



专题三 模块化程序设计

Part B


专题三 模块化程序设计

- 递归程序设计 (10.2)
- 编译预处理 (10.3)
 - 宏定义回顾 (10.3.1-10.3.3)
 - 文件包含 (10.3.4)
 - 编译预处理 (10.3.5)
- 大程序构成 (10.4)
- 大程序开发技巧

宏定义回顾

- 宏定义： **#define** 宏名标识符 宏定义字符串
 - 编译时与宏名相同的标识符，用宏定义字符串替代 (符号常量)
 - 在程序改动一处数值，实现程序中出现的所有实例都加以修改，看上去像函数，但没有函数调用开销
 - 不是C语句，后面不得跟分号
 - 带参数宏定义，嵌套使用
 - 作用范围：定义书写处 → 文件尾/**#undef**
 - 例如：

```
#define F(x) x - 2 // #define F (x) x - 2
#define D(x) x * F(x)
```


计算 $D(3)$, $D(D(3))$ → 先展开后代入参数

宏定义回顾

```
#define SUM(a,b) printf("#a " + " #b " = %d\n", ((a)+(b))  
SUM(1 + 2, 3 + 4); // #a → "a" 字符串化, 字符数组
```

```
#define NAME(n) num ## n
```

```
int num3 = 5;
```

```
printf("%d", NAME(3)); // ## 记号粘贴操作符
```

■ 优点

- 提高程序的可读性
- 方便集中修改，改动一个地方即可
- 连接符**##**实现如函数名、变量名连接的功能

■ 缺点

- 多层宏定义、嵌套宏定义的代码可读性较差
- 无法语法检查 (函数对参数类型进行语法检查)
- 宏定义代码一般不支持单步调试和断点
- 代码体积增大 (预编译时替换，无函数调用代价，但增加代码体积，函数不会增加代码体积)

文件包含 (10.3.4)

- 系统文件以 `stdio.h`、`math.h` 等形式供编程者调用
- 实用系统往往有自己诸多的宏定义，也以 `.h` 的形式组织、调用
 - 多个文件模块程序连接
- 问题：如何把若干 `.h` 头文件连接成一个完整的可执行程序？
 - 文件包含 `include`

文件包含

■ 格式

- `#include <需包含的文件名>` 系统文件夹
- `#include "需包含的文件名"` 当前文件夹+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

