计算机程序设计基础(1)
07 模块程序设计（上）

清华大学电子工程系
杨昉
E-mail: fangyang@tsinghua.edu.cn

上节内容

- 当型循环与直到型循环
  - while语句
  - do-while语句
  - for语句
- 循环的嵌套与其他有关语句
  - 循环的嵌套
  - break 语句
  - continue 语句

课程回顾:依照结构和规则以语句为单位进行结构化程序设计

类型	定义	说明
当型循环(while)	while (表达式) 循环体语句;	**先判断后执行**;条件满足执行循环,不满足退出循环; **可能一次都不执行**
直到循环(do-while)	do 循环体语句 while (表达式);	**先执行后判断**;条件满足退出循环,不满足继续执行; **至少执行一次**
循环结构(for)	for(表达式1;表达式2;表达式3) 循环体语句;	精简地实现当型循环;三个表达式分别给**循环变量赋初值**、**判断循环条件**、**循环变量增值**; 表达式1、3**可省略**,但分号**不可省**;表达式2**必须有**
break	break;	**退出当前循环结构**(或switch结构)
continue	continue;	结束**本次循环**执行，但**不退出循环结构**

循环可嵌套，内循环**完全嵌套**于外循环中，内外循环**不交叉**，内外循环控制变量**不重名**

while和do-while语句循环**次数有限**和循环**次数未知**，需要有**改变循环条件语句**

for语句事先知道循环的**起始点**和**终止点**及其**循环次数**的问题

本节目录

7.1 模块化程序设计与函数

- 基本概念
- 函数的定义
- 函数的调用

7.2 模块间的参数传递

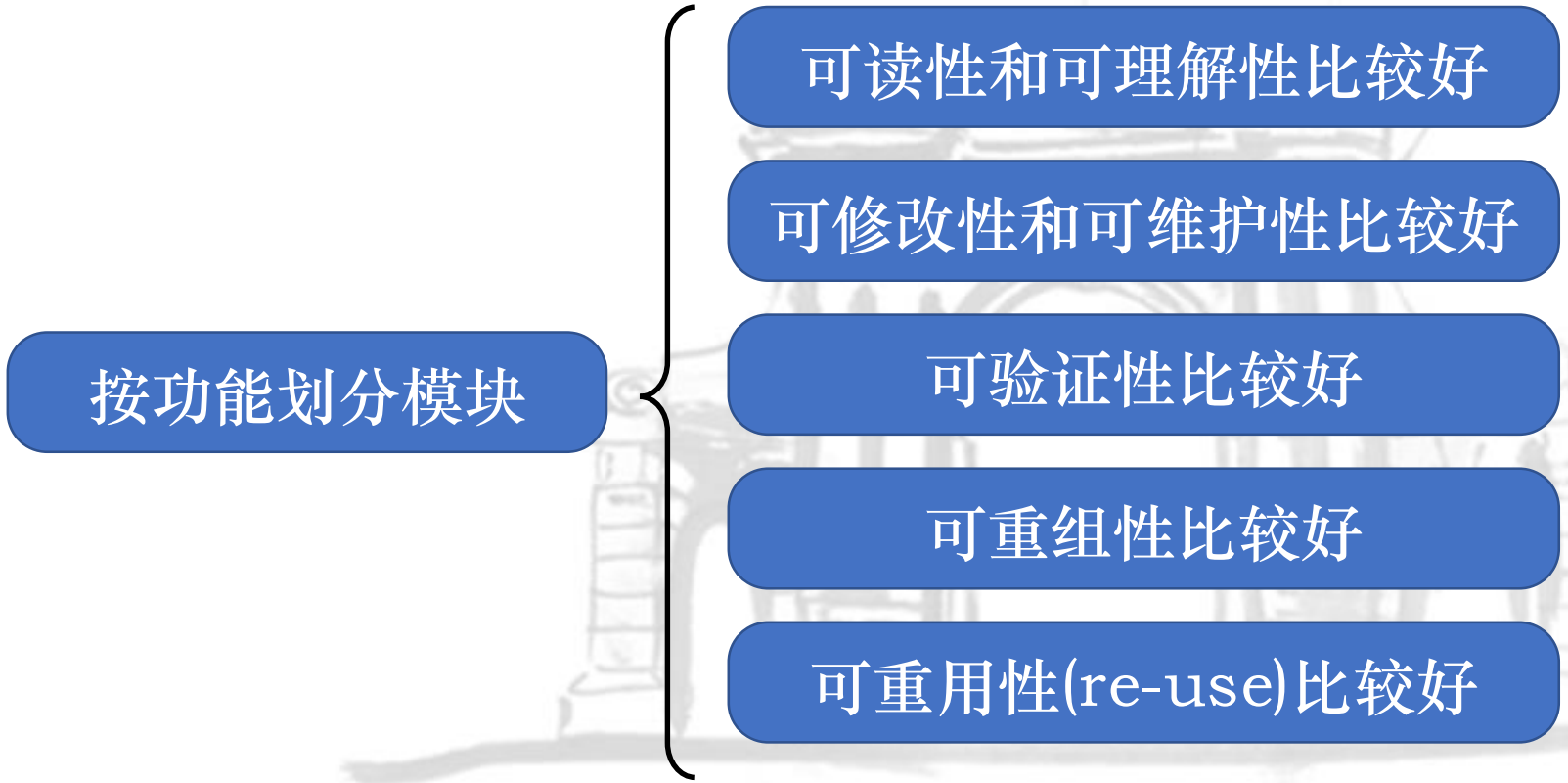
- 形参与实参的结合方式
- 局部变量与全局变量
- 动态存储变量与静态存储变量
- 内部函数与外部函数

7.1 模块化程序设计与函数

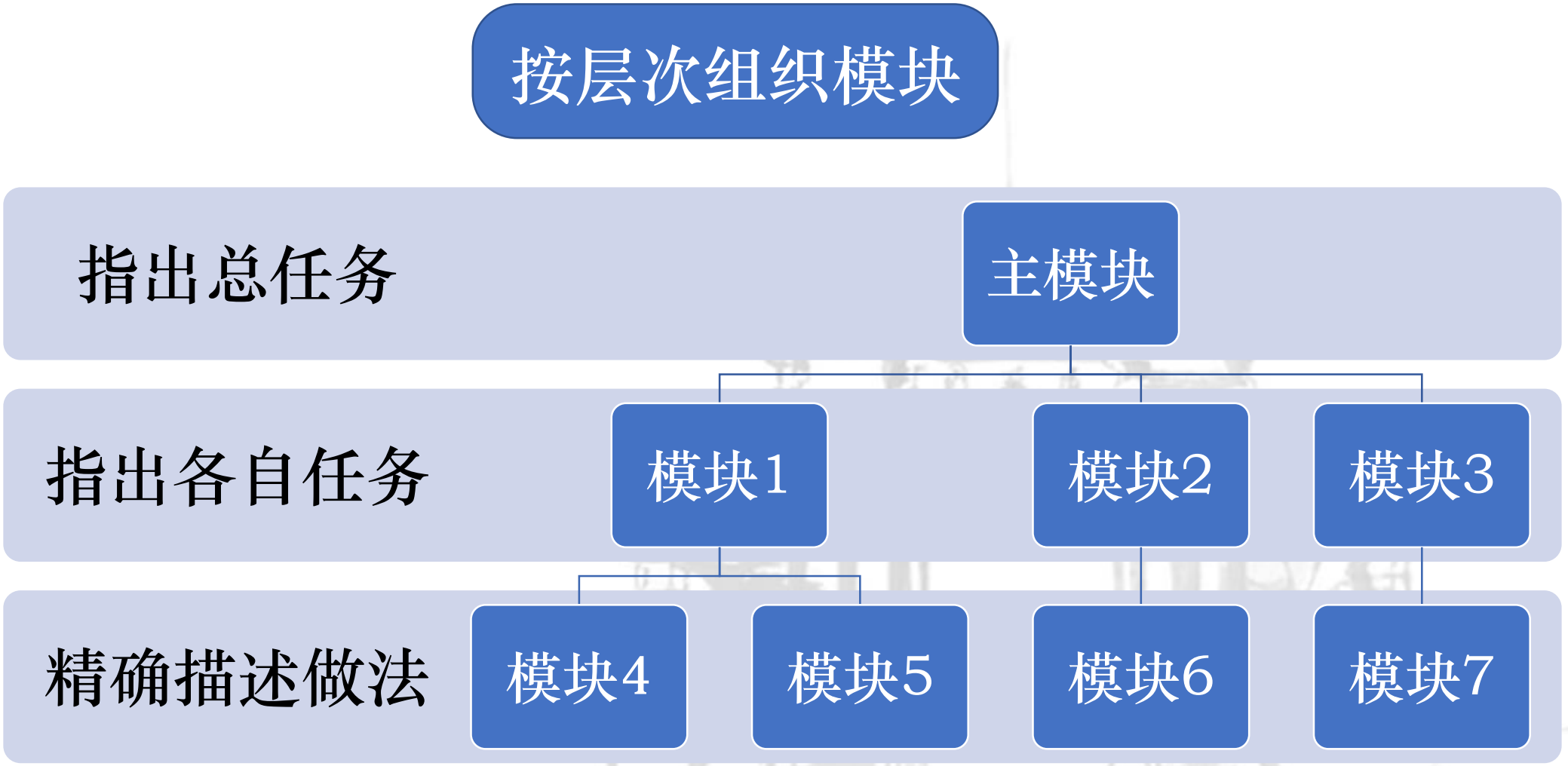
- 基本概念
- 函数的定义
- 函数的调用

模块化程序设计

- 模块化程序设计是指把一个大程序按人容易理解的大小**分解**为若干**模块**



模块化程序设计



模块化程序设计：函数

- C语言中，函数分为两种
  - **标准库**函数
    - scanf( )、printf( )、
    - abs( )、fabs( )、sqrt( )、exp( )、sin( )、cos( )
  - 用户自己定义的函数，函数定义形式：

```
类型标识符  函数名 (形参名称与类型列表) {
        说明部分
        语句部分
}
```

函数的定义

函数定义形式：

```
类型标识符 函数名 (形参名称与类型列表) {
    说明部分
    语句部分
}
```

- **类型标识符**：表示返回的函数值类型
  - 与变量类型说明符等同
  - C语言中可以定义**无类型**（即**void类型**）的函数，这种函数**不返回**函数值，而只是完成某个任务
  - 如果省略函数的类型标识符，则默认是**int型**，但现在的C语言标准不提倡这种省略方法，C++更是不允许

