知识点1【内存的分区】（了解）

知识点2【变量的了解】（了解）

1、普通局部变量

2、普通全局变量

3、静态局部变量

4、静态全局变量

知识点3【全局函数和静态函数】（了解）

1、全局函数（函数默认 都为全局函数）

2、静态函数（加static修饰的函数）

3、案例1

知识点4【gcc 编译的过程】（了解）

知识点5【头文件包含】（了解）

知识点6【#define 宏】（了解）

宏 尽量 大写 和 普通变量 区分开

1、不带参数的宏

2、带参数的宏

知识点7【条件编译】（了解）

案例1：代码裁剪

案例2：代码裁剪

案例3：条件编译 用于防止头文件重复包含　（条件编译方式 推荐）

案例4：条件编译 用于防止头文件重复包含　（windows的方式）

知识点8【动态库和静态库】（了解）

知识点1【内存的分区】（了解）

进程：可执行文件 从运行到结束 整个动态的过程 就叫进程。（占内存空间）

在32位平台　每一个进程 占4G空间（虚拟空间）

知识点2【变量的了解】（了解）

1、普通局部变量

定义形式：在{}里面定义的 普通变量 叫普通局部变量

```c
void test02()
{//复合语句
    int num = 0;//num 局部变量
    {
        int data = 0;//data 局部变量
    }
}
```

作用范围：所在的{}复合语句之间有效

生命周期：所在的{}复合语句之间有效

存储区域：栈区

注意事项：

1、普通局部变量 不初始化 内容 不确定

2、普通局部变量 同名 就近原则（尽量别同名）

```c
void test02()
{                   //复合语句
    int num = 10; //num 局部变量
    {
        int num = 20; //data 局部变量
        printf("%d\n", num);//20 就近原则
    }
    printf("%d\n", num);//10
}
```

2、普通全局变量

定义形式：在函数外定义的普通 变量

```c
int data = 10;//普通全局变量
void test03()
{

}
```

作用范围：当前源文件以及其他源文件 都有效。

生命周期：整个进程。

存储区域：全局区

注意事项：

1、全局变量不初始化 内容为0

2、全局变量 和 局部变量 同名 优先选择局部变量。

```
int data = 10; //普通全局变量
void test03()
{
    int data = 20;//普通局部变量
    printf("data = %d\n", data);//20
}
```

3、其他源文件 使用全局变量 必须对全局变量 进行extern声明。(变量的使用所在的源文件)

extern 声明外部可用。该变量或函数 来自于其他源文件。

01_fun.c

```
//extern 声明data为int类型 来之其他源文件
extern int data;
void add_data(void)
{
    data = data+100;
    return ;
}
```

01_code.c

```
extern void add_data(void);
int data = 10; //普通全局变量
void test03()
{
    add_data();
    printf("data = %d\n", data);
}
```

3、静态局部变量

定义形式：在{}加static定义的局部变量 就叫静态局部变量

```
void test04()
{
    int data1 = 10;//普通局部变量
    static int data2 = 20;//静态局部变量
}
```

作用范围：所在的{}复合语句之间有效

生命周期：整个进程有效

存储区域：全局区

注意事项：

1、静态局部变量不初始化 内容为0

2、静态局部变量 整个进程 都存在（第一次定义有效）

```c
void fun04()
{
    static int num = 10;
    num++;
    printf("num = %d\n", num);
}
void test04()
{
    fun04();
    fun04();
    fun04();
    fun04();
}
```

```
edu@edu: ~/work/c/day04
edu@edu:~/work/c/day04$ sudo gcc 01_code.c
edu@edu:~/work/c/day04$ ./a.out
num = 11
num = 12
num = 13
num = 14
edu@edu:~/work/c/day04$
```

4、静态全局变量

定义形式：在函数外 加static修饰定义的变量 就是静态全局变量

```c
int data3 = 10;//普通全局变量
static int data4 = 20;//静态全局变量
void test05()
{

}
```

