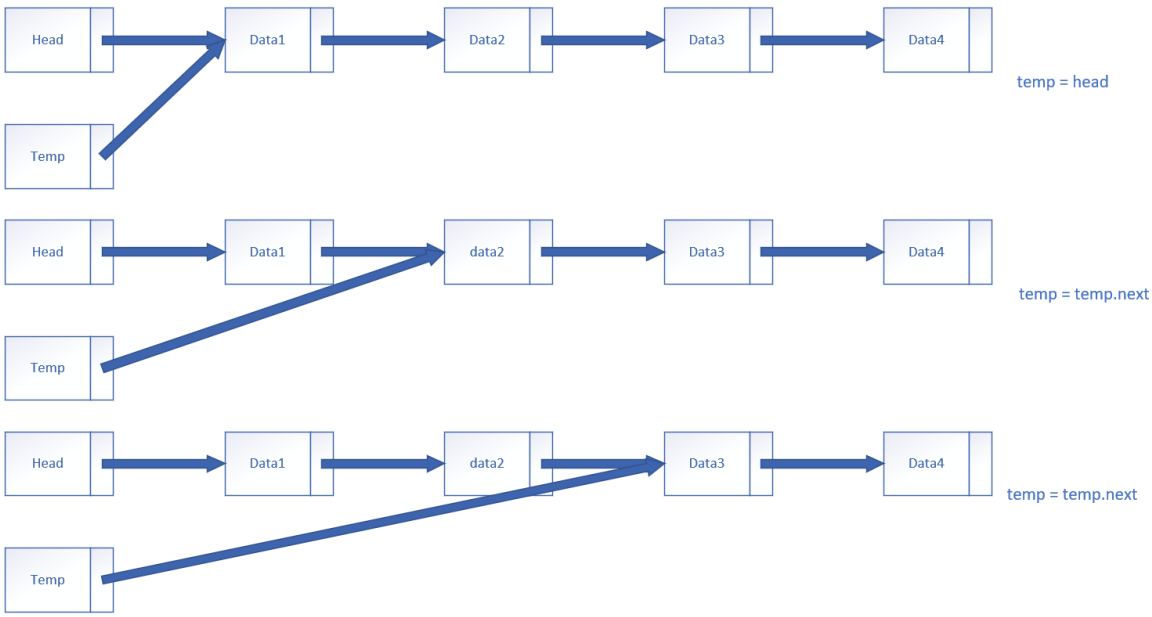
一些便捷操作

理论上链表是若干个结构体通过指针进行连接，每个节点中都会附带一定的数据。但是为了便于操作，在没有特别要求的情况下，我们一般会生命一个数据部分为空，仅为了保存指向第一个数据的空头结点。这样的设计可以在遍历时便于操作。所以建议大家在使用链表时采用空头结点链表而非非空头结点链表。（本讲义中涉及到的链表在没有特殊说明的情况下默认为空头结点链表）

