

学生学籍管理系统 V1.001

开发文档

(第一版)

已在 Github 上开源！

前言：

非常感谢你下载本程序，该程序的源码和库文件以及库的源码一并上传至 Github。你可以下载它们并对其二次修改或引用到你的程序中去。

本文档将介绍该程序中所用的库的全部信息，包括常量、结构体、数据类型和函数一一做了详细的说明。同时为了避免使用时出现不必要的麻烦，该文档对一些函数方法有些提示和警告。

Windows 用户请注意，本程序没有对 Windows 平台提供相应版本，但是您可以通过修改源代码来编写基于 Windows 平台的版本。另外提醒:源代码的编码是 UTF-8，在打开时需要将其转换成 ANSI 编码来确保源代码正常显示。其次，该 C 源代码是基于 C99 标准编写的，请确认你的编译器是否支持 C99 标准。

编写:@非显亮之高光铃丰
最后编写时间:2021-02-09

目录

定义速查表.....	1
常量(宏)	1
数据类型	1
结构体	1
详解	2
常量(宏)	2
<i>HEAD_LONG</i>	2
<i>MEM_POOL_MAX</i>	2
结构体	2
<i>FileInfo</i>	2
<i>StudentData</i>	3
函数	4
文件管理类.....	4
new_file	4
open_file	5
open_fileMem	5
save_file.....	6
get_fileinfo	6
write_FileHead	7
out_Data	8
get_HeadMembers.....	9
get_HeadVersion	9
get_HeadApplInfo	10
get_HeadArgs	10
数据操作类.....	11
add_NewData	11
add_NewDataMem	11
clear_Data.....	12
clear_DataMem	12

list_Data	13
del_Data.....	13
del_DataMem.....	14
search_DataDo	14
edit_DataDo.....	15
edit_Data	15
refresh_Data	16
print_DataInfo	16
search_Data_IDnumber.....	17
search_Data_Name.....	17
search_Data_School	18
内存管理类.....	18
deal_DelData	18
get_MemPool.....	19
get_MemMembers.....	20
clear_MemPool.....	20
get_MemInfo.....	21
编写实例	22
创建你的链表 STDATAP.....	22
创建时间输出指针 TIME_COUT.....	22
定义 GLOBAL_MEMBERS.....	23
定义*FILEP.....	23
定义 MEM_POOL（内存池）	23
学生学籍管理系统-主界面源代码	24
附录	26
静态链接库的使用(.A)	26

定义速查表

常量(宏)

常量名称	值	详解
HEAD_LONG	124	P3
MEM_POOL_MAX	10	P3

数据类型

名称	原型
StudentDataSharp	struct StudentData *
FileInfoSharp	struct FileInfoSharp *

结构体

结构体名称	详解
RegDate	P4
StudentData	P4
FileInfo	P4

详解

常量(宏)

HEAD_LONG

头文件:stdatModels.h

文件中的方法:

```
#define HEAD_LONG 124
```

注释:该常量是代表所保存文件的头部信息长度

MEM_POOL_MAX

头文件:stdatModels.h

文件中的方法:

```
#define MEM_POOL_MAX 10
```

注释:该常量是内存池可存放的最大值，默认为 10

结构体

FileInfo

头文件:stdatModels.h

文件中的方法:

```
struct FileInfo{
    char BuildSoftware[50];
    float VersionInfo;
    char Args[50];
    int year;
```

```
    int mon;  
    int mday;  
    int members;  
};
```

StudentData

头文件:stdatModels.h

文件中的方法:

请注意:该结构体是嵌套结构体

```
struct RegDate{  
    int year;  
    int mon;  
    int mday;  
};  
  
struct StudentData{  
    char name[50];  
    char school[50];  
    unsigned long int IDnumber;  
    int memberID;  
    struct RegDate date;  
    StudentDataSharp next;  
};
```

