Final

I/O in Unix and C

Unix (and C) supports a **unified notion of I/O** known as **file descriptors**
文件描述符：在 Unix 和类 Unix 操作系统中，文件描述符是一个用于访问文件或其他输入/输出资源的抽象化概念。

File Descriptors are Small Integers 什么是文件描述符？数组中的索引——一个小整数。

Today, most **OS's grow** the array **dynamically**.

Most I/O in C Uses Streams
Streams provide a **continuous sequence of bytes** typically including some kind of **buffering**
Buffering means

- **waiting until** a certain **amount or type of data is available** before sending anything, or
- **reading extra data** in anticipation of future requests for data.

OS and Stream Both Buffer on Reads 操作系统和流都在读取时缓冲
A C Stream is a `FILE*` buffering happens in a data structure

- A **stream in C is** a pointer to that structure with **type FILE* (all caps)**. 结构本身通常不使用

默认的三个流 Three Streams by Default

stdin (descriptor 0) keyboard
stdout (descriptor 1) display (normal)
stderr (descriptor 2) display (error)
可以自建流 **FILE *my_file;***

当程序启动时，可以覆盖描述符
Normal and error output distinct reason: 您可以单独覆盖每个描述符，因此，例如，您可以运行一个程序并，将其正常输出保存到文件，但将错误输出传递给显示器，以便您注意到错误。

**程序如何打开文件

```
FILE *fopen (const char* path,  const char* mode)
//path is the file name(从文件目前的工作目录开始)
//mode 决定文件是reading, writing or both
//return a new stream for success or NULL for failure
```

"r" or "rb" read only
"w" or "wb" write only (after deleting, 如果文件不存在，会创建一个新文件，程序从文件开头写入内容。如果文件存在，则该会被截断为零长度，重新写入)
"a" or "ab" append (write only, * at end，您的程序会在已有的文件内容中追加内容)
"r+" / "r+b" / "rb+" open r / w (read/write)
"w+"/"w+b"/"wb+" truncate文件会被截断为零长度, then r/w
"a+"/"a+b"/"ab+" append r/w，读取会从文件的开头开始，写入则只能是追加模式。

**如何关闭这个文件

```
int fclose (FILE* stream);
```

```
//return 0 for success. and EOF(-1) for failure
```

从文件中读取字符

```
int fgetc (FILE* stream);//从stream指向的输入文件中读取一个字符，返回值是读取的字符，如果发生错误则返回 EOF(the int -1),返回0xFF才是byte
int getc(FILE* stream);
```

写入字符

```
int fputc (int c, FILE* stream); //函数 fputc()把参数c的字符值写入到stream所指向的输出流中。如果写入成功，它会返回写入的字符，如果发生错误，则会返回 EOF。
int putc (int c, FILE* stream);
```

从文件中读取字符串

```
char *fgets(char *s, int size, FILE *stream);//s is an array of characters
                                             //size is the size of the array
                                                          //stream is the stream from which to read
                                                                       //returns s or NULL on failure
//注意：从stream所指向的输入流中读取n-1个字符。它会把读取的字符串复制到缓冲区s，并在最后追加一个null字符来终止字符串。
//如果这个函数在读取最后一个字符之前就遇到一个换行符 '\n' 或文件的末尾 EOF，则只会返回读取到的字符，包括换行符。end of a line (ASCII 0x0A or 0x0D),end of the input (such as a file),end of array s (leaving room for a NUL)
```

向文件中写入一个string

```
int fputs (const char* s, FILE *stream);
//把字符串s写入到stream所指向的输出流中。如果写入成功，它会返回一个非负值，如果发生错误，则会返回EOF。
writing a string to stdout
int puts (const char* s);
//puts adds an end of line sequence  (linefeed, ASCII 0x0A, on Unix) to the end of the string (fputs does not).
有一点疑惑问
```

stdin标准输入（键盘），stdout标准输出（屏幕）
用scanf/printf 用于标准I/O with stdin/stdout

```
int scanf (const char* format, ...);
//reads formatted input from stdin.从键盘读取并格式化。
int printf (const char* format, ...);
//writes formatted output to stdout.发送格式化输出到屏幕。
```

用fscanf从文件流stream上读取标准化输入

```
int fscanf (FILE* stream, const char* format, [argument...]]);
//根据数据格式 const char * format, 从文件FILE* stream中 , 读取数据存储到 [argument...]参数中.
//returns number of conversions返回转换次数 or -1 on failure
```

