

计算机程序设计基础(1)

--- C语言程序设计(8)

孙甲松

sunjiasong@tsinghua.edu.cn

电子工程系 信息认知与智能系统研究所
罗姆楼6-104

电话: 13901216180/62796193

2022.10.

第8章 模块设计

8.1 模块化程序设计与C函数

8.1.1 模块化程序设计的基本概念

8.1.2 函数的定义

8.1.3 函数的调用

8.2 模块间的参数传递

8.2.1 形参与实参的结合方式

8.2.2 局部变量与全局变量

8.2.3 动态存储变量与静态存储变量

8.2.4 内部函数与外部函数

8.3 函数的递归调用

8.4 程序举例

1. 梯形法求定积分

2. Hanoi塔问题

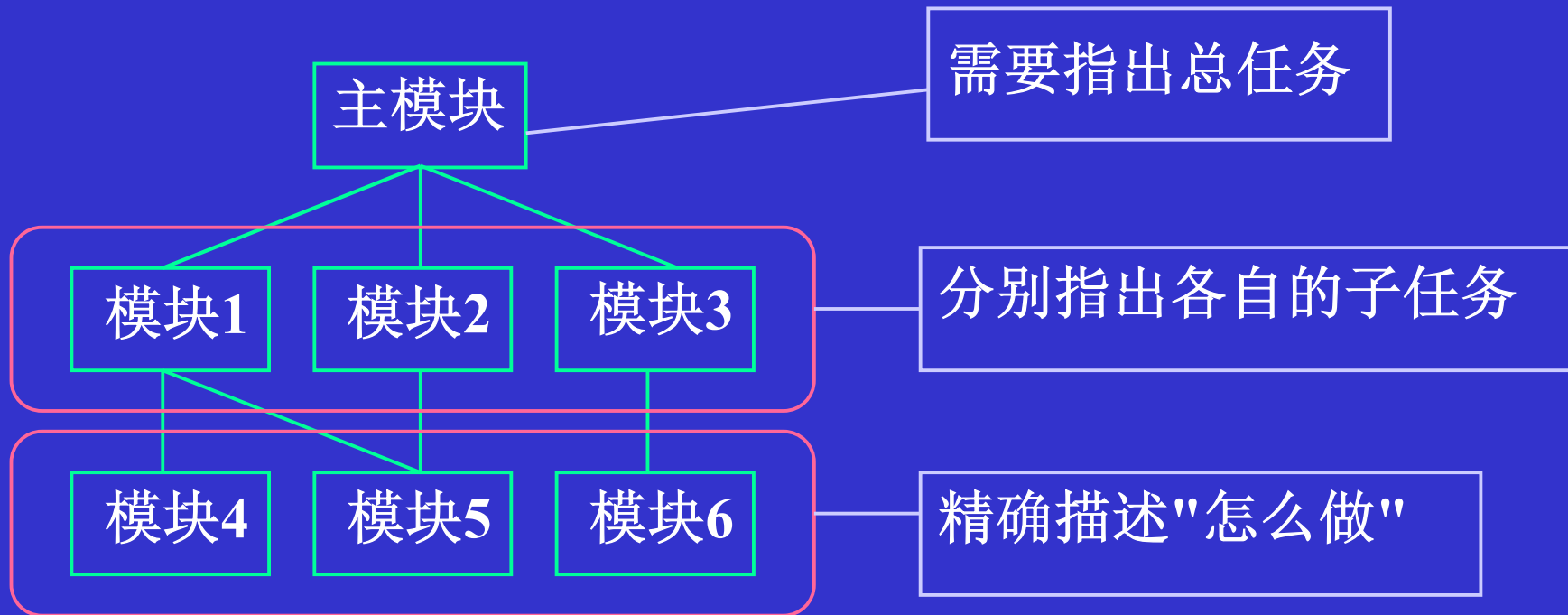
8.1 模块化程序设计与C函数

8.1.1 模块化程序设计的基本概念

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。



按层次组织模块



● 根据模块化设计的原则，一个较大的程序一般应分为若干个程序模块，每一个模块用于实现一个特定的功能。在不同的程序设计语言中，模块实现的方式有所不同。

例如，在FORTRAN语言中，模块用子程序(subroutine)和函数(function)来实现；在PASCAL语言中，模块用过程(procedure)和函数(function)来实现；在C语言中，模块用函数(function)来实现（**强调**：C语言没有子程序和过程）。

在C语言中，函数分为以下两种：

● 标准库函数

例如：scanf()、printf()、fabs()、
log()、sqrt()、exp()、sin()、cos()

● 用户自定义的函数

8.1.2 函数的定义

【例8-1】 计算1到5之间各自然数的阶乘值。

```
#include <stdio.h>
main() /*主函数*/
{ int m;
  int p(int);
  /* 说明要调用的函数p()返回值是int型,有一个int型形参 */
  for (m=1; m<=5; m++)
    printf("%d!=%d\n", m, p(m));
}
int p(int n) /*计算阶乘值的函数, 返回值为int型*/
{ int k, s;
  s=1;
  for (k=1; k<=n; k++)
    s=s*k;
  return(s);
}
```

在这个程序中，共有两个函数：

- 一个是主函数`main()`，它的功能是通过循环调用函数`p()`计算并输出`p(m)`（即 $m!$ ）的值；
- 另一个是函数`p()`，它的功能是计算阶乘值，例如，`p(m)`是计算 $m!$
- 在主函数中用语句

```
int p(int);
```

声明了被调用函数`p`的返回值是`int`型，此函数有一个`int`型形参。这就是 函数向前引用说明。

如果把主函数中的函数向前引用说明 `int p(int);` 去掉:

```
#include <stdio.h>
main() /*主函数*/
{ int m;
  for (m=1; m<=5; m++)
    printf("%d!=%d\n", m, p(m));
}
int p(int n) /*计算阶乘值的函数，返回值为int型*/
{ int k, s;
  s=1;
  for (k=1; k<=n; k++)
    s=s*k;
  return(s);
}
```

编译结果为:

提示: C编译器假设任何未说明的函数返回值类型为int。

test.c(5) : warning C4013: “p”未定义; 假设外部返回 int

勉强通过编译, 编译器假设函数p返回 int类型, 与函数p实际定义的返回类型凑巧一致了。

【例8-2】 下列C程序的功能是计算并输出一个圆台两底面积之和。

```
#include <stdio.h>
main( )
{ double r1,r2;
  double q(double);
  /* 函数q()是双精度实型,有一个双精度实型形参 */
  printf("input r1,r2:"); /* 提示输入 */
  scanf("%lf%lf",&r1,&r2); /* 输入r1与r2 */
  printf("s=%f\n",q(r1)+q(r2));
}
double q(double r) /*计算圆面积的函数,为双精度实型*/
{ double s;
  s=3.1415926*r*r;
  return s;
}
```

q函数可以简化为:

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

在这个程序中，在主函数中输入圆台的两个底半径，然后两次调用计算圆面积的函数q()，并将它们加起来。在主函数中用语句

double q(double);

说明了被调用函数的类型是双精度实型，有一个双精度实型形参。

如果把主函数中的函数向前引用说明 `double q(double);` 去掉:

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

```
main( )
```

```
{ double r1,r2;
```

```
    printf("input r1,r2:"); /* 提示输入 */
```

```
    scanf("%lf%lf",&r1,&r2); /* 输入r1与r2 */
```

```
    printf("s=%f\n",q(r1)+q(r2));
```

```
}
```

```
double q(double r)
```

```
{ return 3.1415926*r*r;
```

```
}
```

编译结果为:

test.c(6) : warning C4013: “q”未定义; 假设外部返回 int

test.c(9) : **error** C2371: “q”: 重定义; 不同的基类型

未通过编译, 编译器假设函数q返回 int类型, 而函数q实际定义
的返回类型为double, 因此编译器认为发生了q重定义致命错误。

- 在C语言中，函数定义的一般形式为：

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
}

- 在定义C函数时要注意以下几点：

- C语言函数中说明部分与语句部分要严格分开。
- 函数类型标识符等同变量类型说明符，它表示返回的函数值类型。在C语言中还可以定义无类型（即void类型）的函数，这种函数不返回函数值，而只是完成某个指定任务。
- 如果省略函数的类型标识符，则默认为是int型，例如main()就是省略了函数类型。但现在的C语言标准不提倡这种省略方法，C++更是不允许。
- C语言允许定义空函数，如 void dummy(){}
- 函数中返回语句：return (表达式); 或 return 表达式;
其作用是：立即停止本函数的执行，将表达式的值作为函数值返回给调用函数，其中表达式的类型应与函数返回值类型一致。

- 如果“形参名称与类型列表”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。

例如，在例8.1中的函数p()中，也可以写成：

```
int p(n)
int n;
{   int k, s;
    s=1;
    for (k=1; k<=n; k=k+1) s=s*k;
    return(s);
}
```

```
int p(int n)
{   int k, s;
    s=1;
    for (k=1; k<=n; k=k+1)
        s=s*k;
    return(s);
}
```

但这是C语言旧的写法，现在已经不提倡了，大家只要能看懂就好。函数头部应写为：**int p(int n)**

- 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。

在例8-1中，程序是从主函数开始执行的，只有执行到计算表达式`p(m)`（`printf()`中的输出项目）值时，才调用`p()`函数以计算`p(m)`的值。

- 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。

例如，在例8-1中，C程序中的两个函数可以分别放在两个文件中(主函数的文件名为`sp.c`，函数`p()`的文件名为`sp1.c`)，见下页

- 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。但C函数不能嵌套定义(不能在一个函数中定义另一个函数)。

```
/* 主函数main()放在文件sp.c中 */  
#include <stdio.h>  
main() /*主函数*/  
{ int n, m;  
  int p(int);  
  /*说明本函数中要调用的函数p()返回值是整型，有一个整型形参*/  
  for (m=1;m<=5;m++)  
    printf("%d!=%d\n",m,p(m));  
}
```

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

如果一个C程序中的多个函数分别放在多个不同的文件中，在VS2008的编译环境下，可以有以下基本处理方式：

- 建立一个空的项目（Project），把多个文件插入到此项目中，用“重新生成”编译链接

例如，对于上述程序，先建立项目sp，再分别建立文件sp.c, sp1.c并插入到项目sp中，用“重新生成”编译链接，编译系统将分别编译这两个函数所在的文件sp.c, sp1.c，然后进行链接，生成一个可执行文件sp.exe。

以下是操作过程演示。

新建项目

项目类型(P):

Visual C++

ATL

CLR

常规

MFC

智能设备

测试

Win32

其他语言

其他项目类型

测试项目

模板(T):

.NET Framework 3.5

Visual Studio 已安装的模板

Win32 控制台应用程序

Win32 项目

我的模板

搜索联机模板...

用于创建 Win32 控制台应用程序的项目

名称(N):

sp

位置(L):

D:\Sun\2020C语言\学生问题

浏览(B)...

解决方案名称(M):

sp

☒ 创建解决方案的目录(D)

