程序设计专题

主讲教师：张引

助教：田沈晶，熊海辉

QQ： 599168489

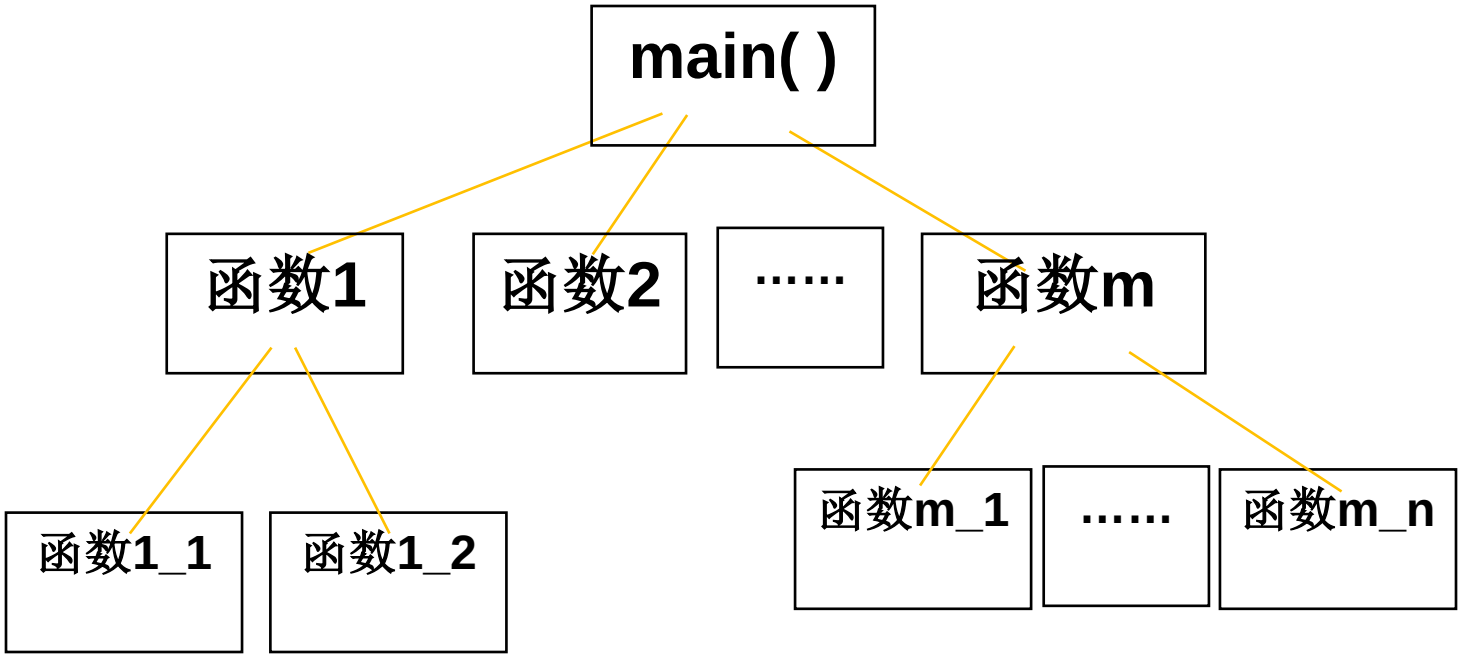
浙江大学计算学院．2018年3月12日

程序设计专题一：
结构化程序设计与递归函数

专题要点

- 用结构化程序设计的思想解决问题
  - 函数嵌套求解复杂的问题
  - 理解和使用函数递归
- 将多个函数组织起来，将多个源程序文件组织起来
  - static和extern
  - 编译预处理
- 理解程序设计规范及其重要性

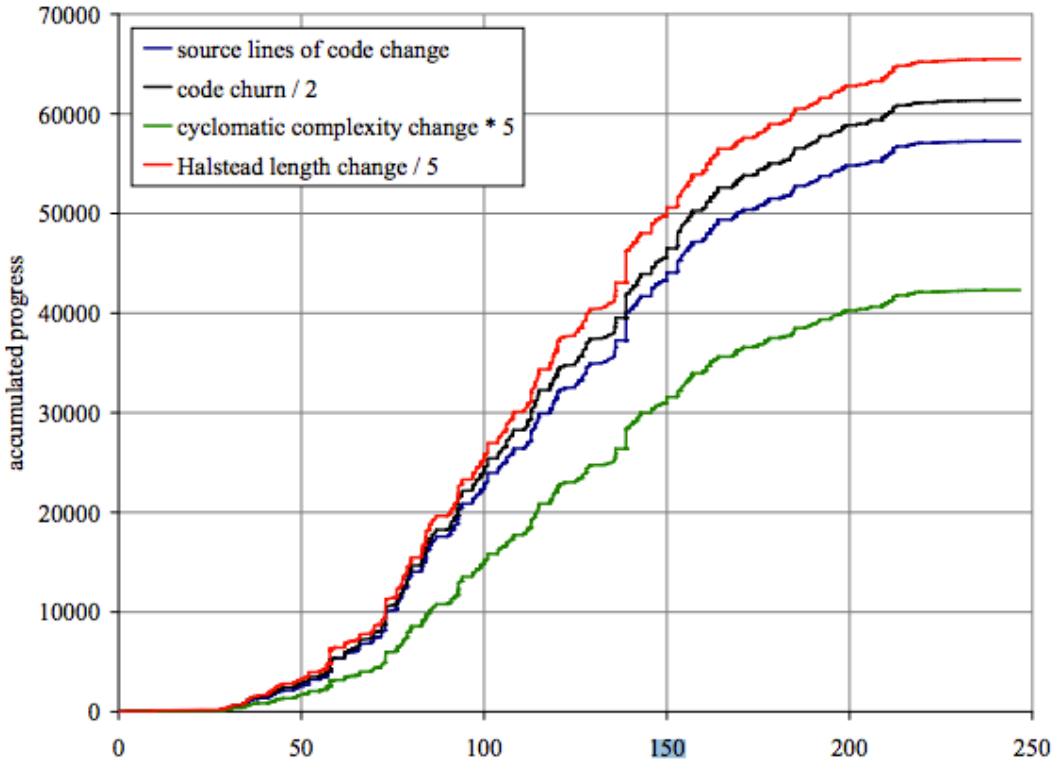
程序结构



子程序（routines）是为实现一个特定功能而编写的一个可被调用的方法（method）、函数（function）或过程（procedure）。如Java中的方法，C/C++里的函数。

