- **可变数组**
  通过实现一个函数库

```
#ifndef _ARRAY_H_
#define _ARRAY_H_

typedef struct{
    int *array;
    int size;
} Array;

Array array_create(int int_size);
void array_free(Array *a);
int array_size(const Array *a);
int* array_at(Array *a,int index);
void array_inflate(Array *a,int more_size);
#endif
```

```
#include "array.h"
#include <stdio.h>
#include <stdlib.h>
# define BLOCK_SIZE 5
//typedef struct{
//    int *array;
//    int size;
//} Array;

Array array_create(int init_size){
    Array a;
    a.size=init_size;
    a.array=(int*)malloc(sizeof(int)*a.size);
    return a;
}

void array_free(Array *a){
    free(a->array);
    a->array=NULL; //防止调用两次free
    a->size=0;
}

int array_size(const Array *a){
    return a->size;
}               //不直接a.size，是为了封装，因为以后可能算法会升级复杂化
```

```c
int* array_at(Array *a,int index){
    if(index>=a->size){
        //array_inflate(a, index-a->size+1);//每次只长到index，不经济
        /*需要有block概念，每次长block*/
        array_inflate(a,(index/BLOCK_SIZE+1)*BLOCK_SIZE-a->size);
    }
    return &(a->array[index]); //因为返回的是int*类型，所以是指针
}

void array_inflate(Array *a,int more_size){
    int *p = (int*)malloc(sizeof(int)*(a->size)+more_size);
    int i;
    for(i=0;i<a->size;i++){
        p[i]=a->array[i];    //拷贝
    }
    free(a->array);
    a->array=p;
    a->size+=more_size;
}

int main(int argc,char const *argv[]){
    Array a=array_create(100);
    printf("%d\n",array_size(&a));
    *array_at(&a,0)=10;
    printf("%d\n",*array_at(&a,0));
    int number=0;
    int cnt=0;
    while(number!=-1){
        scanf("%d",&number);
        if(number!=-1){
            *array_at(&a,cnt++)=number;
        }
        //scanf("%d",array_at(&a,cnt++));
    }

    array_free(&a);

    return 0;
}
```

- **可变数组的缺陷**
  一是每次长大的时候都要申请新的内存空间，包括新的在内的全部的东西要拷贝进去
  首先，拷贝要花时间，其次，存在一种情况，明明有足够的内存，却再也不能申请空间了，如在内存受限的场合，比如前面是n-bs，中间是n，后面是2bs，空余的内存是n+bs，但却是被分隔开来的，因此无法申请到。

可以采用linked blocks，申请block大小的内存，然后，把它们链接起来，一个block走完到下一个block访问，不需要再拷贝了，节约了时间，充分利用内存的角角落落

- **链表**
  链表 节点

```
#ifndef _NODE_H_
#define _NODE_H_

typedef struct _node{ //节点
    int value;
    struct _node *next;
} Node;

#endif
```

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

//typedef struct _node{
//    int value;
//    struct _node *next;
//} Node;

int main(int argc,char const *argv[]){
    Node *head=NULL;
    int number;
    do{
        scanf("%d",&number);
        if(number!=-1){
            /* add to linked list */
            Node *p=(Node*)malloc(sizeof(Node));
            p->value=number;
            p->next=Null; //因为是新的一个
            /*find the last */
            Node *last=head; //每次从开头开始找
            if(last){
                while(last->next){
                last=last->next;
                } //只要next不是null，说明没找到最后一个
                // attach
                last->next=p;//找到最后一个后在其后添加
            }else{
```

```
                head=p;//如果last是null
            }


        }
    }while(number!=-1);

    return 0;
}
```

将我们添加链表元素的过程抽出来作为一个函数
但如果单纯地抽出来的话，head作为参数传入后是没有变化的，因此是错误的，那么方法是第一让函数返回head，在main函数中用head再接受，第二个方法是传入指向head的指针，即指向指针的指针，最后一种就是再定义一个结构

链表搜索

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

//typedef struct _node{
//    int value;
//    struct _node *next;
//} Node;
typedef struct _list{
    Node *head;
} List;

void add(List *plist,number);
void print(List *plist);

int main(int argc,char const *argv[]){
    List list;
    int number;
    list.head=NULL;
    do{
        scanf("%d",&number);
        if(number!=-1){

            add(&list,number)

        }
    }while(number!=-1);
```

```
    /*Node *p;
    for(p=head;p;p=p->next){  // 结束是p还存在
        printf("%d\t",p->value); //链表遍历
    }
    printf("\n");*/

    print(&list);

    scanf("%d",&number);
    isFound=0;
    for(p=list.head;p;p=p->next){
        if(p->value==number){
            printf("找到令");
            isFound=1;
            break;
        }
    }
    if(!isFound){
        printf("没找到");
    }

    Node *q; //删除节点
    for(q=NULL,p=list.head;p;q=p,p=p->next){
        if(p->value==number){
            if(q){
                q->next=p->next;
            }else{
                list.head=p->next;
            }
            free(p);
            break;
        }
    }
    //清除整个链表
    for(p=list.head;p;p=q){
        q=p->next;
        free(p);
    }

    return 0;
}

/* add to linked list */
void add(List *plist,number){
    Node *p=(Node*)malloc(sizeof(Node));
    p->value=number;
    p->next=Null; //因为是新的一个
    /*find the last */
```

```
    Node *last=plist->head; //每次从开头开始找
    if(last){
        while(last->next){
        last=last->next;
        } //只要next不是null，说明没找到最后一个
        // attach
        last->next=p;//找到最后一个后在其后添加
    }else{
        plist->head=p;//如果last是null
    }
}

void print(List *plist){
    Node *p;
    for(p=plist->head;p;p=p->next){  // 结束是p还存在
        printf("%d\t",p->value); //链表遍历
    }
    printf("\n");
}
```

删除链表元素，首先我们需要让前面的一个单元的指针部分指向后面一个单元，并且要将删除的部分的空间给free掉
指针如果出现在->的左边，意味着这个指针不能是NULL，所以在上面问题可能会处在q上面，需要识别，如果为null，则q是第一个数，因此让head指向下一个