确定

取消

Win32 应用程序向导 - sp



欢迎使用 Win32 应用程序向导

概述

应用程序设置

这些是当前项目设置：

- 控制台应用程序

在任一窗口中单击“完成”以接受当前设置。

创建项目后，请参阅该项目的 `readme.txt` 文件，了解有关项目功能和所生成的文件的信息。

< 上一步

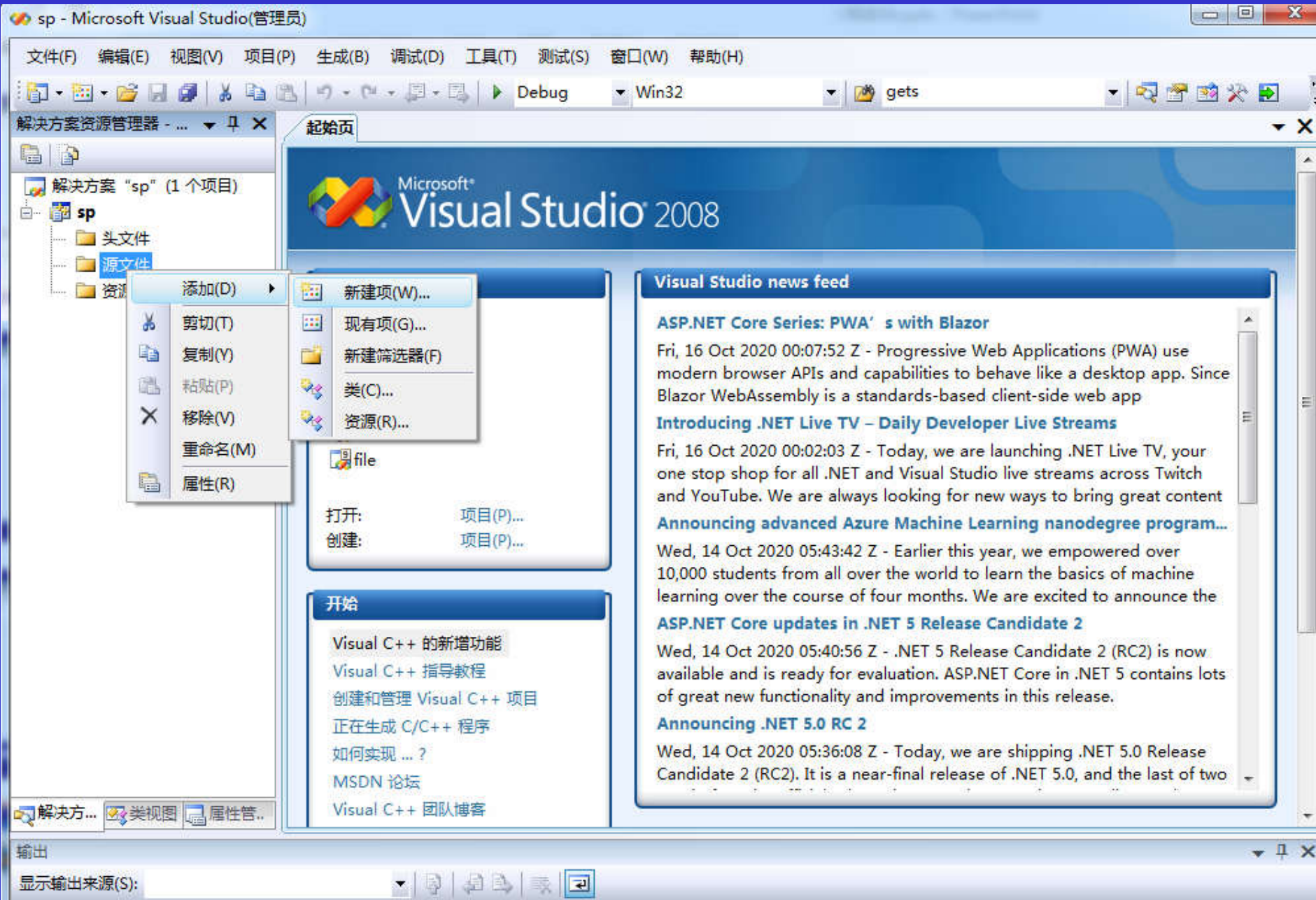
下一步 >

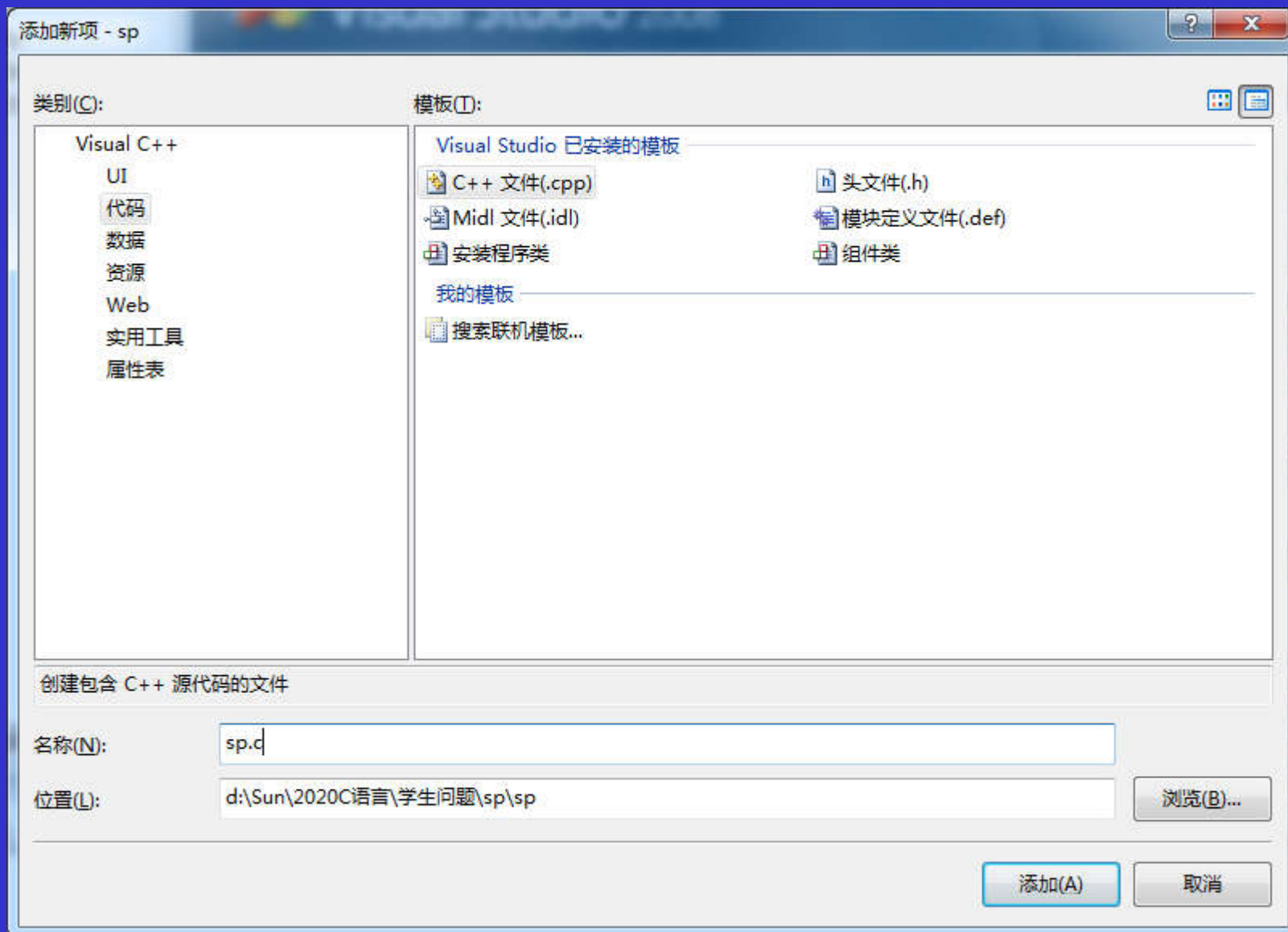
完成

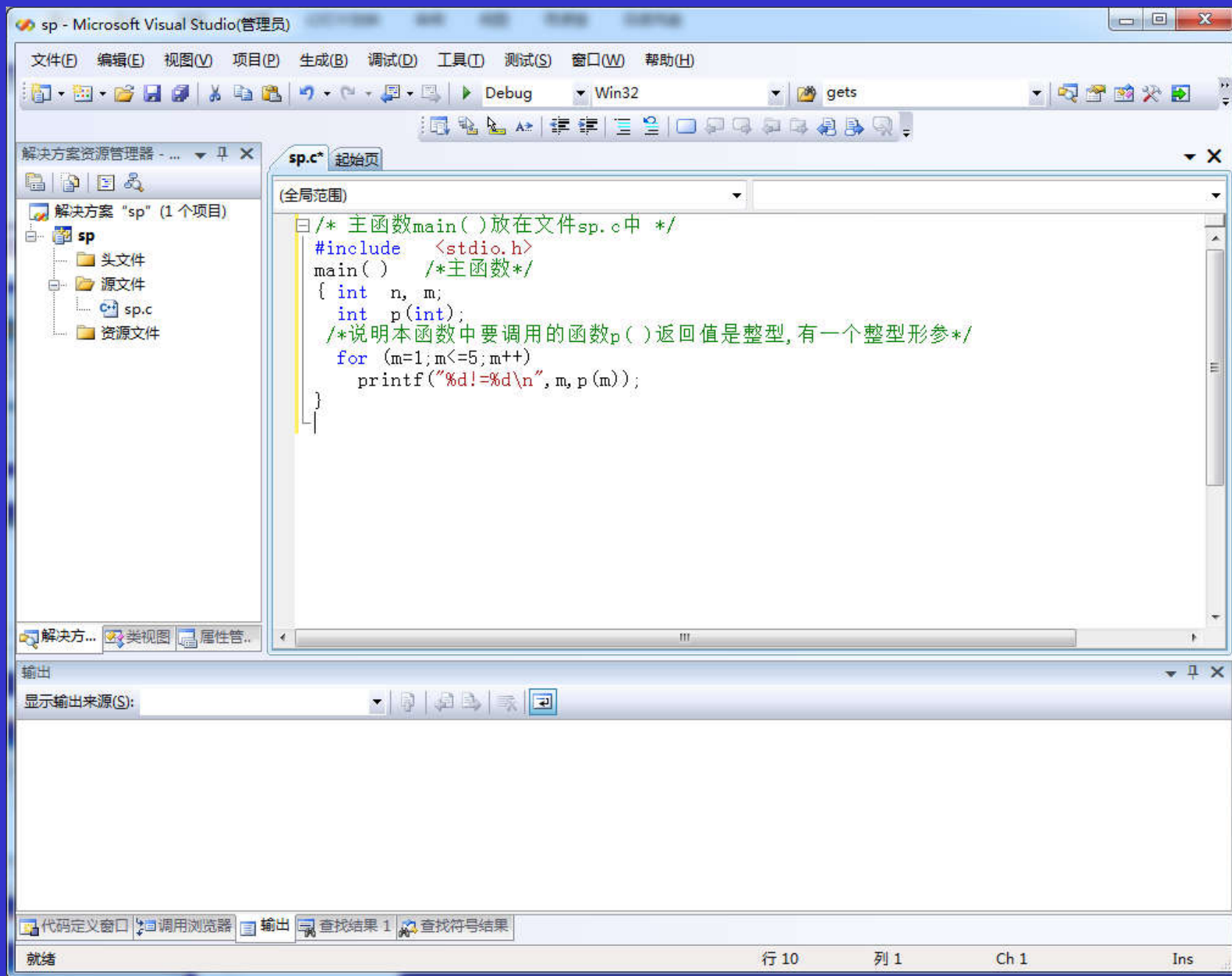