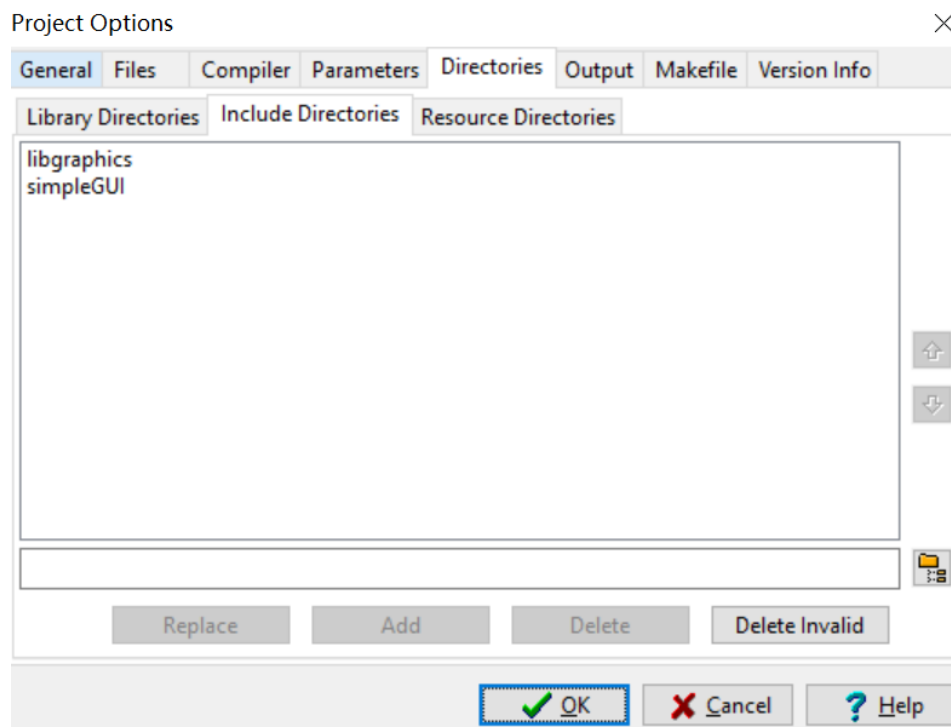
■ 注意

- 编译预处理命令，以**#**开头
- 在程序编译时起作用，不是真正的C语句，行尾没有分号

文件包含

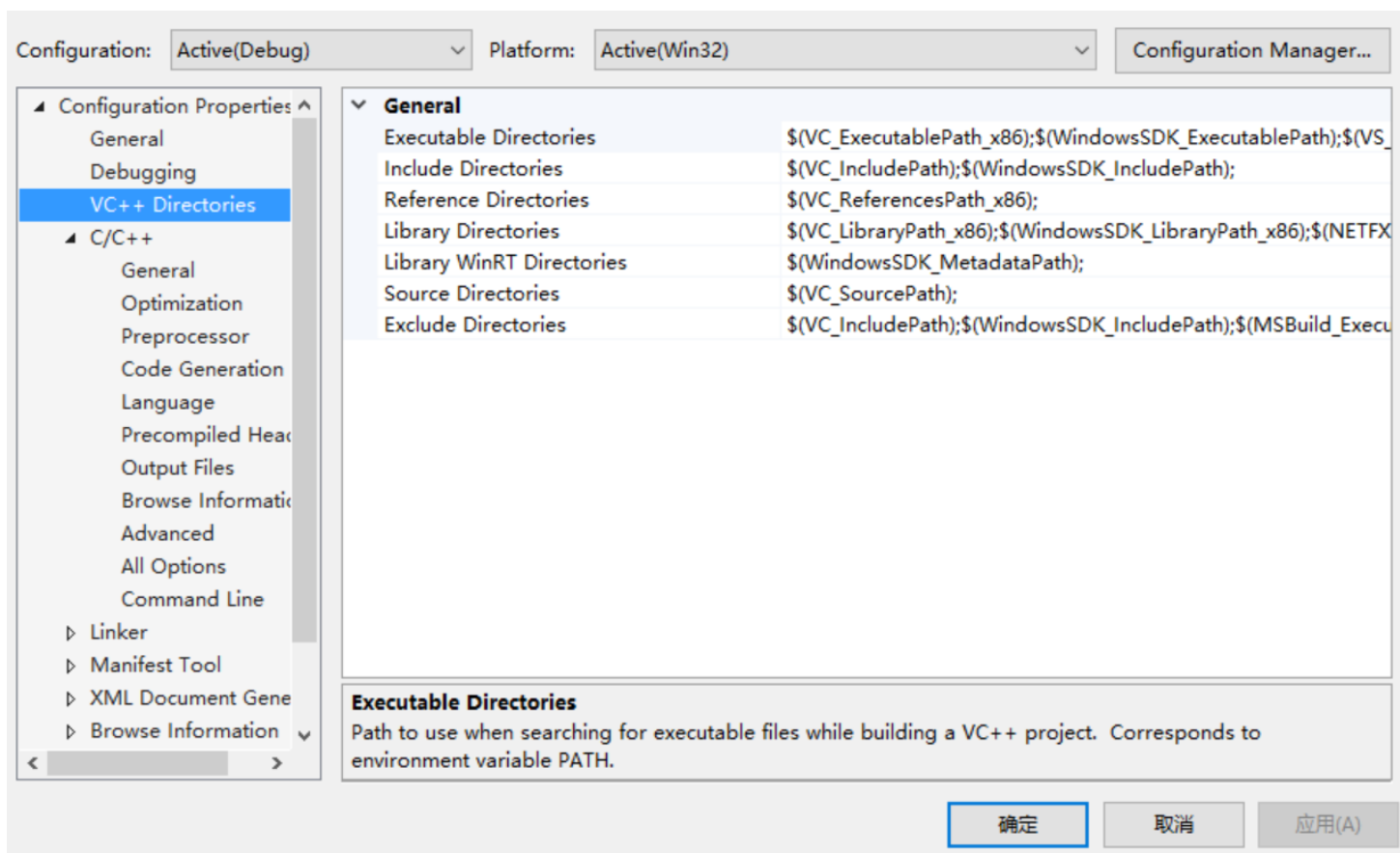
■ Dev-C++ 系统文件夹

- 菜单 Project → Project Options → Directories
→ Include Directories



文件包含

■ Visual Studio系统文件夹



例10-9 将例10-7中长度转换的宏，定义成头文件length.h，并写出主函数文件

头文件length.h源程序

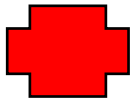
```
#define Mile_to_meter 1609      /* 1英里=1609米 */
#define Foot_to_centimeter 30.48 /* 1英尺=30.48厘米 */
#define Inch_to_centimeter 2.54 /* 1英寸=2.54厘米 */
```

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;  
}
```



编译连接后生成的程序

... *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)                                length.h  
{  
    float mile, foot, inch;  
    .....  
    return 0;  
}                                              prog.c
```

常用标准头文件

- `ctype.h` 字符处理
- `math.h` 与数学处理函数有关的说明与定义
- `stdio.h` 输入输出函数中使用的有关说明和定义
- `string.h` 字符串函数的有关说明和定义
- `stddef.h` 定义某些常用内容
- `stdlib.h` 杂项说明
- `time.h` 支持系统时间函数

编译预处理 (10.3.5)

- 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
- 文件包含(**#include**)和宏定义(**#define**)都是编译预处理指令
 - 在形式上都以"#"开头，不属于C语言中真正的语句
 - 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

c源程序 → 编译预处理 → 编译 → 优化程序 → 汇编程序 → 链接程序 → 可执行文件

编译预处理

- C程序的编译处理，目的是把每一条C语句用若干条机器指令来实现，生成目标程序
- 由于**#define**等编译预处理指令不是C语句，不能被编译程序翻译，需要在真正编译之前作一个预处理，解释完成编译预处理指令，从而把预处理指令转换成相应的C程序段，最终成为由纯粹C语句构成的程序，经编译最后得到目标代码

编译预处理功能

■ 编译预处理的主要功能

□ 宏定义 (**#define**)

- 宏标识符用宏字符串替换

□ 文件包含 (**#include**)

- 把指定的文件模块内容插入到**#include**所在的位置

□ 条件编译

- 如果length.h也包含**#include <stdio.h>**，那么stdio.h的代码在prog.c会重复两次，导致变量或函数重复定义？

编译预处理功能

■ 条件编译

□ 详细内容会在大程序开发技巧小节介绍

```
#define FLAG 1
```

```
#if FLAG
```

```
    程序段1
```

```
#else
```

```
    程序段2
```

```
#endif
```

```
#define FLAG 0
```

```
#if FLAG
```

```
    程序段1
```

```
#else
```

```
    程序段2
```

```
#endif
```

