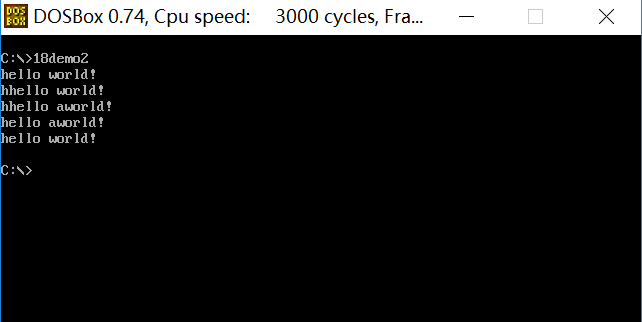
**c语言综合研究十八报告**

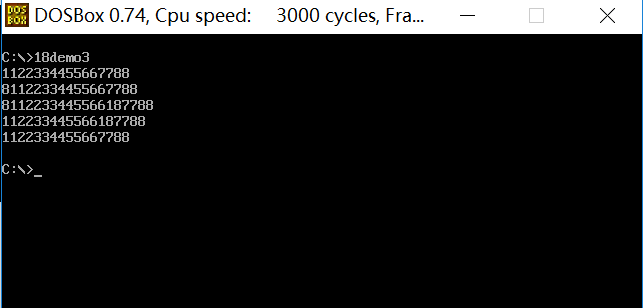
1. **研究过程展示**

**1>.程序18demo2.c部分：**

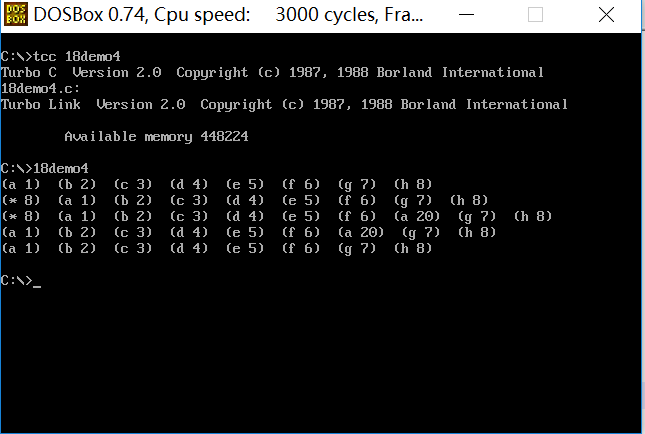
用m.c对list.h进行测试：



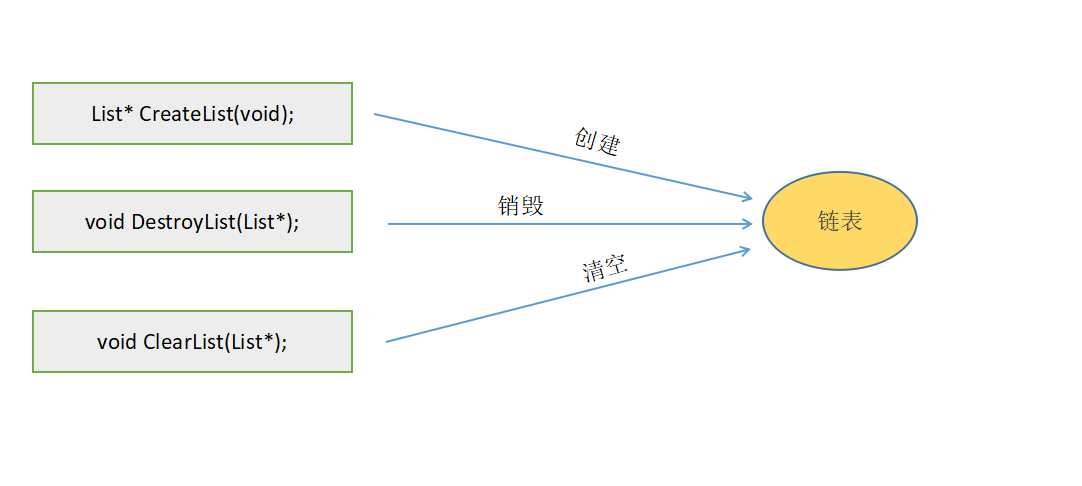
将list.h的第一个语句typedef char EleType改为typedef int EleType，再用m1.c对list.h进行测试：

