单链表 （18.7.3）

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

typedef struct list\_nod

{

int data;

struct list\_nod \*next;

}list\_sing; //单链表的写法

int **main**()

{

list\_sing \*head =NULL;

head = (list\_sing \*)malloc(sizeof(list\_sing));

if(head == NULL)

{

printf("malloc failed \n");

}

memset(head,0,sizeof(head));

head->data=100;

head->next =NULL;

printf("data = %d\n",head->data);

free(head);

return 0;

}

1.开辟空间

2.不为空指针

3.清空，再使用

4使用

5.free()

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define LEN sizeof (list\_init)

typedef struct student

{

int num;

float score;

struct student \*next;

}list\_init;

list\_init stu;

int n;

struct student \***Create**()

{

struct student \*head; //头节点

struct student \*p1 = NULL; //p1保存创建的新节点的地址

struct student \*p2 = NULL; //p2保存原链表最后一个节点的地址

n = 0; //创建前链表的节点总数为0：空链表

p1 = (struct student \*) malloc (LEN); //开辟一个新节点

p2 = p1; //如果节点开辟成功，则p2先把它的指针保存下来以备后用

if(p1==NULL) //节点开辟不成功

{

printf ("\nCann't create it, try it again in a moment!\n");

return NULL;

}

else //节点开辟成功

{

head = NULL; //开始head指向NULL

printf ("Please input %d node -- num,score: ", n + 1);

scanf ("%d %f", &(p1->num), &(p1->score)); //录入数据

}

while(p1->num != 0) //只要学号不为0，就继续录入下一个节点

{

n += 1; //节点总数增加1个

if(n == 1) //如果节点总数是1，则head指向刚创建的节点p1

{

head = p1;

p2->next = NULL; //此时的p2就是p1,也就是p1->next指向NULL。

}

else

{

p2->next = p1; //指向上次下面刚刚开辟的新节点

}

p2 = p1; //把p1的地址给p2保留，然后p1产生新的节点

p1 = (struct student \*) malloc (LEN);

printf ("Please input %d node -- num,score: ", n + 1);

scanf ("%d %f", &(p1->num), &(p1->score));

}

p2->next = NULL; \//此句就是根据单向链表的最后一个节点要指向NULL

free(p1); //p1->num为0的时候跳出了while循环，并且释放p1

p1 = NULL;//特别不要忘记把释放的变量清空置为NULL,否则就变成"野指针"，即地址不确定的指针

return head; //返回创建链表的头指针

}

void **Print**(struct student \*head)

{

struct student \*p;

printf ("\nNow , These %d records are:\n", n);

p = head;

if(head != NULL) //只要不是空链表，就输出链表中所有节点

{

printf("head is %o\n", head); //输出头指针指向的地址

do

{

/\*

输出相应的值：当前节点地址、各字段值、当前节点的下一节点地址。

这样输出便于读者形象看到一个单向链表在计算机中的存储结构，和我们

设计的图示是一模一样的。

\*/

printf ("%o %d %5.1f %o\n", p, p->num, p->score, p->next);

p = p->next; //移到下一个节点

}

while (p != NULL);

}

}

/\*

list\_init \*creat()

{

list\_init \*head;

list\_init \*p1=NULL;

list\_init \*p2=NULL;

p1 = (list\_init \*)malloc(sizeof(LEN));

if(p1 == NULL)

{

printf("sizeof failed \n");

return NULL;

}

head = NULL;

p2 = p1;

printf("node = %d p1->num = %d\n",n+1,p1->num);

while(!p1->num)

{

n =+ 1;

if(n == 1)

{

head =p1;

p2->next = NULL;

}

else

{

p2 -> next =p1;

}

p2 = p1;

p1 = (list\_init \*)malloc(sizeof(LEN));

printf("input %d \n",n+1);

scanf("%d,%f",&(p1->num),&(p1->score));

}

}

list\_init print(list\_init \*head)

{

list\_init \*p;

p = head;

printf ("\nNow , These %d records are:\n", n);

if(head != NULL)

{

printf("head print = %o p = %o \n",head,p);

do

{

printf("p = %o p->num = %d, p->score = %f \n",p,p->num,p->score,p->next);

p= p->next;