程序结构

⌘子程序设计原则



长度适中：“子程序/函数的第一要素就是短小，第二条规则还是短小”，鲍勃大叔如此说。

- 理论上认为子程序的长度最大长度通常是一屏代码，大约50-150行。
- 一项对子程序的研究发现，平均100-150行代码的子程序需要修改的几率最低（Lind and Vairavan 1989）。
- IBM一项研究发现，最容易出错的是那些超过500行代码的子程序。超过500行后，子程序的出错率和代码行数成正比。

程序结构

⌘子程序设计原则

好的命名：

好的命名能清晰的描述子程序所做的一切。

一些命名注意事项：

1. 描述子程序所完成的功能
2. 避免使用无意义的、模拟或表达不清的词
3. 不要仅通过数字来区分不同的子程序名
4. 根据需要确定子程序名字的长度
5. 对返回值要有所描述
6. 一般是动词+名词形式
7. 使用对仗词，如add/remove, begin/end, first/last, get/put, up/down/, show/hide, open/close。

源程序文件组织

- 模块
  - 一个可单独编译的源文件
  - 或该源文件被编译后所生成的目标文件
  - 功能模块：共同完成某个功能的若干程序集合
  - 任何能被装入内存中运行的可执行代码和数据的集合。
    - 可执行的EXE文件（又称为应用程序模块），
    - 动态链接库（DLL，Dynamic Linking Library）模块），
    - 设备驱动程序
  - 也可以特指一段自包含的程序。

源程序文件组织

- 为了避免一个文件过长，可以把程序组织为多个文件
  - 每一个文件可包含若干个函数
- 程序文件➔程序文件模块
  - 程序 ➔ 文件 ➔ 函数
  - 各程序文件模块分别编译，再连接
- 整个程序只允许有一个main()函数

源程序文件组织

例：欧美国家长度使用英制单位，1英里=1609米，1英尺=30.48厘米，1英寸=2.54厘米。请编写程序转换。

```
/*  only 1 file named “ transfer.c */
#include<stdio.h>
#define Mile_to_meter 1609                    /* 1英里=1609米 */
#define Foot_to_centimeter 30.48              /*  1英尺=30.48厘米  */
#define Inch_to_centimeter  2.54              /*  1英寸=2.54厘米  */
int main(void)
{
        float foot, inch, mile;        /* 定义英里，英尺，英寸变量 */
        printf("Input mile,foot and inch:");
        scanf("%f%f%f", &mile, &foot, &inch);
        printf("%f miles=%f meters\n", mile, mile * Mile_to_meter);/* 计算英里的米数 */
        printf("%f feet=%f centimeters\n", foot, foot *
                                                Foot_to_centimeter);      /* 计算英尺的厘米数 */
        printf("%f inches=%f centimeters\n", inch, inch *
                                                Inch_to_centimeter);      /* 计算英寸的厘米数 */
        return 0;
}
```

源程序文件组织

例：欧美国家长度使用英制单位，1英里=1609米，1英尺=30.48厘米，1英寸=2.54厘米。请编写程序转换。

头文件length.h

```
#define Mile_to_meter 1609
#define Foot_to_centimeter 30.48
#define Inch_to_centimeter 2.54
```

主函数文件prog.c

```
#include<stdio.h>
#include “length.h”
int main(void)
{
    float mile,foot,inch;

    ......
    return 0;
}
```