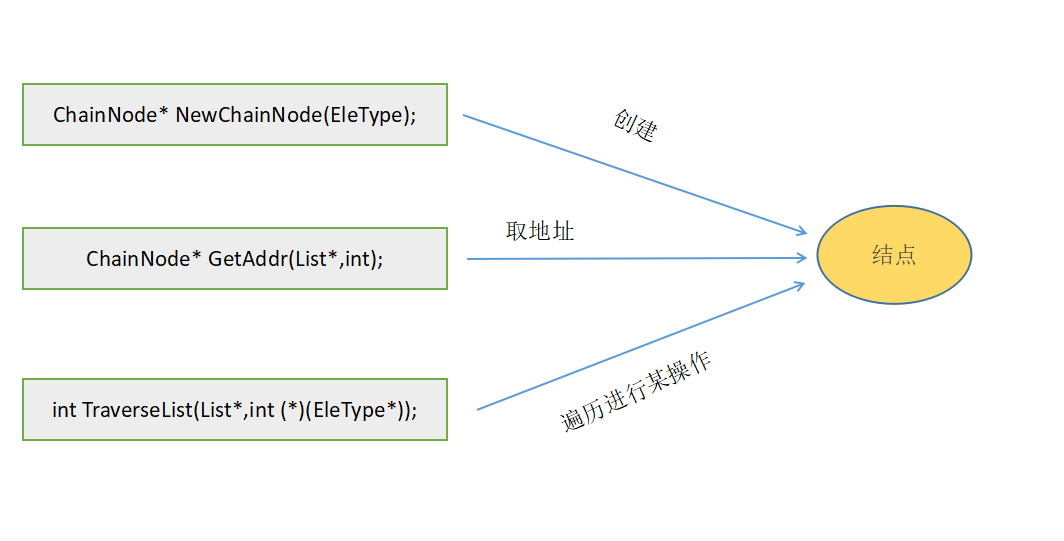


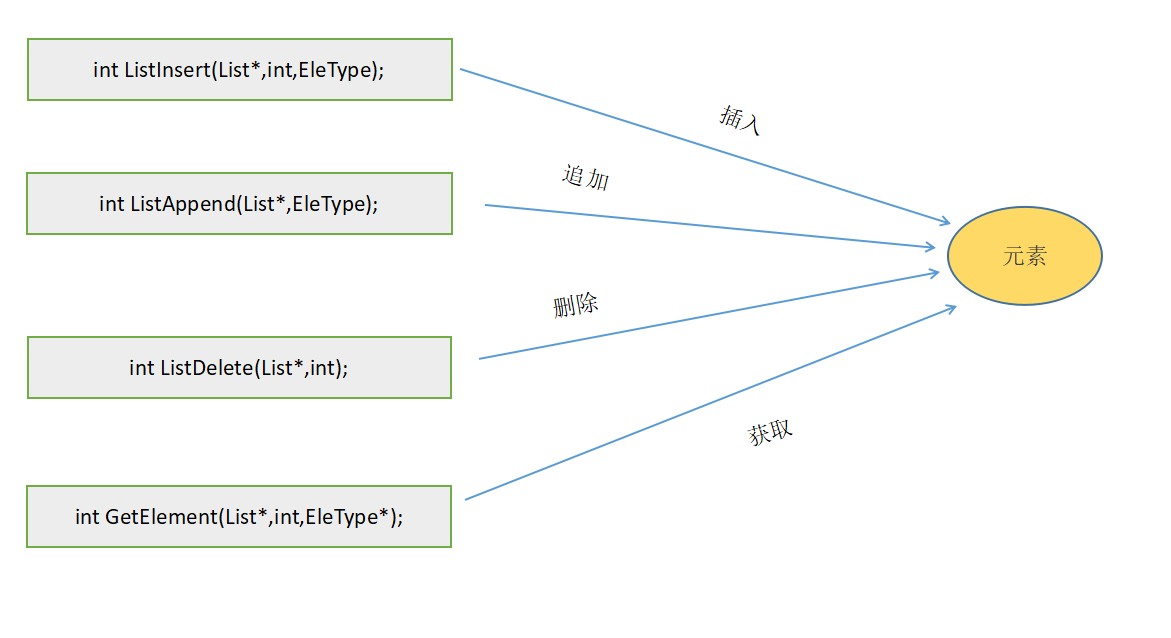
将list.h的第一个语句typedef char EleType改typedef struct{char a;int b;} EleType，再用m2.c对list.h进行测试：