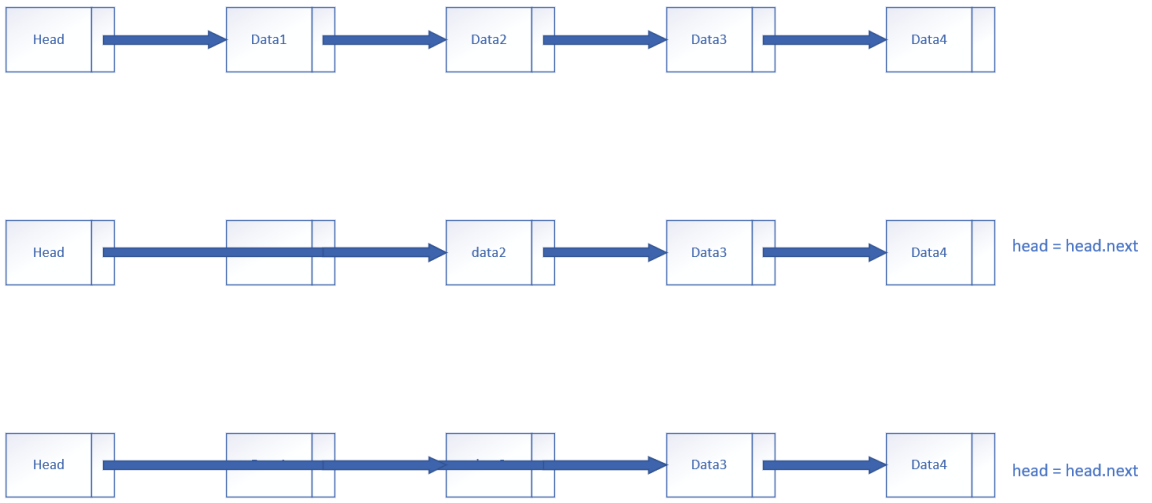
遍历

遍历过程即为通过头结点完成对整个链表的访问。由于指向头结点的指针是唯一的（即如果head被覆盖后无法继续读取到链表的后续接节点），所以在访问时应当先将指向头结点的指针（在这里可以理解成head这一变量名）复制一份，这样就可以避免随着head指针的迭代导致丢失了后方的节点。

Traverse

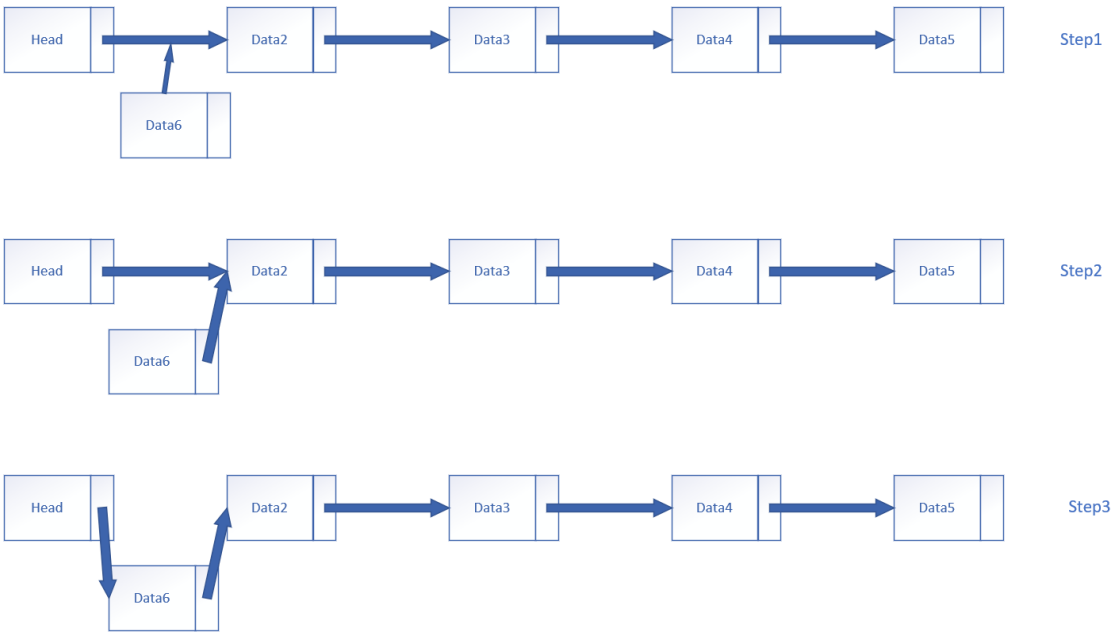


Traverse(error)