函数的定义

函数定义形式：

```
类型标识符 函数名 (形参名称与类型列表) {
    说明部分
    语句部分
}
```

- 形参名称与类型列表
  - 若其中有多个**形式参数**，则它们之间要用","分隔
  - 现在C语言标准提倡在形参表中直接对形参的**类型**进行说明

函数的定义

函数定义形式：

```
类型标识符 函数名 (形参名称与类型列表) {
    说明部分
    语句部分
}
```

- 返回语句（包含在语句部分）
  - 形式：　　return(表达式);　　或　　return 表达式;
  - 将**表达式的值**作为函数值返回给调用函数
  - 表达式的类型应与函数类型标识符**一致**

函数的调用：向前引用说明


- 形式

  函数类型 函数名(形参1类型，形参2类型，…);

  函数类型 函数名(形参1类型 形参名1,形参2类型 形参名2,…);

- 注意
  - 各“形参名”是任意的，可以与被调用函数中形参名不同
  - 说明中的函数类型以及各形参类型（包括形参个数）必须要与被调用函数定义中的一致
  - 例：int p(int); 与 int p(int x); 等价，其中标识符x是任意的

C语言中，上述这种对被调用函数的说明称为函数原型。
这种说明又被称为函数向前引用说明

函数的调用：向前引用说明

- 主要作用
  - 便于在编译源程序时对调用函数的合法性进行**全面检查**
  - 当编译系统发现与函数原型不匹配的函数调用时，就会显示出**警告**性质的错误信息
    - 如函数类型、参数个数、参数类型不匹配等
    - 用户可以根据系统提示的错误信息发现并改正函数调用中的错误
  - 绝对**不能**对这类编译警告性错误提示视而不见！

函数的定义

●例7-1：计算1到5之间各自然数的阶乘值

```
1  #include<stdio.h>
2  void main() { /*主函数*/
3    int n, m;
4    int p(int);
5     //说明要调用的函数p()返回值是int型,有一个int型形参
6    for (m = 1; m <= 5; m++)
7      printf("%d!=%d\n", m, p(m));
8  }
9    //计算阶乘值的函数,返回值为int型
10 int p(int n) {
11   int k, s;
12   s = 1;
13   for (k = 1; k <= n; k++)
14     s = s * k;
15   return(s);
16 }
```

函数的向前引用说明

被调用函数p的**返回值**是int型，此函数有一个int型**形参**

```
1!=1
2!=2
3!=6
4!=24
5!=120
请按任意键继续. . .
```

- 主函数main( )：通过循环调用函数p( )计算并输出m!的值
- p( )：它的功能是计算阶乘值，例如，p(m)是计算m!

函数的定义


●例7-2：计算并输出一个圆台两底面积之和

```
1  #include <stdio.h>
2  void main() {
3    double r1, r2;
4    double q(double);
5    //函数q()是双精度实型,有一个双精度实型形参
6    printf("input r1,r2:");     //提示输入
7    scanf("%lf%lf", &r1, &r2); //输入r1与r2
8    printf("s=%f\n", q(r1) + q(r2));
9  }
10   //计算圆面积的函数,为双精度实型
11 double q(double r) {
12   double s;
13   s = 3.1415926*r*r;
14   return s;
15 }
```

函数的向前引用说明

被调用函数q的返回值是double型，此函数有一个double型形参

q函数书写可以简化为：

```
double q(double r){
    return 3.1415926*r*r;
}
```

```
input r1,r2:1 2
s=15.707963
请按任意键继续. . .
```

课堂练习

练习7-1：阅读下面的代码（两段在同一文件内），若输入为1.1 4，请写出运行结果

```c
#include <stdio.h>
void main(){
   double base;
   int index;
   double pow(double, int);
   printf("input base,index:\n");
   scanf("%lf%d", &base, &index);
   printf("Ans=%lf\n", pow(base, index));
}
```

```c
double pow(double base, int index){
   double ans;
   int cnt;
   ans = 1.0;
   for (cnt = 0; cnt < index; cnt++){
      ans *= base;
   }
   return ans;
}
```

```
input base, index:
1.1 4
```

```
input base, index:
1.1 4
Ans=1.464100
请按任意键继续. . .
```

函数的定义

- 主函数main()
  - 一个完整的C程序可以由若干个函数组成，其中**必须有一个且只能有一个**主函数main( )
  - C程序总是**从主函数开始执行**(不管它在程序中的什么位置)，而其他函数只能被调用
- 函数
  - C语言中的函数没有**从属关系**，各函数之间**互相独立**，可以**互相调用**
  - C函数**不能嵌套定义**(不能在一个函数中定义另一个函数)

C程序放在多个文件

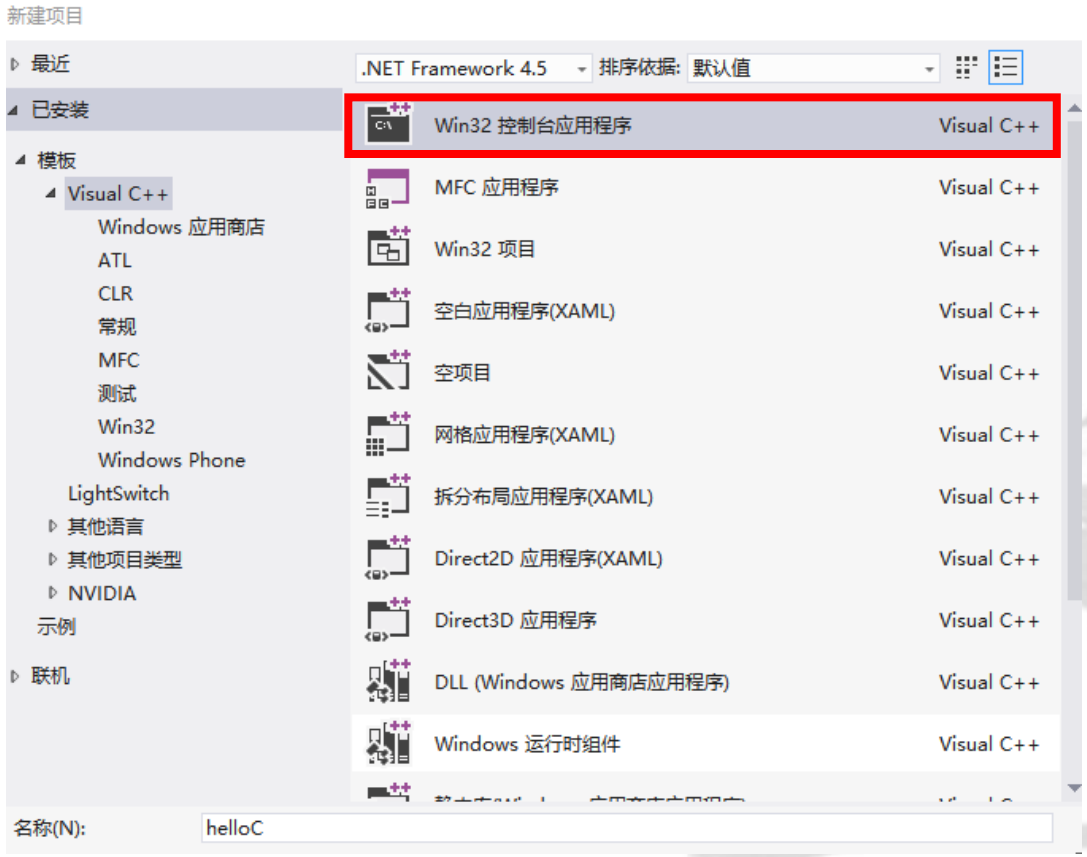
- 一个完整C程序中的所有函数可以放在一个文件中，也可以放在**多个文件**中
- 例7-3：将例7-1写到两个文件中

```c
/* 主函数main( )放在文件sp.c中*/
#include <stdio.h>
void main() {  /*主函数*/
   int m, a;
   int p(int);
   for (m = 1; m <= 5; m++){
      a=p(m);
      printf("%d!=%d\n", m, a);
   }
}
```

```c
/* 函数p( )放在文件sp1.c中*/
/* 计算阶乘的函数,返回值为整型*/
int p(int n) {
   int k, s;
   s = 1;
   for (k = 1; k <= n; k++)
      s = s * k;
   return s;
}
```