取消

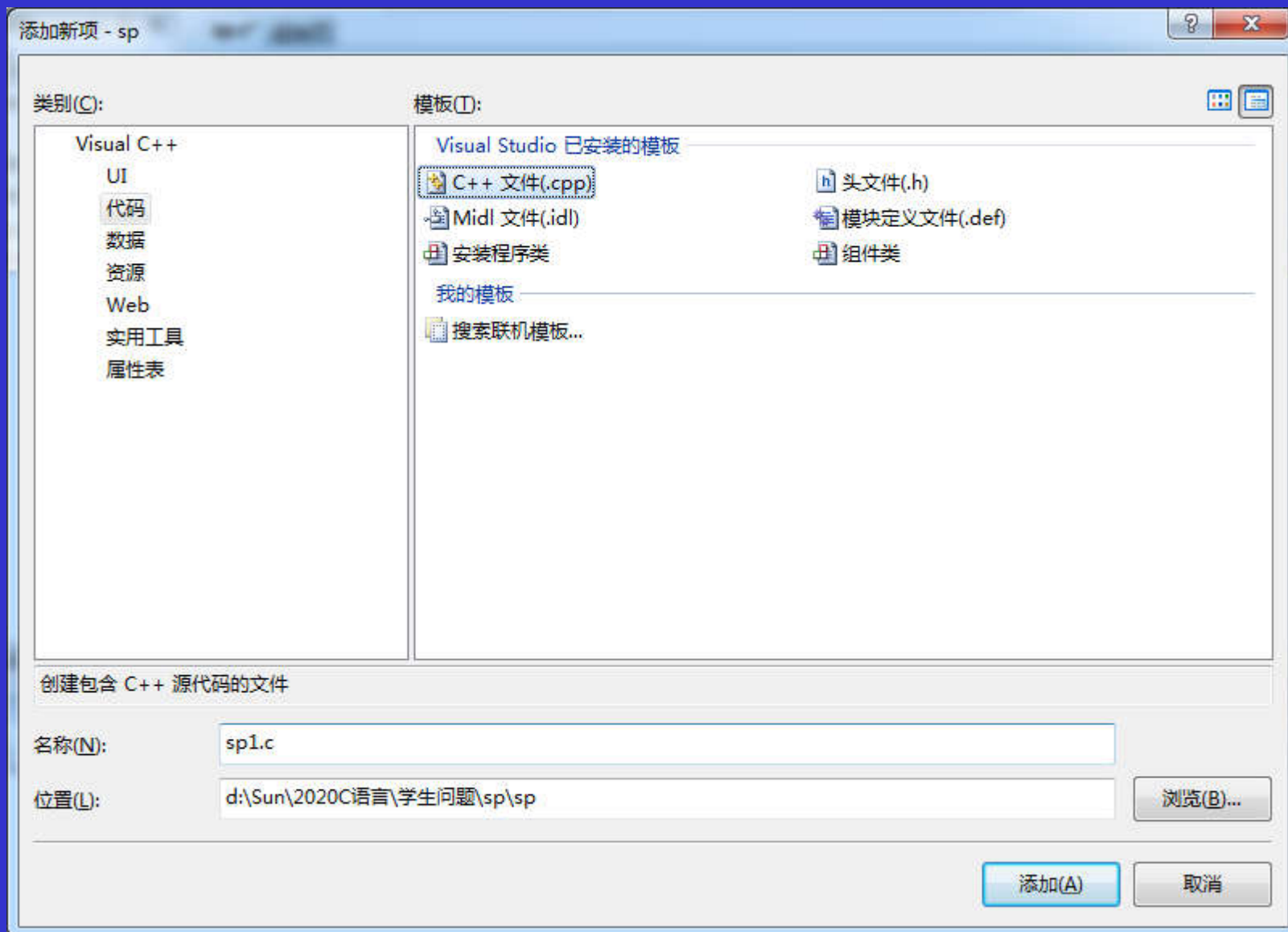


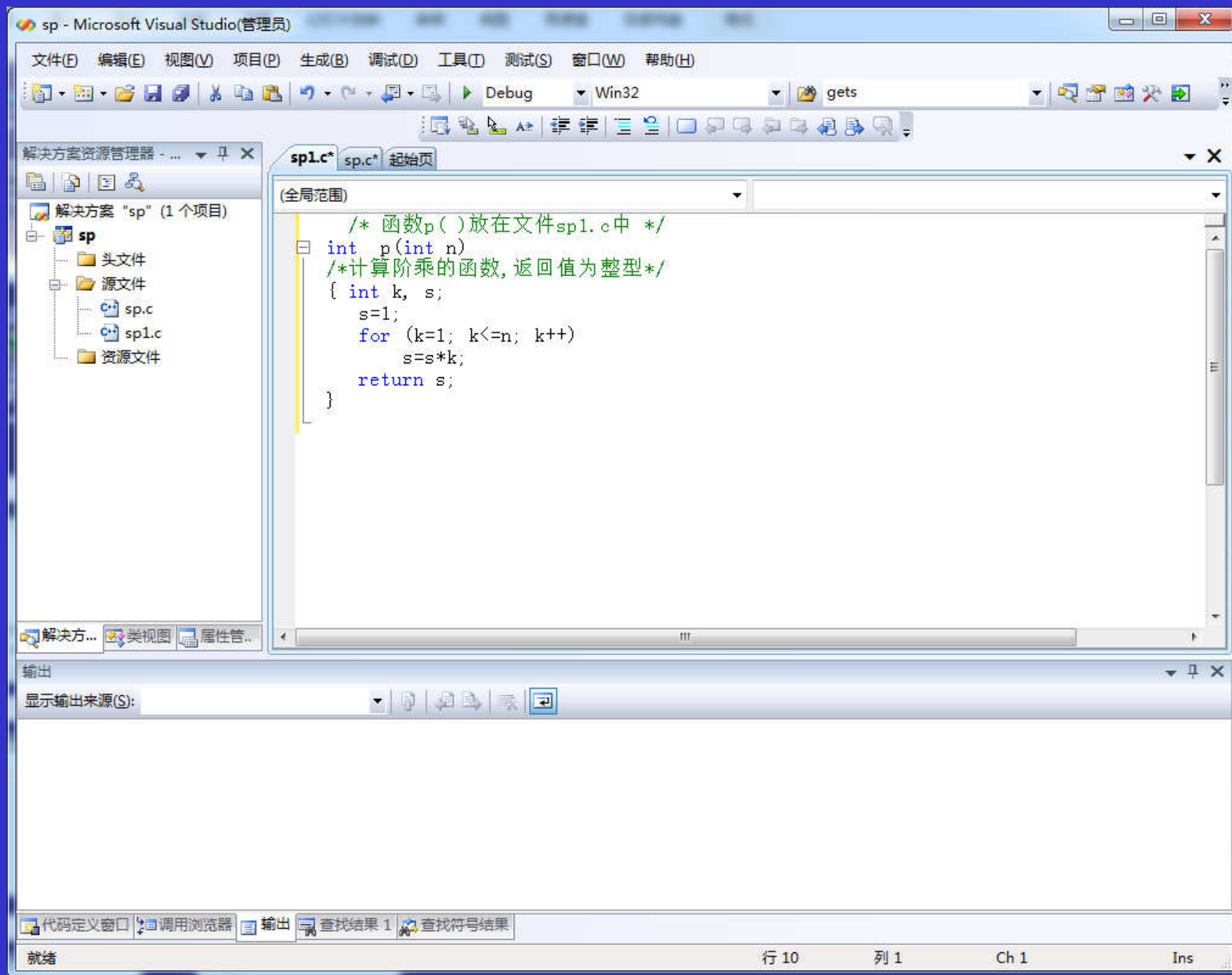
注意：附件选项选择“空项目”。



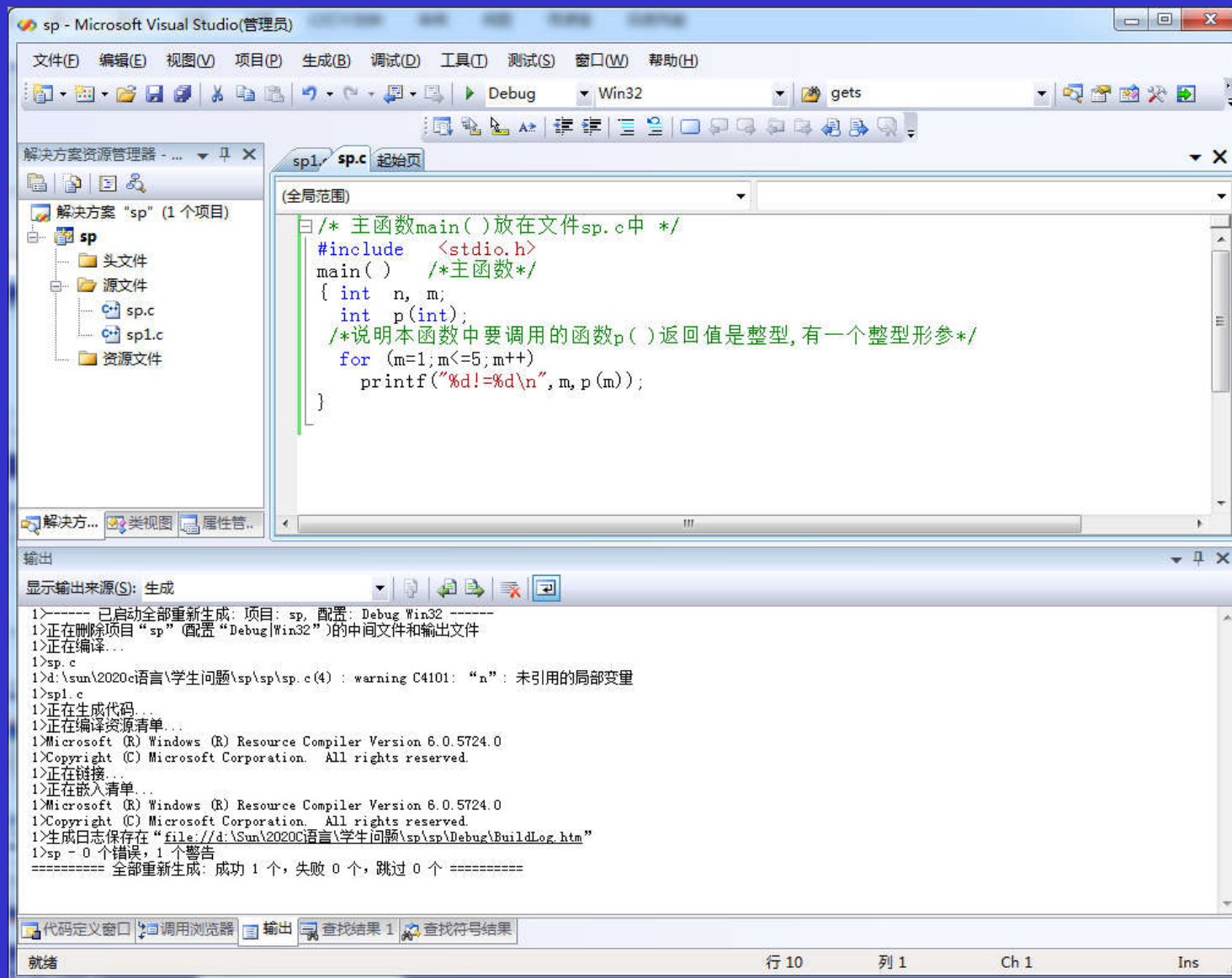




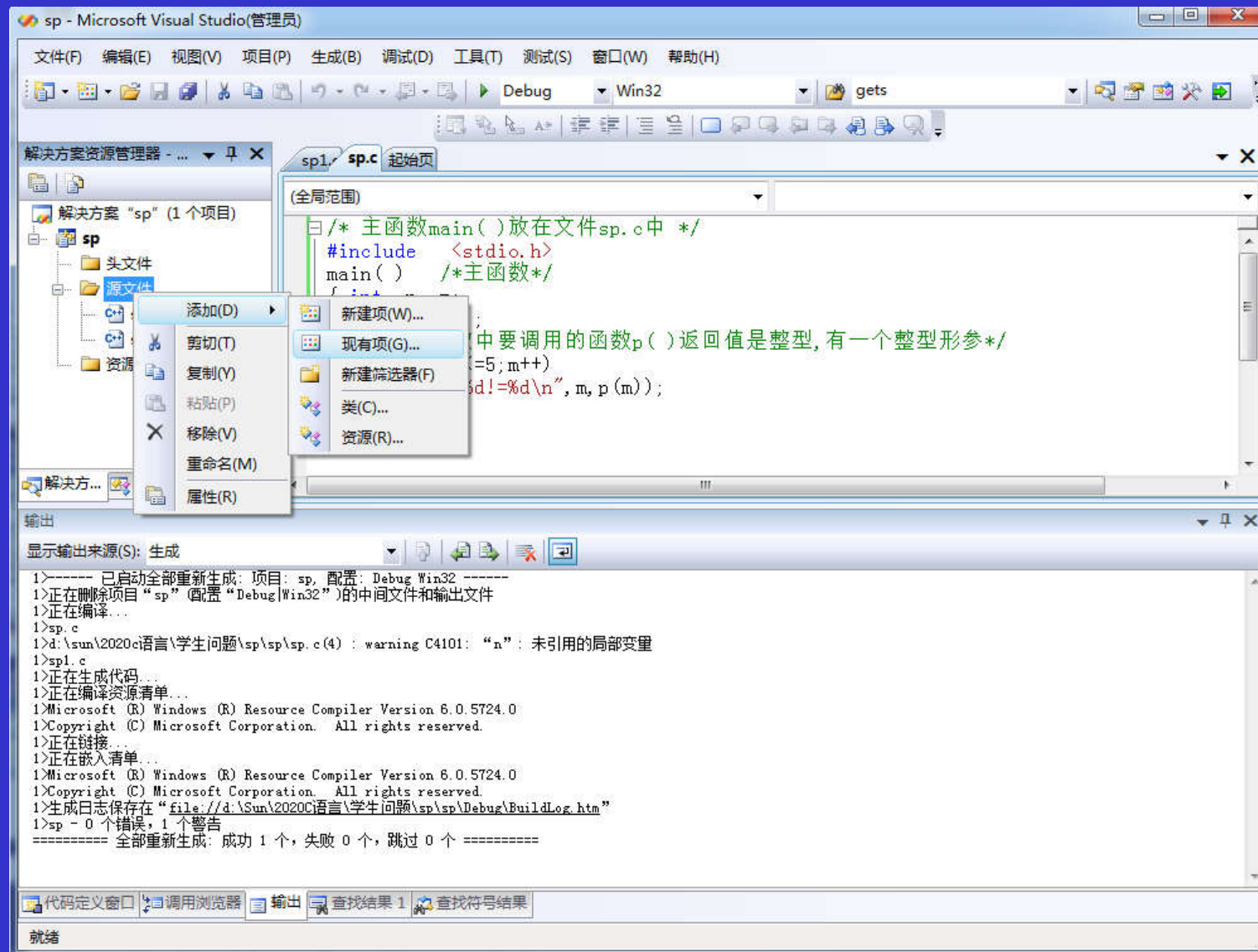




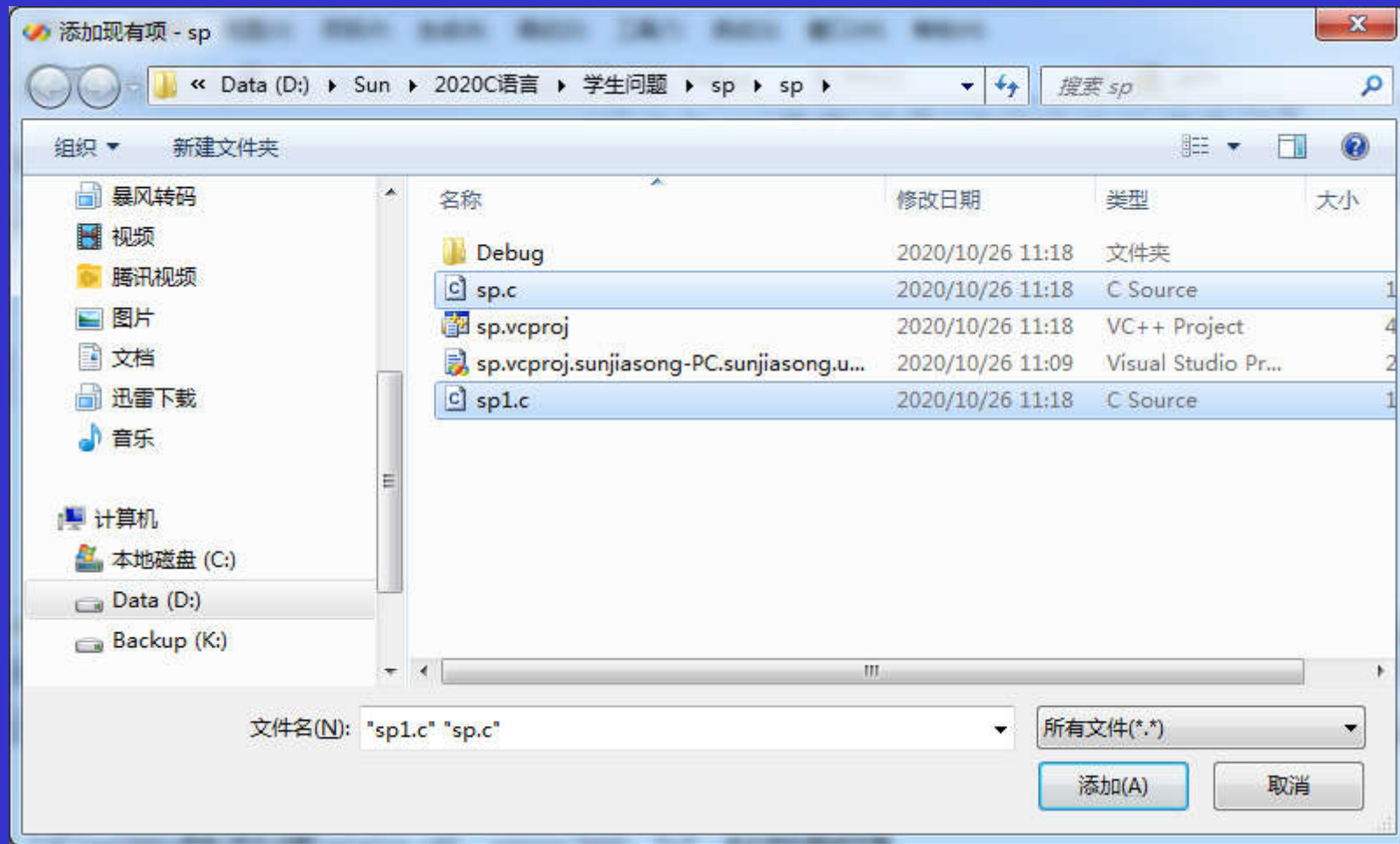
选择“生成”菜单中的“重新生成（Rebuild All）”，会逐个编译项目中的所有.c源程序并链接生成可执行文件。