函数

文件管理类

简要:该类是文件与内存之间的操作类。在此类中你可以对文件进行编辑/重构/文件信息读写等等操作。同时该类提供了“open_fileMem”的函数,它具备“内存回收机制”,可以帮助你提高程序在打开文件的执行效率。

new_file

头文件:stdatModels.h

库文件:libfileManage.a

函数原型:

```
int new_file(char *FileName);
```

该函数会根据所传递的字符变量创建一个以这个字符变量为名字的文件。

注意:如果当前目录有同名的文件的话,将会询问是否要覆盖该文件,覆盖时程序将会先前的文件抹掉再创建新文件

其次,该函数的存储方式是二进制,如果要读取该内容可以使用open_file 函数

该函数的返回值没有任何意义。

使用实例:

例 1.

```
new_file( "test.dat" );
```

例 2.

```
char file[50];
```

```
strcpy(file, "test.dat" );
```

```
new_file(file);
```


open_file

头文件: stdatModels.h

库文件: libfileManage.a

函数原型:

```
int open_file(char *FileName);
```

通过所传递的字符变量来打开以这个字符变量为名的文件，当目标文件被成功打开后函数会将目标文件的信息读取到内存中。

该函数返回的值没有任何意义。

使用实例:

例 1.

```
open_file( "test.dat" );
```

例 2.

```
strcpy(file, "test.dat" );
```

```
open_file(file);
```

open_fileMem

头文件: stdatModels.h

库文件: libfileManage.a

函数原型:

```
int open_fileMem(char *FileName);
```

通过所传递的字符变量来打开以这个字符变量为名的文件，当目标文件被成功打开后函数会将目标文件的信息读取到内存中。

该函数具备“内存回收机制”，调用时将会检查内存池中是否有可用的空余内存并将其回取作为新数据的空间。

该函数返回的值没有任何意义。

使用实例:

例 1.

```
open_fileMem( "test.dat" );
```

例 2.

```
strcpy(file, "test.dat" );
```

```
open_fileMem(file);
```

save_file

头文件:stdatModels.h

库文件:libfileManage.a

函数原型:

```
int save_file(char *FileName);
```

此函数将内存中保存的信息写入到以传递字符变量的值为名的文件中。

如果同目录下有同名的文件，程序将会询问你是否要覆盖该文件。

该函数的返回值没有任何意义。

使用实例

例 1.

```
save_file( "test.dat" );
```

例 2.

```
strcpy(file, "test.dat" );
```

```
save_file(file);
```

get_fileinfo

头文件:stdatModels.h

库文件:libfileManage.a

函数原型:

```
int get_fileinfo(char *FileName);
```

通过所传递的字符变量来打开以这个字符变量为名的文件，并将该文件的头部信息打印至屏幕。

该函数返回的值没有任何意义。

使用实例：

例 1.

```
get_fileinfo( "test.dat" );
```

例 2.

```
strcpy(file, "test.dat" );
```

```
get_fileinfo(file);
```

write_FileHead

头文件: stdatModels.h

库文件: libfileManage.a

函数原型:

```
int write_FileHead(FILE *temp_file,
                   int members,
                   char *softwareInfo,
                   char *fileArgs,
                   float softwareVersion);
```

参数 temp_file: 传递当前打开文件的文件指针。

参数 members: 在文件头部中写入或修改文件中 members 的值(在程序中解释为“当前已注册人数”)。

参数 softwareInfo: 在文件头部中写入或修改文件中 BuildSoftware

的值(在程序中解释为“应用程序描述”)。

参数 fileArgs: 在文件头部中写入或修改文件中 Args 的值(在程序中解释为“文件描述”)。

参数 softwareVersion: 在文件头部中写入或修改文件中 VersionInfo 的值(在程序中解释为“应用版本”)。该参数为浮点型，应用内最大可支持 4 位输出。