用fprintf向文件流stream上写标准化输出

```
int fprintf (FILE* stream, const char* format, ...);
//returns number of characters printed打印的字符数 or negative number on failure
例如：fprintf (out, "%d\n", p->n_nodes);
```

二进制I/O函数的输入输出
用fread从文件流stream上读取binary输入

```
size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *stream);
//ptr指向data存储的地址
//一个element的size
//element的数量
//stream从那个文件中读
//returns number of element read or 0 on failure
```

用fwrite写入binary输出到文件流stream

```
size_t fwrite(const void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *a_file);
//向stream从那个文件中写
//returns number of element written or 0 on failure
```

人是容易编写错误处理代码，对于人类可读的文件，scanf是一项挑战。
相反，我们可以使用 fgets 将每一行读入字符串，然后
使用字符串 "I/O "来解析字符串。
◦ 失败可以用不同的方式重新解析。
◦ 失败的行可以向人类发出警告。

sscanf通常被用来解析并转换字符串
Use sscanf to Read Formatted Input from a String

```
int sscanf (const char* s, const char* format, ...);
//s is the string from which to read
//format is the format specifier
//returns number of conversions or -1 on failure
实例：
int year, month, day;

int converted = sscanf("20191103", "%04d%02d%02d", &year, &month, &day);
printf("converted=%d, year=%d, month=%d, day=%d/n", converted, year, month, day);

输出：
converted=3, year=2019, month=11, day=03
"%04d%02d%02d"是用来解析字符串的格式，%表示格式转换的开始，d表示转换为一个整数，04作为d的修饰，表示这是一个长度为4位的整数，不足4位时以0补齐。

例子返回结果等于3，表示有3个数据成功转换，转换成功数目同时取决于被解析的字符串以及其转换格式，如果我们把例子中的格式改为"%04d%02d"，那么sscanf将只返回2，day的数值不会被sscanf更改。

我们还可以实现浮点数的转换：
double longitude, latitude;
int converted = sscanf("113.123456789 31.123456789", "%lf %lf", &longitude, &latitude);
printf("converted=%d, longitude=%.9lf, latitude=%lf/n", converted, longitude, latitude);
sscanf的格式字符串中，f表示这是一个[浮点数]其修饰词l表示这是一个double的浮点数

输出：converted=2, longitude=113.123456789, latitude=31.123457
```

snprintf用于格式化输出字符串，并将结果写入到指定的缓冲区
Use snprintf to Write Formatted Output to a String

```
int snprintf (char* s, size_t size, const char* format, ...);
//s is the array to which to write
//size is the length of array s, 超过size的部分被截断
//format is the format specifier
// returns number of characters printed or negative number on failure
实例：
int main()
{
    char buffer[50];
```

formatted格式化的字符数

```
    char* s = "runoobcom";
    // 读取字符串并存储在 buffer 中
    int j = snprintf(buffer, 6, "%s\n", s);
    // 输出 buffer及字符数
    printf("string:\n%s\ncharacter count = %d\n", buffer, j);
    return 0;
}
输出结果（不会）
string:
runoo
character count = 10

单精度浮点数
int main() {
  char buffer[50];
  float x = 3.1415926;
  int len = snprintf(buffer, 50, "x = %f", x);
  printf("%s\n", buffer);
  printf("Written characters: %d\n", len);
  return 0;
}
输出：
x = 3.141593
Written characters: 12
```

6 = 5位字符串+1位\n

%f 默认小数点后6位

“x = ”占四位

**实例函数

```
int printlog (const char* fmt, ...)
{
        va_list args; （这是什么意思不会）
        if (NULL == logfile) {
        logfile = fopen ("the_log","a");
        if (NULL == logfile) {
                return -1;
                }
        }
        va_start (args, fmt);
        return vfprintf (logfile, fmt, args);
}
```

int vfprintf(FILE *stream, const char *format, va_list ap);

stream：指向输出流的指针，可以是标准输出 stdout、标准错误 stderr 或者打开的文件指针；
format：格式字符串，包含要插入到结果字符串中的文本和格式说明符；
ap：可变参数列表，用于替换格式字符串中的格式说明符。

**实例：Pyramid Tree I/O Example

任务：

Here’s what we’ll do:

◦ write a tree as ASCII

◦ write a tree as binary

◦ compare the two files, and

◦ rebuild a tree from the binary file.