- 若文件sp.c, sp1.c已经建好, 那么只需要在建立了项目sp后, 在“源文件”上按鼠标右键, 选择“添加”中的“现有项”向项目sp中插入文件sp.c, sp1.c



● 按住Ctrl键用鼠标逐个选中要插入的多个.c或.h文件，或按住Shift键一次选中多个文件，可以一次把多个文件插入到项目中。



如何设置VS2008编译系统的 Release（发布）方式

VS2008编译系统有两种编译方式:

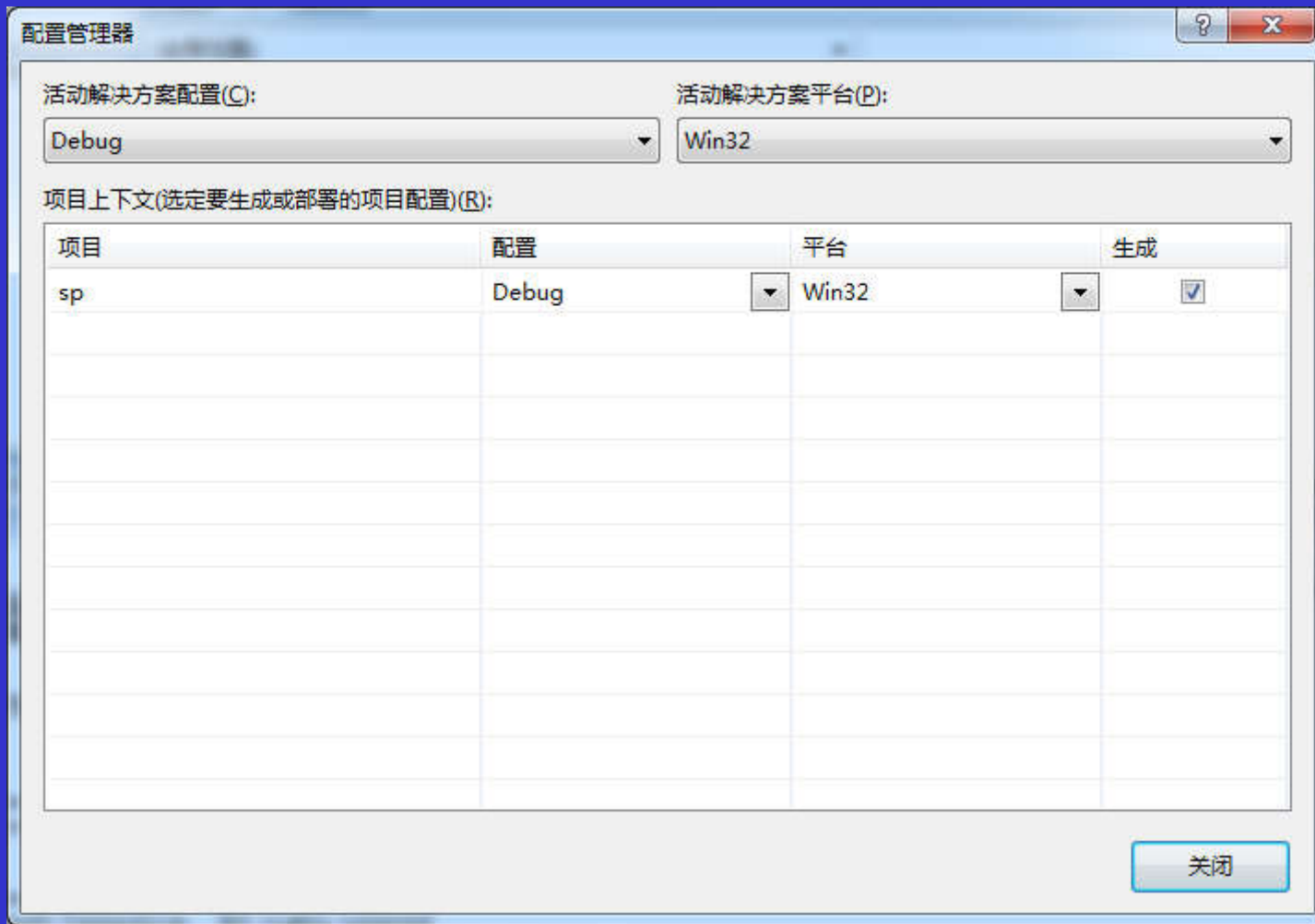
➤ Debug (调试) 方式

➤ Release (发布) 方式

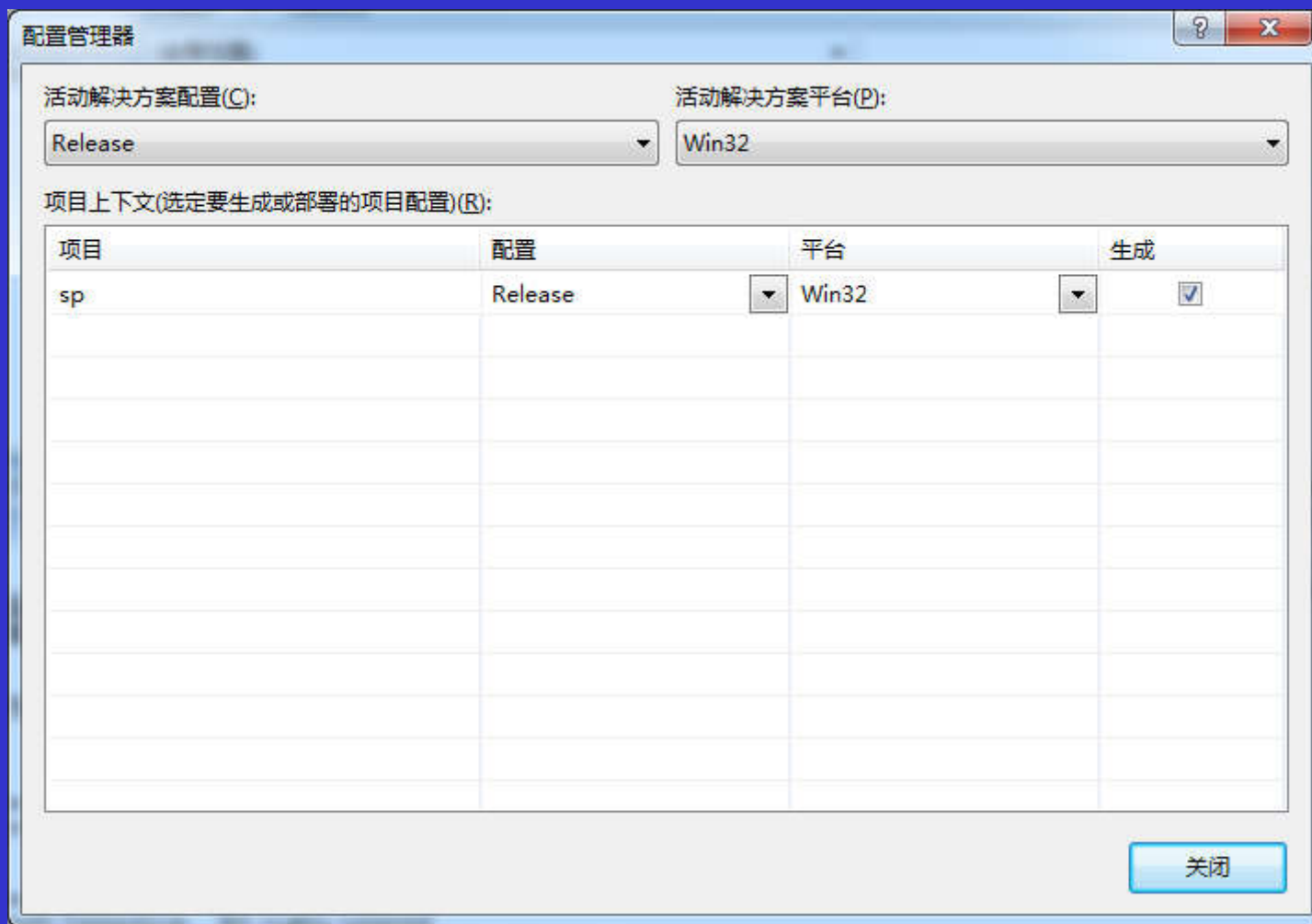
● 当你建立项目进行编译链接时, 编译系统自动默认是 Debug方式, 从编译信息中可以看到
“Configuration: sp - Win32 Debug”。

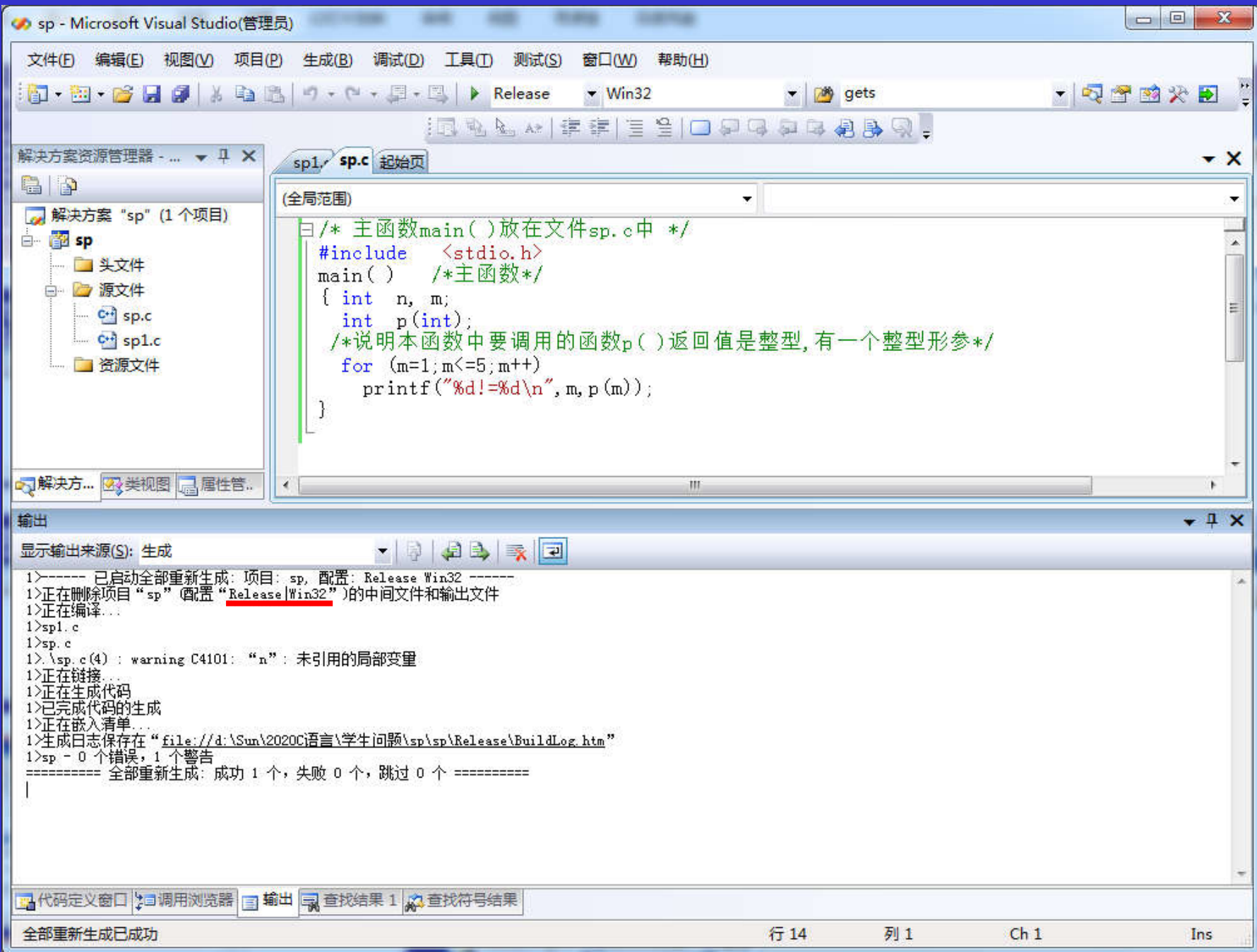
Debug方式方便使用者调试追踪错误, 但编译链接得到的可执行文件一般比Release方式编译链接得到的可执行文件要大, 而且运行速度慢。因为Debug方式自动增加了许多调试追踪信息。