从list.h头文件可以看到，结构大致如下：







**2>.程序18demo3.c部分：**

**3>.分析部分**

程序部分：

1.添加tail指针：

程序18demo6.h:

typedef char EleType;

typedef struct node

{

EleType data;

struct node\* next;

}ChainNode;

typedef struct

{

ChainNode\* head;

ChainNode\* tail;

}List;

List\* CreateList(void);

void DestroyList(List\*);

void ClearList(List\*);

int ListAppend(List\*,EleType);

int ListInsert(List\*,int,EleType);

int ListDelete(List\*,int);

int GetElement(List\*,int,EleType\*);

ChainNode\* GetAddr(List\*,int);

int TraverseList(List\*,int (\*)(EleType\*));

ChainNode\* NewChainNode(EleType);

List\* CreateList(void)

{

List\* p;

EleType\* data = 0;

p = (List\*)malloc(sizeof(List));

if(!p)

{

return 0;

}

p->tail = p->head = NewChainNode(\*data);

if(!p->head)

{

free(p);

return 0;

}

return p;

}

void DestroyList(List\* lp)

{

ClearList(lp);

free(lp->head);

free(lp->tail);

free(lp);

}

void ClearList(List\* lp)

{

while(ListDelete(lp,1));

}

int ListAppend(List\* lp,EleType data)

{

ChainNode\* p;

ChainNode\* newp;

newp = NewChainNode(data);

if(!newp)

{

return 0;

}

p = lp->tail;

p->next = newp;

lp->tail = newp;

return 1;

}

int ListInsert(List\* lp,int n, EleType data)

{

ChainNode\* p;

ChainNode\* newp;

p = GetAddr(lp,n-1);

if(!p)

{

return 0;

}

newp = NewChainNode(data);

if(!newp)

{

return 0;

}

newp->next = p->next;

p->next = newp;

return 1;

}

int ListDelete(List\* lp,int n)

{

ChainNode\* p;

ChainNode\* p1;

if(!lp->head->next)

{

return 0;

}

p = GetAddr(lp,n-1);

if(!(p&&p->next))

{

return 0;

}

p1 = p->next;

p->next = p->next->next;

free(p1);

return 1;

}

int GetElement(List\* lp,int n,EleType\* data)

{

ChainNode\* p;

p = GetAddr(lp,n);

if(!p)

{

return 0;

}

\*data = p->data;

return 1;

}

int TraverseList(List\* lp,int (\*f)(EleType\*))

{

ChainNode\* p;

int a=0;

for(p = lp->head->next;p;p=p->next)

{

if(!f(&(p->data)))

{

return a+1;

}

a++;

}

return 0;

}

ChainNode\* NewChainNode(EleType data)

{

ChainNode\* p;

p = (ChainNode\*)malloc(sizeof(ChainNode));

if(!p)

{

return 0;

}

p->data = data;

p->next = 0;

return p;

}

ChainNode\* GetAddr(List\* lp,int n)

{

ChainNode\* p;

int a;

if(n<0)

{

return 0;

}

p = lp->head;

a = 0;

while(p&&a<n)

{

p = p->next;

a++;

}

return p;

}

3.设计一个不限元素数据类型的通用链表：

程序中实现的链表里的元素类型都是固定的，怎么实现一个链表使它的元素类型为任意类型呢？要在链表中的结点的数据空间中存放任意类型的数据是不可能的，因为每个节点定义类型的时候大小都是固定的。

不过根据（void\*）指针可以任意赋值的特点，可以这样实现：在结点里的数据空间存放指针，指针指向每一个元素处的空间，这个空间的大小可以是任意的，根据我们定义的数据类型而改变，用malloc函数动态分配内存。之后再赋值给（void\*）指针。

但是现在的问题是我们不知道用户传入的数据大小是多少，像printf函数一样用类型说明符只能实现基本数据类型而不能实现用户自定义类型，而用户用结构体定义的自定义大小可以为任意大小，甚至理论上是无穷大的。之前我以为要实现链表的每一个元素的类型都可以是不一样的，