该函数会根据传递的文件指针，在目标文件中重写“FileInfo”，然后根据所传递进来的其塔参数来修改“FileInfo”中相应的值。

该函数将返回目标文件指示器的当前位置。

注意：所传递进来的文件指针必须是带有“W”属性的。

使用实例

例 1.

```
FILE *target;
target=fopen( "test.dat" , "w" );
write_FileHead(target,0,
                "这是我的程序" ,
                " 这是我的程序写出的文件 " ,
                1.0001);
```

out_Data

头文件:stdatModels.h

库文件:libfileManage.a

函数原型:

```
int out_Data(char *OutFile);
```

该函数将链表的信息以 txt 格式输出到以传递字符变量为名的文件中。如果有同名文件，程序会询问是否要覆盖该文件。

该函数返回的值没有任何意义。

使用实例:

例 1.

```
out_Data( "test.dat" );
```

例 2.

```
strcpy(file, "test.dat" );
```

```
out_Data(file);
```

get_HeadMembers

头文件:stdatModels.h

库文件:libfileManage.a

函数原型:

```
int get_HeadMembers(FILE *tempfile);
```

此函数会根据传递的目标文件指针来读取文件头部的“members”项的信息，并将其返回。

注意:传递的文件指针必须带有“r”属性。

使用实例

```
int temp;
FILE *target;
target=fopen( "test.dat" , "r" );
temp=get_HeadMembers(target);
```

get_HeadVersion

头文件:stdatModels.h

库文件:libfileManage.a

函数原型:

```
float get_HeadVersion(FILE *tempfile);
```

此函数会根据传递的目标文件指针来读取文件头部的应用版本信息，然后以 float 的形式将其返回

注意:传递的文件指针必须带有“r”属性。

使用实例

```
float temp;
FILE *target;
target=fopen( "test.dat" , "r" );
```

```
temp=get_HeadVersion(target);
```

get_HeadAppInfo

头文件:stdatModels.h

库文件:libfileManage.a

函数原型:

```
char *get_HeadAppInfo(FILE *tempfile);
```

此函数会根据传递的目标文件指针来读取文件头部的应用程序描述信息，并将其以字符地址的形式返回。

注意:传递的文件指针必须带有“r”属性。

使用实例

```
FILE *target;
target=fopen( "test.dat" , "r" );
printf( "应用程序描述 : %s" ,get_HeadAppInfo(target));
```

get_HeadArgs

头文件:stdatModels.h

库文件:libfileManage.a

函数原型:

```
char *get_HeadArgs(FILE *tempfile);
```

此函数会根据传递的目标文件指针来读取文件头部的文件描述信息，并将其以字符地址的形式返回。

注意:传递的文件指针必须带有“r”属性。

使用实例

```
FILE *target;
target=fopen( "test.dat" , "r" );
printf( "文件描述 : %s" ,get_HeadArgs(target));
```

数据操作类

该类是用于操作内存中用户定义的链表数据，你可以使用该类提供的函数来为你的链表进行添加/删除/查找/编辑等操作。另外此类的函数还包括了具备”内存回收机制”的操作函数，可以使你的程序执行效率更高同时还能减少运行时产生的内存碎片。

add_NewData

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int add_NewData(void);
```

该函数的功能是将用户键入的信息添加到数据链表中。

该函数的返回值没有任何意义。

使用实例

```
add_NewData();
```

add_NewDataMem

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int add_NewDataMem(void);
```

该函数的功能是将用户键入的信息添加到数据链表中。

该函数具备”内存回收机制”的功能，该函数将申请内存空间之前访问内存池是否有多余的内存空间，如果有函数将从中取出一块来存放新数据，反之则会向系统中申请内存空间。

该函数的返回值没有任何意义。

使用实例

```
add_NewDataMem();
```

clear_Data

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int clear_Data(void);
```

该函数的功能是清空用户定义的链表中的所有数据。

该函数的返回值没有任何意义。

使用实例

```
clear_Data();
```

clear_DataMem

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int clear_DataMem(void);
```

该函数的功能是清空用户定义的链表中的所有数据。

该函数具备”内存回收机制”，函数在释放链表数据之前会访问内存池中是否有“空位”，如果有的话则将当前的空间存放到内存池中以便下次备用，如果没有则交给系统释放掉。

该函数的返回值没有任何意义。

使用实例

```
clear_DataMem();
```

list_Data

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int list_Data(void);
```

该函数的功能是将链表中所存在的数据依次打印出来。

如果链表中没有数据则返回提示信息。

该函数的返回值没有任何意义。

使用实例