- 当你的程序在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框。






● 单击“活动解决方案配置”的下拉菜单，选中“Release”
按下“关闭”按钮，则选中激活了Release编译方式。再次
“重新生成”会得到如下一页所示的结果。



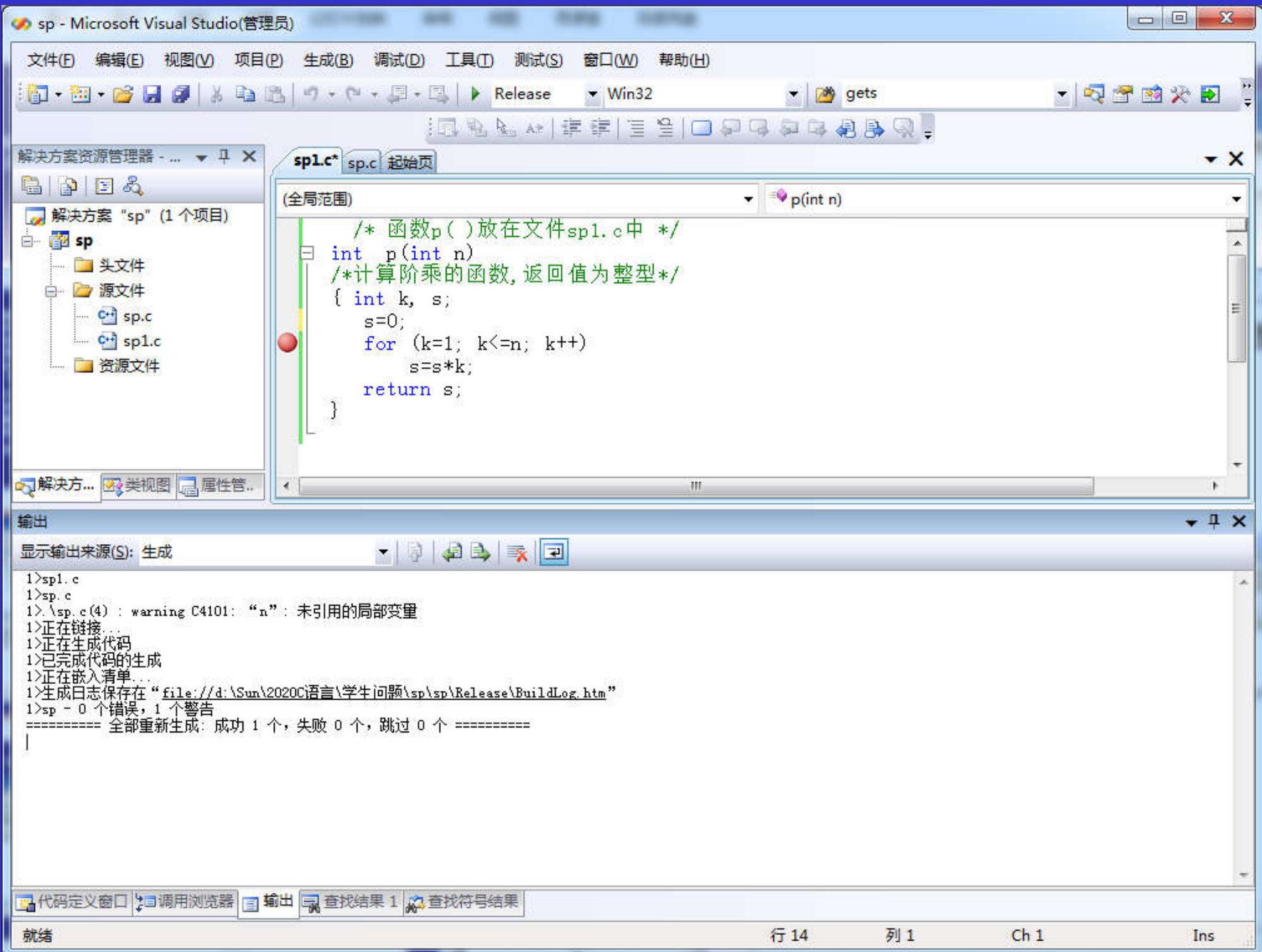


如何跟踪调试程序的逻辑错误（尤其是死循环错误）

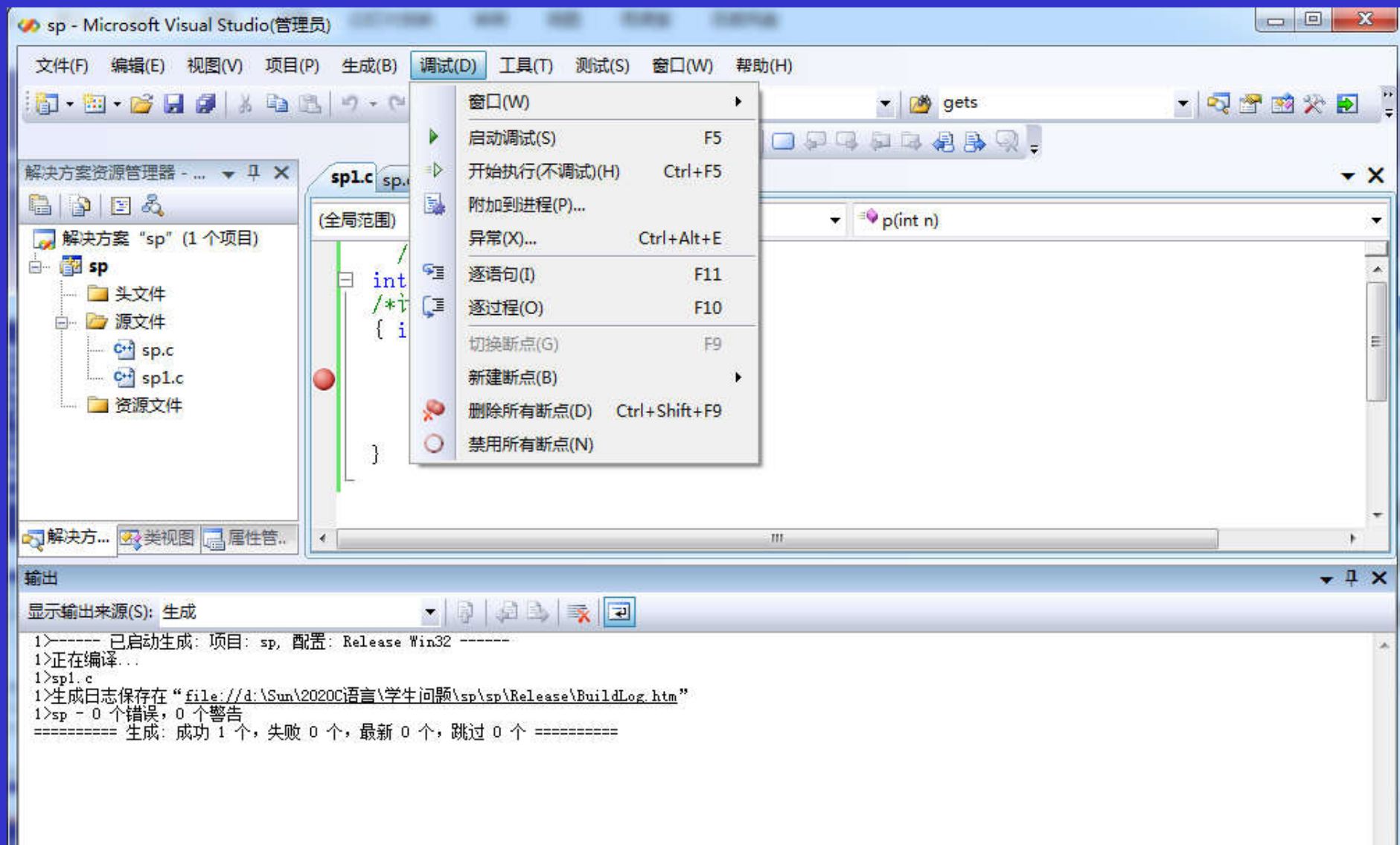
● 在Debug编译方式下，在可能出错的程序的可能出错的程序行，用键盘上方的F9键或者按（此按钮在启动调试后会出现）或者选“调试”菜单中的“切换断点”项，在光标所在的行上设置或取消断点（breakpoint）。当你把光标放到图标上，系统会自动出现提示：“断点”