编译预处理总结

- 在形式上都以"#"开头，不属于C语言中真正的语句
- 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
- 宏定义 (**#define**)
 - 宏标识符用宏字符串替换
- 文件包含 (**#include**)
 - 把指定的文件模块内容插入到**#include**所在的位置
 - **# include <需包含的文件名>** 系统文件夹
 - **# include "需包含的文件名"** 当前文件夹+系统文件夹
- 条件编译
 - 根据宏自动选择代码，例如不同操作系统、debug/release

专题三 模块化程序设计

- 递归程序设计 (10.2)
- 编译预处理 (10.3)
- 大程序构成 (10.4)
 - C程序文件模块 (10.4.2)
 - 文件模块间的通信 (10.4.3)
 - 分模块设计学生信息库系统 (10.4.1)
- 大程序开发技巧

结构化程序设计方法

■ 自顶向下，逐步求精，函数实现

□ 自顶向下

- 先考虑全局目标，后考虑局部目标
- 先考虑总体步骤，后考虑步骤的细节
- 先从最上层总目标开始设计，逐步使问题具体化

□ 逐步求精

- 对于复杂的问题，其中大的操作步骤应该再将其分解为一些子步骤的序列，逐步明晰实现过程

□ 函数实现

- 通过逐步求精，把程序要解决的全局目标分解为局部目标，再进一步分解为具体的小目标，把最终的小目标用函数来实现
- 问题的逐步分解关系，构成了函数间的调用关系

函数设计时应注意的问题

■ 限制函数的长度

- 一个函数语句数不宜过多，既便于阅读、理解，也方便程序调试
- 若函数太长，可以考虑把函数进一步分解实现

■ 避免函数功能间的重复

- 对于在多处使用的同一个计算或操作过程，应当将其封装成一个独立的函数，以达到一处定义、多处使用的目的，以避免功能模块间的重复

■ 减少全局变量的使用

- 定义局部变量作为函数的临时工作单元，使用参数和返回值作为函数与外部进行数据交换的方式
- 只有当确实需要多个函数共享的数据时，才定义其为全局变量

C程序文件模块 (10.4.2)

- 结构化程序设计是编写出具有良好结构程序的有效方法
- 一个大程序会由几个文件组成，每一个文件又可能包含若干个函数
 - 如果程序规模很大，需要几个人合作完成的话，每个人所编写的程序会保存在自己的.c文件中
 - 为了避免一个文件过长，也会把程序分别保存为几个.h/.c文件

我们把保存一部分程序的文件称为程序文件模块

C程序文件模块

- 一个大程序可由几个程序文件模块组成，每一个程序文件模块又可能包含若干个函数
 - 程序文件模块只是函数书写的载体
- 当大程序分成若干文件模块后，可以对各文件模块分别编译，然后通过连接，把编译好的文件模块再合起来，连接生成可执行程序
 - 编译：a.c \rightarrow a.o, b.c \rightarrow b.o (不编译.h文件)
 - 连接：a.o, b.o, 系统库 \rightarrow test.exe
- 问题：如何把若干程序文件模块连接成一个完整的可执行程序？
 - 文件包含 (#include "a.c")
 - 工程文件 (由具体语言系统提供)

C程序文件模块

■ 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

小程序一个.c文件 → 大程序多个.c文件

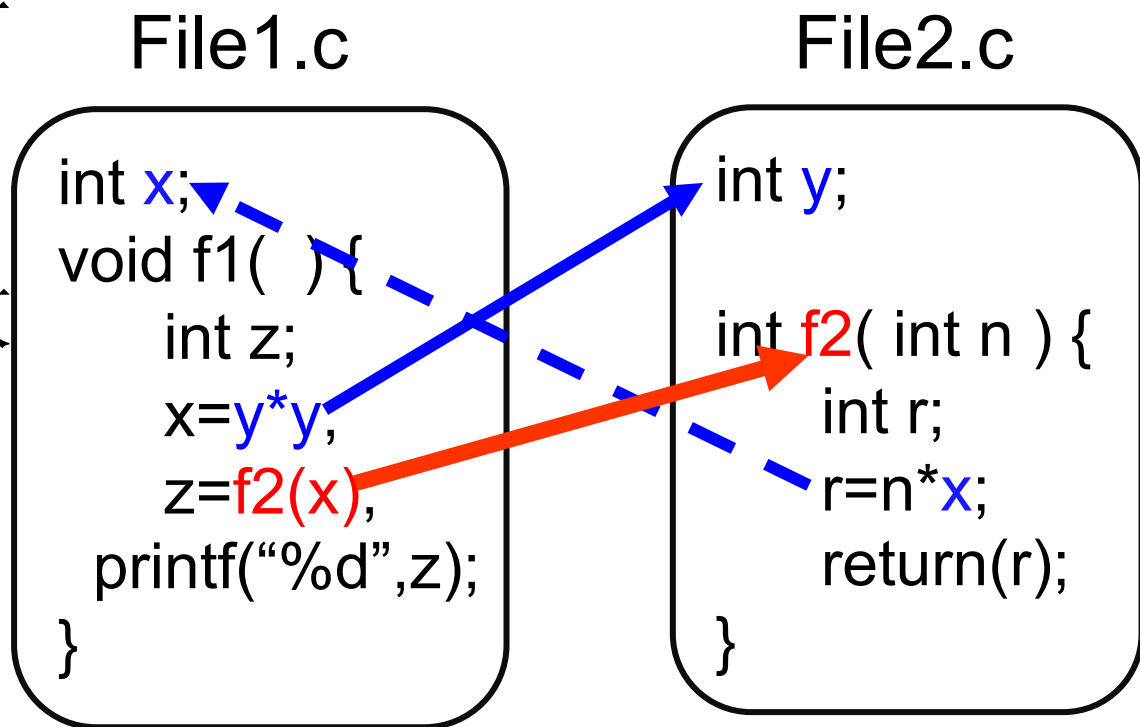
文件模块间的通信 (10.4.3)