增

头插法

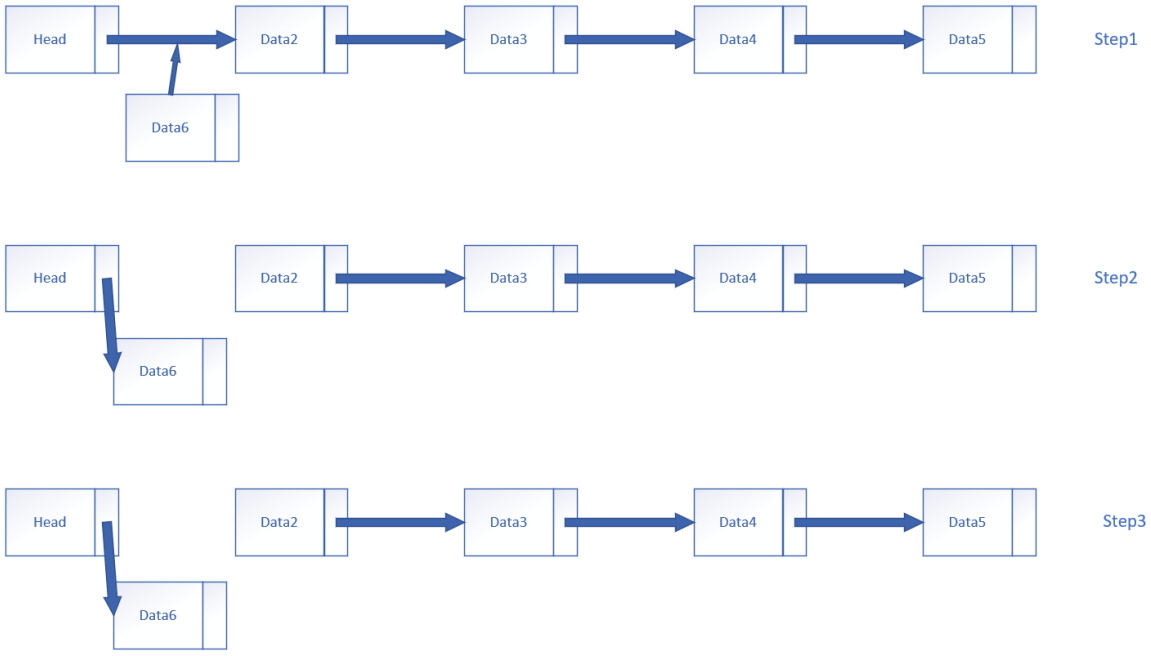


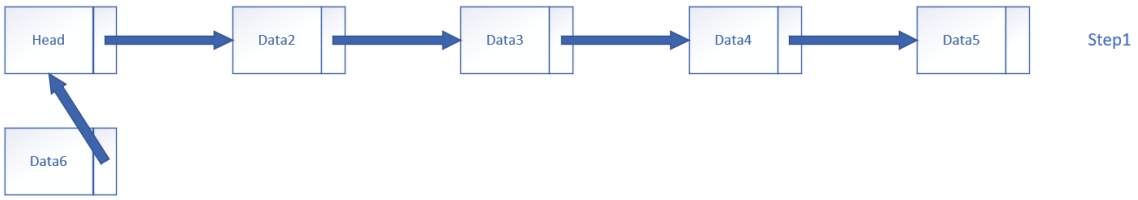
头插法

头插法的意思是将新插入的节点作为链表的第一个数据节点（空头结点链表中为头结点指向的节点，非空节点链表中为头结点）。以上展现的是对于空头结点链表的头插法。

- 空头结点链表：要讲一个新节点作为头结点的下一节点，并且保持链表的完整性，需要现将原来头结点指向的下一个节点复制到新的节点上（step2），这样在完成插入之后仍然介意访问到原来的链表，然后让头结点的next指向要插入的节点。**注意：这两个操作不能颠倒，如果颠倒则丢失了指向第一个数据节点的指针，无法访问原来的链表**
- 非空头结点链表：非空头结点链表的头插法可以直接将新节点的next指向原来的头结点，同时将新节点赋值给head作为新的头结点。