```
list_Data();
```

del_Data

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int del_Data(void);
```

该函数的功能是通过用户键入”memberID”来删除指定链表信息。

该函数的返回值没有任何意义。

使用实例

```
del_Data();
```

del_DataMem

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int del_DataMem(void);
```

该函数的功能是通过用户键入”memberID”来删除指定链表信息

该函数具备“内存回收机制”，释放目标数据之前会访问内存池中是否有可用的“空位”，如果有则存放在内存池中，没有则交给系统释放掉。

该函数的返回值没有任何意义。

使用实例

```
del_DataMem();
```

search_DataDo

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int search_DataDo(void);
```

本函数是”搜索”功能的调用入口，调用后将指导用户选择一种搜索方式来搜索目标数据。其操作内部请参考：

search_Data_IDnumber、search_Data_School、search_Data_Name 函数。

该函数的返回值没有任何意义。

使用实例

```
search_DataDo();
```

edit_DataDo

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int edit_DataDo(void);
```

该函数的功能是通过用户键入”memberID”来定位指定的链表信息。将所得到的链表信息的指针传递给 edit_Data(), 最后通过 edit_Data() 方法将其修改。其内部最终的实现请参考: edit_Data() 函数。

该函数的返回值没有任何意义。

使用实例

```
edit_DataDo();
```

edit_Data

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int edit_Data(StudentDataSharp temp);
```

修改传递目标指针的具体数据值。

该函数的返回值没有任何意义。

使用实例

```
StudentDataSharp temp;
If((temp=(StudentSharp)malloc(sizeof(struct
StudentData)))==NULL)
{
    fprintf(stdree,” 申请内存出错!” );
```

```
exit(EXIT_FAILURE); //防止内存泄漏，强制关闭程序
}

edit_Data(temp);
//启动函数，按照提示修改即可
```

refresh_Data

头文件: stdatModels.h

库文件: libDataOperan.a

函数原型:

```
int refresh_Data(void);
```

刷新链表中成员的 membersID 信息。

建议你执行完添加/删除等操作时调用此方法。保持成员 ID 信息的清晰。

add_NewData、add_NewDataMem、del_Data、del_DataMem、open_file、open_fileMem，执行这些函数时它们会自动调用本方法。

该函数的返回值没有任何意义。

使用实例

```
refresh_Data();
```

print_DataInfo

头文件: stdatModels.h

库文件: libDataOperan.a

函数原型:

```
void print_DataInfo(StudentDataSharp temp);
```

打印传递目标指针的数据。

使用实例

```

    StudentDataSharp temp;
    If((temp=(StudentSharp)malloc(sizeof(struct
StudentData)))==NULL)
{
    fprintf(stderr," 申请内存出错！" );
    exit(EXIT_FAILURE); //防止内存泄漏，强制关闭程序
}

    edit_Data(temp);
    //编辑完成后调用 print_DataInfo 打印数据验证
    print_DataInfo(temp);

```

search_Data_IDnumber

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int search_Data_IDnumber(void);
```

根据数据中的” IDnumber” 属性来查找数据

该函数的返回值没有任何意义。

使用实例

```
search_Data_IDnumber();
```

search_Data_Name

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int search_Data_Name(void);
```

根据数据中的” Name” 属性来查找数据

该函数的返回值没有任何意义。

使用实例

```
search_Data_Name();
```

search_Data_School

头文件:stdatModels.h

库文件:libDataOperan.a

函数原型:

```
int search_Data_School(void);
```

根据数据中的”School”属性来查找数据

该函数的返回值没有任何意义。

使用实例

```
search_Data_School();
```

内存管理类

本程序引进了一项机制——内存回收机制，该机制可以提升程序的运行效率，节省程序在运行过程中产生的内存消耗。无论是文件管理类还是数据操作类，它们的内存回收机制都是基于本类制作的。因此本类提供的函数在开发过程中起到很重要的作用。

deal_DelData

头文件:stdatModels.h

库文件:libMemManage.a

函数原型:

```
void deal_DelData(StudentDataSharp target);
```

将所需要处理的目标空间通过该函数处理。函数接收到目标数据后先去访问 Mem_Pool(内存池)，如果当前存入的数量达到 MEM_POOL_MAX 则交给系统释放。反之将目标存入内存池中。