■ 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



问题：这两个文件之间的变量和函数是否可以访问？

变量和函数 vs. 申明和定义

文件模块间的通信

■ 外部变量 (跨文件模块使用)

- 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

File2.c

```
int y;  
extern int x;  
int f2( int n ) {  
    int r;  
    r=n*x;  
    return(r);  
}
```


文件模块间的通信

■ 静态全局变量 (文件模块内使用)

- 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

别的文件就不能通过extern来访问y这个变量

File2.c

```
static int y;  
extern int x;  
int f2( int n ) {  
    int r;  
    r=n*x;  
    return(r);  
}
```

函数内静态局部变量 → 文件内静态全局变量

文件模块间的通信

类库开发者和使用者，例如
递归函数和封装函数，
`qsort`函数实现者和使用者

■ 文件模块与函数

□ 外部函数 (跨文件模块使用)

- 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);

例如: **extern int f2(int n);**

□ 静态函数 (文件模块内使用)

- 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
- 方法: **static**关键字申明，例如**static int f2(int n);**

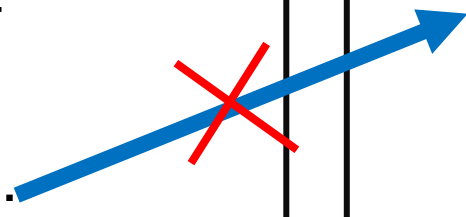
文件模块间的通信

File1.c

```
extern int f2( int n );  
int x;  
void f1( ) {  
    int z;  
    x=y*y;  
    z=f2(x);  
    printf(“%d”,z);  
}
```

File2.c

```
int y;  
static int f2(int n);  
int f2( int n ) {  
    int r;  
    r=n*x;  
    return(r);  
}
```



文件模块间的通信总结

- 跨文件模块使用
 - 外部全局变量 **extern** 变量名表;
 - 外部全局函数 **extern** 函数原型;
- 仅定义的文件模块内使用
 - 静态全局变量 **static** 变量定义;
 - 静态函数 **static** 函数原型;
- 静态全局变量和静态函数作用
 - 增强文件模块的独立性
 - 避免同名变量和函数的相互干扰

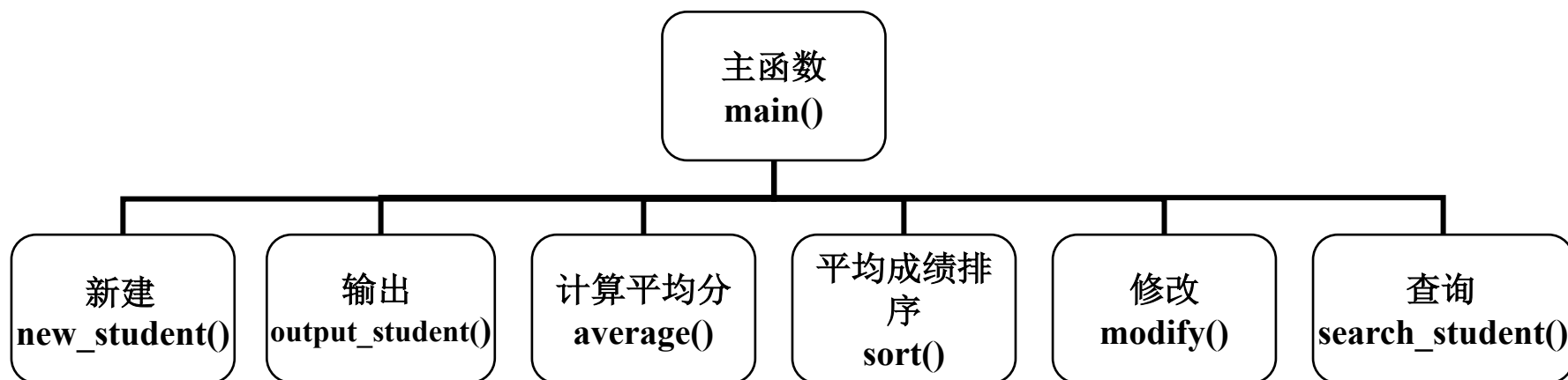
变量类型总结

变量类型	初始化	作用范围	生命周期
函数的形式参数	赋值为实参的数值	变量和形参定义的函数内部	函数调用时，定义变量，分配存储单元，函数调用结束，自动收回存储单元
函数内部的变量 auto, register	随机值(未赋值)		
复合语句内部的变量 auto	随机值(未赋值)	复合语句内部	复合语句执行开始，到复合语句结束时
全局变量	0	定义/申明开始到文件结束，可以在多个文件使用 (extern)	程序开始执行直到程序结束
静态局部变量	0，函数第一次调用时赋初值	变量定义的函数	
静态全局变量	0	定义开始到文件结束，只能在定义文件使用	