Then, as a think-pair-share, you can rebuild a tree from the ASCII file.

```
struct pyr_node_t {
        int32_t x;
        int32_t y_left;
        int32_t y_right;
        int32_t id;
};
struct pyr_tree_t {
        int32_t n_nodes;  //number of nodes in pyramid tree
        pyr_node_t* node;  //指针指向 array of node
};
```

```
int32_t write_pyr_tree_ASCII  (pyr_tree_t* p/*tree*/, const char* fname/*fimlename*/){
                FILE* out;
                if (NULL == (out = fopen (fname,"w")/*open file for writing*/)) {
                        return 0;  //打开失败。
                }
                fprintf (out, "%d\n", p->n_nodes);//print numbers of nodes to stream.
//要注意：N为leaf node的条件：4N + 1 ≥ n_nodes
//计算第一个叶节点
int32_t first_leaf;
int32_t i;
//计算第一个叶节点序列
first_leaf = (p->n_nodes + 2) / 4;
for (i = 0; first_leaf > i; i++) { //循环打印每一个内部点的数据
        fprintf (out, "%d %d %d\n",
                        p->node[i].x,
                        p->node[i].y_left,
                        p->node[i].y_right);
}
// After last loop, i is first_leaf.
for ( ; p->n_nodes > i; i++) { //循环所有叶节点
        fprintf (out, "%d\n", p->node[i].id); //打印叶节点id
}
return (0 == fclose (out));
```

下面是binary version的方法：

```
int32_t write_pyr_tree_binary (pyr_tree_t* p, const char* fname)  {
        FILE* out;
        if (NULL == (out = fopen (fname, "w"))) {
                return 0; //打开不成功
        }
        int32_t rval = (1 == fwrite(&p->n_nodes, sizeof(p->n_nodes),1,out) &&  p->n_nodes == fwrite (p->node,
sizeof (p->node[0]), p->n_nodes, out));
        //return success if both writes succeed
        //write node array to output file
        fclose (out);
        return rval;
//now reconstruct a pyramid tree from a binary file
pyr_tree_t* read_pyr_tree_binary (const char* fname/*file name*/)
{
        FILE* in;  //input stream
        pyr_tree_t* p; //new pyramid tree
        int32_t count; //numbers of node in file
        if (NULL == (in = fopen (fname, "r")) || 1 != fread(&count, sizeof (count),1,in)) { //if file open
success, read number of nodes in file
        if (NULL != in) {
                fclose (in);  //都失败，return failure并且关文件。
        }
        return 0;
}
        if (NULL == (p = malloc (sizeof (*p)))/*allocate space for pyramid tree*/ || NULL == (p->node = malloc
(count * sizeof (p->node[0])))/*allocate space for node array*/ {
        if (NULL != p) { free (p); }
                fclose (in);
                return NULL;  //都失败，free tree，close 文件，  return failure
}
p->n_nodes = count; //write number of nodes into pyramid tree
        if (p->n_nodes != fread (p->node, sizeof (p->node[0]), p->n_nodes, in)) {
        free_pyramid_tree (p);
        fclose (in);
        return NULL;
}
```

**其他的例子
Here's the problem:
◦ given a file name,
◦ count the "words" in the file, and
◦ print a list of the words and their counts in alphabetic order.

**关于类的整理class：

在C++中, 用 "类" 来描述 "对象", 所谓的"对象"是指现实世界中的一切事物。那么类就可以看做是对相似事物的抽象, 找到这些不同事物间的共同点
一个类的定义包含两部分的内容, 一是该类的**属性**, 另一部分是它所拥有的**方法**（描述对象拥有的行为）

```
class 类名{
                public:    //公共的行为或属性
                private:    //公共的行为或属性
                };         //分号一定不能省略
```