使用实例

```
StudentDataSharp temp;
If((temp=(StudentSharp)malloc(sizeof(struct
StudentData)))==NULL)
{
    fprintf(stderr," 申请内存出错！" );
    exit(EXIT_FAILURE); //防止内存泄漏，强制关闭程序
}

deal_DelData(temp);
```

get_MemPool

头文件:stdatModels.h

库文件: libMemManage.a

函数原型:

```
StudentDataSharp get_MemPool(void);
```

访问内存池是否有可用的内存空间，如果有则返回该空间的指针，如果没有则返回 NULL。

提示:使用此函数建议与 get_MemMembers 方法一起搭配，能保障程序的正常运行

使用实例

```
StudentDataSharp target;
If(get_MemMember()!=0){
    //如果内存池不为空则返回其中的一个指针
```

```
        Target=get_MemPool();
    }
    else{
        printf(“内存申请失败！”);
    }
```

get_MemMembers

头文件:stdatModels.h

库文件:libMemManage.a

函数原型:

```
int get_MemMembers(void);
```

获取当前内存池成员的个数，并以 int 的形式返回

使用实例

```
int numbers;
numbers=get_MemMembers();
//其中 numbers 的值为内存池中的成员个数
```

clear_MemPool

头文件:stdatModels.h

库文件:libMemManage.a

函数原型:

```
int clear_MemPool(void);
```

清除内存池中的所有成员，当程序录入新信息时需要向系统申请内存空间。

该函数的返回值没有任何意义。

使用实例

```
clear_MemPool();
```


get_MemInfo

头文件:stdatModels.h

库文件:libMemManage.a

函数原型:

```
void get_MemInfo(void);
```

打印当前内存池的数据。

使用实例

```
get_MemInfo();
```

运行结果:

```
-----
```

当前内存池信息:

总大小:10

已占用:1

缓存大小:136 字节

编写实例

创建你的链表 stdatap

如果你查看过本程序的源代码你可能会发现，该程序所存储数据的模型是单链表模型。但是比较遗憾的是本程序所提供的库没有达到真正的定制化。这意味着在你定义链表名称时需要遵循库的定义。库中的函数会自动定位到该链表然后处理其数据。

注意:该链表的作用域必须是全局。

链表基于的结构体指针名称必须为”stdatap”

定义方法:

```
#include"stdatModels.h"
StudentDataSharp stdatap=NULL;
```

特别提醒:建议你定义完链表后手动把链表赋值为 NULL，这可以使程序代码更严谨。

创建时间输出指针 time_cout

该指针将用于写入文件，添加数据时获取时间信息。指针中存储着当前运行计算机的时间信息，这意味着时间的值由运行的计算机所决定。

注意:该指针的作用域必须是全局。

其内部的赋值操作你可以在函数中进行。

该指针变量名必须为” time_cout”

定义方法:

```
#include"stdatModels.h"

struct tm *time_cout;
void get_time(void)
```

```
{
    //加载时间
    time_t t;
    time(&t);
    time_cout=localtime(&t);
}
```

定义 global_members

这是一个 int 型的全局变量，该变量存储着链表成员的个数。以及存储文件时 save_file 会根据 global_members 的值来确定写入 members 的值。