}

while(p != NULL);

}

}

\*/

int **main**()

{

list\_init \*head;

head = creat();

print(head);

return 0;

}

-----------------------------------------------------------------------------------------------------------------------

共享内存

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <unistd.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define PERM 0666|IPC\_CREAT //066 可读写，如果没有create

int **main**(int argc, char \*\*argv)

{

int shmid;

char \*f\_add,\*p\_add;

if(argc != 2)

{

printf("plase input add :%s\n",argv[0]);

exit(1);

}

if((shmid = shmget(IPC\_PRIVATE,1024,PERM))== -1) // IPC\_PRIVATE 无名管道亲缘

{

printf("creat error \n");

return -1;

}

printf("shmid = %d \n",shmid);

if(fork())

{

f\_add = shmat(shmid, 0,0);

printf("f\_add= %p \n",f\_add);

memset(f\_add,0,sizeof(f\_add));

memcpy(f\_add,argv[1],1024);

exit(0);

}

else

{

sleep(0);

p\_add =shmat(shmid,0,0);

printf("add = %p \n",p\_add);

}

char cmd[80]={0};

sprintf(cmd,"ipcs | grep %d",shmid);

printf("cmd = %s \n",cmd);

system(cmd);

shmdt(shmid);

shmctl(shmid, IPC\_RMID,0); //删除id

}

注意事项：

| //删除之前要脱离映射unmap

| shmdt(p);

| //删除共享共享内存--不一定立即删除--要看是否还有其他进程链接到该进程

| shmctl(shm\_id,IPC\_RMID,NULL);

int shmdt(const void \*shmaddr);

注意：将共享内存段与当前进程脱离不等于删除共享内存段

**./a.out 结果**

shmid = 14286892

f\_add= 0x7f61bc46a000

add = 0x7f61bc46a000

cmd = ipcs | grep 14286892

0x00000000 14286892 yq 666 1024 1

**callback**

#include <stdio.h>

int **add\_ret**() ;

int **multiply**();

int **add**(int a , int b , int (\*add\_value)()) //

{

return (\*add\_value)(a,b);

}

int **multiplk\_p**(int c,int d,int (\*multip\_date)())

{

return (\*multip\_date)(c,d);

}

int **main**(void)

{

int sum = add(3,4,add\_ret);

int multip = multiplk\_p(6,5,multiply);

printf("sum:%d\n",sum);

printf("mutil:%d\n",multip);

return 0 ;

}

int **add\_ret**(int a , int b)

{

return a+b ;

}

int **multiply** (int c, int d)

{

printf("c = %d ,d =%d \n ",c,d);

return c - d;

}

两层指针使用

#include <stdio.h>

const char \*msg[] ={"a","b","c","d","e","f"};

void const **get\_print**(const char \*\*p)

{

static int i =0;

printf("i = %d \n",i);

\*p = msg[i%7];

i++;

}

int **main**()

{

const char \*p\_1 = NULL;

const char \*p\_2 = NULL;

get\_print(&p\_1);

get\_print(&p\_2);

printf("p\_1= %s p\_2= %s \n",p\_1,p\_2);

}

单例模式：

举一个小例子，在我们的windows桌面上，我们打开了一个回收站，当我们试图再次打开一个新的回收站时，Windows系统并不会为你弹出一个新的回收站窗口。，也就是说在整个系统运行的过程中，系统只维护一个回收站的实例。这就是一个典型的单例模式运用

特别地，在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。事实上，这些应用都或多或少具有资源管理器的功能。例如，每台计算机可以有若干个打印机，但只能有一个 Printer Spooler(单例) ，以避免两个打印作业同时输出到打印机中。再比如，每台计算机可以有若干通信端口，系统应当集中 (单例)管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。总之，选择单例模式就是为了避免不一致状态，避免政出多头。

　　综上所述，单例模式就是为确保一个类只有一个实例，并为整个系统提供一个全局访问点的一种方法。

// g\_perm\_arg = optarg;

break;

default:

// g\_print\_help = 1;

ret = -1;

}

}

return ret;