宏定义从定义开始到文件结束或**#undef**

分模块设计学生信息库系 (10.4.1)

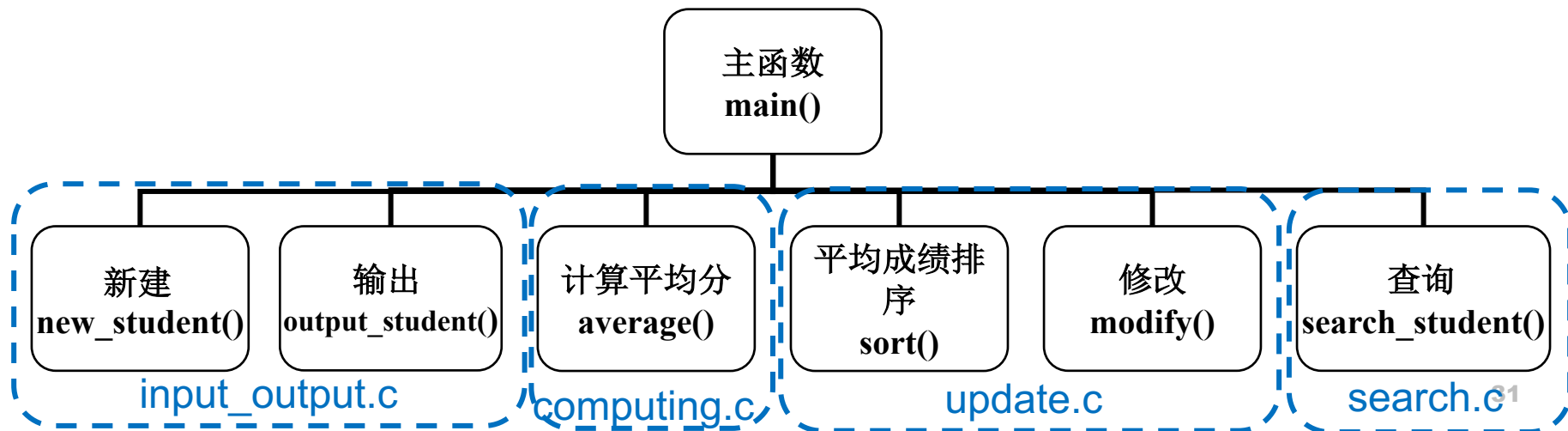
- 例10-10 请综合例9-1、例9-2、例9-3，分模块设计一个学生信息库系统。该系统包含学生基本信息的建立和输出、计算学生平均成绩、按照学生的平均成绩排序以及查询、修改学生的成绩等功能



分模块设计学生信息库系

- 由于整个程序规模较大，按照功能图，分成五个程序文件模块

- 输入输出程序文件input_output.c
- 计算平均成绩程序文件computing.c
- 修改排序程序文件update.c
- 查询程序文件search.c
- 主函数程序文件main.c



例10-10 程序文件模块 - 文件包含

- 分成五个程序文件模块，所有文件存放在同一个文件夹下，采用文件包含的形式进行连接和调用

□ 主函数程序文件main.c

```
#include "input_output.c"
```

```
#include "computing.c"
```

```
#include "update.c"
```

```
#include "search.c"
```

```
int Count = 0;          /* 全局变量，记录当前学生总数 */
```

```
int main(void)
```

```
{ struct student students[MaxSize]; /* 定义学生信息结构数组 */  
  new_student (students);           /* 输入学生信息结构数组 */  
  average(students);                /* 计算每一个学生的平均成绩 */  
  sort(students);                   /* 按学生的平均成绩排序 */  
  output_student(students);         /* 显示排序后的结构数组 */  
  modify(students);                 /* 修改指定输入的学生信息 */  
  output_student(students);         /* 显示修改后的结构数组 */  
  return 0;  
}
```


例10-10 程序文件模块 - 文件包含

■ 输入输出程序文件input_output.c

```
extern int Count;                /* 外部变量声明 */
void new_student (struct student students[ ]) /*新建学生信息*/
{
    .....
}
void output_student(struct student students[ ]) /*输出学生信息*/
{
    .....
}
```

■ 计算平均成绩程序文件computing.c

```
extern int Count;                /* 外部变量声明 */
void average(struct student students []) /*计算个人平均成绩 */
{
    .....
}
```

例10-10 程序文件模块 - 文件包含

■ 修改排序程序文件update.c

```
extern int Count;                /* 外部变量声明 */
void modify(struct student *p)   /* 修改学生成绩 */
{
    .....
}
void sort(struct student students[]) /* 平均成绩排序 */
{
    .....
}
```

■ 查询程序文件search.c

```
extern int Count;                /* 外部变量声明 */
void search_student(struct student students[ ], int num)
/*查询学生信息*/
{
    .....
}
```

例10-10 程序文件模块 - 工程包含

■ 多文件模块连接：文件包含→工程文件

- 头文件student.h (新增)

- 把结构体定义等写成一个头文件

- 输入输出程序文件input_output.c (更新)

- 计算平均成绩程序文件computing.c (更新)

- 修改排序程序文件update.c (更新)

- 查询程序文件search.c (更新)

- 主函数程序文件main.c (更新)

例10-10 程序文件模块 - 工程包含

■ 头文件student.h

```
#ifndef _STUDENT_H_  
#define _STUDENT_H_
```

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

```
#define MaxSize 50
```

```
struct student{  
    int num;  
    char name[10];  
    int computer, english, math;  
    double average;  
};
```