作用范围：只能在当前源文件使用 不能在其他源文件使用。

生命周期：整个进程

存储区域：全局区

注意事项：

1、静态全局变量不初始化 内容为0

2、静态全局变量 只能在当前源文件使用

知识点3【全局函数和静态函数】（了解）

1、全局函数（函数默认 都为全局函数）

全局函数：在当前源文件 以及其他源文件 都可以使用

如果其他源文件使用需要 extern对全局函数 进行声明

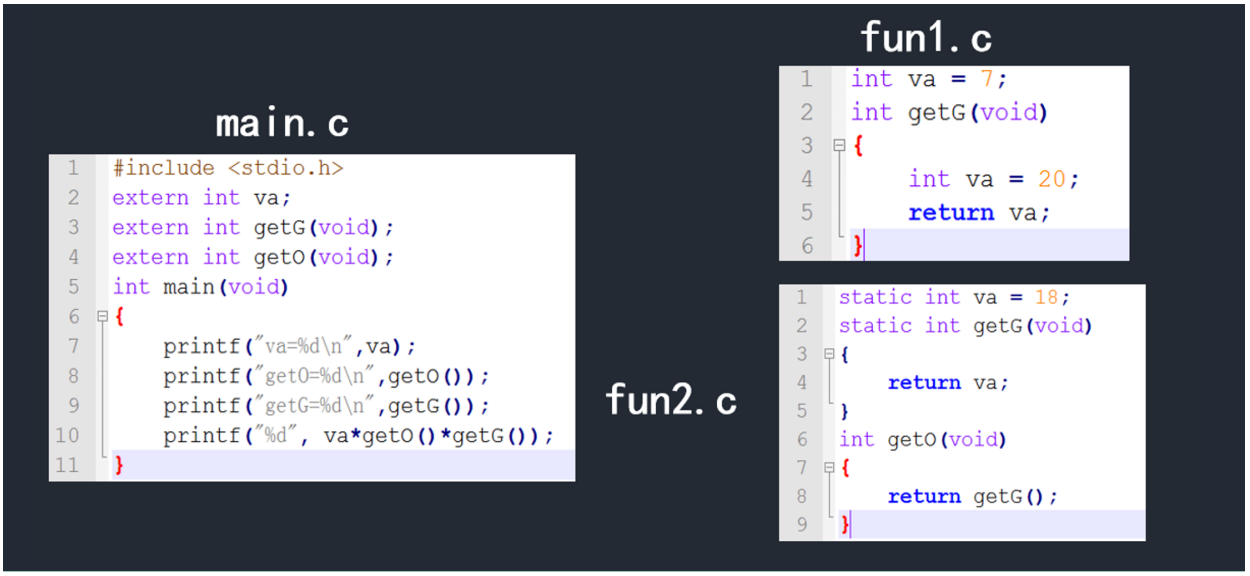
2、静态函数（加static修饰的函数）

静态成员 只能在当前源文件 用

```c
static void func(void)
{

}
```

3、案例1



知识点4【gcc 编译的过程】（了解）

```c
#include <stdio.h>
#define PI 3.14
int main(int argc, char const *argv[])
{
    //打印IP的值
    printf("PI = %lf\n", PI);
    return 0;
}
```

