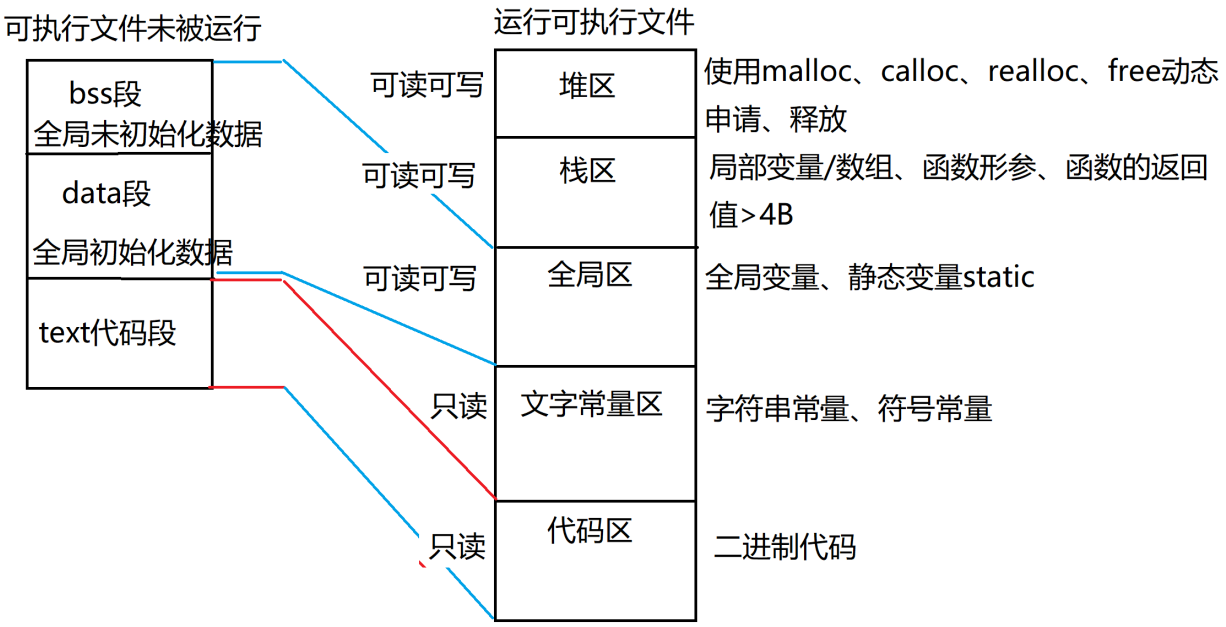
知识点1【内存的分区】（了解）



知识点2【普通局部变量、普通全局变量、静态局部变量、静态全局变量】

1、普通局部变量

定义形式：在{}里面定义的普通变量 就是普通局部变量。

```
void test01()
{//复合语句
    int num1 = 10;//普通局部变量
}
```

作用范围：离它最近的{}之间有效

```
2  void test01()
3  {
4      int num1 = 10;//普通局部变量
5      {
6          int num2 = 100;//num2作用范围为
7      }
8
9      printf("num2 = %d\n",num2);
0  }
```

生命周期：离它最近的{}之间有效，离开{}的局部变量 系统自动回收

存储区域：栈区

注意事项：

a、普通局部变量不初始化 内容不确定

```c
#include<stdio.h>
void test01()
{
    //局部变量 不初始化 内容 不确定
    int data;
    printf("data = %d\n",data);
}
```

```
"C:\WORK\C\CODE\DAY06\00_test\Debug\00_test.exe"
data = -858993460
Press any key to continue
```

b、普通局部变量 同名 就近原则

```c
void test01()
{
 //局部变量 同名 就近原则
 int data = 100;
 {
 int data = 200;
 printf("A:data = %d\n",data);//200
 }

  printf("B:data = %d\n",data);//100
}
```