/*学生信息结构定义*/

/*学号*/

/* 姓名 */

/* 三门课程成绩 */

/* 个人平均成绩 */

/* 其他文件模块定义的全局变量，需要声明为外部变量 */

```
extern int Count;
```

```
#endif
```

例10-10 程序文件模块 - 工程包含

■ 主函数程序文件main.c

```
#include "student.h"
```

```
int Count = 0;          /* 全局变量，记录当前学生总数 */
```

```
int main(void)
```

```
{ struct student students[MaxSize]; /* 定义学生信息结构数组 */  
  new_student (students);           /* 输入学生信息结构数组 */  
  average(students);                /* 计算每一个学生的平均成绩 */  
  sort(students);                   /* 按学生的平均成绩排序 */  
  output_student(students);         /* 显示排序后的结构数组 */  
  modify(students);                 /* 修改指定输入的学生信息 */  
  output_student(students);         /* 显示修改后的结构数组 */  
  return 0;  
}
```

例10-10 程序文件模块 - 工程包含

■ 输入输出程序文件input_output.c

```
#include "student.h"
void new_student (struct student students[ ]) /*新建学生信息*/
{
    .....
}
void output_student(struct student students[ ]) /*输出学生信息*/
{
    .....
}
```

■ 计算平均成绩程序文件computing.c

```
#include "student.h"
void average(struct student students []) /*计算个人平均成绩 */
{
    .....
}
```

例10-10 程序文件模块 - 工程包含

■ 修改排序程序文件update.c

```
#include "student.h"
void modify(struct student *p)  /* 修改学生成绩 */
{
    .....
}
void sort(struct student students[])  /* 平均成绩排序 */
{
    .....
}
```

■ 查询程序文件search.c

```
#include "student.h"
void search_student(struct student students[ ], int num)
    /*查询学生信息*/
{
    .....
}
```

大程序构成总结

■ C程序文件模块

- 多个.h/.c文件，编译.c文件，连接生成.exe
- 文件包含 (不建议) 与 工程文件 (建议)

■ 跨文件模块使用

- 外部全局变量 **extern** 变量名表;
- 外部全局函数 **extern** 函数原型;

■ 仅定义的文件模块内使用

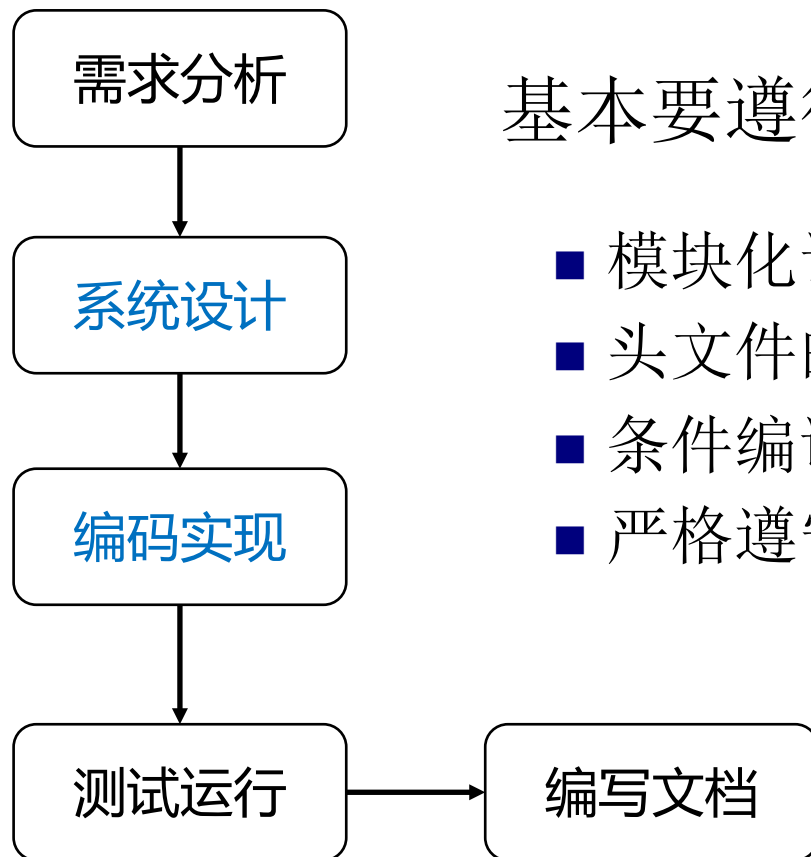
- 静态全局变量 **static** 变量定义;
- 静态函数 **static** 函数原型;