编译的过程：预处理、编译、汇编、链接

预处理：宏替换、删除注释、头文件包含、条件编译 -E （不会报语法错误）

```
gcc -E hello.c -o hello.i 1、预处理
```

编译：将预处理后的文件 编译成 汇编文件 (报语法错误)

```
gcc -S hello.i -o hello.s 2、编译
```

汇编：将汇编文件 生成 二进制文件

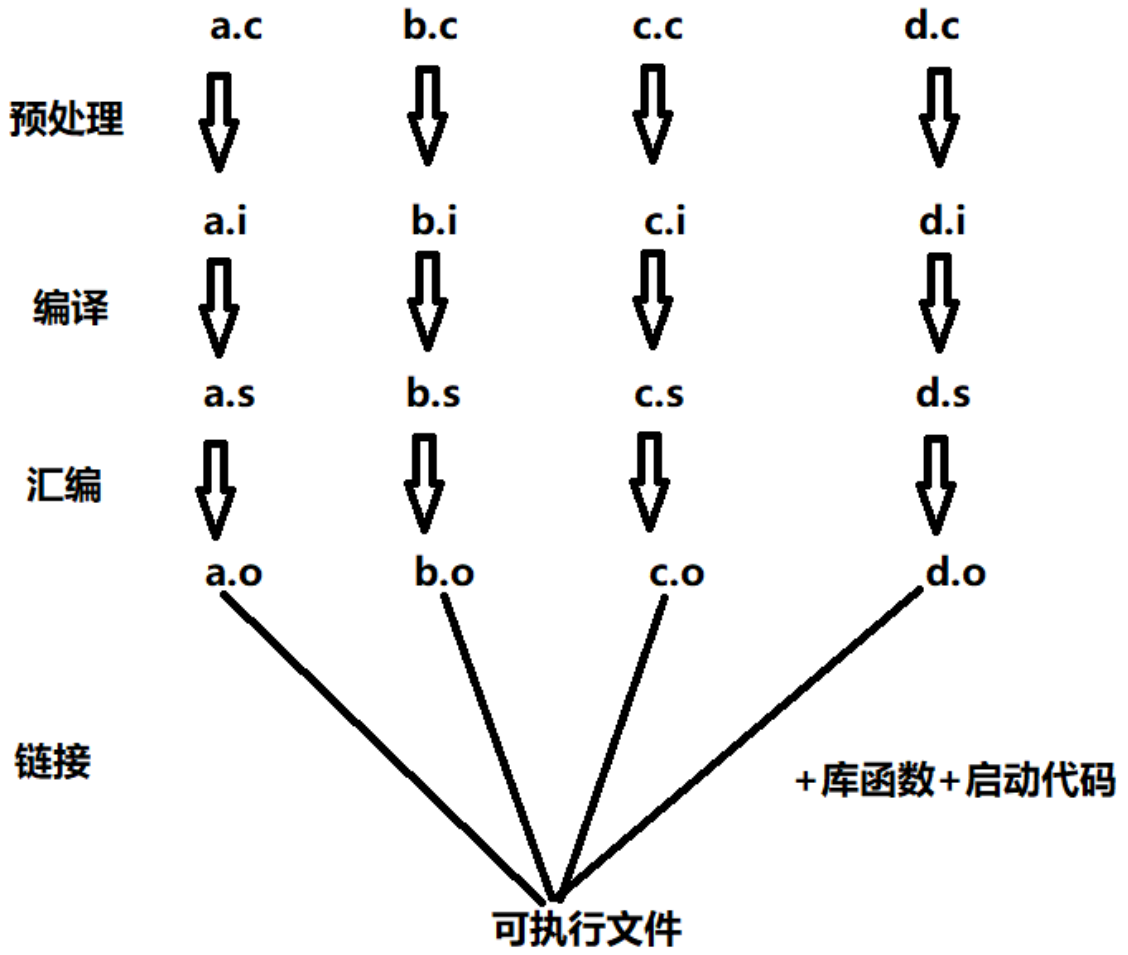
```
gcc -c hello.s -o hello.o 3、汇编
```

链接：将工程的二进制文件 +库函数+启动代码 生成可执行文件

```
gcc hello.o -o hello_elf 4、链接
```

一步到位：

```
gcc main.c -o main
gcc main.c
```



知识点5【头文件包含】（了解）

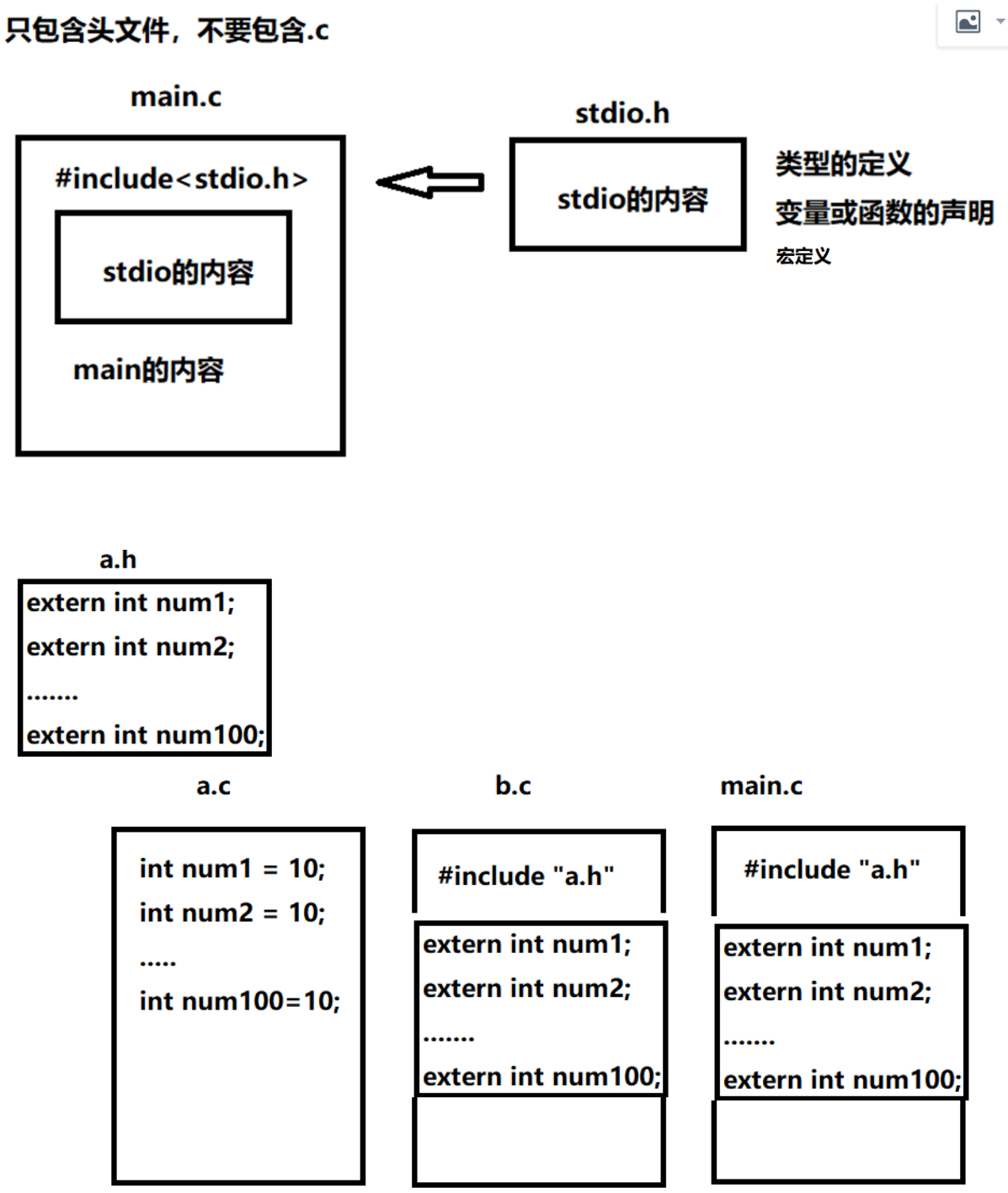
头文件包含：在预处理 结果 将头文件的内容 原封不动的 包含在 目的文件中

#include <head.h> 建议<>包含系统头文件

<>从系统指定目录 寻找head.h头文件

#include "head.h" 建议""包含自定义头文件

“” 先从当前目录 寻找head.h头文件 如果找不到 再到系统指定的目录下寻找



知识点6【#define 宏】（了解）

使用关键字 define 定义 叫宏

```
#define PI 3.14 （宏定义）
```

在预处理结果 使用3.14 替换所有出现PI的位置 （宏展开）

注意：不要再宏后加;分号

```
#define PI 3.14;
if(PI>3.0)
{
  语句;
}
```

宏 尽量 大写 和 普通变量 区分开

1、不带参数的宏

```
#define PI 3.14 （宏定义）
#define MY_STR "hello world"
#define N 100
```

宏的作用范围：是从定义处开始 到 当前文件结束 都有效

#undef可以结束宏的作用域

宏 没有归属 只在当前源文件 有效

2、带参数的宏

```
#define MY_MUL(a, b) a*b

printf("%d\n", MY_MUL(10,20) );//10*20
//printf("%d\n", 10*20 );
```