private表示该部分内容是私密的, 不能被外部所访问或调用, 只能被本类内部访问。
public 表示公开的属性和方法, 外界可以直接访问或者调用。

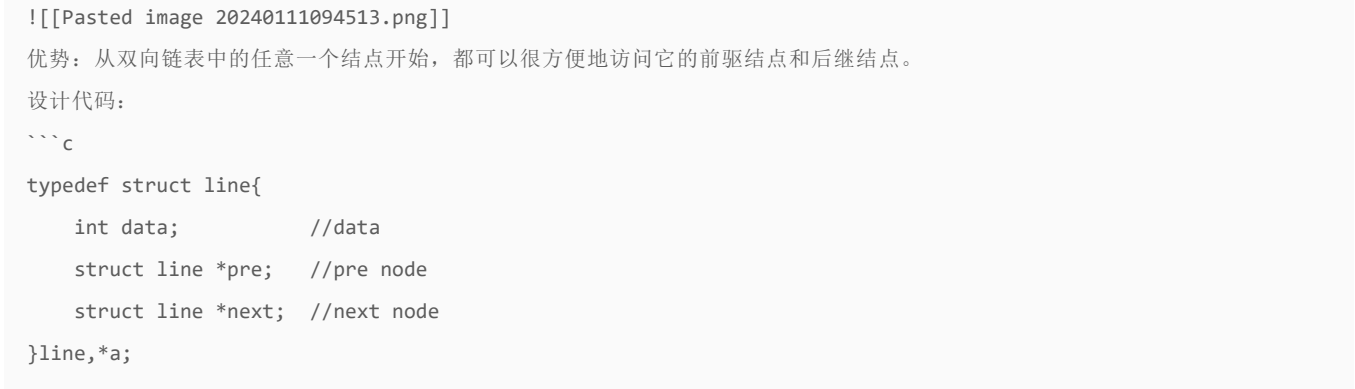
C++类的实现 ：
在上面的定义示例中我们只是定义了这个类的一些属性和方法声明, 并没有去实现它, 类的实现就是完成其方法的过程。类的实现有两种方式, **一种是在类定义时完成对成员函数的定义, 另一种是在类定义的外部进行完成。**
1>. 在类定义时定义成员函数

```
#include <iostream>
    using namespace std;
    class Point {
            public:
            void setPoint(int x, int y) //实现setPoint函数
                {
                        xPos = x;
                    yPos = y;
            }
            void printPoint() //实现printPoint函数
                {
                    cout<< "x = " << xPos << endl;
                    cout<< "y = " << yPos << endl;
                }
            private:
            int xPos;
                int yPos;
    };
    int main(){
                Point M; //用定义好的类创建一个对象 点M
                M.setPoint(10, 20); //设置 M点 的x,y值
            M.printPoint(); //输出 M点 的信息
                return 0;
                }
```

与类的定义相比, 在类内实现成员函数不再是在类内进行声明, 而是直接将函数进行定义, 在类中定义成员函数时, 编译器默认会争取将其定义为 **inline 型函数**
2>. 在类外定义成员函数
在类外定义成员函数通过在类内进行**声明**, 然后在类外通过作用域操作符 **::** 进行实现, 形式如下:
```cpp
返回类型 类名::成员函数名(参数列表)
```

```
{
//函数体
}
#include
using namespace std;
class Point
{
public:
void setPoint(int x, int y); //在类内对成员函数进行声明
void printPoint();
private:
int xPos;
int yPos;
};
void Point::setPoint(int x, int y) //通过作用域操作符 '::' 实现setPoint函数
{
xPos = x;
yPos = y;
}
void Point::printPoint() //实现printPoint函数
{
cout<< "x = " << xPos << endl;
cout<< "y = " << yPos << endl;
}
int main()
{
Point M; //用定义好的类创建一个对象 点M
M.setPoint(10, 20); //设置 M点 的x,y值
M.printPoint(); //输出 M点 的信息
return 0;
}
```

双链表的基本操作



优势：从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。

设计代码：

```c
typedef struct line{
    int data;           //data
    struct line *pre;   //pre node
    struct line *next;  //next node
}line,*a;
```