- 增强文件模块独立性，避免同名变量和函数的相互干扰

专题三 模块化程序设计

- 递归程序设计 (10.2)
- 编译预处理 (10.3)
- 大程序构成 (10.4)
- 大程序开发技巧
 - 模块化程序设计原则
 - 编码规范

大程序开发基本过程



基本要遵循软件工程过程

- 模块化设计原则
- 头文件的运用
- 条件编译的运用
- 严格遵守编码规范

模块化程序设计基本原则

■ 应用自顶向下的设计

- 把一个相对复杂的功能，划分成相对独立的子功能，直到每个子功能相对简单。每个子功能用一个函数来实现

■ 一个函数实现一个简单的功能

- 编程中：如果一个函数的代码行数很大(比如150以上)，最好的方法是把它分成几个相互调用的小函数来完成任务

■ 一个源(.c)文件包含功能相对集中的若干函数定义

- 如果一个源文件中包含很多个函数(比如50个以上)，最好的方法就是把程序再分成几个更小的源文件。每个源文件都包含一组功能相关的函数

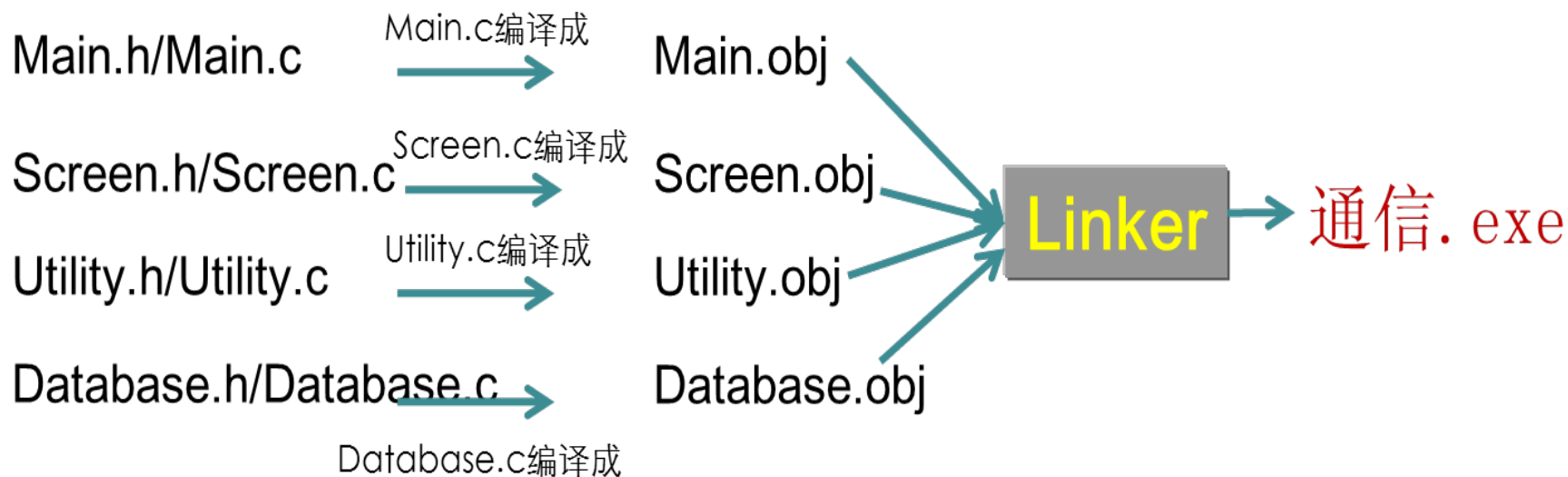
使用头文件

- 头文件：为了方便模块中的函数被别人调用，专门形成一个头文件，内容是函数声明(函数原型要求)、常量定义等，如**Utility.h**
 - 全局变量定义、函数定义在源文件(.c)中，如**Utility.c**
- 正确使用头文件，能够使得代码在可读性、文件大小和性能上大为改观
- 头文件的使用
 - 如果一个源文件如**main.c**中，要使用**Utility.h**中声明的函数、类型和具名符号等，在该源文件开始处

```
#include "Utility.h"
#include <stdio.h>
```

模块设计

- 模块：较小的源文件称为模块，包含main函数的模块叫主模块(main module)
- 独立编译单元：Utility.c → Utility.obj
- Example：某个通信管理数据库系统



怎样写高质量的函数

变量名一般使用“名词”或“形容词+名词”

- 好的函数名字：描述函数所做的所有事情(verbs, 动词或动词+名词)。如init(...)、isPrime(...)、drawTriangle(...)、calcMonthlyRevenues(...)
- 内聚性高，一个函数只实现一个功能
- 函数参数
 - 按照输入-修改-输出的顺序排列参数
 - 考虑对参数采用某种表示输入、修改、输出的命名规则
 - 使用所有的参数
 - 把状态或出错变量放在最后
 - 不要把函数的参数用作工作变量
 - 在接口中对参数的假定加以说明
 - 把函数的参数个数限制在大约7个以内

写头文件的技巧

- 头文件由三部分内容组成
 - 头文件开头处的版权和版本声明
 - 预处理块
 - 函数、结构和枚举类型声明、外部变量声明、具名常量定义、**typedef**和宏等
- 头文件应该只用于声明，而不应该包含或生成占据存储空间的变量或函数的定义

to see 某工程C代码

学习高人很重要



头文件的多重包含(#define保护)

■ 头文件多重包含问题如何避免？

- 所有头文件都应该使用`#define`防止头文件被多重包含(multiple inclusion)

方法：条件编译

■ 使用头文件的注意点

- 虽然函数、变量的声明都可以重复，所以同一个声明出现多次也不会影响程序的运行，但它会增加编译时间，重复引用头文件会浪费编译时间
- 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重复引用

```
#ifndef _HEADERNAME_H
#define _HEADERNAME_H
...//(头文件内容)
#endif
```


部分预处理指令

指令	用途
#	空指令，无任何效果
#include	包含一个源代码文件
#define	定义宏
#undef	取消已定义的宏
#if	如果给定条件为真，则编译下面代码
#ifdef	如果宏已经定义，则编译下面代码
#ifndef	如果宏没有定义，则编译下面代码
#elif	如果前面的# <code>if</code> 给定条件不为真，当前条件为真，则编译下面代码，其实就是 <code>else if</code> 的简写
#endif	结束一个# <code>if</code># <code>else</code> 条件编译块
#error	停止编译并显示错误信息