C程序放在多个文件

- 具体操作
  - 1. 建立一个空的项目（Project）

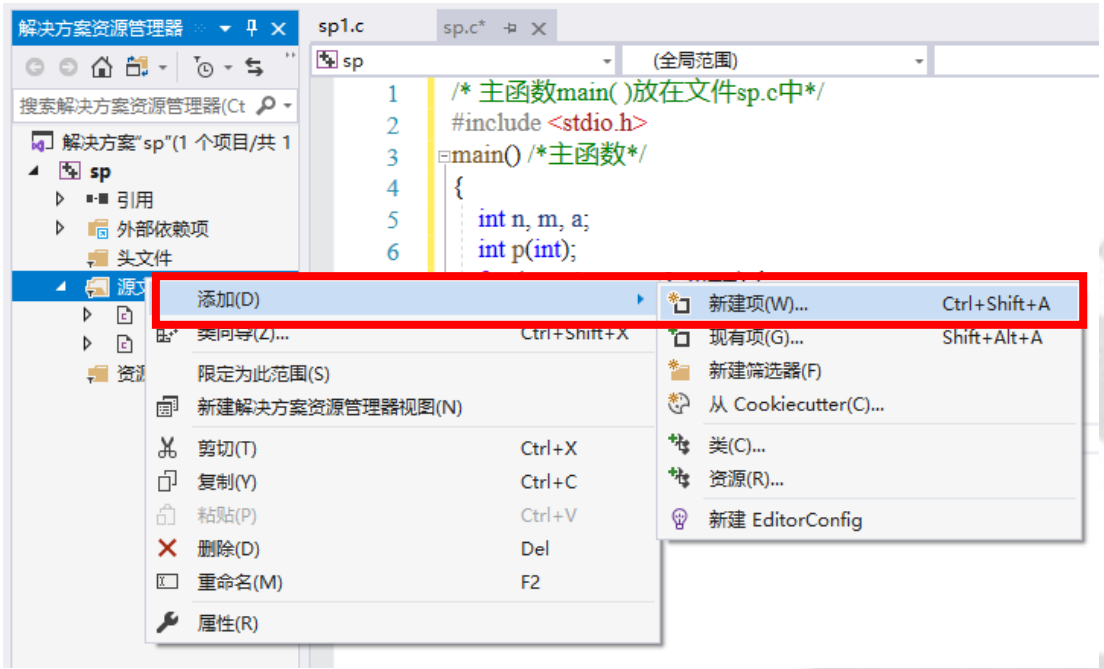


C程序放在多个文件

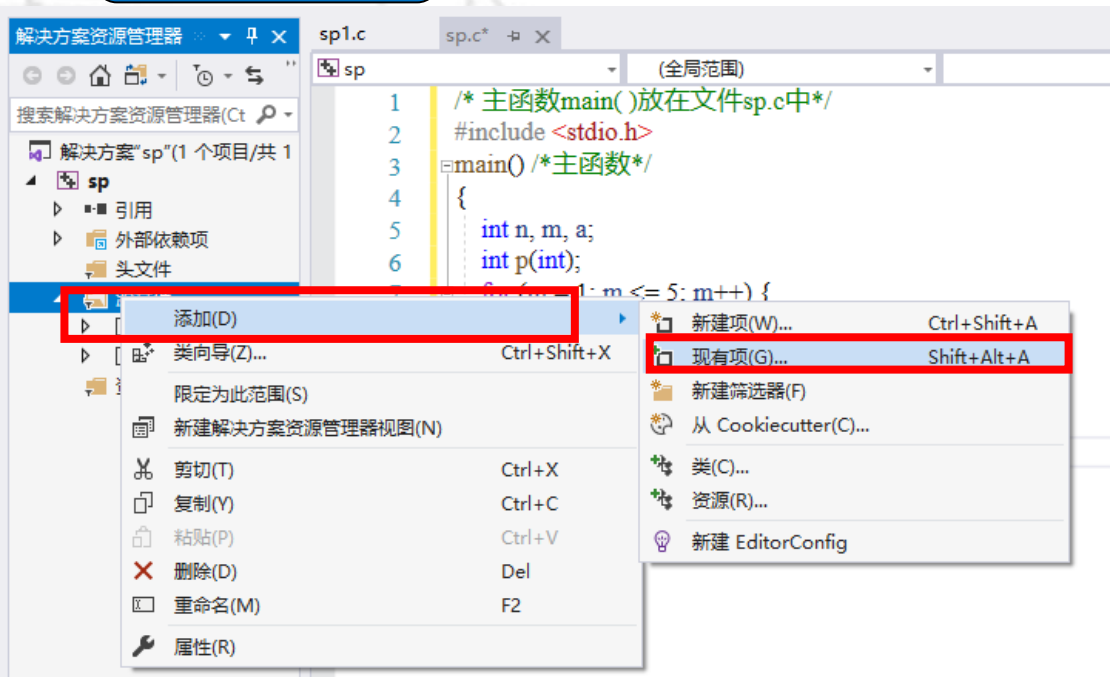
●具体操作

□2. 在项目中新建文件 或者 将多个文件插入到此项目中

新建文件

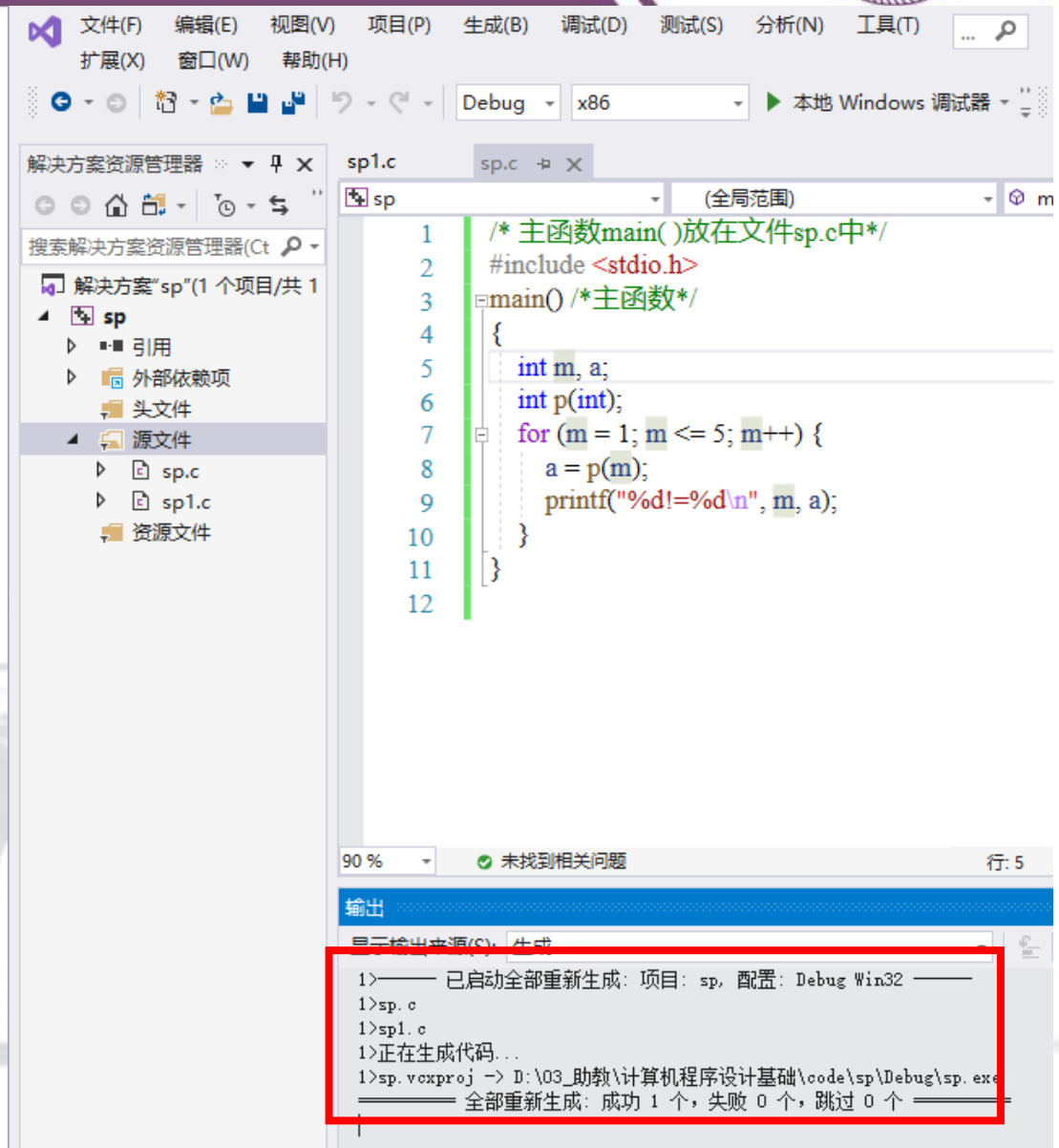


插入文件



C程序放在多个文件

- 具体操作
  - 3.用“重新生成”编译链接

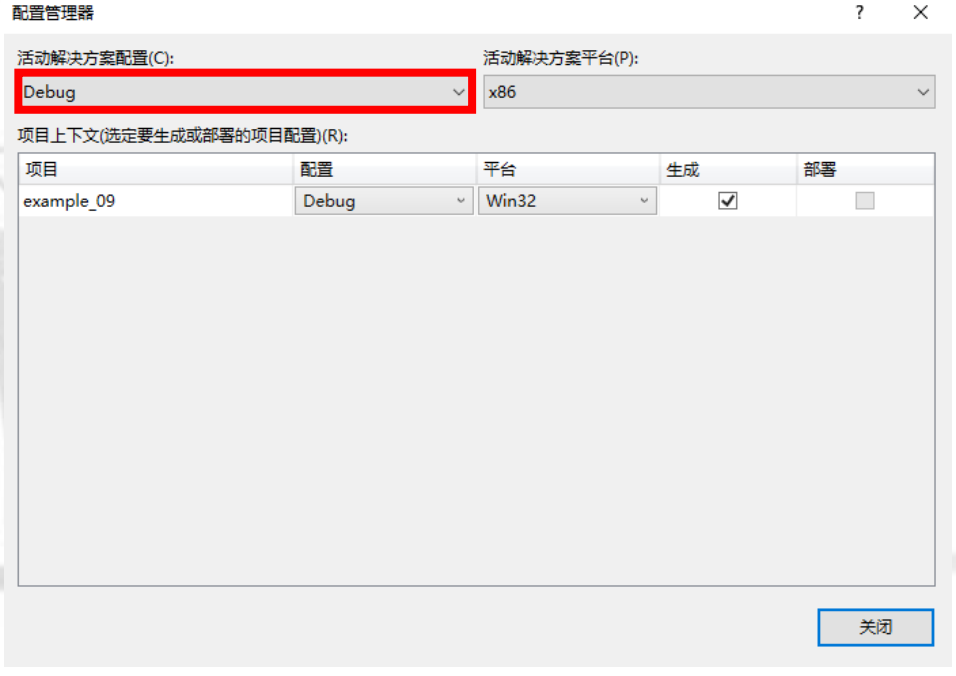


设置编译系统Release方式

- Visual Studio编译系统有两种编译方式
  - Debug（调试）方式
  - Release（发布）方式
  - 建立项目进行编译链接时，编译系统自动默认是Debug方式，从编译信息中可以看到“配置: Debug Win32”
  - Debug方式方便使用者调试追踪错误，但编译链接得到的可执行文件往往比Release方式编译链接得到的可执行文件要大，且运行速度慢
  - Debug方式自动增加了许多调试追踪信息

设置编译系统Release方式

- Visual Studio编译系统有两种编译方式
  - 当程序在Debug方式下确认没有任何错误后，可以切换到Release方式下再编译链接，产生最后的可执行程序
  - 方法是，选择“生成”菜单中的“配置管理器”项，将弹出如下所示的对话窗



设置编译系统Release方式

- Visual Studio编译系统有两种编译方式
  - 单击“活动解决方案配置”的下拉菜单，选中“Release”后按下“关闭”按钮，则选中激活了Release编译方式
  - 再次“重新生成”会得到如图所示的结果

```
1>------ 已启动全部重新生成: 项目: example_09, 配置: Release Win32 ------
1>source.c
1>正在生成代码
1>Previous IPDB not found, fall back to full compilation.
1>All 5 functions were compiled because no usable IPDB/IOBJ from previous compilation was found.
1>已完成代码的生成
1>example_09.vcxproj -> D:\03_助教\计算机程序设计基础\code\example_09\Release\example_09.exe
========== 全部重新生成: 成功 1 个，失败 0 个，跳过 0 个 ==========
```