1、宏的参数不能有类型

```
#define MY_MUL(int a, int b) a*b //error
```

2、宏不能保证参数的完整性

```
#define MY_MUL(a, b) a *b
void test01()
{
    printf("%d\n", MY_MUL(10, 20));           //10*20
    printf("%d\n", MY_MUL(10 + 10, 20 + 20)); //10 + 10*20 + 20 == 230
}
```

MY_MUL(10 + 10, 20 + 20)的结果是230 而不是800

可以使用()的形式 让带参数的宏 具备一定的完整性

```
#define MY_MUL(a, b) ((a) *(b))
```

```
#include <stdio.h>
#define MY_MUL(a, b) a *b
#define MY_MUL2(a, b) ((a) * (b))
void test01()
{
    printf("%d\n", MY_MUL(10, 20));           //10*20
    printf("%d\n", MY_MUL(10 + 10, 20 + 20)); //10 + 10*20 + 20 == 230
```

```
    printf("%d\n", MY_MUL2(10 + 10, 20 + 20));//((10 + 10) * (20 + 20))==800
}
```

3、宏不能作为结构体、类的成员

4、案例：

```
#define MY_MUL(a, b) a *b
#define MY_MUL2(a, b) ((a) * (b))
printf("%d\n", MY_MUL(MY_MUL2(10+10, 20+20), MY_MUL(10+10, 20+20)));
MY_MUL2(10+10, 20+20) *MY_MUL(10+10, 20+20)
((10+10) * (20+20))*10+10 *20+20
20*40*10+10*20+20== 8000+200+20== 8220
```

5、带参宏（宏函数）和带参函数的区别

带参宏被调用多少次就会展开多少次，执行代码的时候没有函数调用的过程，不需要压栈弹栈。所以带参宏，是浪费了空间，因为被展开多次，节省时间。

带参函数，代码只有一份，存在代码段，调用的时候去代码段取指令，调用的时候要，压栈弹栈。有个调用的过程。

所以说，带参函数是浪费了时间，节省了空间。

带参函数的形参是有类型的，带参宏的形参没有类型名。

知识点7【条件编译】（了解）

1、判断存在
#ifdef XXX
语句1;
#else
语句2;
#endif

如果定义了宏XXX编译语句1，否则语句2

2、判断不存在
#ifndef XXX
语句1;
#else
语句2;
#endif

如果没有定义宏XXX编译语句1，否则语句2

3、判断是否成立
#if XXX
语句1;
#else
语句2
#endif

如果XXX为真编译语句1 否则编译语句2

案例1：代码裁剪

```
#include <stdio.h>
#include <string.h>
void test01()
{
    char str[128] = "";
    fgets(str, sizeof(str), stdin);
    str[strlen(str) - 1] = '\0';

    int i = 0;
     while (str[i] != '\0')
     {
```

```c
#ifdef MAX_TO_MIN
        //将大写字母转换小写字母
        if (str[i] >= 'A' && str[i] <= 'Z')
        {
            str[i] += 32;
        }
#else
        //将小写字符转换成大写字母
        if (str[i] >= 'a' && str[i] <= 'z')
        {
            str[i] -= 32;
        }

#endif
        i++;
    }

    printf("%s\n", str);

    return;
}
int main(int argc, char const *argv[])
{
    test01();
    return 0;
}
```

```
edu@edu: ~/work/c/day04
edu@edu:~/work/c/day04$ sudo gcc 04_code.c
edu@edu:~/work/c/day04$ ./a.out
hehe HAHA
HEHE HAHA
edu@edu:~/work/c/day04$ sudo gcc 04_code.c -D MAX_TO_MIN
edu@edu:~/work/c/day04$ ./a.out
hehe HAHA
hehe haha
edu@edu:~/work/c/day04$
```