头插法（error）





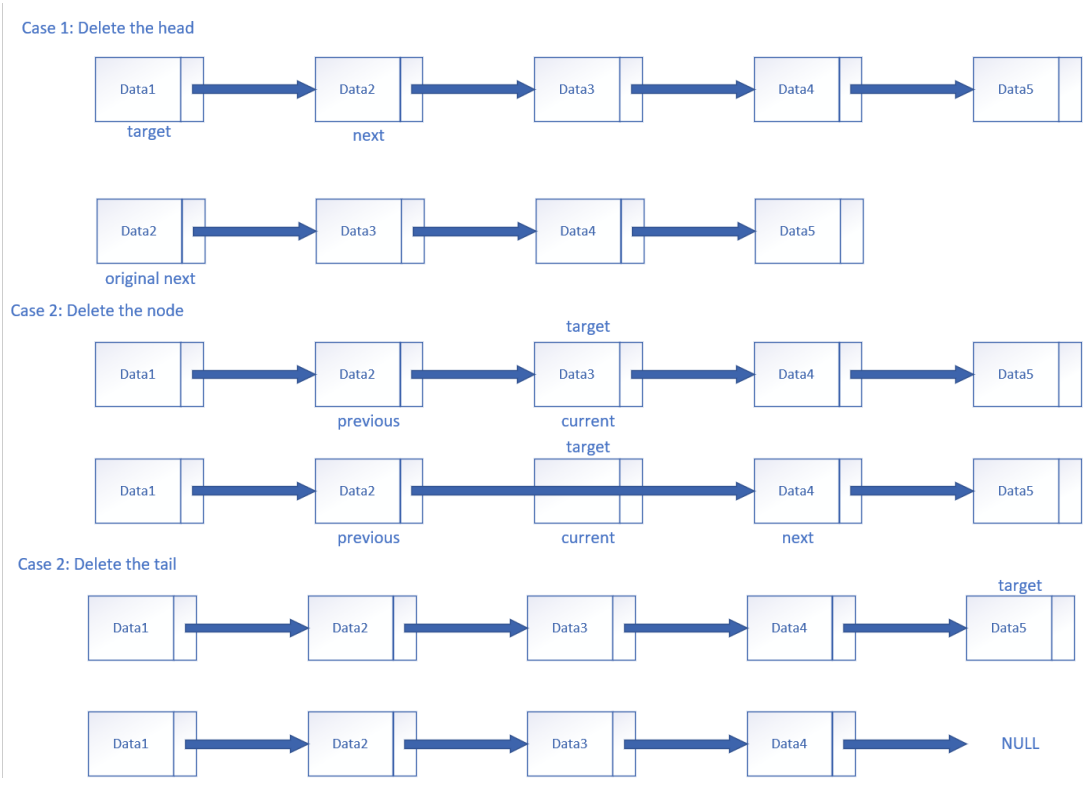
尾插法

尾插法则无所谓空头结点链表与非空节点链表，直接通过遍历找到最后一个节点，将最后一个节点的next指向需要插入的新节点即可。

删

删除需要分三种情况进行讨论，分别为删除第一个数据节点，删除链表中间的数据节点以及链表尾部的数据节点。但是对于空头结点链表而言，其第一个数据节点必然是第二个链表节点，所以其删除第一个数据节点的操作与删除链表中间节点相似，所以在此作为一种情况讨论。

由于链表的基本结构为通过指针链接一系列的结构体，所以链表中的删除操作可以简化为在链表中将指向要删除节点的指针移动到指向要删除节点的后一节点，所以在删除操作中我们需要同时维护两个指针——指向当前节点的指针与指向前置节点的指针。



以上即为删除链表节点的三种情况。

case 1：

这种情况是针对非空头结点链表而言的，即我要删除头结点中的数据，但是我们仍需要保留一个节点作为头结点（否则丢失了后面所有的节点），所以我们将头结点后面的节点作为新的头结点。即将第二个节点的地址赋值给head，然后free原来的头结点。值得注意的是，我们需要备份原来的头结点用于free操作——在赋值完成后head存储的是新的头结点的地址，此时free（head）将直接释放新头结点的空间。同时两个操作不能对调，因为先进行free操作后head.next此时为NULL，无法寻找到原来的链表。

case2:

这种情况同时包括了删除链表的中间节点（无论空头结点链表还是非空头结点链表）以及删除空头结点链表的第一个数据节点。删除空头结点链表的第一个数据节点的情况较为简单，由于已知需要删除的节点的前置节点（头结点），所以可以将指向当前节点的next直接指向当前节点的下一节点，而后free掉当前节点即可完成操作。对于其余两种情况，需要先通过遍历寻找到需要进行删除的节点，而后进行上述的操作。在该过程中同样需要同时维护当前节点指针与前置节点指针，而后才能进行同样的操作。

case 3：

这种情况较为简单，由于已知需要删除的节点位于链表的末尾，所以可以通过遍历寻找到链表的最后一个数据节点，并且需要同时维护当前节点指针（此时应该指向节点的最后的一个数据节点）与前置节点指针。在完成遍历并找到要删除的节点后，直接将指向要删除节点的指针置为NULL，并且free掉需要删除的节点占据的内存空间。

搜索&修改

搜索与修改本质上是对链表进行遍历，逐个比较当前节点是否为需要检索的节点，如果是则进行修改或者返回需要检索的节点等等操作。由于这两项操作主要基于遍历进行，这一部分已经在前面进行了详细的讲解，故而在此部分不再赘述。