● 凡是设置了断点的行，在行的前面会出现一个大圆点，如下页图所示。设置断点的目的是当程序运行到断点所在的行时，会自动暂停下来，让你查看此时各个变量的值。

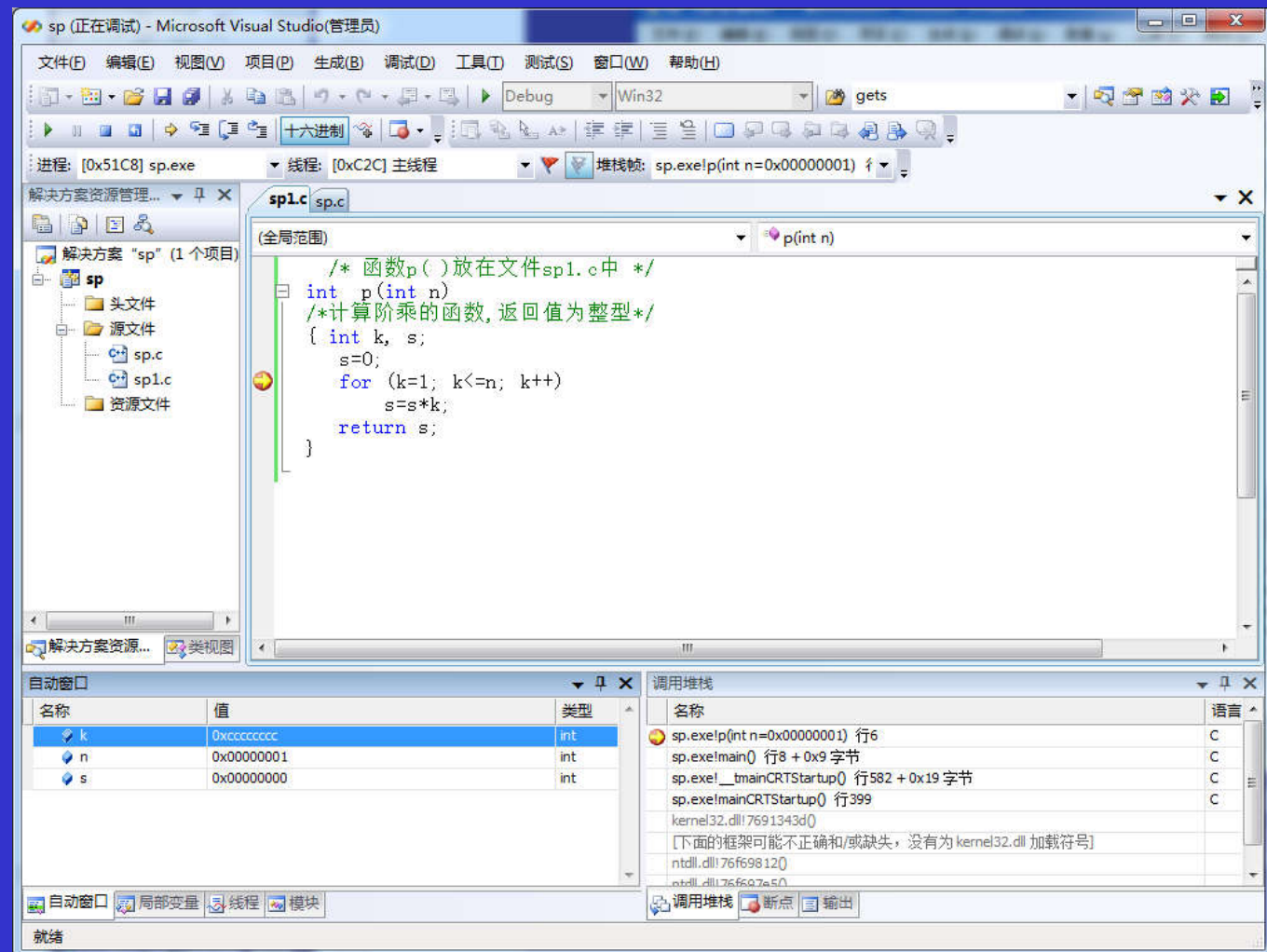
提示：可以在一个程序中同时设置多个断点。

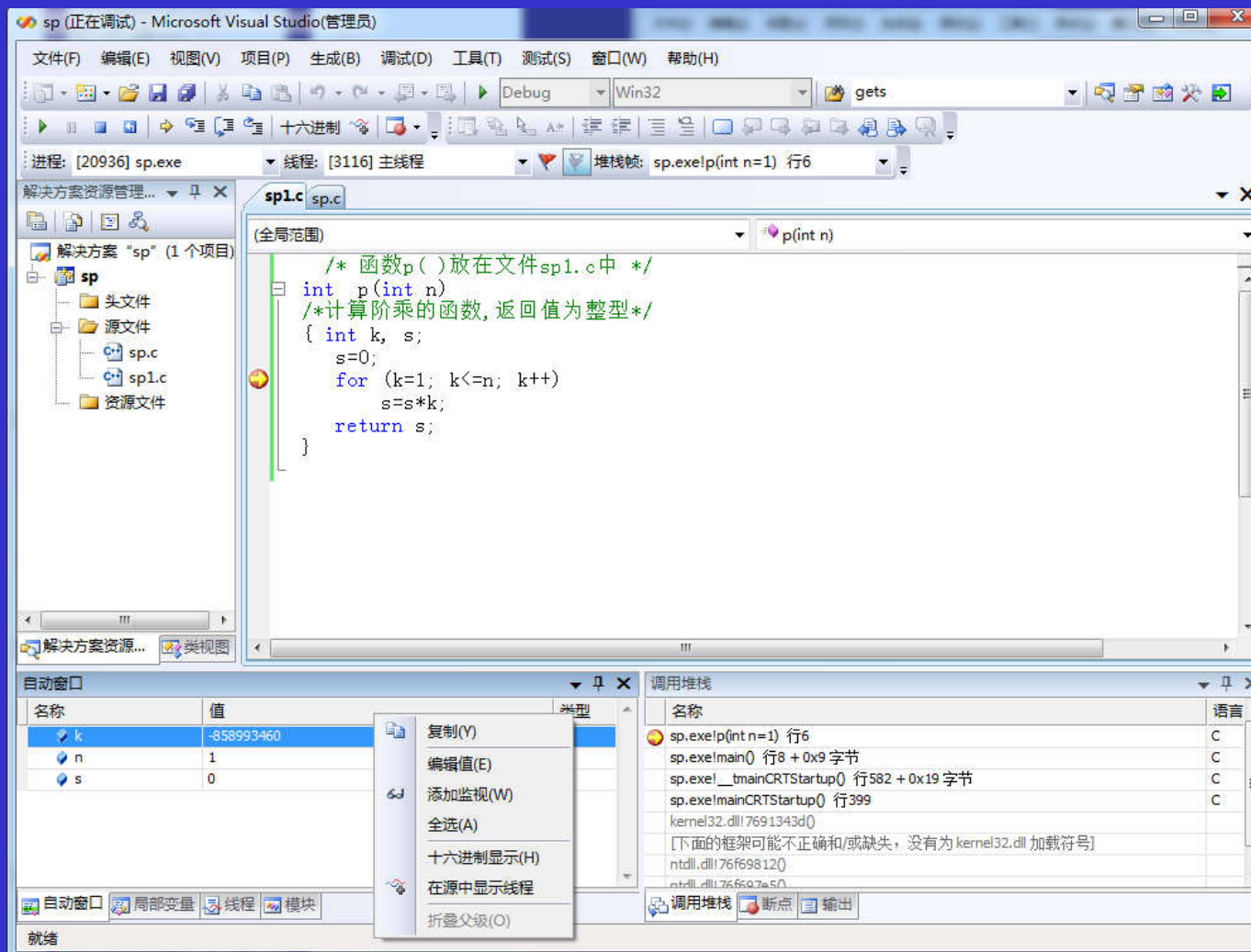


- 选择“调试”菜单，会看到下图所示的各项功能：



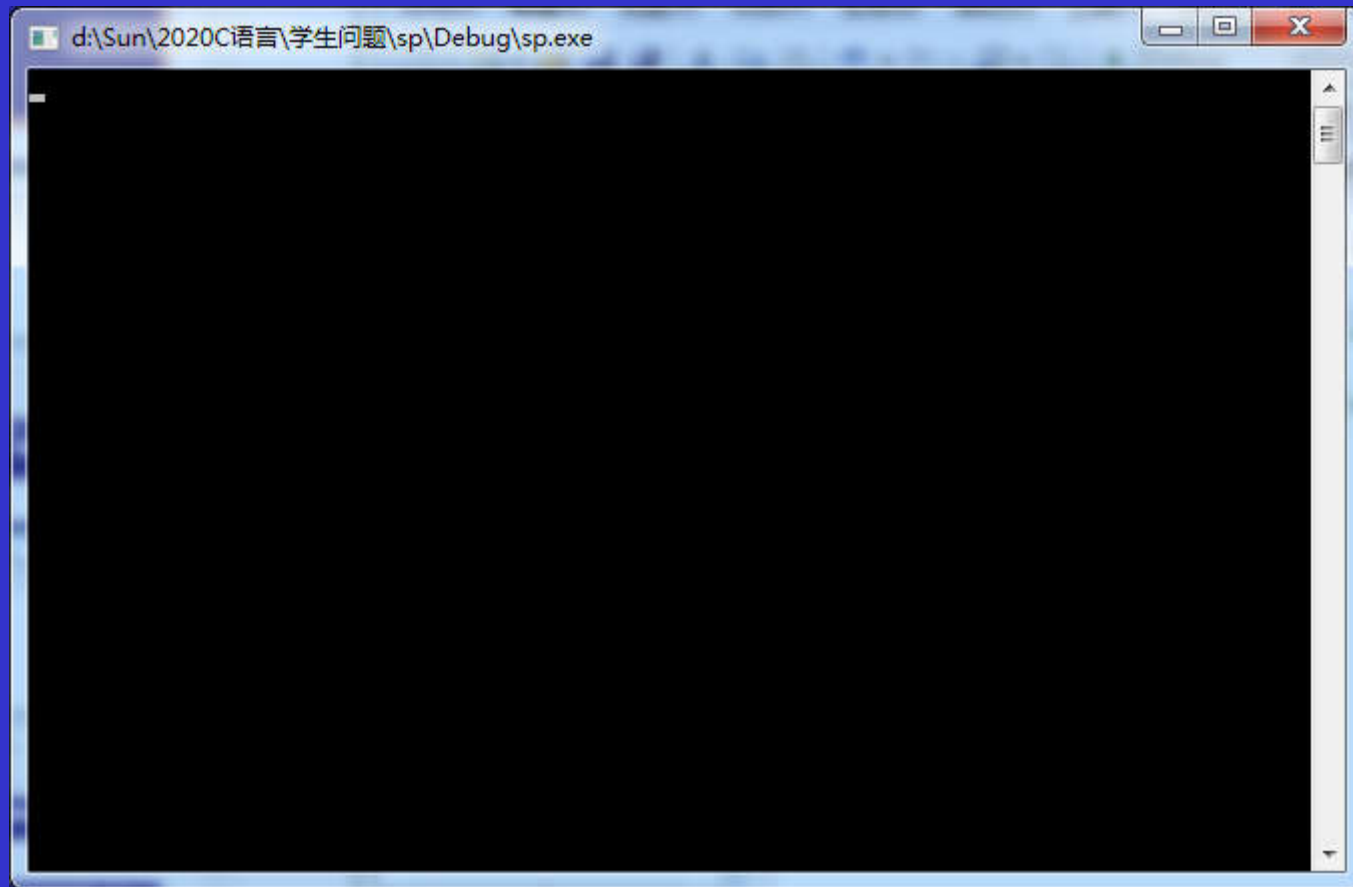
● 设置完断点后，用“调试”菜单中“启动调试”或按F5键或点图标，启动调试（开始Debug），程序执行到断点处会自动停下，显示当前变量的值，如下图所示：



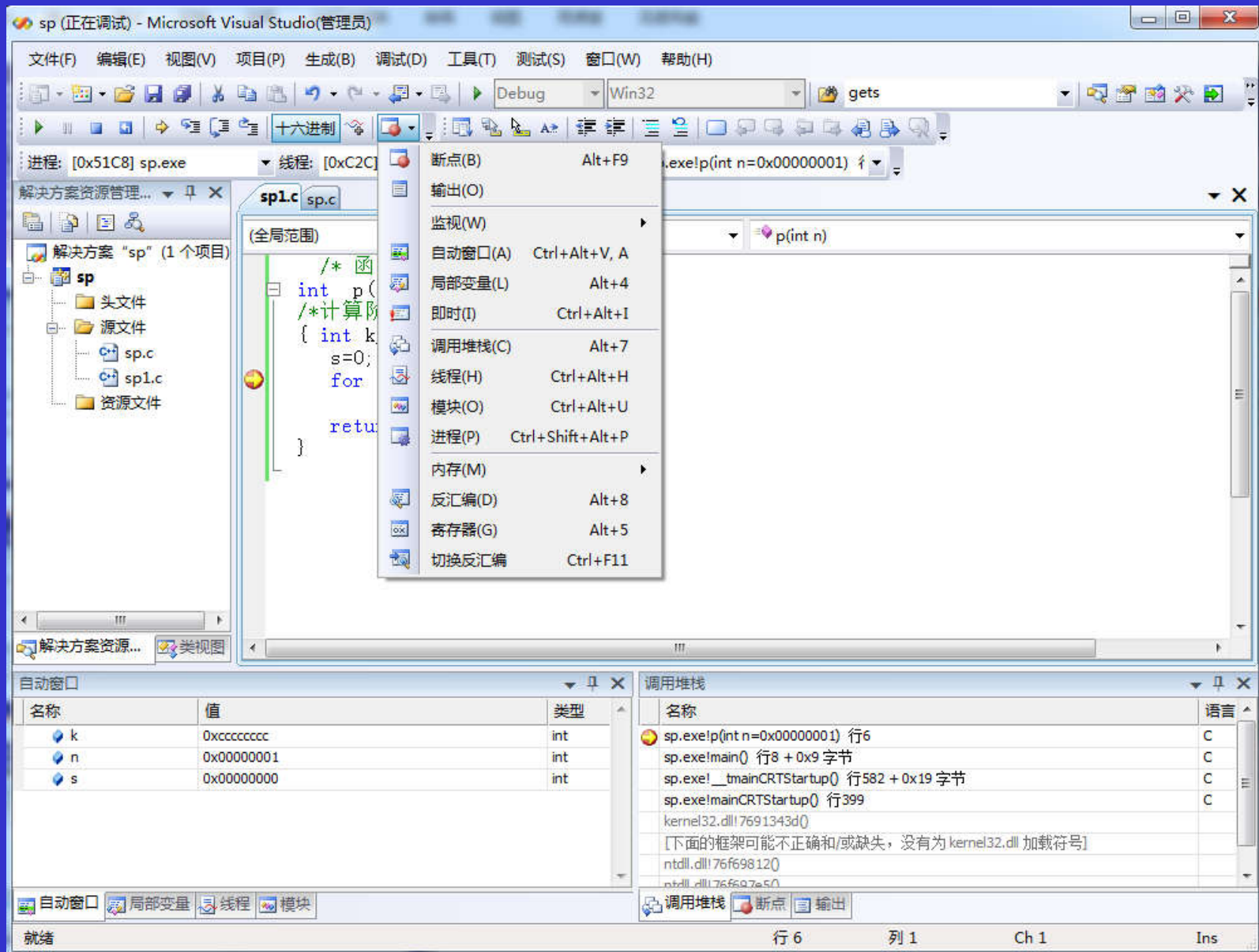


可以不用“十六进制显示”而改为十进制显示各变量结果。此时会看到由于断点所在的行 `for(k=1; k<=n; k++)` 还未被执行，因此`k`的值是一个随机数，而`n`的值为1(将求1!)，`s`的值为0。

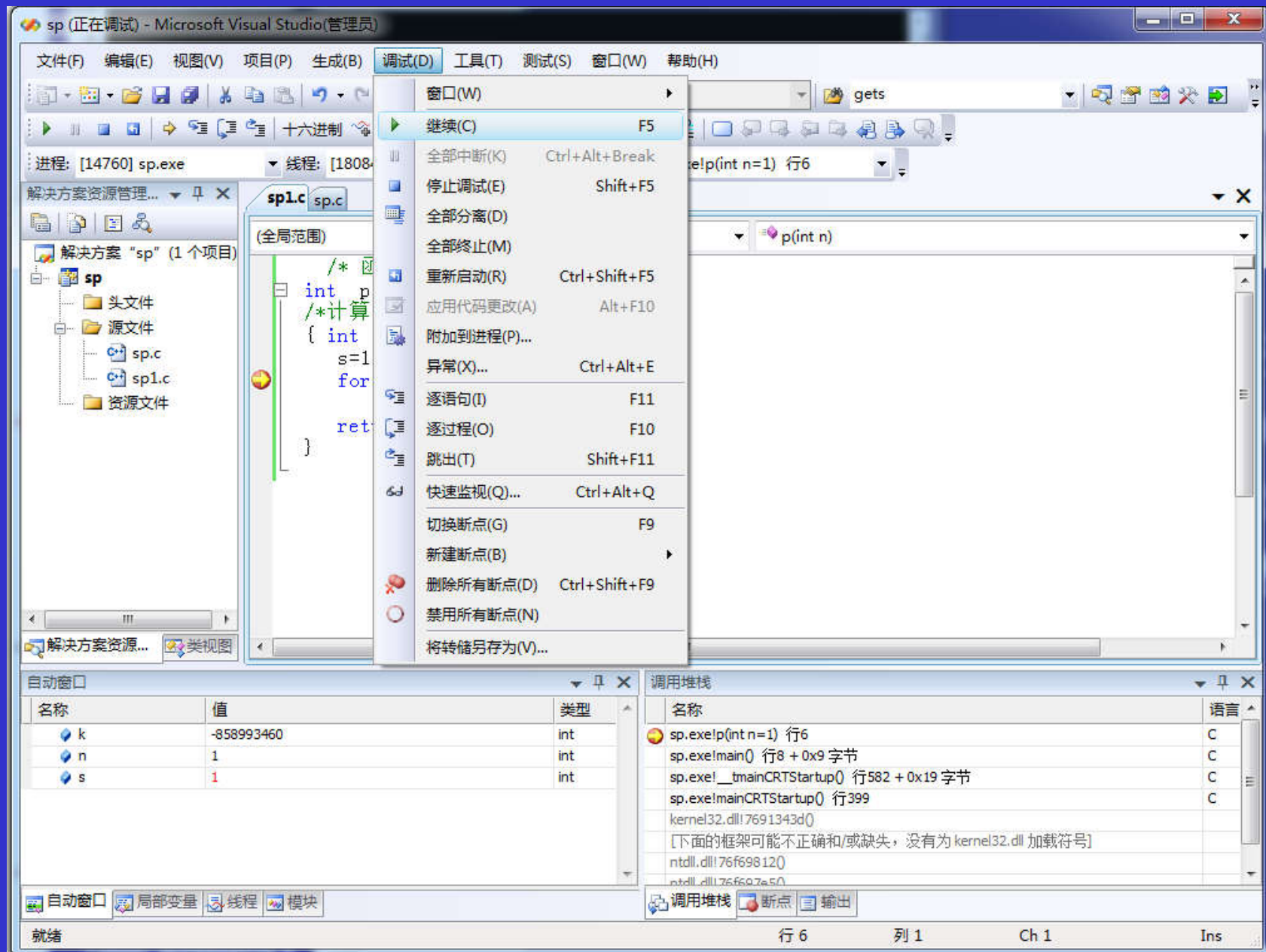
- 同时会在此窗口后面弹出黑色窗口（如果程序有读入操作，点击选中此窗口后进行输入）：