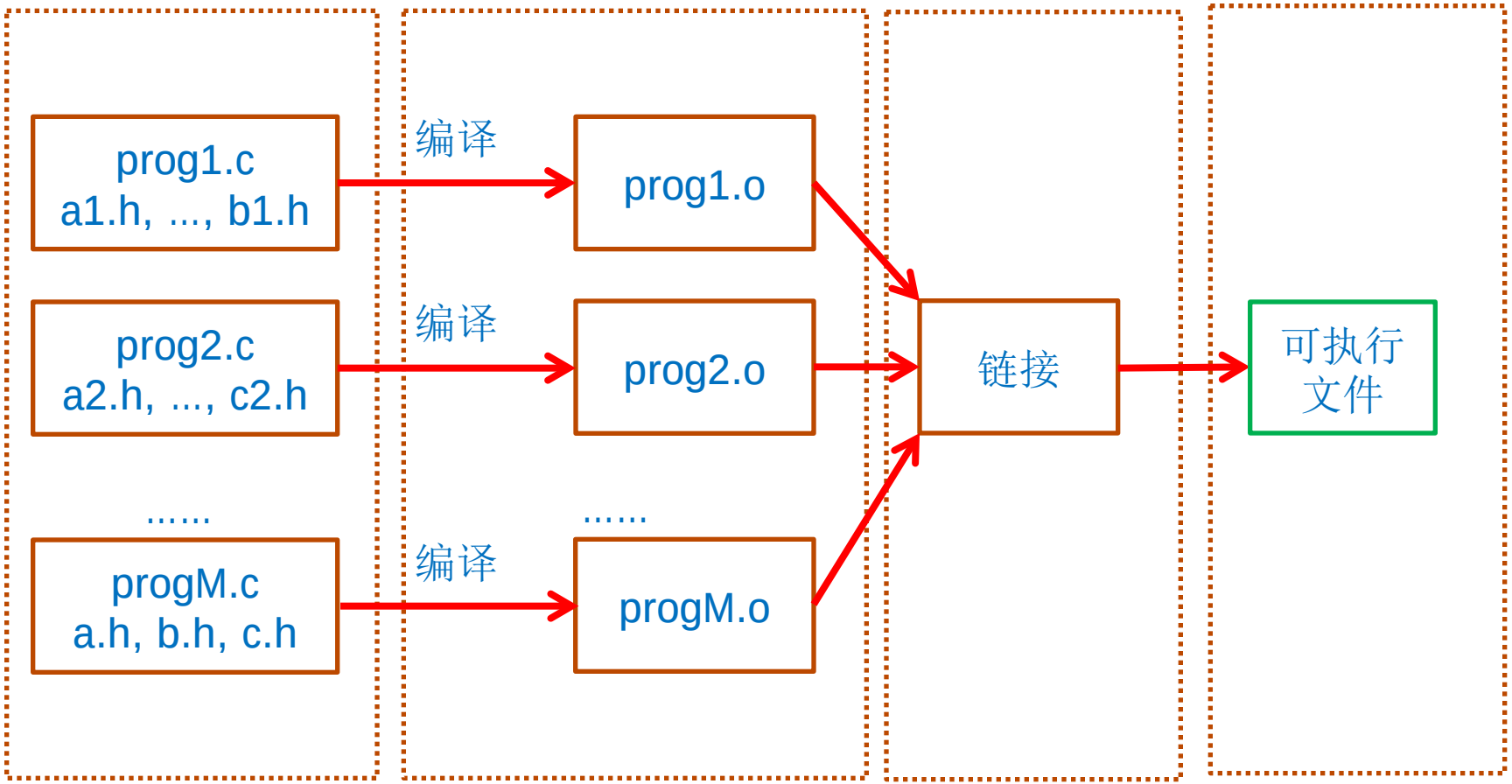
多文件

源程序文件组织

- .c文件：实现文件
  - 函数实现
  - 全局变量定义
- .h文件：自定义的头文件
  - 类型定义，声明，符号常量
- 一个程序通常包含多个.c和.h文件
  - 几个到几千个
  - 系统越大越复杂，文件越多

源程序文件组织

- 模块编译链接



编译预处理

- 编译器对源代码开始编译之前，首先对.c文件进行预处理
- 编译预处理是C语言编译程序的组成部分，它用于解释处理C语言源程序中的各种预处理指令。

编译预处理

你接触过的有？

- 预处理指令
  - 以#号开头的代码行
  - #号必须是该行除了空白字符外的第一个字符。
  - #号后是指令关键字
  - 在关键字和#号之间允许存在任意个数的空白字符
  - 整行构成了一条预处理指令
  - 在形式上都以“#”开头，不属于C语言中真正的语句
  - 编译器通过预处理指令，对源代码做相应转换
  - 增强了C语言的编程功能，改进C语言程序设计环境，提高编程效率

编译预处理

- 编译器对源代码开始编译之前，首先对.c文件进行预处理
  - 读取c源程序，对其中的预处理指令（以#开头的指令）和特殊符号进行处理。
  - 或者说，扫描源代码，对其进行初步的转换，生成新的源代码提供给编译器。

编译预处理 ——常用的预处理指令

- #include 文件包含指令
- #define 宏定义指令
- #undef 宏定义取消指令
- 条件编译指令
  - #ifdef/#ifndef
  - <条件对应的代码>
  - #endif
- 条件编译指令
  - #if表达式
  - <表达式非零时编译的代码>
  - #else
  - <否则编译的代码>
  - #elif

#error 停止编译预处理并输出错误信息

```
#include <stdio.h>
#define _LIB_VERSION 2
#undef _linux

#ifndef M_PI
#define M_PI 3.1415926
#endif

#if _LIB_VERSION ==2
<适合版本2的代码>
#error "Only support V 2"
<适合其他版本的代码>
#endif
```

编译预处理 ——宏定义指令

- #define 宏名标识符 宏定义字符串

  #define PI 3.14

  编译时，把程序中所有与宏名相同的标识符，用宏定义字符串替代

说明:

- 宏名一般用大写字母，以与变量名区别
- 宏定义不是C语句，后面不得跟分号
- 宏定义可以嵌套使用

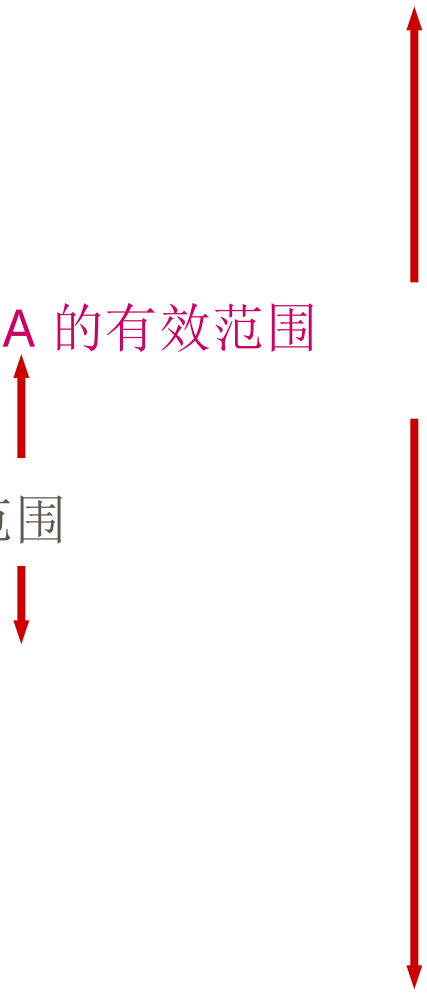
  #define PI 3.14

  #define S 2*PI*PI
- 多用于符号常量、简单的操作和函数等

编译预处理 ——宏定义指令

- 宏定义可以写在程序中任何位置，它的作用范围从定义书写处到文件尾。可以通过“#undef”强制指定宏的结束范围。

```
#define  A  “This is the first macro”
void  f1()
{
    printf( “A\n” );
}
#define  B  “This is the second macro”
void  f2( )
{
    printf( B ) ;
}
#undef  B
int main(void)
{
    f1( );
    f2( );
    return 0;
}
```