因为不知道数据大小到底是多少，所以在线性表中加入一个数据项，int length以表示数据大小，并在main函数中创建线性表时用sizeof计算数据元素大小并传给length;

传入ListAppend函数的链表数据是一个局部变量，保存在栈段中，并且在函数返回后会被释放，所以要另外开辟空间来存储它。这里&lp表示传入的线性表lp在栈中的地址，&lp+1、&lp+2分别表示下一个参数，即我们要添加、插入的数据在栈中的地址。

程序18demo7.h:

typedef struct node

{

void\* datap;

struct node\* next;

}ChainNode;

typedef struct

{

int length;

ChainNode\* head;

ChainNode\* tail;

}List;

List\* CreateList(int);

void DestroyList(List\*);

void ClearList(List\*);

int ListAppend(List\*,...);

int ListInsert(List\*,int,...);

int ListDelete(List\*,int);

int GetElement(List\*,int,void\*);

ChainNode\* GetAddr(List\*,int);

int TraverseList(List\*,int (\*)(void\*));

ChainNode\* NewChainNode(void\*);

List\* CreateList(int size)

{

List\* p;

void\* datap = 0;

p = (List\*)malloc(sizeof(List));

if(!p)

{

return 0;

}

p->tail = p->head = NewChainNode(datap);

p->length = size;

if(!p->head)

{

free(p);

return 0;

}

return p;

}

void DestroyList(List\* lp)

{

ClearList(lp);

free(lp->head);

free(lp->tail);

free(lp);

}

void ClearList(List\* lp)

{

while(ListDelete(lp,1));

}

int ListAppend(List\* lp,...)

{

void\* pt;

void\* ps;

ChainNode\* newp;

ps = &lp + 1;

if(!ps)

{

return 0;

}

pt = (void \*)malloc(lp->length);

if(!pt)

{

return 0;

}

memcpy(pt, ps , lp->length);

newp = NewChainNode(pt);

if(!newp)

{

return 0;

}

lp->tail->next = newp;

lp->tail = newp;

return 1;

}

int ListInsert(List\* lp,int n, ...)

{

ChainNode\* p;

ChainNode\* newp;

void\* pt;

void\* ps;

ps = &lp + 2;

p = GetAddr(lp,n-1);

if(!p)

{

return 0;

}

pt = (void\*)malloc(lp->length);

if(!pt)

{

return 0;

}

memcpy(pt,ps,lp->length);

newp = NewChainNode(pt);

if(!newp)

{

return 0;

}

newp->next = p->next;

p->next = newp;

return 1;

}

int ListDelete(List\* lp,int n)

{

ChainNode\* p;

ChainNode\* p1;

if(!lp->head->next)

{

return 0;

}

p = GetAddr(lp,n-1);

if(!(p&&p->next))

{

return 0;

}

p1 = p->next;

p->next = p->next->next;

free(p1);

return 1;

}

int GetElement(List\* lp,int n,void\* datapp)

{

ChainNode\* p;

p = GetAddr(lp,n);

if(!p)

{

return 0;

}

datapp = (void\*)malloc(lp->length);

if(!datapp)

{

return 0;

}

datapp = p->datap;

return 1;

}

int TraverseList(List\* lp,int (\*f)(void\*))

{

ChainNode\* p;

int a=0;

p = (ChainNode\*)malloc(sizeof(ChainNode));

if(!p)

{

return 0;

}

for(p = lp->head->next;p;p=p->next)

{

if(!f( p->datap ))

{

return a+1;

}

a++;

}

return 0;

}

ChainNode\* NewChainNode(void\* pt)

{

ChainNode\* p;

p = (ChainNode\*)malloc(4);

if(!p)

{

return 0;

}

p->datap = pt;

p->next = 0;

return p;

}

ChainNode\* GetAddr(List\* lp,int n)

{

ChainNode\* p;

int a;

if(n<0)

{

return 0;

}

p = lp->head;

a = 0;

while(p&&a<n)

{

p = p->next;

a++;

}

return p;

}

打印整型的程序18demo8.c:

#include "18demo7.h"

typedef struct

{

char a;

int b;

}EleType;

EleType a[8] = {'a',1,'b',2,'c',3,'d',4,'e',5,'f',6,'g',7,'h',8};

void showlist(List\*);

int putelement(void\*);

main()

{

List\* lp;

int n;