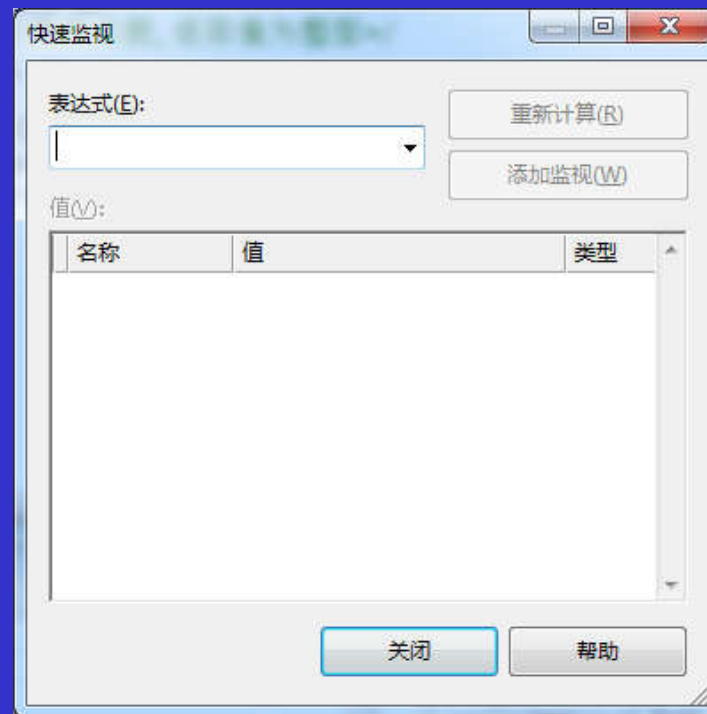
● 选择右边的下拉菜单，会看到下图所示的各项功能，其中 Alt+4 键是显示当前函数的局部变量值：



- 此时再去查看“调试”菜单，会看到下图所示的功能：



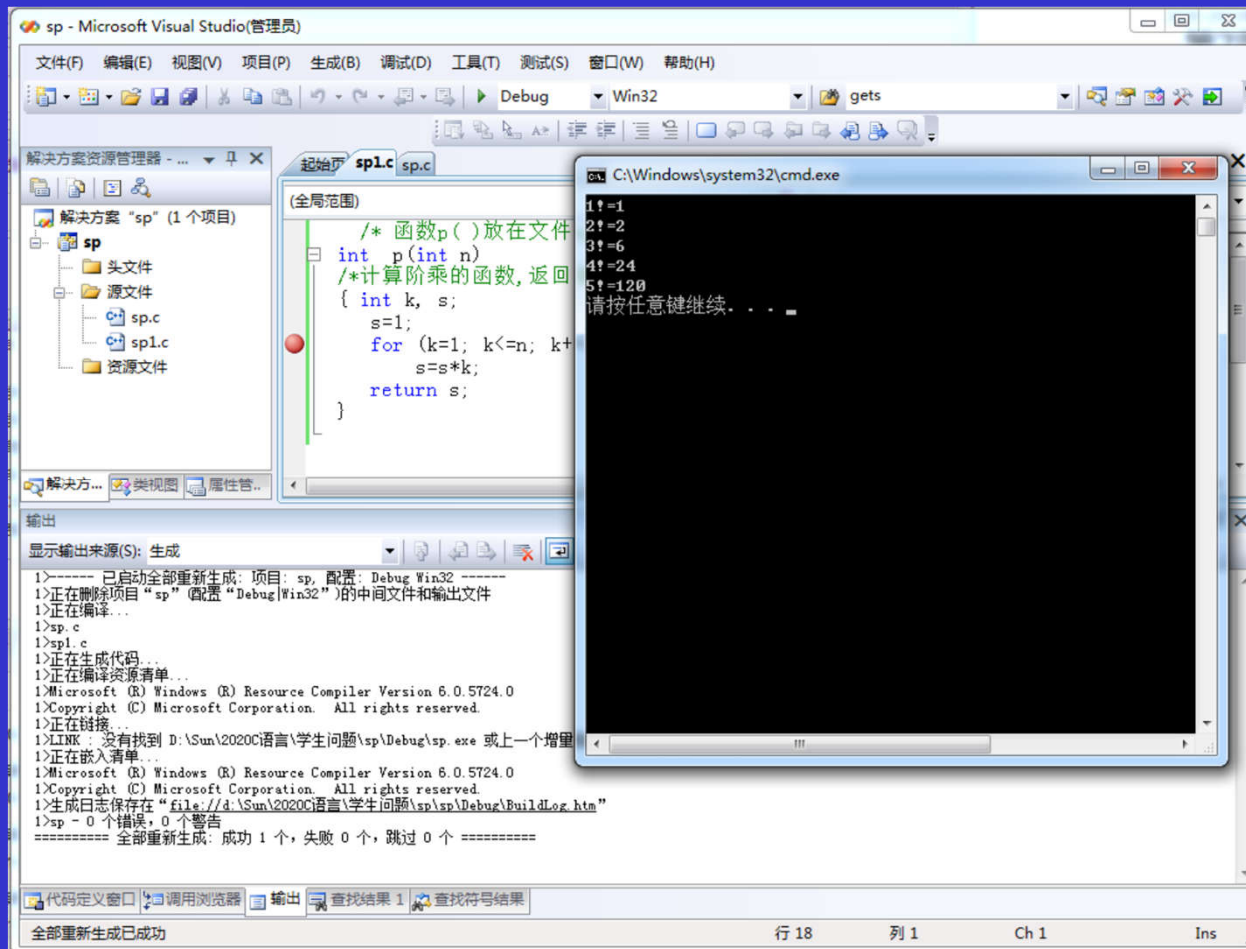
- 可以用：F10键（逐个语句）、F11键（逐个过程）单步执行程序，考察变量值的变化，找出错误。
- 还可以用Shift-F11，Ctrl-F10（执行一次直到断点停）等组合键，进行单步执行程序，这些功能键功能略有差异，大家可以自己摸索领会。
- 还可以用Shift-F9开一个快速监视窗，观察你指定的变量或者表达式的值。如下图所示。



- 也可以在左下方的“自动窗口”中空白行上“名称”列中，输入你想跟踪观察的变量名，回车后，会立刻显示此变量的当前值。

- 经过跟踪我们会发现，由于s初值为0，因此无论for循环乘多少次k，s还是0，显然是s的初值不对，s初值应该改为1。此时可以选择“调试”菜单中的“停止调试”或用 Shift-F5 停止调试 (Debug) 状态进行程序修改，或者用 Ctrl-Shift-F5 重新开始新的调试。

●修改后程序再次编译链接运行，得到了预期的结果。如下图所示。



●此时可以再把光标移到相应行上用F9取消各个断点，重新编译链接。如果确认无错，可以选择Release方式再次编译链接，看运行结果是否正确。**注意：**Debug方式下运行正确的程序在Release方式下运行不一定正确。

8.1.3 函数的调用

- 函数调用的一般形式为：函数名(形参名称与类型列表)

如在上述例子中，如果要计算5!并赋给变量s，则调用语句为： `s=p(5);`

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、函数名以及形参的类型。

说明方式：

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,...);

在后一种形式中，各“形参名”可以是任意的，可以与被调用函数中的形参名一样，也可以不一样。但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。

在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。

在例8-1的主函数中，用说明语句 `int p(int);`

说明了本函数中所要调用的函数`p()`为整型，有一个整型形参。这个说明与 `int p(int x);`等价，其中标识符`x`可以是任意的。而在例8-2的主函数中，用说明语句 `double q(double);`说明了本函数中所要调用的函数`q()`为双精度实型，有一个双精度实型形参。这个说明与`double q(double x);`等价，其中标识符`x`可以是任意的。

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（如函数类型不匹配，参数个数不一致，参数类型不匹配等）时，就会给出警告性质的错误信息，用户可以根据系统提示的错误信息去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！

● 需要特别说明的是：在以前的C版本中，在调用函数中对被调用函数进行说明时不是采用函数原型，而只说明函数名和函数类型，即采用以下形式：函数类型 函数名()；

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。下面举一个例子来说明这个问题。

例如有以下程序：

```
#include <stdio.h>
main( )
{ float x;
  double y, f( ); /*未采用函数原型说明*/
  x=1.0f;
  y=f(x);
  printf("y=%f\n", y);
}
double f(float x)
{ double y;
  y=2*x+1.0;
  return(y);
}
```

程序运行的结果为:

y=1.000000

这个运行结果显然是错误的。这个错误是什么原因造成的呢？

- 在C语言中，实型运算统一默认为双精度运算，因此，实参中的实型表达式的值一定是双精度型的，其对应的形参必须也是双精度型的。也就是说，在C语言中，函数中的实型形参类型缺省是双精度实型（即double型），而不是单精度实型（即float型）。

- 本例中，主函数中的变量x虽然被定义为float型，但执行 $y=f(x)$ ；时，首先取实参x的值转换成双精度型，因此，最后传送给函数f()的值是双精度型的，而函数f()中的形参却是float型的，造成实参与形参的类型不一致，导致形参接收到的数据是错误的，从而计算得到的返回值也是错误的。

- 这种实参类型与形参类型的不一致，编译系统无法发现，因为主函数中只说明了被调用函数f()的返回值类型，但没有说明其形参的类型，即没有用函数原型说明，导致编译器将不(无法)检查实参类型与形参类型是否一致。

将函数f()中的形参x定义成double型，即程序改为：

```
#include <stdio.h>
main( )
{ float x;
  double y, f( );
  x=1.0f;
  y=f(x);
  printf("y=%f\n", y);
}
double f(double x)
{ double y;
  y=2*x+1.0;
  return(y);
}
```

此时程序的运行结果为：

y=3.000000

●在这种情况下，虽然在主函数中没有对函数f()的形参类型进行说明，但由于实参与形参的类型恰好一致，运行结果就正确了。这个例子也说明了另外一个问题，即在进行数值运算时，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。

●如果在主函数中采用函数原型对被调用函数进行说明，即程序改为：

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

```
main( )
```

```
{ float x;
```

```
    double y, f(float x); /*采用函数原型进行函数向前引用说明*/
```

```
    x=1.0f;
```

```
    y=f(x);
```

```
    printf("y=%f\n", y);
```

```
}
```

```
double f(float x)
```

```
{ double y;
```

```
    y=2*x+1.0;
```

```
    return(y);
```

```
}
```

程序运行的结果为：

y=3.000000

采用函数原型进行函数向前引用说明，使实参与形参的类型一致，因此运行结果正确。

强调：在调用函数中应采用函数原型对被调用函数进行说明。

C语言规定，在以下两种情况下可以不在调用函数中对被调用函数作说明：

① 被调用函数的定义出现在调用函数之前。

例如，如果将例8-2中的函数q()放在主函数的前面，即程序改成：

```
#include <stdio.h>
double q(double r)
{ return 3.1415926*r*r;
}
main( )
{ double r1, r2;
  printf("input r1,r2: ");
  scanf("%lf%lf",&r1,&r2);
  printf("s=%f\n",q(r1)+q(r2));
}
```

② 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。

在文件开始处用外部函数向前引用说明了被调用的函数原型后，其后整个文件中所有调用此函数的函数中都不需要再进行说明而可以直接使用。

例如：`#include <math.h>`
就是进行了全部数学库函数的向前引用说明，因此在本文件的任意函数中你可以随意使用数学库函数。

```
#include <stdio.h>
double q(double);
main( )
{ double r1,r2;
  printf("input r1,r2: ");
  scanf("%lf%lf",&r1,&r2);
  printf("s=%f\n",q(r1)+q(r2));
}
double q(double r)
{ return 3.1415926*r*r;
}
```