2、普通全局变量

定义形式：定义在函数外边的变量 就是普通全局变量

```c
int data;//普通全局变量 在函数外边定义
void test02()
{

}
```

作用范围：

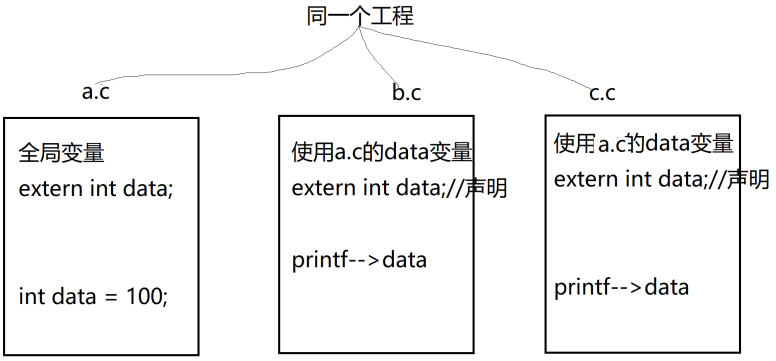
a、当前源文件 都有效

```c
#include<stdio.h>
extern int data;//声明一下 不要赋值
void test01()
{
 printf("test01 中 data = %d\n",data);//100
}

int data=100;//普通全局变量 在函数外边定义

 void test02()
 {
```

```
 printf("test02 中 data = %d\n",data);//100
}
int main(int argc,char *argv[])
{
 printf("main 中 data = %d\n",data);//100
 test01();
 test02();

}
```

b、其他源文件使用全局变量时 必须加extern声明。

//extern 本质：告诉编译器 变量/函数 来至其他源文件，请通过编译



代码：

main.c

```
#include<stdio.h>
extern void my_printf(void);
extern int data;//声明
int main(int argc,char *argv[])
{
 printf("main 中 data = %d\n",data);//100

 my_printf();

}
int data = 100;
```

fun.c

```
#include<stdio.h>
extern int data;//声明data来至其他源文件
void my_printf(void)
{
 printf("在fun.c中 data = %d\n",data);
}
```

运行结果：

```
main 中 data = 100
在fun.c中 data = 100
Press any key to continue
```

生命周期：整个进程 都有效（程序结束的时候 全局变量 才被释放）

存储区域：全局区

注意事项

a、全局变量 不初始化 内容为0。

b、如果全局变量 要在其他源文件中使用 必须在所使用的源文件中加extern声明。

c、如果全局变量 和 局部变量 同名 在{}中优先使用局部变量

```
int num = 10;//全局变量
int main(int argc,char *argv[])
{
    int num = 100;//局部变量

    printf("num = %d\n",num);//100 优先识别局部变量

}
```

```
"C:\WORK\C\CODE\DAY06\00_test\Debug\00_test.exe"
num = 100
Press any key to continue
```

3、静态局部变量

定义形式：在{}中定义 前面 必须加static 修饰 这样的变量 叫 静态局部变量。

```
1  void test01()
2  {
3   static int num;//静态局部变量
4   return;
5  }
```

作用范围：离它最近的{}之间有效。

```
1  void test01()
2  {
3   {
4   static int num;//静态局部变量
5   }
6   //说明 静态局部变量 只在离它最近的{}有效
7   printf("num = %d\n",num);//err 不识别num
8   return;
9  }
```

生命周期：整个进程， （程序结束的时候 静态局部变量 才被释放）

----------案例：普通局部变量----------

```
1  #include<stdio.h>
```

```c
void fun1(void)
{
 int num = 10;//普通的局部变量
 num++;

 printf("num = %d\n",num);
 return;
}
 int main(int argc,char *argv[])
 {
  fun1();//11
  fun1();//11
  fun1();//11
  fun1();//11
  return 0;
 }
```

----------案例：静态局部变量----------

```c
#include<stdio.h>
void fun1(void)
{
 //静态局部变量 只能被初始化一次
 //静态局部变量 生命周期 是整个进程
 static int num = 10;//静态局部变量
 num++;
 printf("num = %d\n",num);
 return;
}
int main(int argc,char *argv[])
{
 fun1();//11
 fun1();//12
 fun1();//13
 fun1();//14
 return 0;
}
```