Debug tips

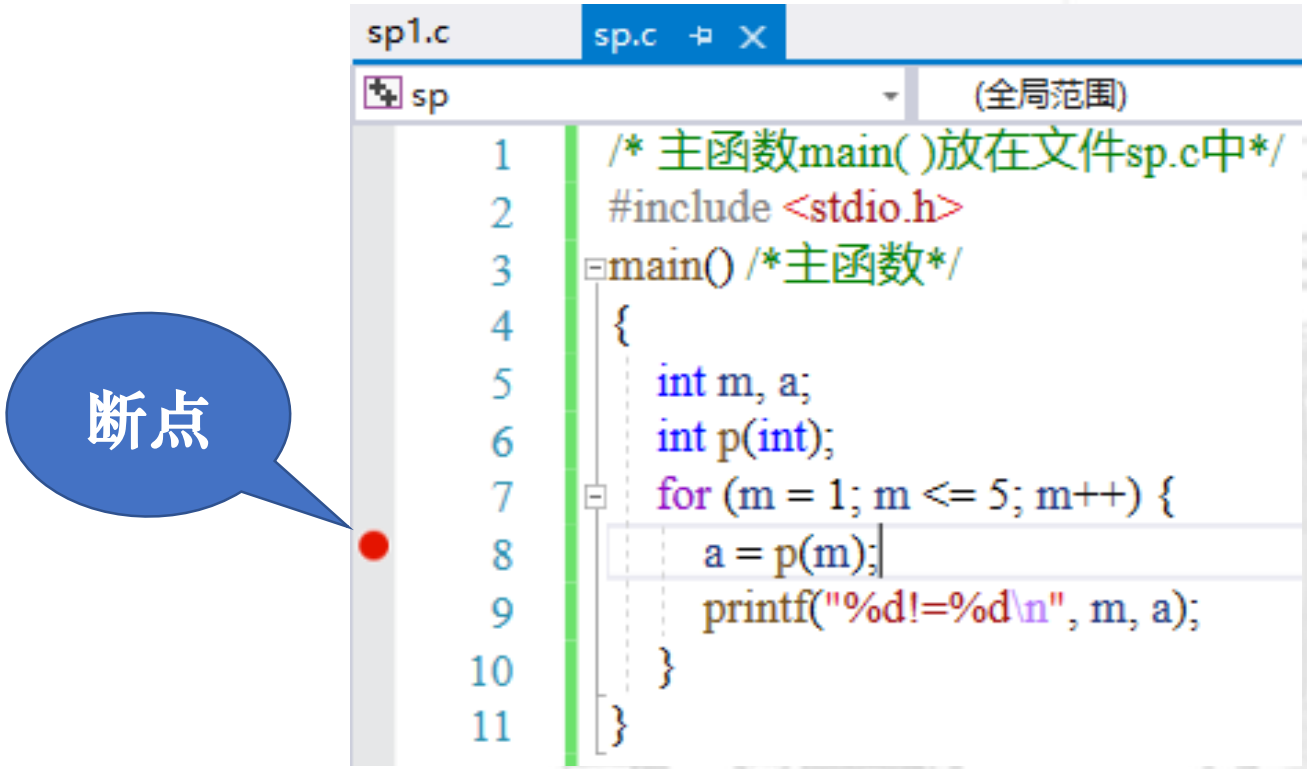
- 如何跟踪调试程序的逻辑错误
  - **断点**
    - 方便快捷，容易使用
  - **采用输出语句**
    - 较为直观，但须在代码中添加输出语句

Debug tips：断点

- 断点作用
  - 程序运行到断点所在行时，会暂停下来，让你查看此时**各个变量的值**
- Visual Studio添加断点的几种方式
  - 1. 点击需要加断点的行，**按F9键**
  - 2. 点击需要加断点的行，选**“调试”**菜单中的**“切换断点”**项
  - 3. 在需要加断点的行左侧点击**灰色区域**