条件编译指令：DEBUG功能

```
#define DEBUG    /*此时#ifdef DEBUG为真*/
```

```
//#define DEBUG 0 //此时为假
```

```
int main()
```

```
{
```

```
    #ifdef DEBUG
```

```
        代码段A;
```

```
    #else
```

```
        代码段B;
```

```
    #endif
```

```
    代码段C;
```

```
    return 0;
```

```
}
```

这样就可以实现debug功能，每次要输出调试信息前，只需要#ifdef DEBUG判断一次。

不需要了就在文件开始定义#define DEBUG 0

规则2.4(建议): 代码段不应被“注释掉”(comment out)。当源代码段不需要被编译时，应该使用条件编译来完成

举例1: 条件编译指令

```
// 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*/
typedef struct ;
{ int i;
  struct A m;
} B;
void Foo2() // 函数定义，error
{ }
int a2; // 全局变量定义，error
#endif
```

```
//oneFile.c
#include "header.h"
int a1 = 0; /*定义正确，头文件已声明*/
int a2 = 0; /*不合规则*/
static int a3 = 0; /*正确，是静态的*/
static void Foo2() /*正确，是静态的*/
{ //... }
void Foo1()
{ //... } /*正确，在头文件中声明了*/
```

举例2: 多重定义oneFile.c与otherFile.c编译报错

```
// 头文件header.h
```

```
#ifndef _HEADER_H
```

```
#define _HEADER_H
```

```
char school[] = “浙江大学”;
```

```
#endif
```

```
// oneFile.c
```

```
#include "header.h"
```

```
// otherFile.c
```

```
#include "header.h"
```

link-time error:
"multiple definition of

说明：在其他文件中只要包含了header.h就会独立的解释，然后每个.c文件生成独立的标示符。在编译器链接时，就会将工程中所有的符号整合在一起，由于文件中有重名变量，于是就出现了重复定义的错误

头文件和源(实现)文件

■ 头文件(.h)

- 早期的编程语言如Basic、Fortran没有头文件的概念，C/C++语言的初学者虽然会使用头文件，但常常不明其理
- 通过头文件来调用函数功能
 - 在很多场合，源代码不便(或不准)向用户公布，只要向用户提供头文件和二进制的库即可。用户只需要按照头文件中的接口声明来调用库功能，而不必关心接口怎么实现的。编译器会从库中提取相应的代码
- 头文件能加强类型安全检查
 - 如果某个接口被实现或被使用时，其方式与头文件中的声明不一致，编译器就会指出错误，这一简单的规则能大大减轻程序员调试、改错的负担

■ 源(实现)文件(.c)由三部分内容组成

- 定义文件开头处的版权和版本声明
- 对一些头文件的引用
- 程序的实现体 (包括数据和代码)

头文件有哪三部分组成？

C程序架构设计

■ 模块代码 (train.h/train.c)

- 通常，每一个.c文件(C源文件)都有一个对应的.h文件(头文件)，也有一些例外，如单元测试代码和另包含main()的.c文件

■ 普通头文件

- 包含全局函数声明、全局变量声明、struct结构定义、符号常量定义、typedef。如：baseType.h

■ 标准库文件

- C标准库头文件里是什么？

- stdio.h里面是格式化输入输出函数scanf()、printf()等函数声明

- C标准库文件源代码在哪里？

- 标准库文件是以lib这种编译好的静态库的形式提供，一般在lib文件夹里，厂家不可能给源文件，而且源文件不一定是C写的，也有些是汇编写的，就算是C不同厂家写的也不一定一样，接口一样就行

编码规范

■ 什么是高质量的程序

- 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
-

■ 编码规范很重要，特别是团队开发的时候

■ 注释、缩进对齐、空行、空格

- 提高代码可读性

《C编码规范》

《C本学期作业代码自检规范》

巧用typedef

- 不直接使用基础类型，应该使用指示了大小和符号的**typedef**以代替基本数据类型
- 比如，《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如下的**typedef**。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;
```


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 (firstName, lastName)`

■ 慎用全局变量

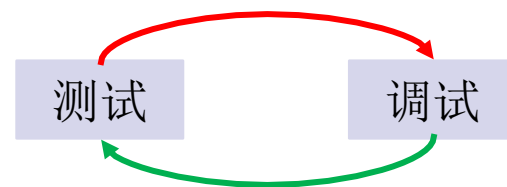
- 多线程代码中非常数全局变量是禁止使用的。内建类型的全局变量是允许的，但使用时务必三思

■ 用访问器子程序来取代全局数据

- 把数据隐藏到模块里面。用 `static` 关键字来定义该数据，写出可以 `read`、`write` 和 `initialize` 初始化该数据的子程序来。要求模块外部的代码使用该访问器子程序来访问该数据，而不是直接操作它

测试

- 测试用例：[输入数据、相应的预期输出]
- 测试用例尽可能多，覆盖各种可能情况
 - 各个判断分支
- “边界”情形、“极限”情形的测试
 - “==”的判断
 - 分母是非常小/大的数字
 - 字符串最长，个数最多
 -
- 未通过测试，说明程序中存在错误
 - 调试：查找并改正程序中的错误



专题三 模块化程序设计

- 递归程序设计 (10.2)
- 编译预处理 (10.3)
- 大程序构成 (10.4)
- 大程序开发技巧
 - 模块化程序设计原则
 - 编码规范