双链表的创建：

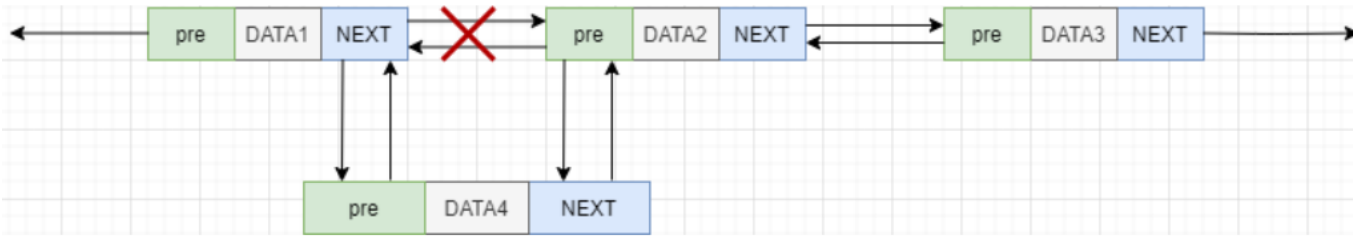
先创建头结点，然后逐步的进行添加双向链表的头结点是有数据元素的，也就是头结点的data域中是存有数据的，这与一般的单链表是不同的。

对于逐步添加数据，先开辟一段新的内存空间作为新的结点，为这个结点进行的data进行赋值，然后将已成链表的上一个结点的next指针指向自身，自身的pre指针指向上一个结点。

```
//创建双链表
line* initLine(line * head){
    int number,pos=1,input_data;
    //三个变量分别代表结点数量，当前位置，输入的数据
    printf("请输入创建结点的大小\n");
    scanf("%d",&number);
    if(number<1){return NULL;} //输入非法直接结束
    //////头结点创建///////
    head=(line*)malloc(sizeof(line));//重要的部分
    head->pre=NULL;
    head->next=NULL;
    printf("输入第%d个数据\n",pos++);
    scanf("%d",&input_data);
    head->data=input_data;

    line * list=head;
    while (pos<=number) {
        //创建新节点
        line * body=(line*)malloc(sizeof(line));
        body->pre=NULL;
        body->next=NULL;
        printf("输入第%d个数据\n",pos++);
        scanf("%d",&input_data);
        body->data=input_data;
        //头结点和新节点相互连接
        list->next=body;
        body->pre=list;
        list=list->next;
    }
    return head;
}
```

双向链表的插入操作

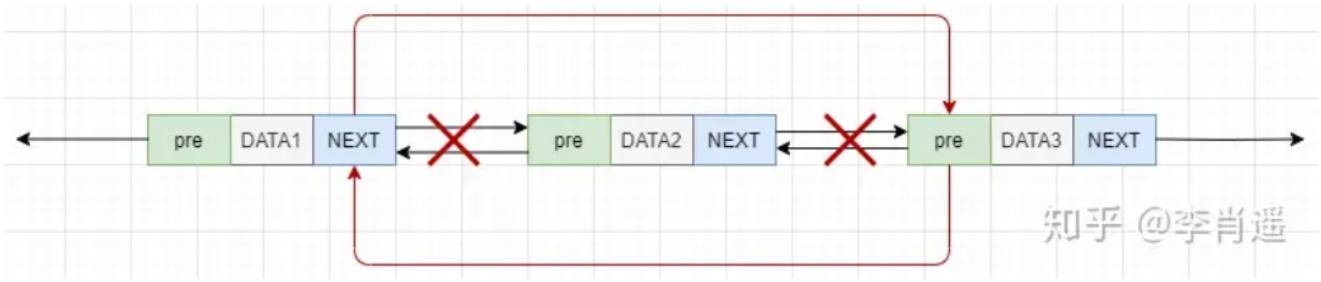


对于每一次的双向链表的插入操作，首先需要创建一个独立的结点，并通过malloc操作开辟相应的空间；
实现代码：

```
//插入数据
line * insertLine(line * head,int data,int add){
    //三个参数分别为：进行此操作的双链表，插入的数据，插入的位置
    //新建数据域为data的结点
    line * temp=(line*)malloc(sizeof(line));
    temp->data=data;
    temp->pre=NULL;
    temp->next=NULL;
    //插入到链表头，要特殊考虑
    if (add==1) {
        temp->next=head;
        head->pre=temp;
        head=temp;
    }else{
        line * body=head;
        //找到要插入位置的前一个结点
        for (int i=1; i<add-1; i++) {
```

```
            body=body->next;
        }
        //判断条件为真，说明插入位置为链表尾
        if (body->next==NULL) {
            body->next=temp;
            temp->pre=body;
        }else{
            body->next->pre=temp;
            temp->next=body->next;
            body->next=temp;
            temp->pre=body;
        }
    }
    return head;
}
```