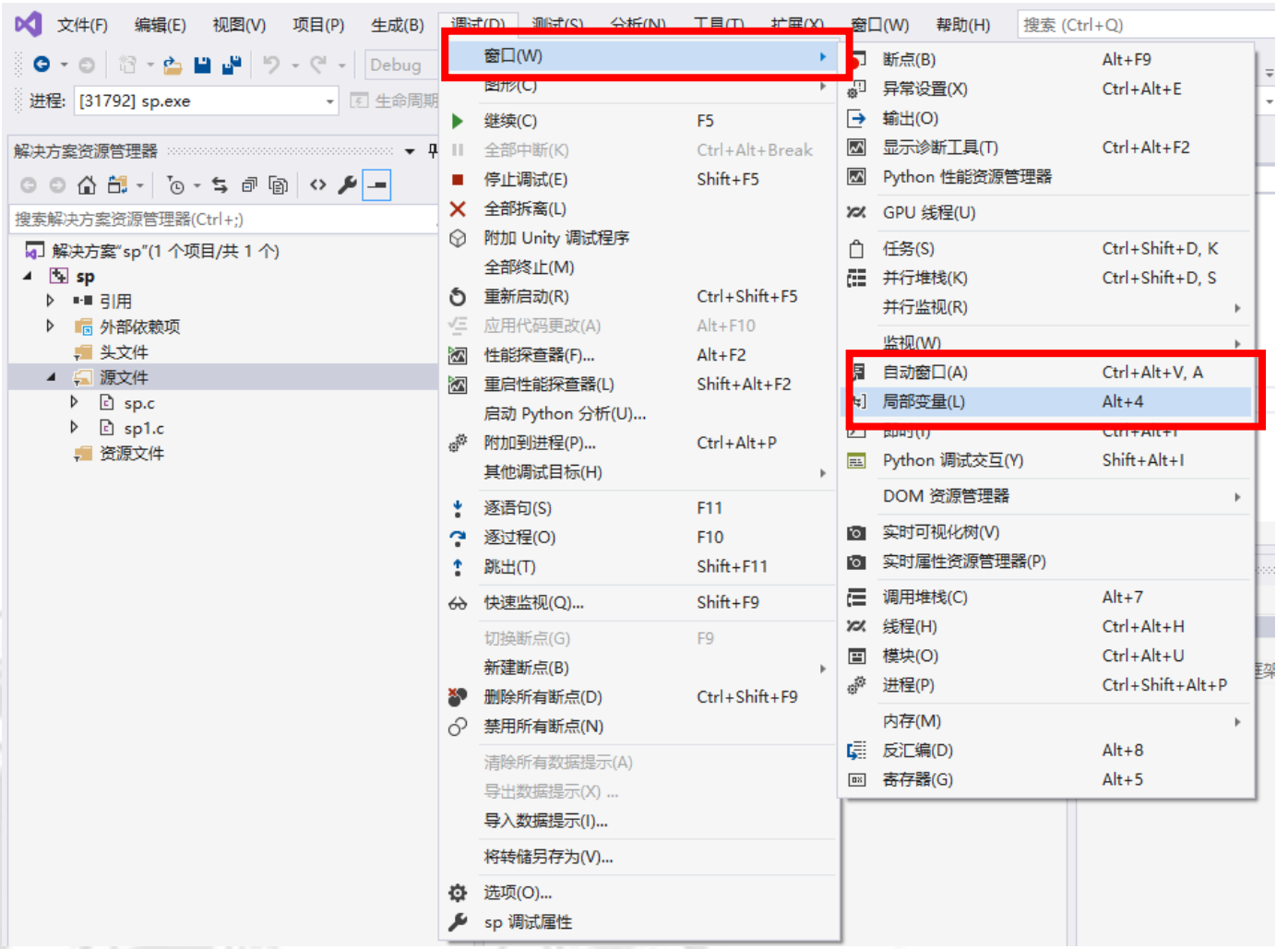
Debug tips：断点

- 添加断点后效果



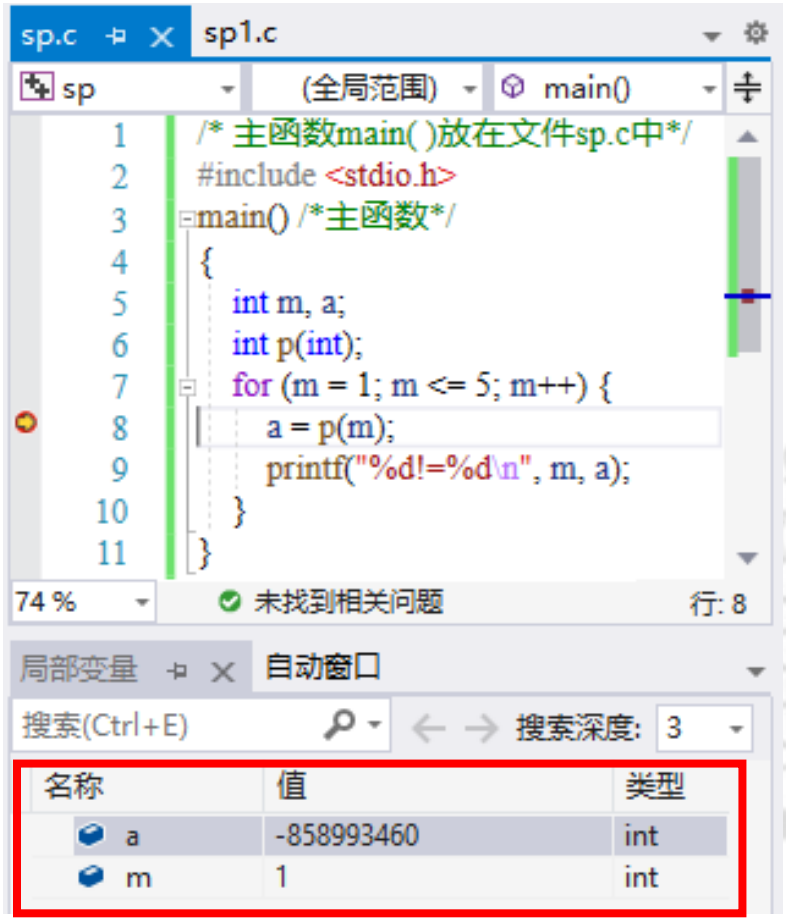
Debug tips：断点

- 添加断点后查看局部变量值
  - 调试->窗口->局部变量
  - 调试->窗口->自动窗口



Debug tips：断点

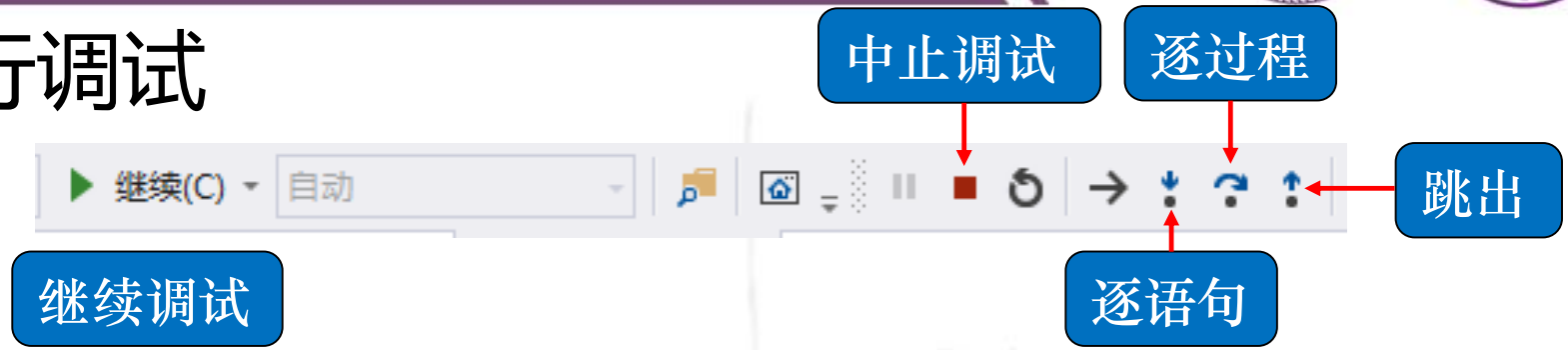
- 设置断点后进行调试
  - 程序执行到断点处会自动停下，显示当前变量的值



- for循环第一次执行，a尚未被赋值，为随机数值
- m值为1

Debug tips：断点

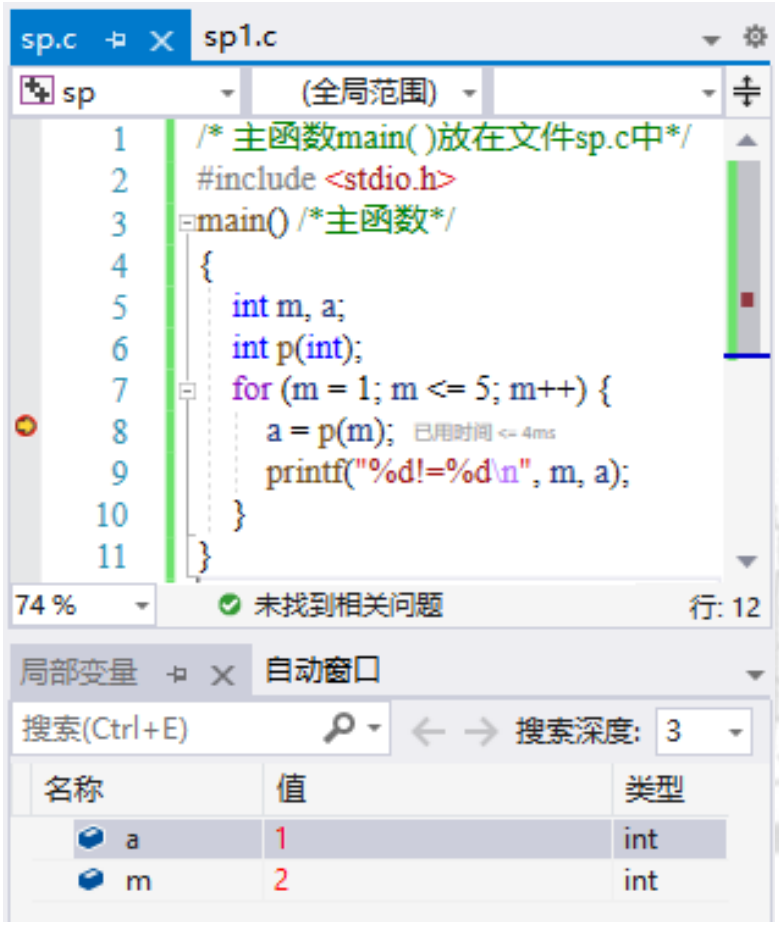
- 设置断点后进行调试
  - 工具栏



  - “继续调试”，程序将运行至**下一次断点出现的位置**
  - “逐语句”，每次执行一行语句
    - 如果碰到函数调用，它就会**进入到函数里面**
  - “逐过程”，每次执行一行语句
    - 碰到函数时不进入函数，**把函数调用当成一条语句执行**
  - “跳出”
    - 进入到函数内，直接执行函数内剩余的语句，返回到该函数被调用时的后面的语句处

Debug tips：断点

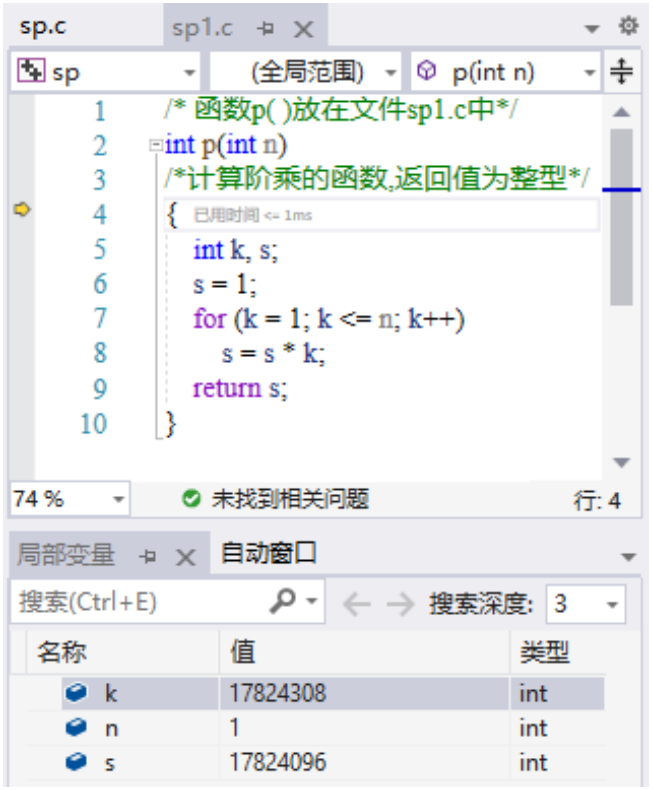
- 设置断点后进行调试
  - “继续调试”：程序将运行至**下一次断点出现的位置**