编译预处理

⌘ 带参数的宏定义

例：各位数字的立方和等于它本身的数。
例如153的各位数字的立方和是
$1^3+5^3+3^3=153$

```
#define  f(a)  (a)*(a)*(a)
int     main( )
{
   int i,x,y,z;
   for (i=1; i<1000; i++)
   {
       x=i%10; y=i/10%10; z=i/100 ;
       if  ( f(x) + f(y) + f(z) == i )
                  printf("%d\n" ,i);
   }
    return 0;
}
```

- 带参数的宏定义不是函数，宏与函数是两种不同的概念
- 宏可以实现简单的函数功能

f(x+y) == (x+y)³ ?

x+y*x+y*x+y　　**(x+y)*(x+y)*(x+y)**

编译预处理 ——宏定义指令

⌘ 嵌套的宏定义

```
#define F(x)  x - 2
#define D(x)  x*F(x)
void main()
{
    printf("%d,%d", D(3), D(D(3))) ;
}
```

计算 D(3), D(D(3))

⌘先全部替换好，最后再统一计算

⌘不可一边替换一边计算，更不可以人为添加括号

D(3) = 3*F(3)
= 3*3-2
= 7

D(D(3)) = D(3)*F(D(3))
= 3*F(3)*D(3)-2
= 3*3-2*3*F(3)-2
= 3*3-2*3*3-2-2
= -13

编译预处理 ——宏定义指令

- 应用示例
  - 定义宏LOWCASE，判断字符c是否为小写字母。

```
#define  LOWCASE(c)  (((c) >= 'a') && ((c) <= 'z') )
```

  - 定义宏CTOD将数字字符（‘0’～‘9’）转换为相应的十进制整数，-1表示出错。

```
#define  CTOD(c)  (((c) >= '0') && ((c) <= '9') ? c - '0' : -1)
```

  - 最大值、最小值

```
#define  MAX(a,b)  ((a) >= (b) ?  (a) : (b))
#define  MIN(a,b)  ((a) <= (b) ?  (a) : (b))
```

编译预处理 ——宏定义指令

- 宏替换仅仅是以文本串代替宏标识符，前提是宏标识符必须独立的识别出来，否则不进行替换。

```
#define XYZ this is a tes
printf("XYZ");
```

- 如果串长于一行，可以在该行末尾用一反斜杠’\’续行。

```
#defineLONG_STRING"this is a very long\
string that is used as an example"
```

编译预处理 ——文件包含预处理指令

- 格式
  - # include <需包含的文件名>
  - # include “需包含的文件名”

系统文件夹

当前工作文件夹+系统文件夹

- 作用

把指定的文件模块内容插入到 #include 所在的位置，当程序编译连接时，系统会把所有 #include 指定的文件拼接生成可执行代码。

例：欧美国家长度使用英制单位，1英里=1609米，1英尺=30.48厘米，1英寸=2.54厘米。请编写程序转换。

头文件length.h源程序

```
#define Mile_to_meter 1609              /* 1英里=1609米 */
#define Foot_to_centimeter 30.48        /* 1英尺=30.48厘米 */
#define Inch_to_centimeter  2.54        /* 1英寸=2.54厘米 */
```

主函数文件prog.c源程序

```
#include<stdio.h>
#include “length.h”                     /* 包含自定义头文件 */
int main(void)
{
    float foot, inch, mile;         /* 定义英里，英尺，英寸变量 */
    printf("Input mile,foot and inch:");
    scanf("%f%f%f", &mile, &foot, &inch);
    printf("%f miles=%f meters\n", mile, mile * Mile_to_meter);
    printf("%f feet=%f centimeters\n", foot, foot * Foot_to_centimeter);
    printf("%f inches=%f centimeters\n", inch, inch * Inch_to_centimeter);
    return 0;
}
```

头文件length.h

```
#define Mile_to_meter 1609
#define Foot_to_centimeter 30.48
#define Inch_to_centimeter 2.54
```

主函数文件prog.c

```
#include<stdio.h>
#include “length.h”
int main(void)
{
    float mile,foot,inch;

    ……
    return 0;
}
```

文件包含预处理后

```
…  stdio.h的内容
#define Mile_to_meter 1609
#define Foot_to_centimeter 30.48
#define Inch_to_ centimeter 2.54

int main(void)
{
    float mile,foot,inch;
    scanf("%f%f%f", &mile, &foot, &inch);
    printf("%f miles=%f meters\n",    mile, mile *
Mile_to_meter);
    printf("%f feet=%f centimeters\n", foot, foot *
Foot_to_centimeter);
    printf("%f inches=%f centimeters\n", inch, inch
* Inch_to_centimeter);
    return 0;
}
```