双向链表的删除



选择需要删除的结点->选中这个结点的前一个结点->将前一个结点的next指针指向自己的下一个结点->选中该节点的下一个结点->将下一个结点的pre指针修改指向为自己的上一个结点。在进行遍历的时候直接将这一个结点给跳过了，之后，我们释放删除结点，归还空间给内存

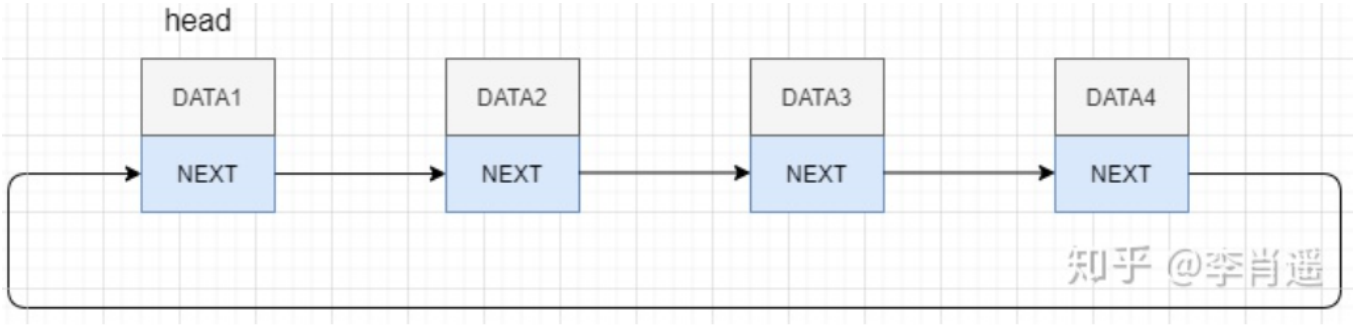
```
//删除元素
line * deleteLine(line * head,int data){
    //输入的参数分别为进行此操作的双链表，需要删除的数据
    line * list=head;
    //遍历链表
    while (list) {//直到null都没找到，停止操作
        //判断是否与此元素相等
        //删除该点方法为将该结点前一结点的next指向该节点后一结点
        //同时将该结点的后一结点的pre指向该节点的前一结点
        if (list->data==data) {
            list->pre->next=list->next;
            list->next->pre=list->pre;
            free(list);
            printf("--删除成功--\n");
            return head;
        }
        list=list->next;
    }
    printf("Error:没有找到该元素，没有产生删除\n");
    return head;
}
```

遍历双链表

```
//遍历双链表,同时打印元素数据
void printLine(line *head){
    line *list = head;
    int pos=1;
    while(list){
        printf("第%d个数据是:%d\n",pos++,list->data);
        list=list->next;
```

```
    }
}
```

循环链表



定义代码：

```
typedef struct list{
    int data;
    struct list *next;
}list;
//data为存储的数据，next指针为指向下一个结点
```