重载运算符 友元

/\*

练习 重载运算符 友元

operator friend

operator：注意重载的运算的返回

操作符重载的概念：

注意事项：

将现有操作符与一个成员函数相关联，并将该操作符与其成员对象（操作数）一起使用。

1） 重载不能改变操作符的基本功能，以及该操作符的优先级顺序。

2） 重载不应改变操作符的本来含义。

3） 只能对已有的操作符进行重载，而不能重载新符号。

4） 操作符重载只对类可用。

5） 以下运算符不能被重载：

. 原点操作符（成员访问符）

\* 指向成员的指针

:: 作用域解析符

? : 问号条件运算符

sizeof 操作数的字节数

操作符函数的一般格式：

return\_type operator op(argument list);

return\_type：返回类型（要得到什么）

op：要重载的操作符

argument list：参数列表（操作数有哪些）

###################################

friend:

友元类：在一个类中指明其他的类（或者）函数能够直接访问该类中的private和protected成员

-注意：friend在类中的声明可以再public、protected和private的如何一个控制域中，而不影响其效果。例如，如果你在protected域中有这样的声明，那么aClass类同样可以访问该类的private成员

友元函数：它允许一个函数不需要通过其public接口就能够访问到类中的private和protected成员变量

-注意：友元函数 不是类成员函数 可以声明再类内或者类外 不加 类的限定符 ::

\*/

#include <iostream>

#include <string>

#include <stdio.h>

using namespace std;

class Ram

{

private:

int a;

int **add**(int b,int c);

friend class Box; //Box class call Ram private data must define in there

};

int Ram::**add**(int b,int c)

{

return b+c;

}

class Box

{

public:

int b;

int l;

int g;

void **set\_b**(int a);

void **set\_l**(int a);

int **count**();

int **call\_f**();

friend Box **operator** + (const Box& x, const Box& y);

//friend class Ram;

private:

Ram \*ram\_f;

#if 0 //method 2

Box operator + (const Box& x)

{

Box box;

box.b = this -> b + x.b;

box.l = this -> l + x.l;

box.g = this -> g + x.g;

return box; // remember return ！！！！

}

#endif

};

#if 1 //method 1

Box **operator** + (const Box& x, const Box& y)

{

Box box;

box.b = y.b + x.b;

box.l = y.l + x.l;

box.g = y.g + x.g;

return box; // remember return ！！！！

}

#endif

void Box::**set\_b**(int a)

{

b = a;

}

void Box::**set\_l**(int a)

{

l = a;

}

int Box::**count**()

{

return b \* l;

}

int Box::**call\_f**()

{

return ram\_f->add(6,7);

}

int **main**()

{

int t;

Box box1;

Box box2;

Box box3;

/\*

box1.set\_b(1);

box1.set\_l(2);

box2.set\_b(1);

box2.set\_l(2); //same under way

\*/

box1.b = 1;

box1.l = 2;

box1.g = 3;

box2.b = 1;

box2.l = 2;

box2.g = 3;

box3 = box1 + box2;

t =box3.count();

cout << "box3.b = " <<box3.b<<endl;

cout << "t = "<<t<<endl;

out << "Box call Ram private data call\_f = " << box1.call\_f()<<endl;

}

/\*\*\*\*\*\*\*\*\*\*

\*

Vector<类型>标识符

Vector<类型>标识符(最大容量)

Vector<类型>标识符(最大容量,初始所有值)

Int i[5]={1,2,3,4,5}

Vector<类型>vi(I,i+2);//得到i索引值为3以后的值

Vector< vector< int> >v; 二维向量//这里最外的<>要有空格。否则在比较旧的编译器下无法通过

clear();//清除容器中所以数据 不会 释放内存

data（）//打印 内存区块

push\_back(i); //尾部 install data

pop.back(i) //delete top data

capacity（）// 返回vector的实际存储空间的大小，这个一般大于或等于vector元素个数，注意与size()函数的区别

#include <algorithm> //使用 swap

erase()// 直接用 会产生 野指针问题 配合 remove\_if() 使用

vector<int> A; //创建一个空的的容器

vector<int> B(10,100); //创建一个10个元素,每个元素值为100

vector<int> C(B.begin(),B.end()); //使用迭代器,可以取部分元素创建一个新的容器

vector<int> D(C); //复制构造函数,创建一个完全一样的容器

//重载=

vector<int> E;