宏定义预处理后

```
int main(void)
{
  float mile,foot,inch; float foot, inch, mile;
  printf("Input mile,foot and inch:");
  scanf("%f%f%f", &mile, &foot, &inch);
  printf("%f miles=%f meters\n", mile, mile * 1609);
   printf("%f feet=%f centimeters\n", foot, foot * 30.48);
   printf("%f inches=%f centimeters\n", inch, inch * 2.54);
```

编译预处理 ——文件包含指令

- 头文件(.h)由三部分内容组成：
  - 版权声明和版本信息（起始处）
  - 预处理块
  - 定义和声明
    - 结构和枚举类型定义
    - typedef定义和宏定义
    - 具名常量定义
    - 外部变量声明
    - 函数声明
- 头文件应该只用于声明，不应该包含“占据存储空间的变量或函数”的定义
- 正确使用头文件可提高代码的可读性、可维护性、以及开发效率

编译预处理 ——文件包含指令

- 头文件可以嵌套包含
  - A.h包含B.h，B.h包含C.h，。。。
- 避免循环包含，避免头文件被多重复包含
  - 虽然函数、变量的声明都可以重复，不会影响程序编译和运行，但会增加编译处理的时间
  - 当头文件中包含结构的定义、枚举定义等一些定义时，这些定义是不可以重复的，必须通过一定措施防止发生

编译预处理 ——文件包含指令

vecmat.h

```
typedef struct {
   float x, y, z;
} vec;

typedef struct {
   float m[3][3];
} mat;
```

vfunc.h

```
#include "vecmat.h"
vec vAdd(vec* a, vec* b, vec* r);
vec vSub(vec* a, vec* b, vec* r);
```

mfunc.h

```
#include "vecmat.h"
vec mxv(mat* m, vec* v, vec* r);
vec vxm(vec* v, mat* m, vec* r);
```

myprog.c

```
#include "vfunc.h"
#include "mfunc.h"
```

编译预处理 ——文件包含指令

myprog.c

```
#include "vfunc.h"
#include "mfunc.h"

int main( void )
{
}
```

问题：

vecmat.h被重复包含了

解决方法

#ifndef

#define

#endif

#define保护

```
#ifndef _HEADERNAME_H
#define _HEADERNAME_H

<头文件内容>

#endif
```

注意宏定义
_HEADERNAME_H
在模块中

- 必须是唯一的
- 不要重复

vecmat.h

```
#ifndef _VEC_MAT_H
#define _VEC_MAT_H

typedef struct {
   float x, y, z;
} vec;

typedef struct {
   float m[3][3];
} mat;

#endif // ifndef _VEC_MAT_H
```

#define保护

```
#ifndef _VEC_MAT_H
#define _VEC_MAT_H

typedef struct {
   float x, y, z;
} vec;

typedef struct {
   float m[3][3];
} mat;

#endif // ifndef _VEC_MAT_H
```

通过#define保护，当头文件被重复include的时候，
内容不会被重复include
<在同一个程序模块中不重复>

#define保护

- 保证了在编译一个c文件时，头文件的不被重复编译
  - 在编译其他.c文件时，该头文件任然会被编译
    - 记住：编译c文件是独立进行的
  - 因此，一般地，头文件中不能出现函数定义和变量的定义。
  - 否则如果有多个c文件包含它时，仍然会编译生成重复的变量和函数定义。
    - 重复的变量或函数定义会导致Link错误！

举例：不规范的头文件

```
// in header.h
#ifndef _HEADER_H
#define _HEADER_H
extern void Foo1();    /*函数声明       */
extern int a1;         /*外部变量声明   */
struct A;              /*前置声明结构A */
int a2;               //全局变量定义，应当避免
void Foo2()           //函数定义，应当避免
{}
typedef struct         /* 结构B定义     */
{
   int i;
   struct A m;
}B;
#endif
```

如果有多个程序模块文件，例如A.c和B.c，他们都include了该头文件。那么当他们被分别编译的，都生成了一份全局变量a2和函数Foo2。编译是ok的，但是在Link阶段，会出现重复定义的冲突，导致失败。

头文件的作用

- 通过头文件了解函数功能
  - 用户经常不能拿到源代码（涉密），但是可以拿到头文件和库文件（编译后的二进制文件）。
  - 用户可以按照头文件中的（接口）函数声明，调用库函数，而不必关心接口怎么实现的。
    - 编译器会从库中提取相应的二进制代码。
    - 例如使用printf,scanf标准库函数
- 通过头文件能加强类型安全检查
  - 如果某个接口/函数被实现或被使用时，其方式与头文件中的声明不一致，编译器就会报错
  - 该简单规则能大大减轻程序员调试和改错负担

编译预处理 ——条件编译

```
#define _flag 1
#define _flag 0
#if _flag
程序段1
#else
程序段2
#endif
```

条件编译

```
#define _zhongwen
#ifdef _zhongwen
#define msg "早上好"
#else
#define msg "Good morning"
#endif

main()
{
   printf(msg);
}
```

文件模块间的通信

- 程序—文件—函数关系
  - 小程序：主函数+若干函数 → 一个文件
  - 大程序：若干程序文件模块（多个文件） → 每个程序文件模块可包含若干个函数 → 各程序文件模块分别编译，再连接
  - 整个程序只允许有一个main()函数