双端链表与环形链表

双端列表即维护链表的头尾节点，可以同时使用头插法与尾插法。而环形链表则是维护两个相邻的链表节点，同时链表的尾节点的next指向头结点构成一个环形的链表结构。其增删改查的操作与单向链表差异不大，故而在此不进行详细的介绍。

C语言中的字符串

由于C语言中不存在string类型的变量，所以需要表示字符串时，需要使用char数组来存储字符串。

字符串的声明，初始化以及输出

```
#include<stdio.h>
int main(){
    char temp[] = "This is a test string.";
    char temp1[6];
    temp1[0] = 'H';
    temp1[1] = 'e';
    temp1[2] = 'l';
    temp1[3] = 'l';
    temp1[4] = 'o';
    printf("%s\n", temp1);
    temp1[5] = '\0';
    printf("%s\n", temp1);
    return 0;
```

```
}
```

对于一个char数组可以有三种不同的初始化方式。

1. 在声明的同时进行初始化。此时不需要填写中括号内部的内容。
2. 声明与初始化分离进行。但是生命结束后不允许进行整体的赋值，只能对单个元素逐一赋值。
3. 声明与初始化分离进行，调用strcpy函数对数组进行赋值。

在这里需要强调一点，在char数组声明时如果填写了元素的个数，那么需要比字符串长度大1。这是因为字符串需要以'\0'结尾表示字符串的结束。如声明"hello"时，char数组的长度应为6。如果按照字符串长度声明数组，可能在读取时会出现越界以及野指针的问题。

在IO中，C语言提供的printf函数可以接受%s作为转义字符，直接传入char数组函数名即可进行解码并输入对应的字符串。但是scanf在读取输入时会存在一个问题：scanf在读取过程中遇到空白字符自动终止（空白字符可能包括空格，换行符\n，制表符\t等等）。

```c
#include<stdio.h>
int main(){
    char temp[100];
    scanf("%s", temp);//此处不用取地址是因为数组名即为数组的首地址
    printf("The text you input is %s", temp);
    return 0;
}
```

（以上的代码在输入"Hello World!"与"HelloWorld!"时会出现不同的结果）

在需要读取带空格的字符串时，一般使用gets函数。

```c
#include<stdio.h>
int main(){
    char temp[100];
    gets(temp);
    printf("The text you input is %s", temp);
    return 0;
}
```

string.h的使用

由于有对字符串进行整体操作的需求，C语言提供了string.h库用于处理字符串的相关操作。但是在实际使用中比较常用的一般是以下几个。

int strcmp（char*， char*)

strcmp函数主要用于传入的两个字符串是否相同（值是否相同，在没有指针的语言中会涉及到地址是否相同的比较）。需要注意的是，strcmp函数的处理逻辑是从左到右逐个比较字符的ASCII码值。

```c
#include<stdio.h>
#include<string.h>
int main(){
    char str1[] = "hello";
    char str2[] = "hella";
    char str3[] = "hfho";
    printf("%d\n", strcmp(str1, str2));
    printf("%d\n", strcmp(str1, str3));
    return 0;
}
```

在比较str1与str2时，系统会从左至右比较字符的ASCII值，'h'相同，下一个，'e'相同，下一个.....指到比较到'o'时，'o' > 'a',所以strcmp返回值是一个正数。

在比较str1与str3时， 系统同样从左到右比较ASCII值，'h'相同，下一个，'e' < 'f'，所以strcmp的返回值是一个负数。

所以如果两个字符串时一样的，则strcmp会返回0。

void strcpy（char*, char*)

strcpy函数主要用于复制一个字符串中的内容，这个复制为值传递的过程，系统重新开辟一块内存空间用于存储对应的数据。

```c
#include<stdio.h>
#include<string.h>
int main(){
    char str1[] = "hello";
    char str2[64];
    strcpy(str2, str1);
    printf("fuck\n");
    printf("The address of str1 is %p\n", str1);
    printf("The address of str2 is %p\n", str2);
    return 0;
}
```