存储区域：全局区

注意事项：

a、静态局部变量 不初始化 内容为0.（全局区）

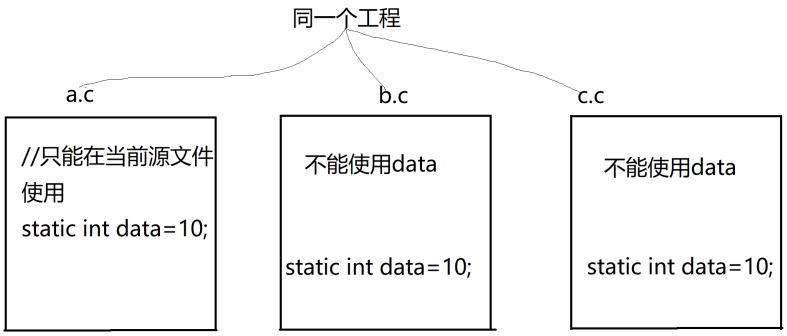
b、只能被定义一次（重要）

4、静态全局变量

定义形式：在函数外边定义 同时加static 这样的变量就是 静态全局变量

```c
#include<stdio.h>
static int data = 10;//静态全局变量
int main(int argc,char *argv[])
{

 return 0;
}
```

作用范围：当前源文件 有效 不能在其他源文件中使用。



生命周期：整个进程，（程序结束 静态全局变量才被释放）

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 只在当前源文件 有效

知识点3【全局函数（普通函数） 和 静态函数（局部函数）】

1、全局函数：普通函数

```c
void my_fun(void)
{
 printf("(全局函数)普通函数\n");
 return;
}
```

特点：其他源文件 可以使用 全局函数，必须加extern 声明

2、静态函数（局部函数）

```
static void my_static_fun(void)
{
 printf("(静态函数)局部函数\n");
 return;
}
```

特点：只能在当前源文件使用 不能在其他源文件使用。

注意：如果想在其他源文件 调用 静态函数 需要将静态函数 封装在 全局函数中。同时全局函数 和静态函数 必须是同一个源文件。

fun.c

```

static void my_static_fun(void)
{
 printf("(静态函数)局部函数\n");
 return;
}

void my_fun(void)
{
  printf("(全局函数)普通函数\n");
  my_static_fun();
  return;
 }
```

main.c

```
#include<stdio.h>
extern void my_fun(void);
//static void my_static_fun(void);
int main(int argc,char *argv[])
{
 my_fun();
 //my_static_fun();
 return 0;
}
```

案例：

main.c

```c
#include <stdio.h>
extern int va;
extern int getG(void);
extern int getO(void);
int main(void)
{
    printf("va=%d\n",va);
    printf("getO=%d\n",getO());
    printf("getG=%d\n",getG());
    printf("%d", va*getO()*getG());
}
```

fun1.c

```c
int va = 7;
int getG(void)
{
    int va = 20;
    return va;
}
```

fun2.c

```c
static int va = 18;
static int getG(void)
{
    return va;
}
int getO(void)
{
    return getG();
}
```

知识点4【gcc编译过程】（了解）

```
➢ gcc -E hello.c -o hello.i        1、预处理
➢ gcc -S hello.i -o hello.s        2、编译
➢ gcc -c hello.s -o hello.o        3、汇编
➢ gcc hello.o -o hello_elf         4、链接
```

```c
#include<stdio.h>
```