文件模块间的通信

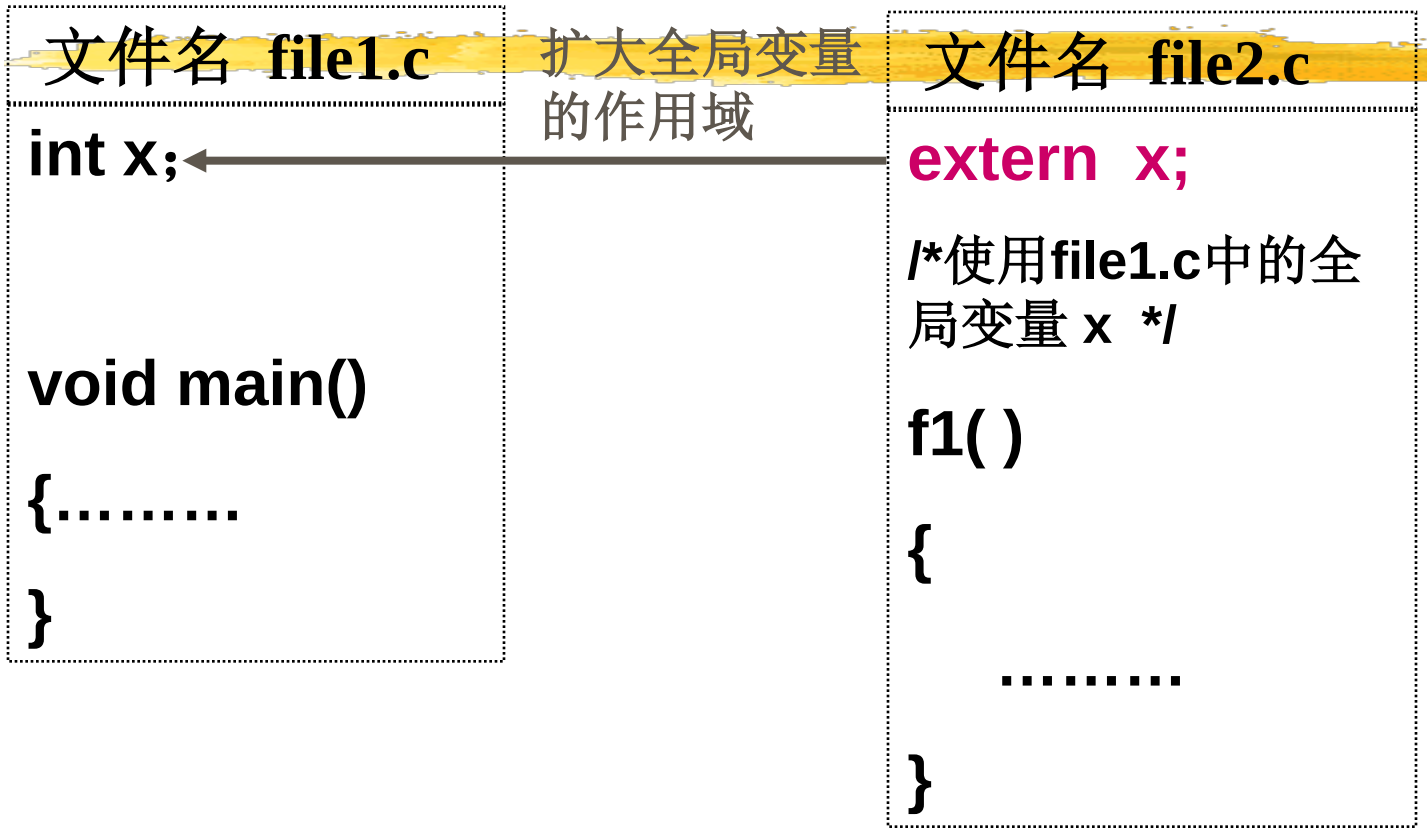
- 文件模块与变量
  - 外部变量
  - 静态全局变量
- 文件模块与函数
  - 外部函数
  - 静态的函数

文件模块间的通信

- 外部变量
  - 全局变量只能在某个模块中定义一次，如果其他模块要使用该全局变量，需要通过外部变量的声明

    外部变量声明格式为：

    extern 变量名表;
  - 如果在每一个文件模块中都定义一次全局变量，模块单独编译时不会发生错误，一旦把各模块连接在一起时，就会产生对同一个全局变量名多次定义的错误
  - 反之，不经声明而直接使用全局变量，程序编译时会出现“变量未定义”的错误。