E = B;

\*

\*/

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class car

{

public:

int sp;

};

bool **cmp**(int a,int b){

return a > b;

}

bool **cmpp**(car& c)

{

return c.sp >100;

}

int **main**()

{

vector<int> vec;

vector<int> vc\_yq(5,6);

vector<int>:: iterator it;

vector<int>::iterator begin=vec.begin();

vector<int>::iterator end=vec.end();

#if 1 // ;like array[]

for(int i =0;i < 5;++i)

{

vec.push\_back(i); //尾部 install data 并没有头插

cout<<vec[i]<<",";

}

cout <<"push\_back end "<<endl;

for(int a = 0;a < 5; a++)

{

//vec.pop\_back(); //尾删 例如 尾插 0-9 这里 尾删 4-9

cout<<vec[a]<<",";

}

/\*\* iterator use

1)只能读取容器中的元素，而不能修改

\*\*/

cout <<"pop\_back"<<endl;

// 遍历

for(it = vec.begin();it != vec.end();++it)

{

//\*it =3; error static

cout<< "iterator vec = "<<\*it<<endl;

}

for(int i =0 ; i < vec.size();i++) //size\_t

{

cout<< " at fu = "<< vec.at(i)<<endl;

}

cout << "=== 成员 交换==="<<endl;

reverse(vec.begin(), vec.end());//交换

for(it = vec.begin();it != vec.end();++it)

{

//\*it =3; error static

cout<< "reverse iterator vec = "<<\*it<<endl;

}

//从小到大

cout << "===ascend sort 从小到大==="<<endl;

sort(vec.begin(), vec.end());

for(it = vec.begin();it != vec.end();++it)

{

//\*it =3; error static

cout<< "sort iterator vec = "<<\*it<<endl;

}

//从小到大

cout << "===descend sort 从大到小==="<<endl;

sort(vec.begin(),vec.end(),cmp); // 按降序排序

for(it = vec.begin();it != vec.end();++it)

{

//\*it =3; error static

cout<< "descend iterator vec = "<<\*it<<endl;

}

cout << "=== vector交换==="<<endl;

cout<< " befor vec memory dir = "<< vec.data()<<endl;

cout<< " befor vc\_yq memory dir = "<< vc\_yq.data()<<endl;

vec.swap(vc\_yq);

cout<< " after vec memory dir = "<< vec.data()<<endl;

cout<< " after vc\_yq memory dir = "<< vc\_yq.data()<<endl;

for(it = vec.begin();it != vec.end();++it)

{

cout<< "swap iterator vec = "<<\*it<<endl;

}

for(it = vc\_yq.begin();it != vc\_yq.end();++it)

{

cout<< "swap iterator vc\_yq = "<<\*it<<endl;

}

// 查找

it = find( vec.begin() ,vec.end(),3);

if(it != vec.end())

{

cout<<"find = "<< \*it<<endl;

}

// use unique 去重前 先排序 //下面是 去重 两个相同的 元素

cout << "=== unique 使用 ==="<<endl;

int a[]={1,200,3,470,9,4,4,100,100,300,600};

vector<int> unq(a,a+11);

sort(unq.begin(),unq.end());

for(it = unq.begin();it != unq.end();++it)

{

cout<< " sort end = "<<\*it<<endl;

}

int t =0,count = 0;

for(auto i = unq.begin();i != unq.end();++i)

{

++t;

//cout<< " \*(unq.begin()+1) unique iterator unq = "<<\*(unq.begin()+t)<<" \*it == "<< \*it<< " unq.capacity() = " << unq.capacity()<< " unq.size() = "<<unq.size()<<endl;

cout<< " \*i = "<<\*i<<" \*(i+1) = "<<\*(i+1)<< " unq.size() = "<<unq.size() << " t ="<< t <<" count ="<<count<<endl;

if((\*i == \*(i + 1)) && (t < unq.size()-1))

{

i =unq.erase(i);

cout << "i ="<<\*i<<endl;