EleType data;

lp = CreateList(sizeof(EleType));

if(!lp)

{

printf("Create?\n");

return;

}

for(n = 0; n < 8 ; n++)

{

ListAppend(lp,a[n]);

}

showlist(lp);

data.a = '\*';

data.b = 8;

ListInsert(lp,1,data);

showlist(lp);

data.a = 'a';

data.b = 20;

ListInsert(lp,8,data);

showlist(lp);

ListDelete(lp,1);

showlist(lp);

ListDelete(lp,7);

showlist(lp);

}

void showlist(List\* lp)

{

TraverseList(lp,putelement);

printf("\n");

}

int putelement(void\* data)

{

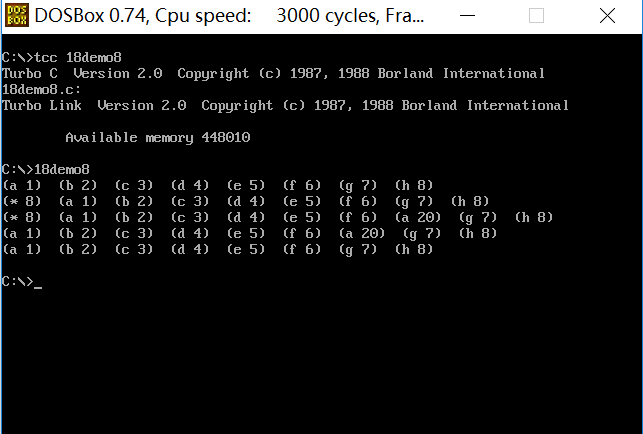
EleType\* test = (EleType\*)data;

printf("(%c %d) ",test->a,test->b);

return 1;

}

运行之后：



这是对自定义数据类型的处理，其他基本数据类型也可以处理。

1. **已思考研究并已解决问题汇总**

1.List 只有一个数据项“ChainNode\* head”,为什么还要定义这个数据类型？

因为表是一个整体的概念，就像之后扩展的时候，还会增加tail指针，变量length一样。这样做就好像给了“表”这个对象更多的可能性。

1. 是怎样的设计保证了线性表实现方法的改动，没有引起使用线性表的程序的改动？

共性和个性的区分，遍布各处的“return 0”、“return 1”这里就是返回一个调用状态，说明函数之间尽量遵循了一个独立的原则。而我们在用的时候，也是只需要知道他的参数是什么，功能是什么，返回值是什么就可以了。不需要了解内容功能修改了没有，修改了什么，怎么修改的等问题。

1. 为什么要使用一个头结点？

这里我觉得就是一个统一的问题，比较一下按位计数法，最常见的10进制计数法，0的作用是什么呢？举个数字“2503”，0在这里的作用是什么？对的，脱口而出，就是“没有”，0在这里就是表示没有，但是，可不可以没有“没有”呢？那不就是“253”了吗？明显不对了。所以这里的0表示“没有”但是不可或缺，也就是起了“占位”的作用。同样的，乐谱中的休止符也是可以由此联想的。头结点就像0一样，它有具体的空间，它占了一个位。而在数学中10的幂次从0...5...100...依次增长，这里为什么 有0呢?因为还有...-100...-5...0

的次幂，如果我们把10的0次只作为1来特殊化处理，就好像人为地建立了一个隔断，所以0还有统一标准的作用。头结点同样如此，统一标准，就像是程序员的一枚小细作，程序员控制头结点，头结点像领头羊一样带领着屁股后面的一串结点。

1. void \*memcpy( void \*dest, const void \*src, size\_t count);

Memcpy函数的原型如上，即从指针src指向的空间拷贝count个字节到指针dest指向的空间里。

1. 对void\* 指针强制类型转换时的体会

就像是用基本类型（系统地默认模板）或自定义数据类型（自定义模板）对一块已经密密麻麻摆满的数据的文本进行了相应的数据类型的格式化排版，这也就解决了相应的打印输出问题。

1. **已思考研究并未解决问题汇总**

1、销毁的时候，只有free，没有赋值为0,不怕出现野指针吗？

2、void\*指针还有其他什么神通广大的地方？

3、缺乏一个系统的总结

1. **研究感想（心得体会）**

为什么我感觉本来的程序样式就挺好了，加入泛型设计就那么好吗？我的悟性是真的不高呀。