- for循环第二次执行，a在上一次循环中被赋值为p(1)=1
- m值为2

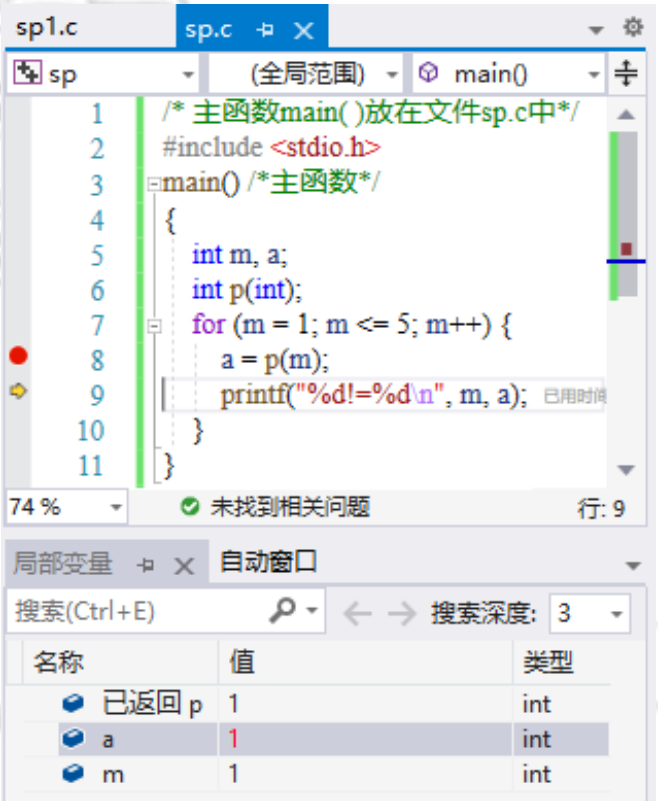
Debug tips：断点

- 设置断点后进行调试
  - “**逐语句**”：每次执行一行语句（进入函数）
  - “**逐过程**”：每次执行一行语句（不进入函数）



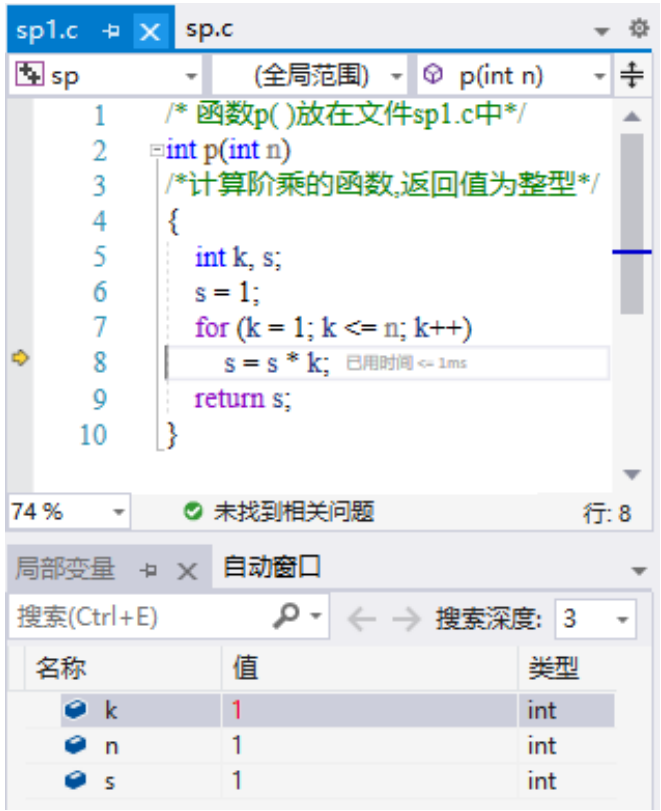
- 逐语句 进入到函数p()中

- 逐过程 a=p(m); 语句执行完毕

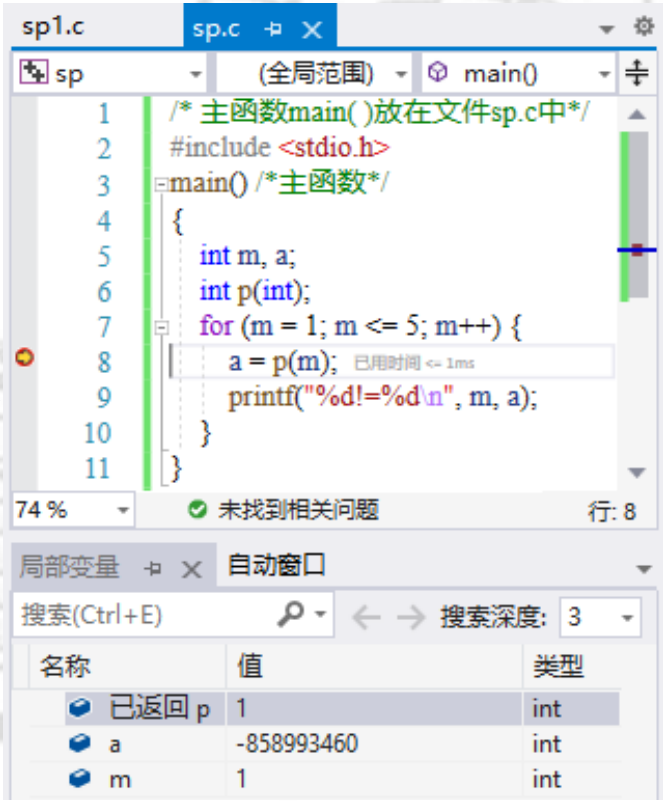


Debug tips：断点

- 设置断点后进行调试
  - “跳出”：进入到函数内，跳出在执行函数内剩余的语句时，直接返回到该函数被调用时的后面的语句处



跳出后



此时p(m)已经执行完毕，值已经被返回，再点击逐语句会进行a=p(m)的赋值过程

Debug tips：使用输出语句

- 通过输出语句来查看程序的运行是否出现问题
- 例7-4：使用输出语句来查看例7-3运行过程

```c
/* 主函数main( )放在文件sp.c中*/
#include <stdio.h>
void main() {  /*主函数*/
   int m, a;
   int p(int);
   for (m = 1; m <= 5; m++) {
      a=p(m);
      printf("%d!=%d\n", m, a);
   }
}
```

```c
/* 函数p( )放在文件sp1.c中*/
/*计算阶乘的函数,返回值为整型*/
int p(int n) {
   int k, s;
   s = 1;
   for (k = 1; k <= n; k++) {
      s = s * k;
      printf("%d ", s);
   }
   printf("\n");
   return s;
}
```

```
1
1!=1
1 2
2!=2
1 2 6
3!=6
1 2 6 24
4!=24
1 2 6 24 120
5!=120
请按任意键继续. . .
```

函数的调用

- 调用形式

  函数名(形参名称与类型列表)

  - 上例 printf(“%d!=%d\n”, m, p(m)); 调用函数p( )

- 注意:

  - 函数调用可以出现在**表达式**中(**有函数值返回**)，也可以单独作为**一个语句**(**无函数值返回**)
  - 在调用函数中，通常要对**被调用函数进行说明**(一般在调用函数的函数体中的说明部分)，包括函数值的**返回类型**、**函数名**以及**形参类型**

函数的调用：向前引用说明

- 以前的C版本中函数向前引用说明
  - 说明形式：**函数类型 函数名( );**
- 例7-5：阅读下面代码

```
1  #include <stdio.h>
2  void main() {
3    float x;
4    double y, f(); /*未采用函数原型说明*/
5    x = 1.0;
6    y = f(x);
7    printf("y=%f\n", y);
8  }
9  double f(float x) {
10   double y;  y = 2 * x + 1.0;
11   return(y);
12 }
```

在这种说明方式下，编译系统也就不检查参数的个数和类型。C语言的国际标准也兼容这种用法，但**不提倡这种用法**，因为这种用法很容易出错

```
y=1.000000
请按任意键继续. . .
```

函数的调用：向前引用说明

- 错误原因
  - 在C语言中，实型运算统一默认为**双精度运算**
  - 函数中的实型形参类型缺省是double型，而不是float型
  - main中的x虽然被定义为float型，但执行y=f(x);时，首先取实参x的值转换成double。最后传送给f( )的值是double型的，而函数f( )中的形参是float型的
  - 实参与形参类型不一致，导致形参接收到的数据错误，从而计算得到的**返回值错误**

主函数中没有说明f( )形参的类型，因此编译系统未发现实参类型与形参类型的不一致

函数的调用：向前引用说明

- 将函数f( )中的形参x定义成double型
  - 由于实参与形参的类型一致，运行结果就正确了
- 例7-6：将例7-5中实型定义为double型

```
1  #include <stdio.h>
2  void main() {
3    float x;
4    double y, f(); /*未采用函数原型说明*/
5    x = 1.0f;
6    y = f(x);
7    printf("y=%f\n", y);
8  }
9  double f(double x) {
10   double y;  y = 2 * x + 1.0;
11   return(y);
12 }
```

进行数值运算时，建议将所有的实型变量定义成double型，因为在C语言中所有的实型运算都是采用双精度运算的

```
y=3.000000
请按任意键继续. . .
```

函数的调用：向前引用说明

- 采用**函数原型**进行函数向前引用说明
- 例7-7：将例7-5中采用函数原型进行向前引用说明

```
1  #include <stdio.h>
2  void main() {
3    float x;
4    double y, f(float x);
5    x = 1.0f;
6    y = f(x);
7    printf("y=%f\n", y);
8  }
9  double f(float x) {
10   double y;
11   y = 2 * x + 1.0;
12   return(y);
13 }
```

注意：在调用函数中应采用函数原型对被调用函数进行说明

```
y=3.000000
请按任意键继续. . .
```

函数的调用：向前引用说明

- 两种情况下可以**不在**调用函数中对被调用函数作说明
  - 被调用函数的定义**出现在调用函数之前**
  - 在**调用函数之前的外部说明**中进行了被调用的函数原型说明

- 例7-8：阅读以下代码

```
1  #include <stdio.h>
2  double q(double r) {
3    return 3.1415926 * r * r;
4  }
5  void main() {
6    double r1, r2;
7    printf("input r1,r2: ");
8    scanf("%lf%lf", &r1, &r2);
9    printf("s=%f\n",q(r1)+q(r2));
10 }
```

- 例7-9：阅读以下代码

```
1  #include <stdio.h>
2  double q(double);
3  void main(){
4    double r1, r2;
5    printf("input r1,r2: ");
6    scanf("%lf%lf", &r1, &r2);
7    printf("s=%f\n", q(r1)+q(r2));
8  }
9  double q(double r) {
10   return 3.1415926 * r * r;
11 }
```

函数的调用

●强调

- 实参表中的各实参可以是表达式，但它们的类型和个数应与函数中的形参一一对应。各实参之间也要用“,”分隔
- C语言虽不允许嵌套定义函数，但可以嵌套调用函数
  - 在例7-1中，可以写表达式p(p(3))，这实际上是计算6!
  - p(p(3))=p(3!)=p(6)=6!=720

函数的调用

●例7-10：判断素数

□编写一个函数，其功能是判断给定的正整数是否是素数，若是素数则函数返回值1，否则函数返回值0

```
1  #include <math.h>
2  int sushu(int n) {
3    int k, i, flag;
4    k = (int)sqrt((double)n);
5    i = 2;
6    flag = 1;
7    while ((i <= k) && (flag == 1)) {
8      if (n % i == 0) flag = 0;
9      i = i + 1;
10   }
11   return flag;
12 }
```

要调用求平方根的函数sqrt( )，因此要包含C库**函数头文件**<math.h>

函数的调用

●例7-10：判断素数

□调用上一页的函数输出3到100之间的所有素数

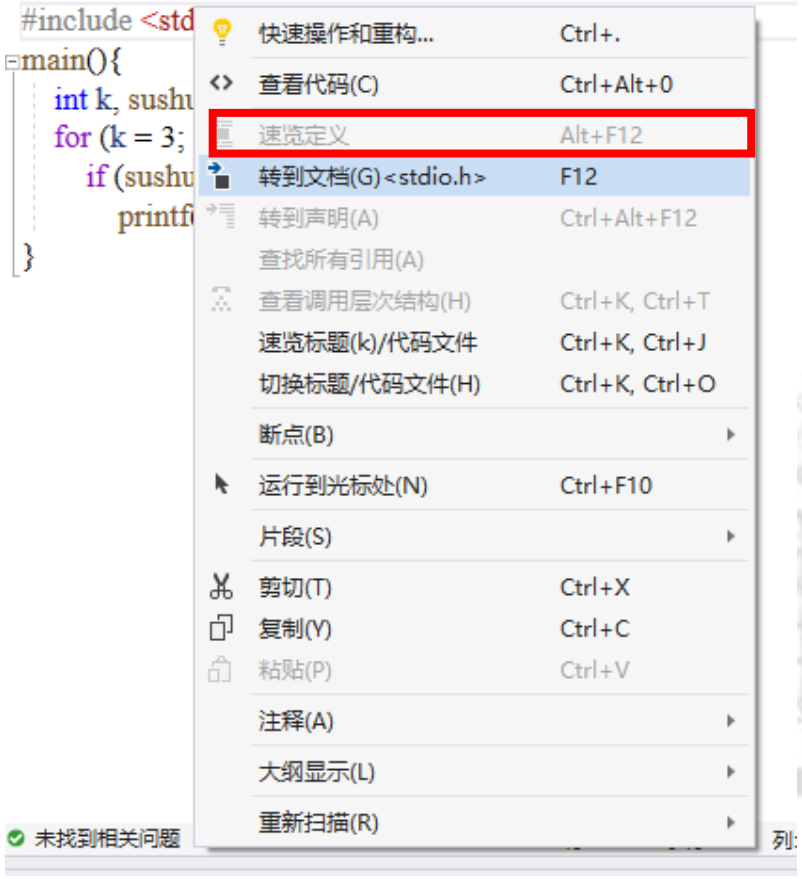
```c
#include <math.h>
#include <stdio.h>
int sushu(int n) {
  int k, i, flag;
  k = (int)sqrt((double)n);
  i = 2;
  flag = 1;
  while ((i <= k) && (flag == 1)){
    if (n % i == 0) flag = 0;
    i = i + 1;
  }
  return flag;
}
```

```c
void main() {
  int k, sushu(int);
  for (k = 3; k < 100; k = k + 2)
    if (sushu(k)) // 或if (sushu(k)==1)
      printf("%d\n", k);
  printf("\n");
}
```

```
3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
请按任意键继续. . .
```

函数的调用

- 查看**编译系统**提供的.h文件的内容
  - 在相应的.h文件名上按鼠标右键，会弹出下拉菜单：




```
//
// stdio.h
//
//      Copyright (c) Microsoft Corporation. All rights reserved.
//
// The C Standard Library <stdio.h> header.
//
#pragma once
#ifndef _INC_STDIO // include guard for 3rd party interop
#define _INC_STDIO

#include <corecrt.h>
#include <corecrt_wstdio.h>

_CRT_BEGIN_C_HEADER

/* Buffered I/O macros */

#define BUFSIZ  512
```