定义方法：

```
int global_members=0;
```

定义*filep

这是一个非常重要的全局文件指针，它记录着文件打开的信息，库中对于文件管理的函数都要使用此指针。

定义方法：

```
FILE *filep=NULL;
```

定义 Mem_Pool（内存池）

这是一个结构体指针，其类型与 stdatap 相同，用于存放删除/添加修改 stdatap 时为其提供的容器。

提示:该指针变量是”内存回收机制”所要用到的。如果你的程序中没有用到与”内存回收机制”相关的函数，则可以不定义该指针。

定义方法：

```
StudentDataSharp Mem_Pool=NULL;
```

学生学籍管理系统-主界面源代码

本部分是 main.c 中的代码，该文件已经上传至 github。（main.c 中的代码已经引用了与”内存回收机制”相关的函数）

```
#include "stdatModels.h" ///综合头文件
///设置内存池，和文件标
StudentDataSharp stdatap=NULL;
StudentDataSharp Mem_Pool=NULL;
int global_members=0;
///设置文件指针
FILE *filep=NULL;
///时间全局变量
struct tm *time_cout;
int main(void)
{
    char chose[FILENAME_MAX];
    {
        //加载时间
        time_t t;
        time(&t);
        time_cout=localtime(&t);
    }
    char FileName[FILENAME_MAX];
    strcpy(FileName,"stData.data");
    system("cat help.txt");
    for(;;){
        printf("\n->>");
        scanf("%s",chose);
        if(strcmp(chose,"new")==0)
            new_file(FileName);
        else if(strcmp(chose,"open")==0)
```

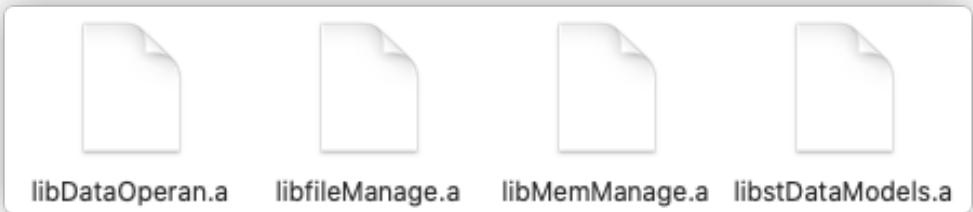
```
        open_fileMem(FileName);
    else if(strcmp(chose,"save")==0)
        save_file(FileName);
    else if(strcmp(chose,"info")==0)
        get_fileinfo(FileName);
    else if(strcmp(chose,"add")==0)
        add_NewDataMem();
    else if(strcmp(chose,"edit")==0)
        edit_DataDo();
    else if(strcmp(chose,"search")==0)
        search_DataDo();
    else if(strcmp(chose,"del")==0)
        del_DataMem();
    else if(strcmp(chose,"list")==0)
        list_Data();
    else if(strcmp(chose,"clear")==0)
        clear_DataMem();
    else if(strcmp(chose,"out")==0)
        out_Data("StudenInfo.txt");
    else if(strcmp(chose,"getmeminfo")==0)
        get_MemInfo();
    else if(strcmp(chose,"exit")==0)
        exit(1);
    else
        printf("\n 找不到相关指令，请重试");
}
return 0;
}
```

附录

静态链接库的使用(.a)

上传源代码时已将库文件编译为静态链接库，主要用于提高编译效率而使用的。本部分将介绍如何使用这些文件。
(注意:实机演示的是 macos，终端部分操作可能与其它终端不一样，本演示只供参考。)

静态链接库主要包括以下文件:



文件名	对应类
libDataOperan.a	数据操作类
libfileManage.a	文件管理类
libMemManage.a	内存管理类
libstDataModels.a	包含以上的所有类

当编译时你需要将库文件和 stdatModels.h 文件与你的源码放在一起。(在使用本库中的函数时你需要在你的代码中包含 stdatModels.h 文件，另外你不必将所有库文件一起复制，这里提供了 libstDataModels.a 文件，是将之前所有的库文件打包在了一起，你可以将它拿出来单独使用。)