这个函数的调用中，第一个参数为字符串的目的地，后一个为原字符串。需要注意的是，此处传入*str2将不会有反应。（声明指针时并没有分配对应的内存空间，所以相当于对一个没有分配内存空间的变量赋值，这一操作不会报错，但是无法正常执行）

void strcat（char*, char*)

strcat函数主要用于拼接两个字符串。在操作逻辑中，该函数会首先查询原来的数组后面有没有位置可以继续使用，如果有则保持原来的首地址，否则重新申请内存空间。

```c
#include<stdio.h>
#include<string.h>
int main(){
    char str1[] = "hello";
    char str2[] = " world!";
    printf("The address of str1 is %p\n", str1);
    strcat(str1, str2);
    printf("The address of str1 is %p\n", str1);
    return 0;
}
```

int strlen（char*）

strlen函数主要用于获取字符串的长度。这一函数与sizeof的区别在于，strlen只计算有效字符的数量（从str[0]开始直到'\0'的位置，不含'\0'），但是sizeof计算整个数组的大小，无关其中的有效字符。

```c
#include<stdio.h>
#include<string.h>
int main(){
    char str1[64];
    strcpy(str1, "Hello");
    printf("The length of str1 is %d\n", strlen(str1));
    printf("The size of str1 is %d\n", sizeof(str1));
    return 0;
}
```

string中的指针

由于string本身是一个char数组，所以也可以使用指针访问。同时由于char类型只占据一个字节的大小，所以在遍历的时候可以这样遍历。

```c
#include<stdio.h>
#include<string.h>
int main(){
    char str1[] = "Hello";
    char *p = str1;
    for(int i = 0;i < strlen(str1);i++)
        printf("%c\n", *(p + i));
    return 0;
}
```

同时由于数组名字就是数组首地址的特性，在给定一个字符串时，可以用下面的方式截取字符串中的一部分。

```c
#include<stdio.h>
#include<string.h>
int main(){
    char str1[] = "Hello World";
    char *p = str1;
    for(int i = 0;i < strlen(str1)/2;i++)
        printf("%c\n", *(p + (2 * i)));
    printf("%s\n", (6+p));
    return 0;
}
```

C语言中的文件

我们需要文件为程序的运行提供数据，同时在程序运行中产生的数据以及更新的数据也需要写入到文件并且保存到硬盘中。所以这部分讨论的内容即为文件的读写。在开始具体讲解之前，有一些些前情提要。

文件的类型

C语言将文件分成了两个类：

- 文本类：只包含纯文本文件，主要是txt文件，doc文件（只包含文字的doc）json（java数据传输的中间文件）， html（网页前端代码文件）一类的文件
- 二进制类：可以理解成非文本类文件，图片，音频，视频都包括在这一部分中。

需要指出的是，pdf文件属于二进制类文件。因为pdf本质上是一张无法修改的图片，无法对其中的文字进行修改。

文件的路径

目前大部分同学使用的主要是Windows系统，所以此处的文件路径主要针对Windows系统。（Linux系统与MacOS的文件操作方式与Windows不同）。

文件的路径主要分为两种：相对路径与绝对路径。

绝对路径

在Windows中，任何文件，任何文件夹都属于一个盘（C盘，D盘一类），所以我们可以通过"D:/dirA/dirB/hello.txt"找到在D盘dirA文件夹下的dirB文件夹下的hello.txt文件。这个操作无论是我在E盘还是C盘，只要是在这台电脑的硬盘里面操作，我都可以用这个地址找到hello.txt文件（只要他存在）。这个路径不会和你在那个文件夹里面操作相关联，所以叫做绝对路径。

相对路径

与绝对路径相反，一个文件的相对路径会和当前的工作目录相关。先介绍两个目录的表示：.与..。

.

表示当前的工作目录。当我现在在dirA文件夹里面的时候，.表示的就是文件夹dirA；在dirB文件夹里面的时候，.表示的就是文件夹dirB

..

表示上一级工作目录。当我现在在dirA文件夹里面的时候，..表示的就是D:;在dirB文件夹里面的时候，..表示的就是dirA

所以当我在dirA文件夹里面的时候，此时我想找dirB下面的hello.txt,那此时hello.txt相对于dirA文件夹的路径就是(./)dirB/hello.txt;当我在dirB文件夹里面的时候，此时我想找hello.txt,那此时hello.txt相对于dirB文件夹的路径就是(./)hello.txt;

在使用时，一般打开一个固定位置的文件（比方说某些放在C盘下面的固定的系统文件），推荐使用绝对路径。但是在提交项目作业的时候，我们的工作目录一般是代码文件所在的文件夹，那么此时应当使用相对路径（因为你也不确定老师会把你的文件放在D盘还是E盘运行）