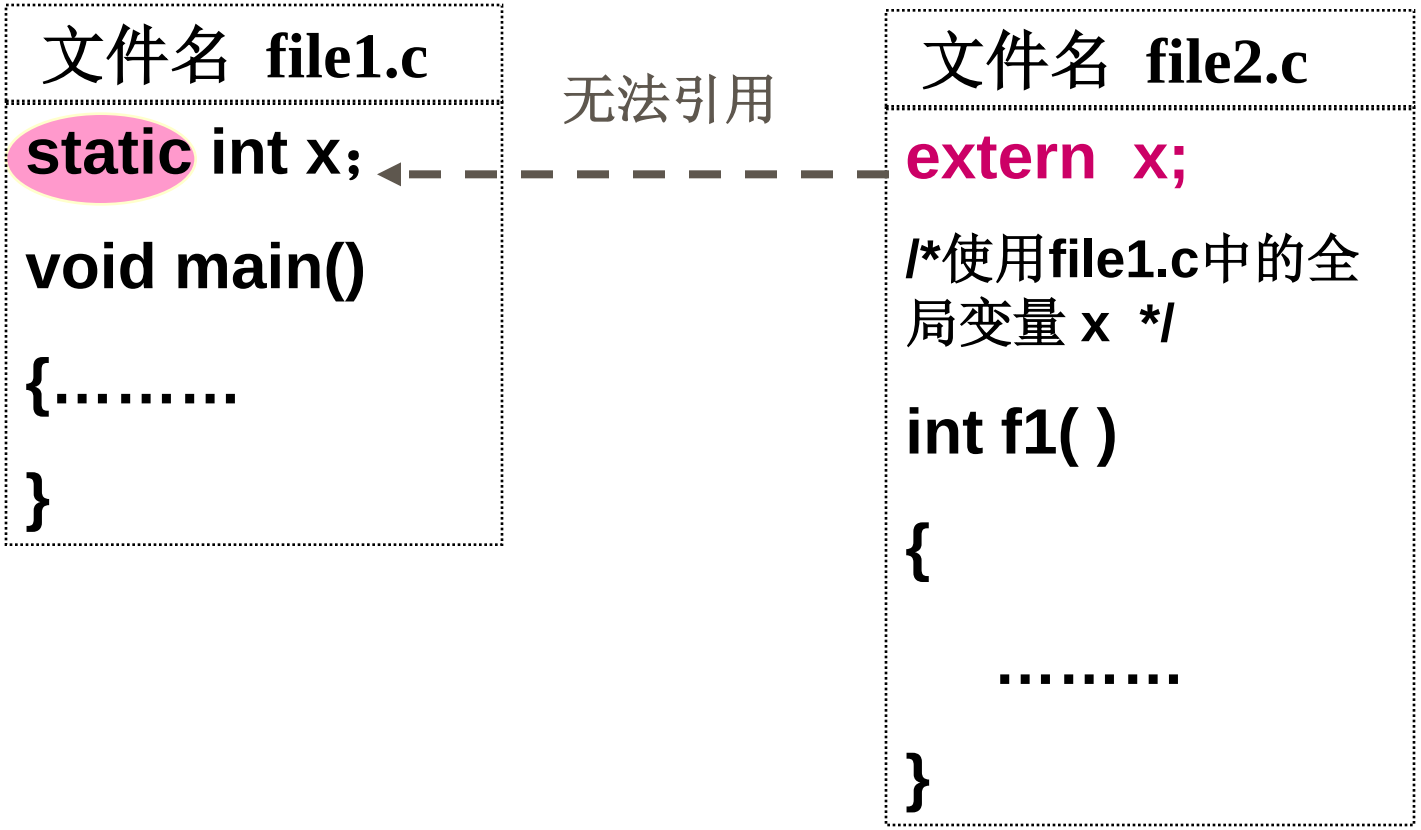

文件名 file1.c
扩大全局变量
的作用域
文件名 file2.c
int x;
void main()
{.....
}
extern x;
/*使用file1.c中的全
局变量 x */
f1( )
{
.....
}

文件模块间的通信

- 静态全局变量
  - 当一个大的程序由多人合作完成时，每个程序员可能都会定义一些自己使用的全局变量
  - 为避免自己定义的全局变量影响其他人编写的模块，即所谓的全局变量副作用，静态全局变量可以把变量的作用范围仅局限于当前的文件模块中
  - 即使其他文件模块使用外部变量声明，也不能使用该变量。

静态全局变量

使全局变量只限于本文件引用，而不能被其他文件引用



文件模块间的通信

文件模块与函数

外部函数

- 如果要实现在一个模块中调用另一模块中的函数时，就需要对函数进行外部声明。声明格式为：

  extern 函数类型 函数名(参数表说明)；

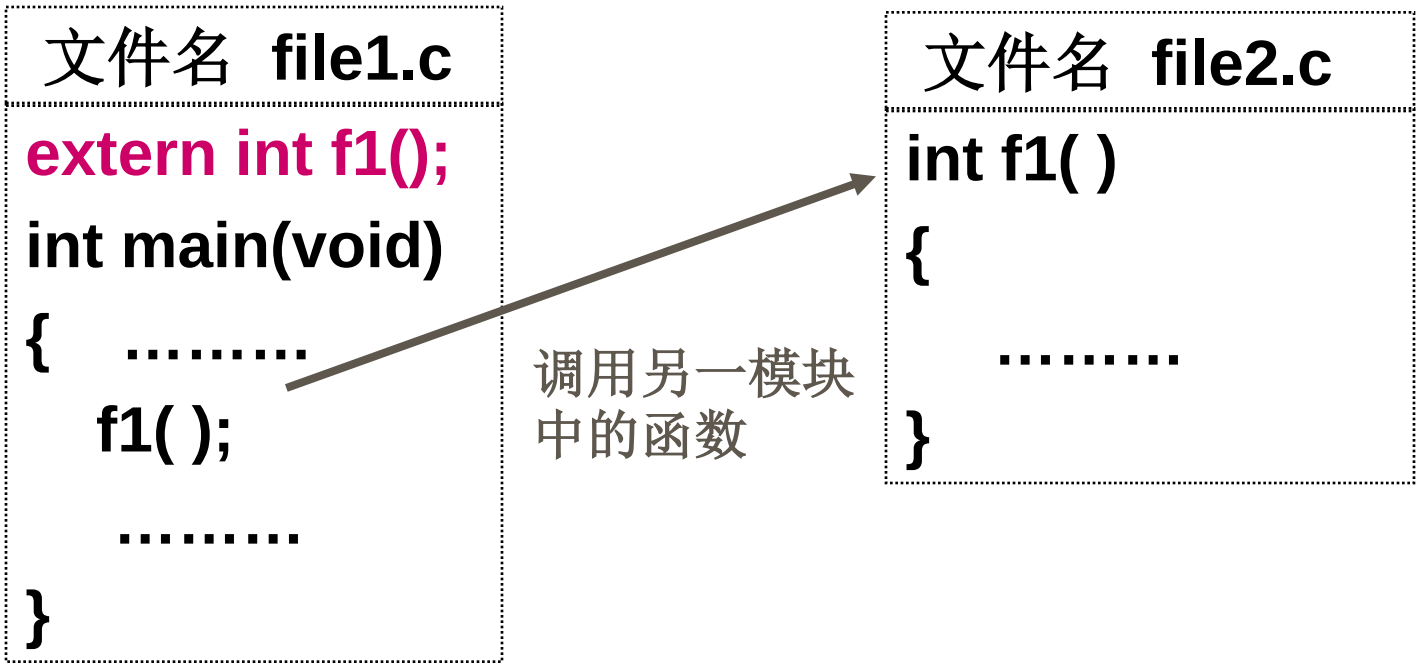
静态的函数

- 把函数的使用范围限制在文件模块内，不使某程序员编写的自用函数影响其他程序员的程序，即使其他文件模块有同名的函数定义，相互间也没有任何关联，
- 增加模块的独立性。