//i--;

++count;

//cout<< " \*(unq.begin()+1) unique iterator unq = "<<\*(unq.begin()+t)<<endl;

//cout << "unq.erase(it) "<<\*(unq.erase(it))<<endl;

}

//if(count)

//cout<< " erase unique iterator unq = "<<\*it<<" \*(it+1) = "<<\*(it+1)<<endl;

}

#if 0

for(t= 0;t<unq.size();++t)

{

//cout<< " \*(unq.begin()+1) unique iterator unq = "<<\*(unq.begin()+t)<<" \*it == "<< \*it<< " unq.capacity() = " << unq.capacity()<< " unq.size() = "<<unq.size()<<endl;

cout<< "unq[t] = "<<unq[t]<<" unq[t+1] = "<<unq[t+1]<<endl;

if((unq[t] == unq[t+1]) && (t < unq.size()-1))

{

//cout<< " \*(unq.begin()+1) unique iterator unq = "<<\*(unq.begin()+t)<<endl;

++count;

cout << "unq.erase(it) "<<\*(unq.erase(unq.begin()+t))<<endl;

}

//if(count)

//cout<< " erase unique iterator unq = "<<\*it<<" \*(it+1) = "<<\*(it+1)<<endl;

}

#endif

cout<< " =============== ===============" <<endl;

//remove\_if use

vector<car> cars;

cars.erase( std::remove\_if(std::begin(cars), std::end(cars), cmpp), std::end(cars) );

/\*

for(it = cars.begin();it != cars.end();++it)

{

cout<< " befor unique iterator unq = "<<\*it<<endl;

}

\*/

cout<< " =============== ===============" <<endl;

//unique(unq.begin(),unq.end());

for(it = unq.begin();it != unq.end();++it)

{

cout<< " after unique iterator unq = "<<\*it<<endl;

}

// 删除问题

/\*\*\*

if(Iter == v1.end()) //要控制迭代器不能超过整个容器

{

break;

}

\*\*/

// swap memory 释放操作

vec.clear();

cout<< " size vec memory dir = "<< vec.size()<<endl;

//cout<< " clear vec memory dir = "<< vec.clear()<<endl;

cout<< " capacity vec memory dir = "<< vec.capacity()<<endl;

cout << "=== vector release memory==="<<endl;

vector <int>().swap(vc\_yq); //和下面的 方式效果相同

{ //花括号 是 为了 退出 tmp 时自动析构

vector <int> tmp;

vec.swap(tmp);

}

cout<< " clear vc\_yq memory dir = "<< vc\_yq.data()<<endl;

cout<< " clear vec memory dir = "<< vec.data()<<endl;

for(it = vc\_yq.begin();it != vc\_yq.end();++it)

{

//\*it =3; error static

cout<< "swap iterator vc\_yq = "<<\*it<<endl;

}

#endif

return 0;

}

1.const :

const 与 #define 区别

1)const 有类型检测 后者没有

2)有些集成化的调试工具可以对 const 常量进行调试,但是不能对宏常量进行调试。

2.类中的const

1)企图在类声明中初始化 const 数据成员 错误

2)数据成员的初始化只能在类构造函数的初始化表中进行 A::A(int size) : SIZE(size) // 构造函数的初始化表

3)整个类中都恒定的常量呢?别指望 const 数据成员了,应该用类中 的枚举常量来实现：enum { SIZE1 = 100, SIZE2 = 200};

枚举常量不会占用对象的存储空间,它们在编译时被全部求值。枚举常量的缺点是: 它的隐含数据类型是整数,其最大值有限,且不能表示浮点数(如 PI=3.14159)。

return int(x + y); // 创建一个临时变量并返回它

应该：

int temp = x + y;

return temp;

引用：

(1)引用被创建的同时必须被初始化(指针则可以在任何时候被初始化)。

(2)不能有 NULL 引用,引用必须与合法的存储单元关联(指针则可以是 NULL)。

(3)一旦引用被初始化,就不能改变引用的关系(指针则可以随时改变所指的对象)

注意 当 数组 作为 函 数的 参数 进 行传 递时 , 该数 组自 动 退化 为同 类 型的 指针