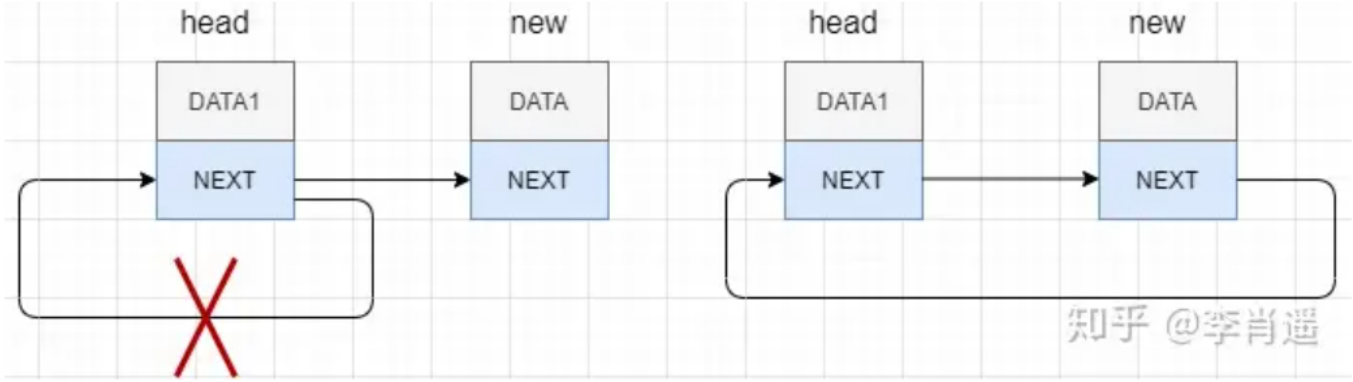
循环单链表初始化

先创建一个头结点并且给其开辟内存空间，在开辟内存空间成功之后，将头结点的next指向head自身，创建一个init函数来完成；

```
//初始结点
list *initlist(){
    list *head=(list*)malloc(sizeof(list));
    if(head==NULL){
        printf("创建失败，退出程序");
        exit(0);    exit(0) 直接退出这个程序，并返回退出码0
    }else{
        head->next=NULL;
        return head;
    }
}
    //////////初始化头结点//////////////
    list *head=initlist();
    head->next=head;
```

为了重复创建和插入，我们可以在init函数重新创建的结点next指向空，而在主函数调用创建之后，将head头结点的next指针指向自身。

循环链表的创建



通过逐步的插入操作，创建一个新的节点，将原有链表尾结点的next指针修改指向到新的结点，新的结点的next指针再重新指向头部结点，然后逐步进行这样的插入操作，最终完成整个单项循环链创建。

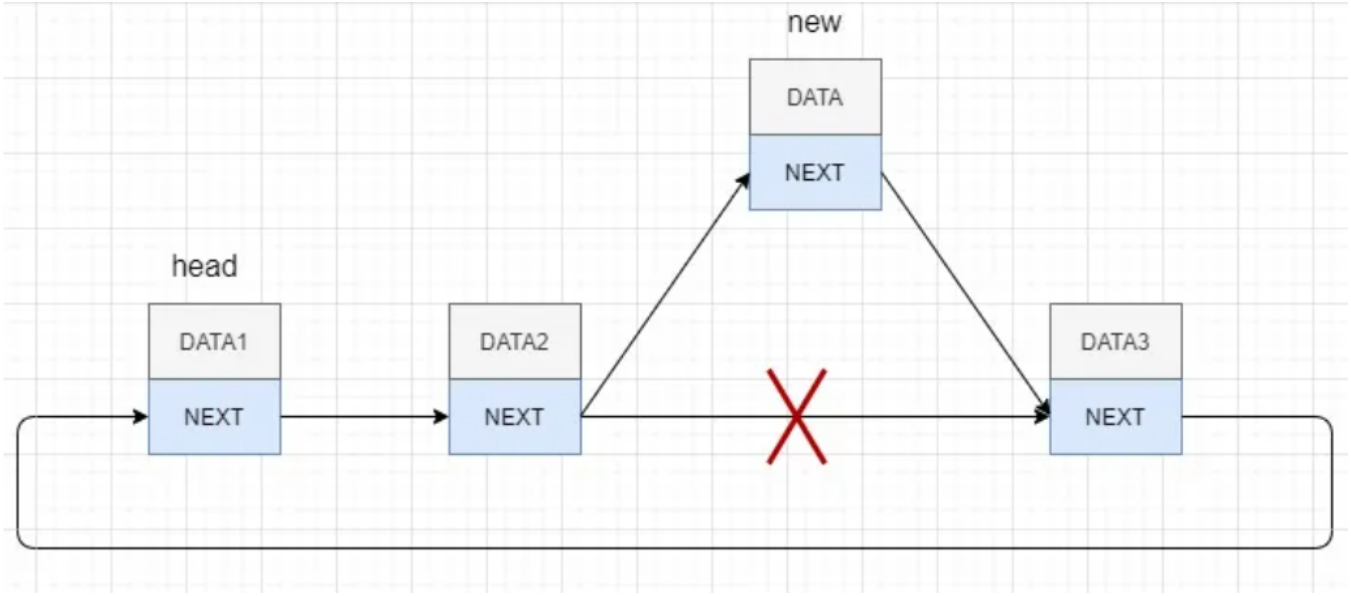
```c
//创建—插入数据
int insert_list(list *head){
    int data;   //插入的数据类型
    printf("请输入要插入的元素：");
    scanf("%d",&data);

    list *node=initlist();
    node->data=data;
    //初始化一个新的结点，准备进行链接

    if(head!=NULL){
        list *p=head;
        //找到最后一个数据
        while(p->next!=head){
            p=p->next;
        }
        p->next=node;
        node->next=head;
        return 1;
    }else{
        printf("头结点已无元素\n");
        return 0;
    }
}
```