给程序传递 宏

案例2：代码裁剪

```c
 int i = 0;
while (str[i] != '\0')
```

```
{
#if MAX_TO_MIN
    //将大写字母转换小写字母
    if (str[i] >= 'A' && str[i] <= 'Z')
    {
        str[i] += 32;
    }
#else
     //将小写字符转换成大写字母
     if (str[i] >= 'a' && str[i] <= 'z')
     {
         str[i] -= 32;
     }

#endif
     i++;
}
```

```
edu@edu:~/work/c/day04$ sudo gcc 04_code.c -D MAX_TO_MIN=0
edu@edu:~/work/c/day04$ ./a.out
hehe HAHA
HEHE HAHA
edu@edu:~/work/c/day04$ sudo gcc 04_code.c -D MAX_TO_MIN=1
edu@edu:~/work/c/day04$ ./a.out
hehe HAHA
hehe haha
edu@edu:~/work/c/day04$
```

案例3：条件编译 用于防止头文件重复包含　（条件编译方式 推荐）

```
#ifndef __05_A_H__
#define __05_A_H__

int num=10;

#endif
```

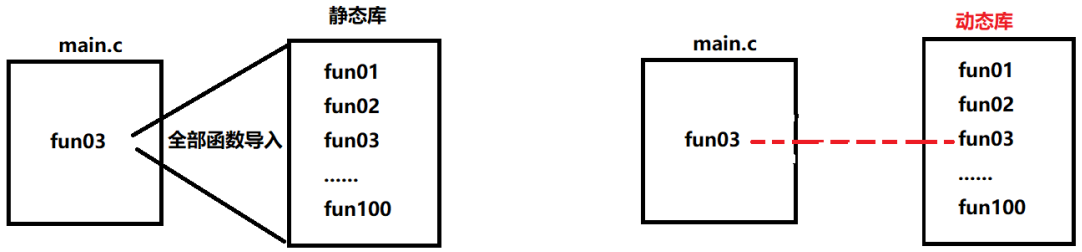
案例4：条件编译 用于防止头文件重复包含　（windows的方式）

```
#pragma once
int num=10;
```

知识点8【动态库和静态库】（了解）

1、库的介绍

库：将源文件生成 二进制文件 只需要链接 就可以生成可执行文件



链接阶段：不管项目调用了库中多少个函数 都是将静态库的所有函数导入项目

优点：

对库的依赖较小

缺点：

生成的可执行文件特点大

链接阶段：仅仅建立与动态库中fun03关系

运行阶段：才导入fun03的二进制

优点：

生成可执行文件小

缺点：

对库的依赖特别大

```
edu@edu:~/work/c/day04$ sudo gcc 05_code.c -o main01
[sudo] edu 的密码：
edu@edu:~/work/c/day04$ sudo gcc 05_code.c -o main02 -static
edu@edu:~/work/c/day04$ ls -lh main*
-rwxr-xr-x 1 root root 8.4K 7月  26 16:36 main
-rwxr-xr-x 1 root root 8.5K 7月  27 10:22 main01
-rwxr-xr-x 1 root root 888K 7月  27 10:22 main02
edu@edu:~/work/c/day04$
```

动态

静态

2、制作静态库

```
1 gcc -c fun.c -o fun.o
2 ar rc libtestlib.a fun.o
```

静态库libtestlib.a 是以lib开头 .a结尾 中间才是库的名称testlib

静态库使用：库和工程在同一目录

```c
#include <stdio.h>
#include "fun.h"
int main(int argc, char const *argv[])
{

    printf("%d\n", my_add(10, 20));
    printf("%d\n", my_sub(10, 20));
    printf("%d\n", my_mul(10, 20));
    printf("%d\n", my_div(10, 20));

    return 0;
}
```

```
edu@edu: ~/work/c/day04/06_code
edu@edu:~/work/c/day04/06_code$ ls
fun.c  fun.h  libtestlib.a  main.c
edu@edu:~/work/c/day04/06_code$ sudo gcc main.c libtestlib.a
edu@edu:~/work/c/day04/06_code$ ./a.out
30
-10
200
0
edu@edu:~/work/c/day04/06_code$
```