本节目录

7.2 模块间的参数传递

- 形参与实参的结合方式
- 局部变量与全局变量
- 动态存储变量与静态存储变量
- 内部函数与外部函数

模块间参数传递：形参与实参的结合

- 地址结合
  - 在一个模块调用另一个模块时，并不是将调用模块中的**实参**值直接传送给被调用模块中的**形参**，而只是将存放实参的**地址**传送给形参
  - 被调用程序中**对形参的操作实际上就是对实参的操作**，实现了数据的**双向传递**
  - 在这种方式中，被调用函数中改变了形参值，同时也就改变了调用函数中的实参值
  - 因此，在这种结合方式中的**实参只能为变量(左值)**

模块间参数传递：形参与实参的结合

●数值结合

- 调用模块中的实参地址与被调用模块中的形参地址是**互相独立**的
- 在一个模块调用另一个模块时，直接将**实参值**传送给形参并被存放在**形参地址**中
- 被调用程序中**对形参的操作不影响调用程序中的实参值**，因此只能实现数据的**单向传递**，即在调用时将实参值传送给形参
- 由于被调用函数中改变了形参值但不会改变调用函数中的实参值。在这种结合方式中的实参可以是**变量**、**表达式**或**常量**

模块间参数传递：形参与实参的结合

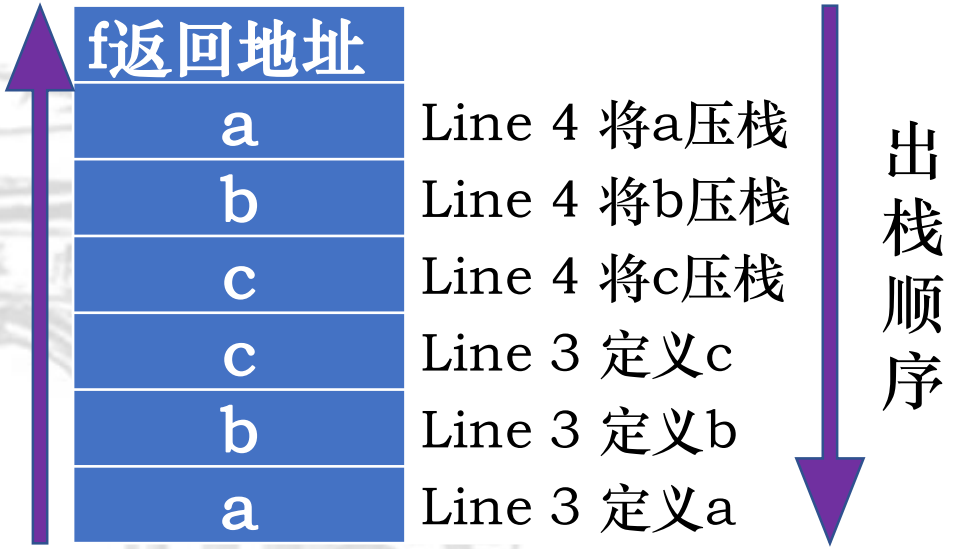
- C语言函数之间的参数传递是**传值**(**数值结合**)
  - 通过**栈**来传递，是单向传递
  - 栈：先入后出
  - 压栈顺序：**函数参数从右向左传递**
  - 一个函数**可以通过**参数把变量值传递给被调用函数，但被调用函数**不能通过**参数把变量值传回调用它的函数

模块间参数传递：形参与实参的结合

- C语言函数之间的参数传递是**传值**
- C语言函数返回(return)值是**传值**
- 例7-11：压栈顺序

```
1  #include <stdio.h>
2  void main(){
3    int a = 3, b = 2, c = 1, f(int, int, int);
4    f(a, b, c);
5    printf("c=%d\n", c);
6  }
7  int f(int a, int b, int c) {
8    c = a + b;
9    printf("c=%d\n", c);
10   return c;
11 }
```

```
c=5
c=1
请按任意键继续. . .
```



后续数字逻辑与处理器基础课程会详细介绍

模块间参数传递：形参与实参的结合

- C语言函数之间的参数传递是**传值**
  - 当形参为简单变量时，均采用数值结合
  - 在这种情况下，一个函数只能通过函数名返回一个值，而无法同时返回多个值
- 例7-12：用迭代法求下列方程的一个实根，其精度要求为$\varepsilon = 0.000001$

$$x - 1 - \arctan(x) = 0$$

迭代公式为

$$x_{n+1} = 1 + \arctan(x_n)$$

不动点法解方程（微积分知识）

模块间参数传递：形参与实参的结合

●例7-12：迭代法求根

□首先编写一个用迭代法求实根的函数如下：

```
#include <stdio.h>
#include <math.h>
int subroot(double x, double eps){
  int m;
  double x0, f(double);
  m = 0;
  do {
    m = m + 1;                          /*迭代次数加1*/
    x0 = x;                             /*保存上次迭代值*/
    x = f(x0);                          /*计算新的迭代值*/
  } while ((m <= 100) && (fabs(x - x0) >= eps));
  /*迭代次数没有超过100次且不满足精度要求则继续迭代*/
  if (m > 100) printf("FAIL!\n"); /*迭代次数超过100次,显示错误信息*/
  printf("x=%f\n", x);                  /*输出最后迭代值*/
  return(m);                            /*返回迭代次数*/
}
```

此函数适用于任意 $x_{n+1} = f(x_n)$ 形式的方程求根

模块间参数传递：形参与实参的结合

●例7-12：迭代法求根

□主函数以及计算迭代值的函数f(x)如下：

```
1  #include <stdio.h>
2  #include <math.h>
3  void main() {
4    int m, subroot(double, double);
5    double x, eps;
6    x = 1.0;
7    eps = 0.000001;
8    m = subroot(x, eps);
9    printf("m=%d\n", m); /*输出迭代次数*/
10   printf("x=%f\n", x); /*输出方程根*/
11 }
12 double f(double x) {   /*计算迭代值的函数*/
13   return 1.0 + atan(x);
14 }
```

```
x=2.132268
m=10
x=1.000000
请按任意键继续. . .
```

实参x的**值**被传递到函数subroot中，但是并没有返回subroot中的x，因此最终输出结果不正确

函数如何返回多个值?

- 利用**指针**，后续课程会讲
- 利用**全局变量**

局部变量与全局变量

- 按照**作用域**对变量进行分类
- 局部变量
  - 在**函数内部**定义的变量，只在该**函数范围内**有效
  - 不同函数中的局部变量**可以重名**，**互不混淆**
  - 函数中的形参也是**局部变量**
- 全局变量
  - 在**函数外**定义的变量，有效范围是从定义变量的位置开始到本源文件结束
  - 同一个程序中的**所有函数**都可以访问

局部变量与全局变量

●例7-13：迭代法求根

□在例7-12中采用全局变量

```
1  #include <stdio.h>
2  double x;  //定义全局变量
3  int subroot(double eps){
4    int m;
5    double x0,f(double);
6    m = 0;
7    do{
8      m = m + 1;
9      x0 = x;
10     x = f(x0);
11   } while ((m <= 100)&&(fabs(x - x0)>=eps));
12   if (m > 100) printf("FAIL!\n");
13   printf("x=%f\n", x);
14   return(m);
15 }
```

```
16 void main(){
17   int m, subroot(double);
18   double eps;
19   x = 1.0;
20   eps = 0.000001;
21   m = subroot(eps);
22   printf("m=%d\n", m);
23   printf("x=%f\n", x);
24 }
25 double f(double x) {
26   return 1.0 + atan(x);
27 }
```

```
x=2.132268
m=10
x=2.132268
请按任意键继续...
```

局部变量与全局变量：注意事项


- 全局变量的引用说明
  - 若将全局变量x的定义放在subroot( ){ }与main( ){ }之间，则应在程序开头处加上全局变量的**引用说明**：

    extern double x;

  - 先说明x是double型，其目的是让subroot( )函数在使用x时知道其类型是double，否则编译时会出现x**未定义的错误信息**

```
1  #include <stdio.h>
2  #include <math.h>
3  extern double x; //引用全局变量
4  int subroot(double eps) {
5  }
6  double x;          //定义全局变量
7  main() {
8  }
9  double f(double x) {
10 }
```

此处省略函数具体内容

局部变量与全局变量：注意事项

- 如果局部变量与全局变量**同名**，则在该局部变量的作用范围内，**全局变量不起作用**。这称为全局变量被**掩蔽**（masked）
- 除非十分必要，一般不提倡使用全局变量，原因如下：
  - 全局变量属于程序中的**所有函数**，因此，在程序的执行过程中，全局变量一直需要**占用存储空间**
  - 在函数中使用全局变量后，要求在所有调用该函数的调用程序中都要使用这些全局变量，从而会**降低函数的通用性**
  - 使用全局变量后，使各函数模块之间的**互相影响比较大**，从而使函数模块的"内聚性"差，而与其他模块的"耦合性"强
  - 在函数中使用全局变量后，会**降低程序的清晰性**，可读性差

动态存储变量与静态存储变量

- 变量的存储方式
  - 依据变量的**存在时间（生存期）**对变量进行分类
  - **静态存储**方式：程序运行期间由系统分配固定的存储空间
    - 整个程序运行期间都不释放
  - **动态存储**方式：程序运行期间根据需要动态分配存储空间
    - 函数调用开始时分配存储空间，调用结束后释放空间

动态存储变量与静态存储变量

- 数据类型
  - int, long, short, float, double …
- 数据的存储类别
  - 自动类型(auto)
  - 静态类型(static)
  - 寄存器类型(register)
  - 外部类型(extern)
  - 数据的存储类别决定了该数据的**存储区域**、**作用域**、**生存期**

```
1  auto unsigned char i;
2  register int a;
3  static double b;
```

动态存储变量与静态存储变量

- 自动类型(auto)
  - 函数中的局部变量，如不声明为static存储类别，都是采用**动态存储方式**，函数调用结束后即释放
  - 本函数内有效
  - 函数中的**形参**、函数中定义的**局部变量**
  - 定义变量时不写auto则**默认**为自动类型

动态存储变量与静态存储变量

●静态变量

- □用static说明的局部变量或外部变量，采用**静态存储方式**
- □在函数调用结束后其内存不会消失而**保留原值**，即其占用的存储单元不释放，在下一次调用时仍为**上次调用结束时的值**
- □形参**不能**定义成静态存储类型
- □对局部静态变量赋初值是在编译时进行的，在调用时**不再赋初值**
- □定义局部静态变量时若不赋初值，则在编译时将**自动赋初值0**

动态存储变量与静态存储变量

- 静态变量说明
  - 用static说明的局部变量或外部变量
  - 局部静态变量会造成多次运行函数的结果之间有关联效应，**尽量不用局部静态变量**
  - 外部静态变量的作用范围仅限**本文件**，一个函数**不能访问**本文件外的外部静态变量