预处理：头文件包含、宏替换、条件编译、删除注释　不做语法检查

编译：将预处理后的文件 生成 汇编文件 语法检查

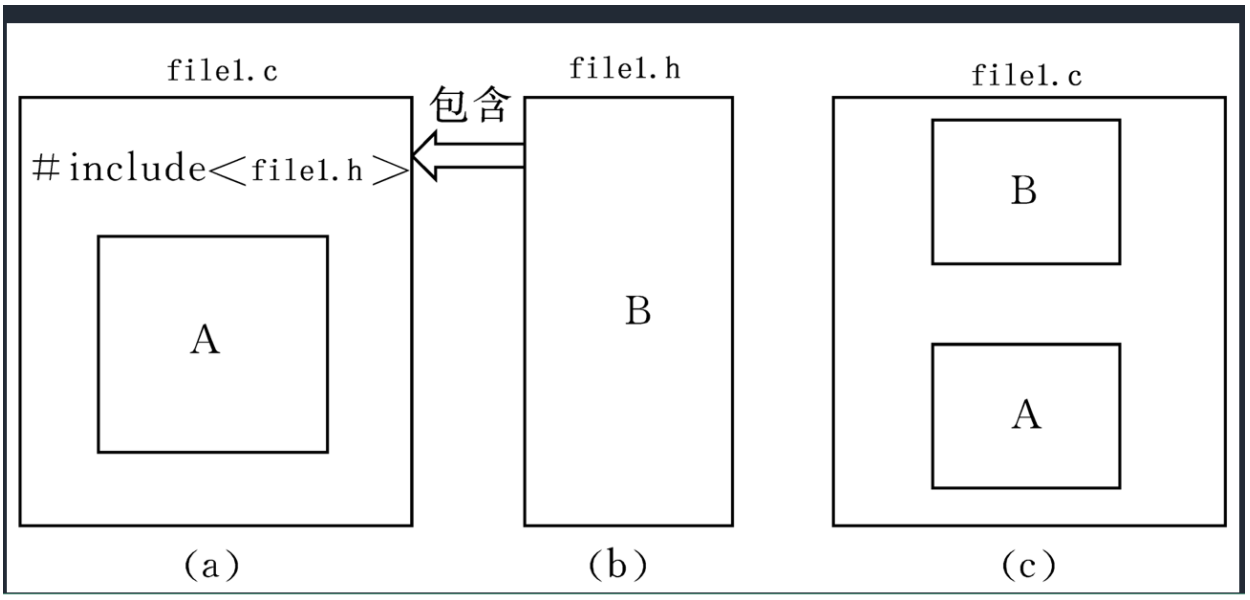
汇编：将汇编文件 编译 二进制文件

链接：将众多的二进制文件+库+启动代码 生成 可执行文件

总结：一步到位 编译：　gcc 源文件 -o 可执行文件

```
ubuntu@VM-0-4-ubuntu:~/work/c$ gcc a.c -o haha
ubuntu@VM-0-4-ubuntu:~/work/c$ ./haha
PI = 3.140000
ubuntu@VM-0-4-ubuntu:~/work/c$
```

知识点5【头文件包含<>,""】（了解）



#include <hehe.h> 表示从系统的指定目录下寻找hehe.h

#include "hehe.h" 表示 先从源文件 所在的目录寻找 如果找不到 再到系统指定的目录下找。

my_fun.h

```
#define PI 3.14f
```

main.c

```
#include<stdio.h>//用于包含系统的头文件
#include"my_fun.h"//用户包含 用户自定义的 头文件

int main(int argc,char *argv[])
{
 printf("PI = %f\n",PI);
 return 0;
}
```

知识点6【define 宏】、

宏只在当前源文件有效

宏 一般为 大写。(将宏 和 普通变量 区分开)

1、不带参数的宏

```
#include<stdio.h>

//宏 后面不要加;
```

```c
#define N "hehe"
void test01()
{
 //在预处理阶段 "hehe" 替换 代码中所有出现的N （宏展开）
 printf("%s\n",N);
 return;
}
int main(int argc,char *argv[])
{
 test01();
 return 0;
}
```

终止宏 的作用范围： #undef 宏名

```c
#include<stdio.h>

//宏 后面不要加;
#define N "hehe"
void test01()
{
 printf("%s\n",N);//OK 识别的

 //使用#undef N终止 N的作用
#undef N

 //printf("%s\n",N);//err 不识别N

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

2、带参数的宏 （宏 函数）

#define 宏名(参数1，参数2，...) 字符串

```c
//宏的参数 a b 不能写类型
//#define MY_ADD(int a, int b) a+b //错误
#define MY_ADD(a,b) a+b

```

```
//调用 宏名(参数)
MY_ADD(10,20);// 10+20
```

案例1：

```
//宏展开 本质 就是替换
#define MY_MUL1(a,b) a*b
#define MY_MUL2(a,b) ((a)*(b))
void test01()
{
 printf("%d\n", MY_MUL1(10,20));//MY_MUL1(10,20) == 10*20