静态库使用：库和库的头文件在 自定义目录（推荐）

```
sudo gcc main.c -I./lib -L./lib -ltestlib
```

```
-I后面跟头文件的路径
-L后面跟库的路径
-l后面跟库的名称（lib和.a之间的名称）
```

```c
#include <stdio.h>
#include "fun.h"
int main(int argc, char const *argv[])
{

    printf("%d\n", my_add(10, 20));
    printf("%d\n", my_sub(10, 20));
    printf("%d\n", my_mul(10, 20));
    printf("%d\n", my_div(10, 20));

    return 0;
}
```

```
edu@edu: ~/work/c/day04/06_code
edu@edu:~/work/c/day04/06_code$ ls
fun.c  lib  main.c
edu@edu:~/work/c/day04/06_code$ sudo gcc main.c -I./lib -L./lib -ltestlib
edu@edu:~/work/c/day04/06_code$ ./a.out
30
-10
200
0
edu@edu:~/work/c/day04/06_code$
```

静态库使用：库和库的头文件在 系统目录

将库拷贝到/usr/lib下

将头文件拷贝到/usr/include下

```
edu@edu:~/work/c/day04/06_code$ ls
a.out  fun.c  lib  main.c
edu@edu:~/work/c/day04/06_code$ sudo cp lib/libtestlib.a /usr/lib
edu@edu:~/work/c/day04/06_code$ sudo cp lib/fun.h /usr/include
edu@edu:~/work/c/day04/06_code$
```

```c
#include <stdio.h>
#include <fun.h>
int main(int argc, char const *argv[])
{

    printf("%d\n", my_add(10, 20));
    printf("%d\n", my_sub(10, 20));
    printf("%d\n", my_mul(10, 20));
    printf("%d\n", my_div(10, 20));

    return 0;
}
```

```
edu@edu: ~/work/c/day04/06_code
edu@edu:~/work/c/day04/06_code$ ls
a.out  fun.c  lib  main.c
edu@edu:~/work/c/day04/06_code$ sudo gcc main.c -ltestlib
edu@edu:~/work/c/day04/06_code$ ./a.out
30
-10
200
0
edu@edu:~/work/c/day04/06_code$
```

只需要给库的名字

3、制作动态库

```
gcc -shared fun.c -o libtestlib.so
```

libtestlib.so　以lib开头　以.so结尾　红色部分为库名称

```
edu@edu:~/work/c/day04/06_code$ sudo gcc -shared fun.c -o libtestlib.so
[sudo] edu 的密码：
edu@edu:~/work/c/day04/06_code$ ls
a.out  fun.c  lib  libtestlib.so  main.c
edu@edu:~/work/c/day04/06_code$
```

动态态库使用：库和工程 在同一目录

```
edu@edu: ~/work/c/day04/06_code
edu@edu:~/work/c/day04/06_code$ sudo gcc main.c -o main libtestlib.so
edu@edu:~/work/c/day04/06_code$ ./main
./main: error while loading shared libraries: libtestlib.so: cannot open shared ob
ject file: No such file or directory
edu@edu:~/work/c/day04/06_code$
```

找不到库

将当前路径./添加到库的搜索目录 就可以

```
export LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH
```

```
edu@edu: ~/work/c/day04/06_code
edu@edu:~/work/c/day04/06_code$ sudo gcc main.c -o main libtestlib.so
edu@edu:~/work/c/day04/06_code$ ./main
./main: error while loading shared libraries: libtestlib.so: cannot open shared ob
ject file: No such file or directory
edu@edu:~/work/c/day04/06_code$ export LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH
edu@edu:~/work/c/day04/06_code$ ./main
30
-10
200
0
edu@edu:~/work/c/day04/06_code$
```