动态存储变量与静态存储变量

●例7-14：静态变量与自动变量区别

```
1  #include <stdio.h>
2  int ksum(int n){
3    static int x = 0;
4    x = x + n;
5    return(x);
6  }
7  void main(){
8    int k, ksum(int);
9    for (k = 1; k <= 5; k++)
10     printf("sum(%d)=%d\n", k, ksum(k));
11 }
```

```
sum(1)=1
sum(2)=3
sum(3)=6
sum(4)=10
sum(5)=15
请按任意键继续...
```

去掉static

```
1  #include <stdio.h>
2  int ksum(int n) {
3    int x = 0;
4    x = x + n;
5    return(x);
6  }
7  void main() {
8    int k, ksum(int);
9    for (k = 1; k <= 5; k++)
10     printf("sum(%d)=%d\n", k, ksum(k));
11 }
```

```
sum(1)=1
sum(2)=2
sum(3)=3
sum(4)=4
sum(5)=5
请按任意键继续...
```

课堂练习

练习7-2：阅读下面的代码，请写出运行结果。

```
1  #include <stdio.h>
2  void main(){
3    int i, a = 2;
4    int ff(int);
5    for (i = 0; i < 3; i++)
6      printf("%3d", ff(a));
7  }
8  int ff(int a){
9    int b = 0;
10   static int c = 5;
11   b++;
12   c++;
13   return (a + b + c);
14 }
```

```
9 10 11请按任意键继续. . .
```

- a=2;b=1;c=6
- a=2;b=1;c=7
- a=2;b=1;c=8

动态存储变量与静态存储变量

- 外部变量
  - 用extern说明的局部变量或外部变量
  - 全局变量如果在文件开头定义,则在整个文件范围内的所有函数都可以使用该变量
  - 如果不在文件开头定义全局变量，则只限于在定义点到文件结束范围内的函数使用该变量

动态存储变量与静态存储变量

- 外部变量用途
  - 在同一文件中，为了使全局变量定义点之前的函数中也能使用该全局变量，则应在函数中用**extern**加以说明
  - 使一个文件中的函数能用**另一个文件中的全局变量**
  - 利用**静态外部变量**，使全局变量只能被本文件中的函数引用，**控制其作用域**

动态存储变量与静态存储变量

●例7-15：用**extern**使**全局变量定义点之前的函数**中也能使用该全局变量，实现两个变量值交换

```
1  #include <stdio.h>
2  void main() {
3    extern int x, y; /*x与y定义为外部变量*/
4    void swap();
5    scanf("x=%d,y=%d", &x, &y);
6    swap();
7    printf("x=%d,y=%d\n", x, y);
8  }
9  int x, y;
10 void swap() {
11   int t;
12   t = x; x = y; y = t;
13   return;
14 }
```

```
x=1,y=2
x=2,y=1
请按任意键继续...
```

用extern 说明了变量x和y是外部变量

若在主函数中**没有用extern**说明变量x和y，则x和y变成主函数的局部变量，**掩蔽**全局变量x和y。swap不能将主函数中的x和y的值交换

```
x=1,y=2
x=1,y=2
请按任意键继续...
```

动态存储变量与静态存储变量


●例7-16：**extern**使一个文件中的**函数能用另一个文件中的全局变量**，实现两个变量值交换

```c
/* file1.c */
#include <stdio.h>
int x, y;
void main() {
  void swap();
  scanf("x=%d,y=%d", &x, &y);
  swap();
  printf("x=%d,y=%d\n", x, y);
}
```

```c
/* file2.c */
extern int x, y;
void swap(){
  int t;
  t = x; x = y; y = t;
  return;
}
```

extern使file2中的函数能使用file1中的全局变量

```
x=1,y=2
x=2,y=1
请按任意键继续. . .
```

- 同一程序的两个函数文件中**不能同时定义同名的全局变量**
- 在VS2008以上的编译系统上，同一程序的两个文件中可以同时定义相同的全局变量，系统会自动认为是同一个全局变量

动态存储变量与静态存储变量

- 利用**静态外部变量**，使全局变量只能被本文件中函数引用，控制其作用域
- 例7-17：实现两个变量值交换

静态外部变量，只适用于本文件，使得file2中extern无效

- 下列程序错误

```c
/* file1.c */
#include <stdio.h>
static int x, y;
void main(){
    void swap();
    scanf("x=%d,y=%d", &x, &y);
    swap();
    printf("x=%d,y=%d\n", x, y);
}
```

```c
/* file2.c */
extern int x, y;
void swap(){
    int t;
    t = x; x = y; y = t;
    return;
}
```

动态存储变量与静态存储变量

●寄存器变量

- □用register说明的变量，本函数内有效
- □采用**动态存储方式**
- □允许将局部变量的值放在运算器的寄存器中，需要时直接从寄存器取出参加运算，从而提高执行效率
- □如果有一些变量使用频繁，则存取变量的值要花不少时间，可以使用**寄存器变量**提高执行效率
  - 例如，在一个函数中执行10000次循环，每次循环中都要引用某局部变量

变量存储位置

●变量的存储位置

- □依据变量的**存储位置**对变量进行分类
- □**静态存储区**：静态变量、外部变量
- □**动态存储区**：自动变量、函数形参
- □**CPU中的寄存器**：寄存器变量

程序区	存放程序
静态存储区	在程序开始执行时就分配的**固定存储**单元
动态存储区	在函数调用过程中**动态分配**的存储单元

内部函数与外部函数

- 内部函数
  - 只能被**本文件中其他函数**调用的函数
  - 定义内部函数的形式：

    static 类型标识符 函数名(形参表)
- 外部函数
  - 能被**其他文件中函数**调用的函数
  - 定义外部函数的形式：

    [extern] 类型标识符 函数名(形参表)

    - 如果省略extern说明符，则默认为是外部函数

内部函数与外部函数

●例7-18：外部函数举例

```c
/* file.c */
#include <stdio.h>
void main() {
   int x = 10, z, y, f(int), g(int);
   y = f(x); z = g(x);
   printf("x=%d\ny=%d\n", x, y);
   printf("z=%d\n",z);
}
```

```c
/* file1.c */
static int f(int x) { /*f是内部函数*/
   return x * x;
}
/* g调用本文件中的内部函数f */
int g(int x){
   return f(x);
}
```

同一个项目中虽然有同名函数f，但因为其中一个是static类型的，是**内部函数**，因此编译时不会出现重名错误

```c
/* file2.c */
int f(int x) {
   return x * x * x;
}
```

```
x=10
y=1000
z=100
请按任意键继续...
```

算法举例

●例7-19：**梯形法求定积分**

$$S = \int_a^b f(x)dx$$

□用梯形近似，每一个梯形S_i面积：

$$S_i = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

$$S = \int_a^b f(x)dx = \sum_{i=0}^{n-1} S_i = \frac{h}{2}\sum_{i=0}^{n-1}[f(x_i) + f(x_{i+1})]$$

$$= \frac{h}{2}[f(a) + f(b)] + h\sum_{i=1}^{n-1} f(x_i)$$

算法举例

```
1  #include <stdio.h>
2  #include <math.h>
3  double tab(double a, double b, int n){
4    int k;
5    double h, x, s, p = 0.0, f(double);
6    h = (b - a) / n; /* 步长*/
7    s = h * (f(a) + f(b)) / 2;
8    for (k = 1; k < n; k++) {
9      x = a + k * h; p += f(x);}
10     s += p * h;
11   return s;
12 }
13 double f(double x){
14   return exp(-x * x);}
15 void main(){
16   int n; double a = 0.0, b = 1.0;
17   printf("input n: ");
18   scanf("%d", &n);
19   printf("%16.12f\n", tab(a, b, n));
20 }
```

- 例7-19 梯形法求定积分

$$S = \int_0^1 e^{-x^2} dx$$

```
input n: 50
 0.746799607189
请按任意键继续. . .
```

```
input n: 500
 0.746823887559
请按任意键继续. . .
```

```
input n: 5000
 0.746824130360
请按任意键继续. . .
```

```
input n: 100000000
 0.746824132812
请按任意键继续. . .
```

模块化程序设计：协作性与团队精神


分工协作

编写一个计算器程序



字符串处理

运算逻辑

异常处理

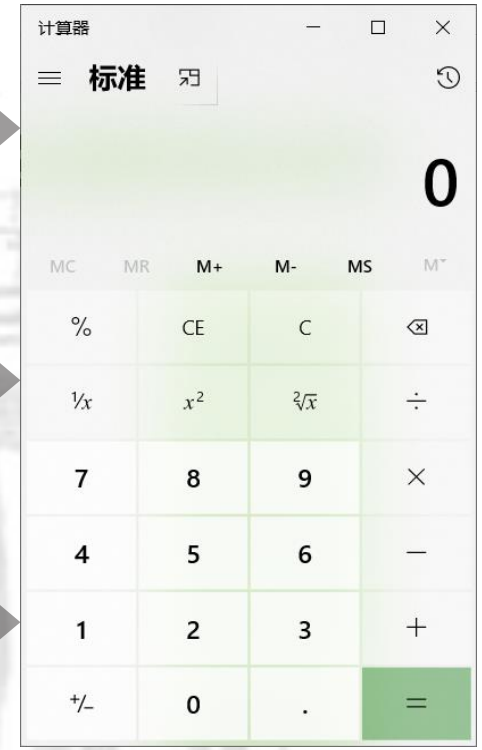
界面显示

读写存储器

输入模块

运算模块

输出模块



将复杂任务拆分成若干小任务，由整个团队共同完成，体现了大型程序编写中的**协作性**，也体现了**部分与整体**对立统一的辩证关系

本节总结

- **模块化**程序设计与函数
  - 模块化程序设计的基本概念
  - 函数的定义：　类型标识符 函数名（形参名称与类型列表）{}
  - 函数的调用：　函数向前引用说明
- 模块间的**参数传递**
  - 形参与实参的结合方式
  - 局部变量与全局变量
  - 动态存储变量与静态存储变量
  - 内部函数与外部函数：static, extern

本节作业

- 作业7
  - 课本第八章习题1,4,5
- 上机实验
  - 课本第八章习题12（要求写实验报告）

本节作业

●附加作业

- 1、编写函数求解鸡兔同笼问题，输入鸡和兔的总数x，脚的数量y，返回鸡的数量z。问题无合理解的情况下返回-1并打印“Error!”。
- 2、编写函数，输入浮点数a,b和字符c，c可能为“+”“-”“*”“/”之一，输出a和b执行c所规定的运算后的结果，如输入(1,2,“/”)，输出0.5。注意处理分母为0的情况。
- 3、利用例7-12类似的迭代法，求解下列方程的根

$$e^x - 3x = 0$$

得到其中一个根即可,精度要求自行设定。


THANKS