文件路径的写法

盘符和文件夹之间，文件夹和文件名之间会用/或者\隔开（只有windows允许使用\作为分隔符），但是在C语言中我们更推荐使用/作为文件路径的分隔符（因为在字符串处理的时候，\需要作为转义字符或者占位符出现，为了防止系统误解，使用时必须使用\\表示）

打开文件

文件有三种不同的打开方式

- 只读：打开之后只能从文件里面读取数据，不能向其中写入数据。如果打开的文件不存在，返回空的文件指针。这使用"r"或者"rb"表示（需要传入一个字符串）
- 只写：打开之后只能向文件里面写数据，不能读取数据。这里需要注意，如果打开的文件之前已经存在，那么打开的时候会清空原来文件里面所有的内容，然后写入数据。如果文件不存在，那么会创建一个新的文件用于写入数据。这使用"w"或者"wb"表示
- 读写：既可以对打开的文件进行读数据操作，也能进行写数据操作。这使用"r+"表示。同时如果是先创建然后写的方式，可以使用"w+"表示。

读文件

读文件的主要流程应该为：

1. 打开文件。
2. 从文件中读取内容，并且根据一定的格式存储成程序使用的变量
3. 关闭文件

下面我们逐步来实现

```
int loadUser(char* filePath, UNode *head){
    FILE* file = fopen(filePath, "r");
    if(NULL == file)
        return 1;
    if(NULL == head)
        return 2;
    while(!feof(file)){
        char buf[5][60];
        //char *name, char *id, char *phone, char *password
        for(int i = 0; i < sizeof(buf); i++)
            fscanf(file, "%s", buf[i]);
        if(!strcmp(buf[0], "User"))
            insertUser_list(initUser_load(buf[1], buf[2], buf[3], buf[4]),
head);
        else
```

```
            return 3;
    }
    fclose(file);
    return 0;
}
```

(这个是我偷懒从项目里面复制粘贴了一个从文件里面读取数据的函数，但是用来理解文件读取的过程已经足够)

先介绍几个比较重要的东西：

- fopen函数：获取文件的路径（可以是绝对路径或者相对路径）与打开文件的方式检查是否可容易打开文件，如果文件可以以指定的方式打开（只读文件不能以只写形式打开），则返回文件指针，如果打开失败，则返回空指针。
- FILE* 文件指针：在成功打开文件后，文件指针会放置在文件的第一个位置（指向文件的第一个字符），此后跟随读取或者写入的进行移动到对应的位置。文件指针主要用于表示读取或者写入到的位置。
- feof（）：传入一个文件的文件指针，判断文件指针是否位于文件的末尾，如果位于末尾，函数返回true，否则返回false。feof函数主要用于检查操作是否已经到达文档末尾。
- fscanf()：作用方式同scanf，但是需要传递三个参数——文件指针（标识文件读取的位置），读取的类型（一般是%s或者%d，%lf），写入的变量地址。这里有一些建议，在写入或者自行生成文档时应该保证一定的格式，每个字段之间可以用空格或者其他的间隔符间隔开（在使用scanf时可以直接读入一段不带空格字符串）。同时如果一个文档里面同时存了多种对象的信息，最好加一个校验（在User前面写"User "，读进来的时候先判断第一个是不是User，是再执行复制的操作）

  fscanf支持同时读取多个字段，如上面的读取可以写成（fscanf(file, "%s %s %s %s %s", buf[0], buf[1], buf[2], buf[3], buf[4]);）但是同时需要注意，单独的一个字段内不允许有空格的存在。

  如果需要直接读取一整行的数据，可以用fgets（variable_name）读取。
- fclose()：关闭当前的文件指针。由于文件指针可以对文件进行读写，这可能导致混乱。所以在完成读写操作后应尽快关闭文件指针。

写文件

```
int saveUser(char* filePath, UNode *head){
    FILE* file = fopen(filePath, "w");
    if(NULL == file)
        return 1;
    if(NULL == head)
        return 2;
    UNode *p = head->next;
    while(p){
        fprintf(file, "User %s %s %s %s\n", p->user.name, p->user.id, p-
>user.phone, p->user.password);
        p = p->next;
    }
    fclose(file);
    return 0;
}
```

写文件的操作与读文件类似，先打开要写的文件（如果文件不存在则会创建一个新的文件），然后向文件中写入需要写入的内容，最后关闭文件。

C语言的头文件