将当前路径添加到 库的搜索路径中

动态态库使用：库在自定义目录

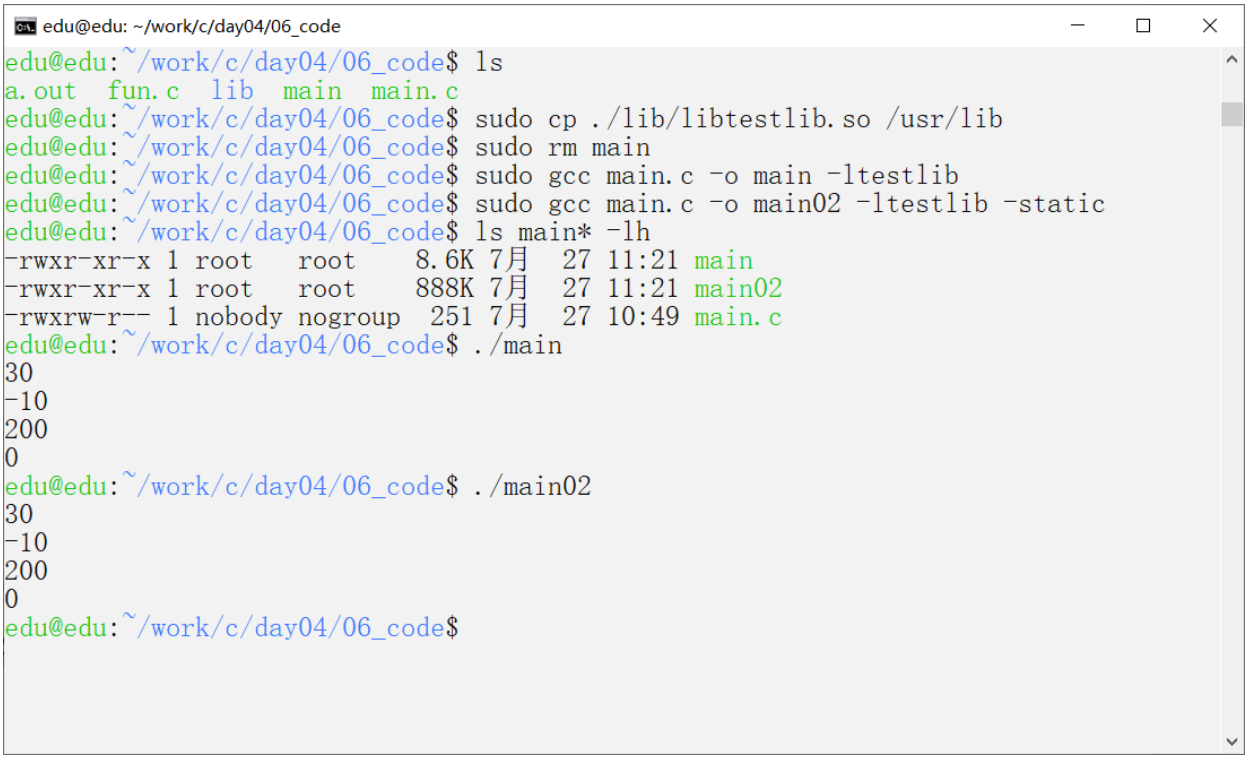
```
edu@edu: ~/work/c/day04/06_code
edu@edu:~/work/c/day04/06_code$ ls
a.out  fun.c  lib  main.c
edu@edu:~/work/c/day04/06_code$ sudo gcc main.c -o main -I./lib -L./lib -ltestlib
edu@edu:~/work/c/day04/06_code$ ./main
./main: error while loading shared libraries: libtestlib.so: cannot open shared ob
ject file: No such file or directory
edu@edu:~/work/c/day04/06_code$
```

将当前路径./lib添加到库的搜索目录 就可以

```
1 export LD_LIBRARY_PATH=./lib:$LD_LIBRARY_PATH
```

```
edu@edu: ~/work/c/day04/06_code
edu@edu:~/work/c/day04/06_code$ ls
a.out  fun.c  lib  main.c
edu@edu:~/work/c/day04/06_code$ sudo gcc main.c -o main -I./lib -L./lib -ltestlib
edu@edu:~/work/c/day04/06_code$ ./main
./main: error while loading shared libraries: libtestlib.so: cannot open shared ob
ject file: No such file or directory
edu@edu:~/work/c/day04/06_code$ export LD_LIBRARY_PATH=./lib:$LD_LIBRARY_PATH
edu@edu:~/work/c/day04/06_code$ ./main
30
-10
200
0
edu@edu:~/work/c/day04/06_code$
```

动态态库使用：库在系统目录/usr/lib　头文件在/usr/include



如果静态库和动态库 同时存在 默认编译选择动态库 只有加-static修饰才能链接静态库