文件模块间的通信

- 外部函数
  - 函数能够被程序中的其他程序文件模块调用
  - 在其他文件模块中调用该函数前，声明为外部函数

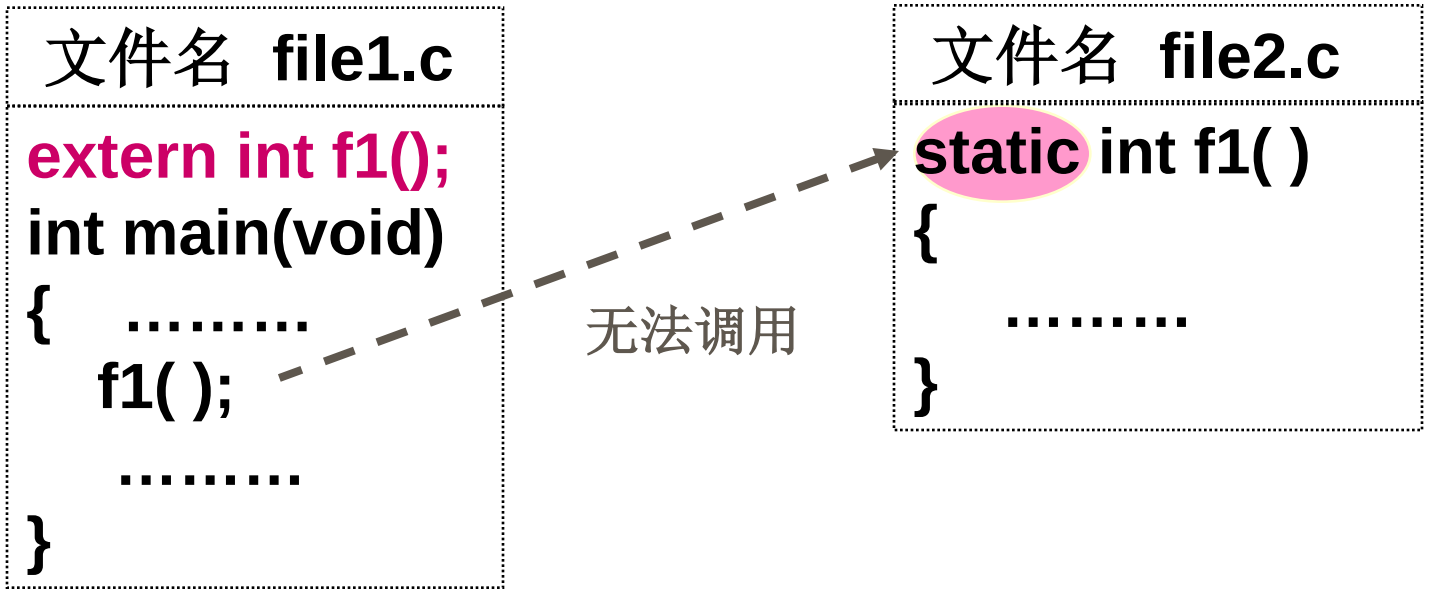
extern 函数类型 函数名(参数表说明);



⌘静态函数

使函数只能在本程序文件模块中被调用

static 函数类型 函数名(参数表说明);



编码规范

- 高质量的程序
  - 正确性：语法正确、功能正确。使之可行
  - 可读性：通用的、必需的习惯用语和模式可以使代码更加容易理解。使之优雅
  - 可维护性：程序应对变化的能力。使之优化
  - ……

编码规范

- 阅读（100下载）
  - 《C本学期作业代码自检规范》（课程project实践）
- 网上搜索资料（bing或者百度C编码规范）
  - http://cn.bing.com/search?q=c%E7%BC%96%E7%A0%81+%E8%A7%84%E8%8C%83&FORM=AWRE

- 规范很多，仅供参考

若干C代码规范

- 不直接使用基础类型
  - 利用typedef重新定义，大小和符号
  - 代替基本数据类型

若干C代码规范

- 《MISRA—C-2008工业标准》建议为所有基本数值类型和字符类型使用typedef重新定义。
- 对于32位计算机，它们是：

```
typedef char char_t;
typedef signed char  int8_t;
typedef signed short int16_t;
typedef signed int  int32_t;
typedef signed long  int64_t;
typedef unsigned char  uint8_t;
typedef unsigned short uint16_t;
typedef unsigned int  uint32_t;
typedef unsigned long  uint64_t;
typedef float  float32_t;
typedef double float64_t;
typedef long double float128_t;
```

若干C代码规范

- 变量、函数的命名符合编码规范
- Pascal命名规则：当变量名和函数名称是由二个或二个以上单字连结在一起，而构成的唯一识别字时:
  - 第一个单字首字母采用大写字母
  - 后续单字的首字母亦用大写字母
  - 例如：FirstName、LastName。

若干C代码规范

- 小心使用全局变量
  - 用访问器子程序来取代全局数据
    - GetValue/SetValue
  - 把数据隐藏到模块里面
    - 用static关键字来定义该数据
  - 写子程序读/写/初始化该数据
  - 要求模块外部的代码使用该访问器子程序来访问该数据，而不直接（使用变量名字）操作它。

程序设计专题

- 教学安排
  - 下下周一(26号）6-8节在东1B-208进行第一次的专题研讨课