 //MY_MUL1( 10+10, 20+20 )==10+10*20+20
 printf("%d\n", MY_MUL1(10+10,20+20));//230 不能保证完整性

  //MY_MUL2(10+10,20+20) == ((10+10)*(20+20))
  printf("%d\n", MY_MUL2(10+10,20+20));//800 保证完整性

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

3、带参数的宏（宏函数） 和 普通函数 有啥区别

带参数的宏（宏函数） 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,也不需要函数的出入栈，所以带参数的宏 浪费空间　节省了时间。

代参的函数：代码只有一份，存在代码段， 调用的时候去代码段读取函数指令，调用的时候 要压栈（保存调用函数前的相关信息），调用完出栈（恢复调用函数前的相关信息），所以函数浪费了时间 节省空间。

```
#include<stdio.h>

//宏展开 本质 就是替换
//宏函数
#define MY_MUL1(a,b) a*b
#define MY_MUL2(a,b) ((a)*(b))

//普通函数
int my_mul(int a,int b)//a=10+10=20 b=20+20=40
```

```
{
 printf("a = %d\n",a);//20
 printf("b = %d\n",b);//40

 return a*b;
}
void test01()
{
 //预处理阶段完成 宏的替换
 printf("%d\n", MY_MUL1(10,20));//MY_MUL1(10,20) == 10*20
 printf("%d\n", MY_MUL1(10+10,20+20));//10+10*20+20=230 不能保证完整性
 printf("%d\n", MY_MUL2(10+10,20+20));//((10+10)*(20+20))=800 保证完整性

 //执行
 printf("%d\n", my_mul(10+10, 20+20));//

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

案例：

```
#define MY_ADD(a,b) a+b
#define MY_MUL1(a,b) a*b

printf("%d\n",MY_MUL( MY_ADD(10+10, 20+20), MY_MUL(10+10,20+20) ) );
```

知识点7【条件编译】

测试不存在

```
#ifndef XXX
    语句1;
#else
    语句2;
#endif
```

测试存在

```
#ifdef  XXX
    语句1;
#else
    语句2;
#endif
```

判断条件是否成立

```
#if  表达式
    语句1;
#else
    语句2;
#endif
```

案例1：测试不存在

```c
#include<stdio.h>
int main(int argc,char *argv[])
{
#ifndef HEHE
        printf("---001----\n");
#else
        printf("---002----\n");
#endif
    return 0;
}
```

运行结果：

```c
int main(int argc,char *argv[])
{

 printf("---001----\n");



    return 0;
}
```

案例2：测试存在

```c
#include<stdio.h>
int main(int argc,char *argv[])
{
#ifdef HEHE
        printf("---001----\n");
#else
        printf("---002----\n");
#endif
    return 0;
}
```

运行结果：

```c
int main(int argc,char *argv[])
{



 printf("---002----\n");

    return 0;
}
```

案例3：判断表达式

```c
int main(int argc,char *argv[])
{
#if 0
        printf("---001----\n");
#else
        printf("---002----\n");
#endif
    return 0;
}
```

综合案例：通过条件编译 控制大小写的转换

```c
#include<stdio.h>

int main()
{
 char buf[128]="";
 int i=0;
 printf("请输入字符串:");
 //fgets 会获取 换行符 '\n'
 fgets(buf,sizeof(buf), stdin);
  //去掉换行符 strlen返回的是字符串是长度 不包含'\0'
  //strlen(buf)-1 这是 换行符 的下标位置
  buf[strlen(buf)-1] = 0;

  //while(buf[i] != '\0')

  //char buf[128];
  //buf[i]是取数组中的第i个元素的值。

  while(buf[i])//最后一个元素是'\0' == 0==假 循环进不去
```

```
 {
#if 1
 if(buf[i]>='A' && buf[i]<='Z')
 buf[i] = buf[i]+32;
#else
 if(buf[i]>='a' && buf[i]<='z')
 buf[i] = buf[i]-32;
#endif
 i++;
 }


 printf("buf = %s\n",buf);
 return 0;
}
```