循环单链表的插入

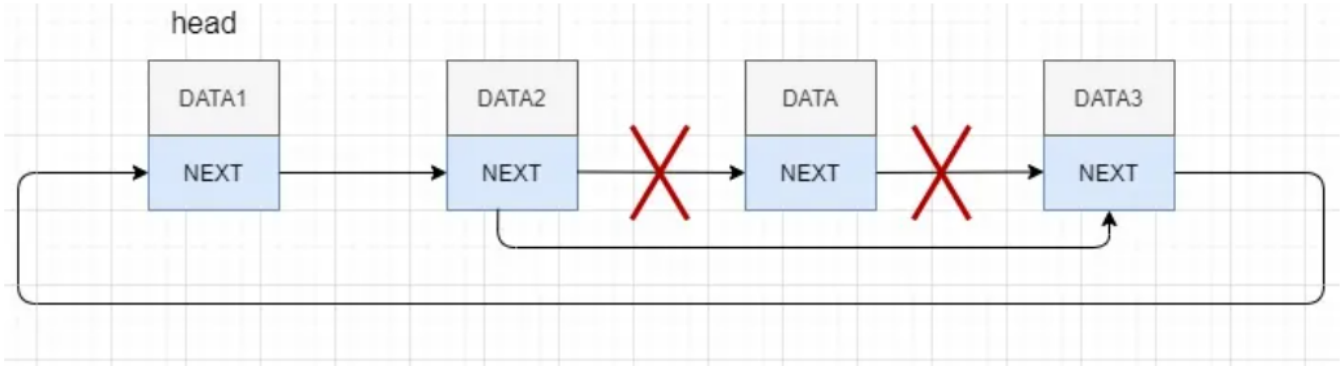
如图，对于插入数据的操作，可以创建一个独立的结点，通过将需要插入的结点的上一个结点的next指针指向该节点，再由需要插入的结点的next指针指向下一个结点的方式完成插入操作。




```c
//插入元素
list *insert_list(list *head,int pos,int data){
    //三个参数分别是链表，位置，参数
    list *node=initlist();  //新建结点
    list *p=head;       //p表示新的链表，
    list *t;            //和temp类似
    t=p;
    node->data=data;
    if(head!=NULL){
        for(int i=1;i<pos;i++){
            t=t->next;  //走到需要插入的位置处
        }
        node->next=t->next;
        t->next=node;
        return p;
```

```
    }
    return p;
}
```

循环单链表的删除




```
//删除元素
int delete_list(list *head) {
    if(head == NULL) {
        printf("链表为空！\n");
        return 0;
    }
    //建立临时结点存储头结点信息（目的为了找到退出点）
    //如果不这么建立的化需要使用一个数据进行计数标记，计数达到链表长度时自动退出
    //循环链表当找到最后一个元素的时候会自动指向头元素，这是我们不想让他发生的
    list *temp = head;
    list *ptr = head->next;

    int del;
    printf("请输入你要删除的元素：");
    scanf("%d",&del);

    while(ptr != head) {
        if(ptr->data == del) {
            if(ptr->next == head) {
                temp->next = head;
                free(ptr);
                return 1;
            }
            temp->next = ptr->next;    //核心删除操作代码
            free(ptr);
            //printf("元素删除成功！\n");
            return 1;
        }
        temp = temp->next;
        ptr = ptr->next;
    }
    printf("没有找到要删除的元素\n");
    return 0;
}
```

循环单链表的遍历

先找到尾节点的位置，由于尾节点的next指针是指向头结点的，所以不能使用 链表本身是否为空（NULL） 的方法进行简单的循环判断，我们需要通过判断结点的next指针是否等于头结点的方式进行是否完成循环的判断。

```
//遍历元素
int display(list *head) {
    if(head != NULL) {
        list *p  = head;
        //遍历头节点到，最后一个数据
        while(p->next != head ) {
```

```
            printf("%d   ",p->next->data);
            p = p->next;
        }
        printf("\n");   //换行
        //把最后一个节点赋新的节点过去
        return 1;
    } else {
        printf("头结点为空!\n");
        return 0;
    }
}
```