在这里我们演示一下调用库中的 get_MemInfo 函数。

新建一个”test.c”文件,并输入下方代码:

```
#include"stdatModels.h"
```

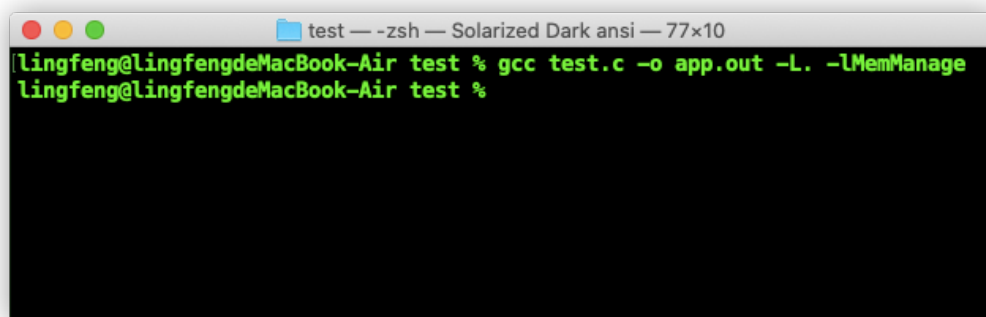
```
StudentDataSharp Mem_Pool=NULL;
int main()
{
    get_MemInfo();
    return 0;
}
```

将 test.c、stdatModels.h、libMemManage.a 文件拷贝至同一目录。

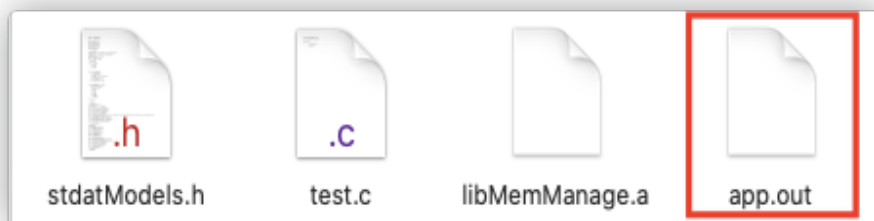


打开系统的”终端”应用并输入下方的代码

```
gcc test.c -o app.out -L. -lMemManage
```



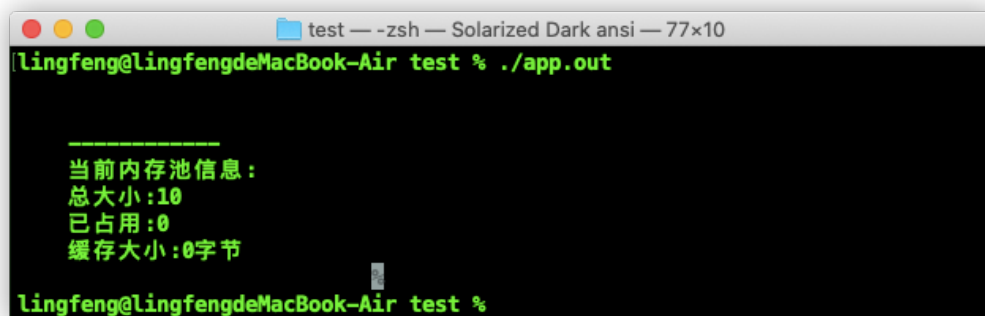
编译成功后你将会看到目录下出现 app.out 文件。



在终端输入下方指令以验证:

```
./app.out
```

最后查看其结果，程序运行正常。



```
lingfeng@lingfengdeMacBook-Air test % ./app.out

-----
当前内存池信息:
总大小:10
已占用:0
缓存大小:0字节

lingfeng@lingfengdeMacBook-Air test %
```

The image shows a terminal window titled 'test - zsh - Solarized Dark ansi - 77x10'. The user 'lingfeng' is at the 'lingfengdeMacBook-Air' machine in the 'test' directory. They run the command './app.out'. The output shows a separator line '-----' followed by the text '当前内存池信息:' (Current memory pool information:), then three lines of data: '总大小:10' (Total size: 10), '已占用:0' (Used: 0), and '缓存大小:0字节' (Cache size: 0 bytes). The prompt returns to 'lingfeng@lingfengdeMacBook-Air test %'.

这是尾页

至此本文档内容到此结束

如有反馈请通过下方的联系方式联系我:

miuul@outlook.com