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。例如，在例8.1中，可以写表达式 $p(p(3))$ ，这实际上是计算 $6!$ 。

$$p(p(3))=p(3!)=p(6)=6!=720。$$

下面再举一个例子来说明函数的应用。

【例8-3】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。

其C函数如下：

```
#include <math.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;
}
```

●在这个函数中，因为要调用求平方根的函数sqrt()，因此要引用C库函数头文件<math.h>。

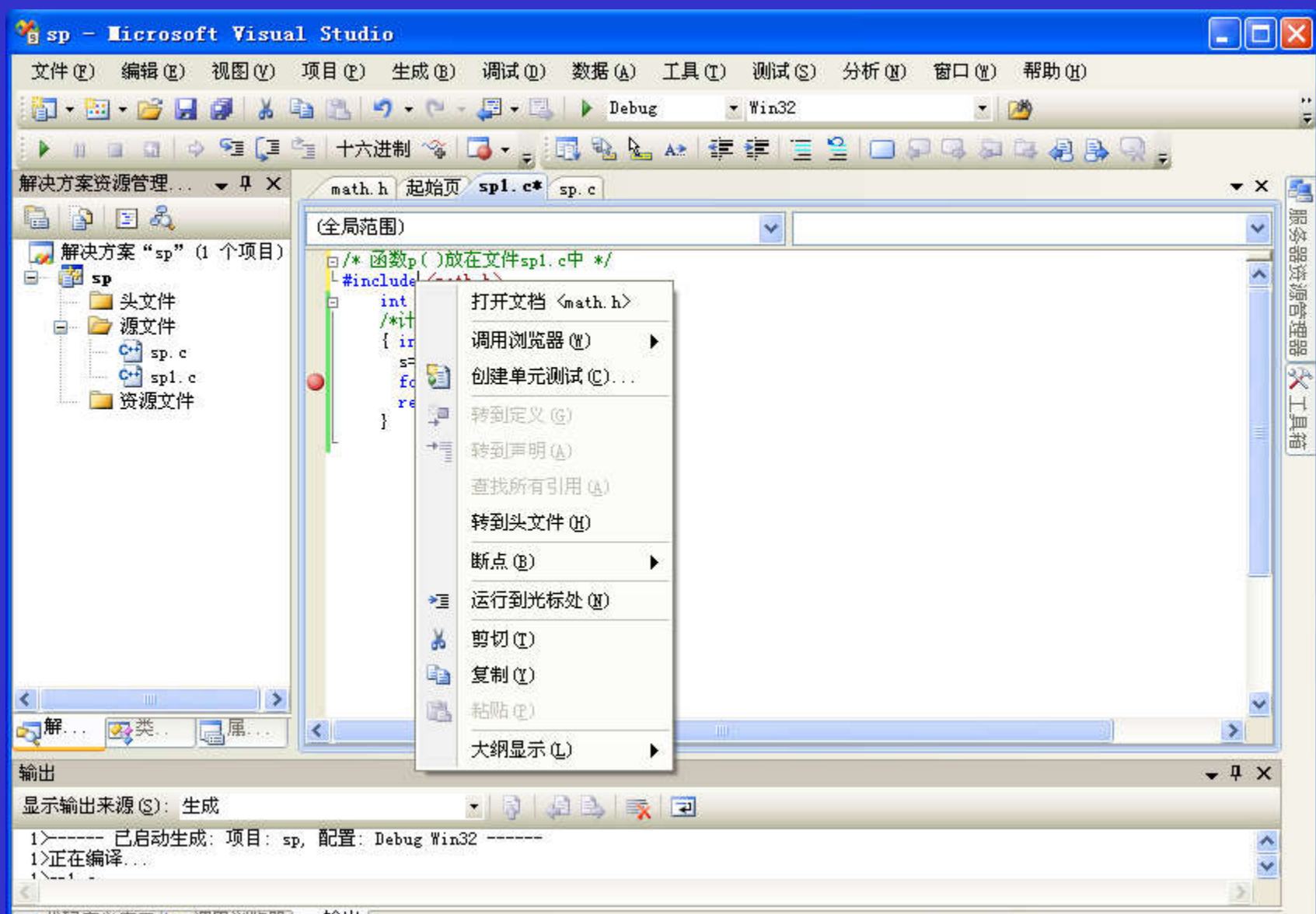
●反过来，你应该明白，C库函数头文件<math.h>中主要是各个数学函数的原型说明。

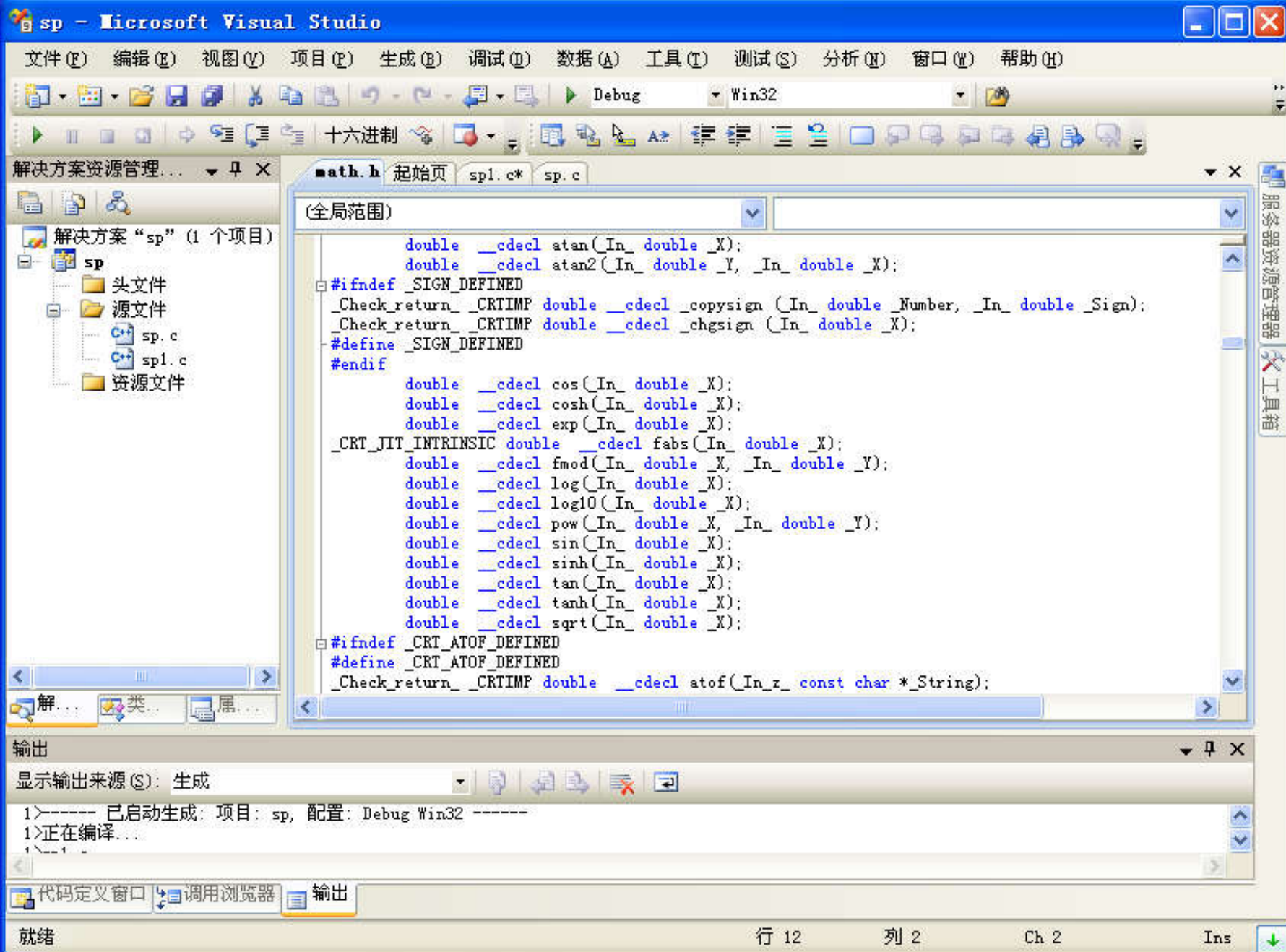
●有了这个函数sushu()后，如果需要输出3到100之间的所有素数，可以用下列主函数来调用它：


```
#include <stdio.h>

main( )
{ int k, sushu(int);
  /*如果主函数在函数sushu( )定义之后,
    就可以不要说明sushu(int) */
  for (k=3; k<100; k=k+2)
    if (sushu(k)) /* 或 if (sushu(k)==1) */
      printf("%d\n", k);
}
```

如何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：





8.2 模块间的参数传递

8.2.1 形参与实参的结合方式

地址结合

地址相同

● 是指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)**。

数值结合

地址不相同

● 是指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，也可以是表达式或常量。

- C语言函数之间的参数传递是**传值**；是通过栈(stack)来传递，是单向传递，压栈顺序是：函数参数从右向左传递。结果是：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。例如：

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

```
main( )
```

```
{  int a=3, b=2, c=1, f (int, int, int);  
    f(a,b,c);  
    printf("c=%d\n", c);  
}
```

```
int f(int a, int b, int c)
```

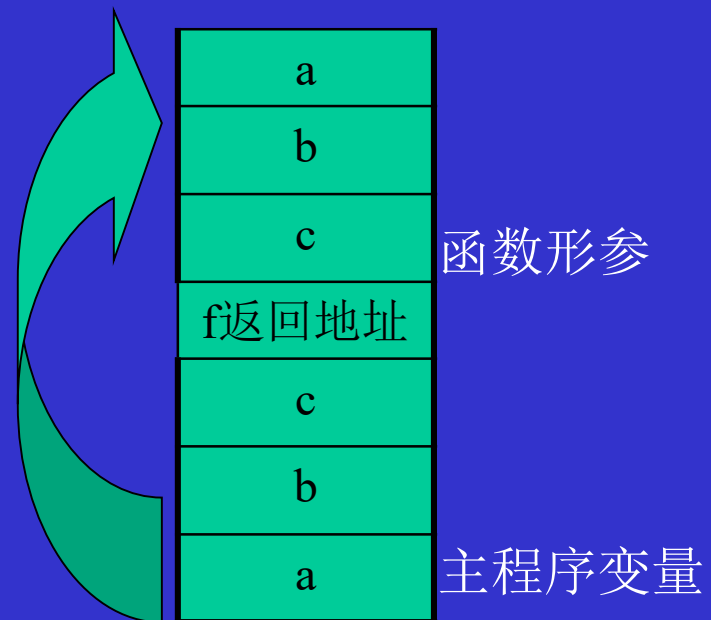
```
{  c= a+b;  
    printf("c=%d\n", c);  
    return c;  
}
```

执行结果是：

c=5

c=1

函数中的赋值c=a+b并没有影响主程序中的c值。



● 在C语言中，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。

● 说明

【例8-4】 用迭代法求方程

$x - 1 - \arctan x = 0$ 的一个实根。精度要求为 $\varepsilon = 0.000001$ 。

其迭代公式为：

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

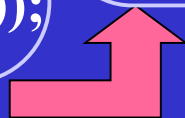
现在用函数来编写这个C程序。

为了通用性，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);
  /*说明计算迭代值的函数为双精度
    型，一个双精度型形参*/
  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*/
}
```

此函数是通用的，可以对任意
 $x_{n+1} = f(x_n)$ 求根。独自放在
一个.c文件中。



然后编写一个主函数以及计算迭代值的函数 $f(x)$ ，放在另一个.c文件中如下：

```
#include <stdio.h>
#include <math.h>
main( )
{ int m, subroot(double, double);
  double x, eps;
  x=1.0;
  eps=0.000001;
  m=subroot(x, eps);
  printf("m=%d\n", m); /*输出迭代次数*/
  printf("x=%f\n", x); /*输出方程根*/
}
double f(double x) /*计算迭代值的函数*/
{ return 1.0+atan(x);
}
```

程序的运行结果为：

$x=2.132268$

$m=10$

$x=1.000000$

显然,最后主程序输出的方程根值是不对的, x 的值并没有改变,因为求根函数 `subroot` 返回的是迭代次数 m , C 语言函数调用参数是值传递, 函数中的 x 值并没有返回到主程序中。

8.2.2 局部变量与全局变量

在C语言中，函数调用时，实参与形参是采用数值结合的方式。因此，除了能将函数值通过函数名返回给调用函数外，不能将形参值传递给调用函数中的实参。但可以通过定义全局变量的方法来实现各函数之间的参数传递。

● 局部变量

在函数内定义的变量

函数内定义的变量，只在该函数范围内有效。因此，不同函数中的局部变量可以重名，互不混淆互不相干。

特别要指出的是，函数中的形参也是局部变量。

● 全局变量

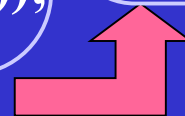
在函数外定义的变量

函数外定义的变量，同一个程序中的所有函数都可以访问。

该程序中的所有函数都可以直接访问

```
#include <stdio.h>
double x; /* 定义为全局变量 */
int subroot(double eps)
{ int m;
  double x0, f(double);
  /*说明计算迭代值的函数为双精度
    型，一个双精度型形参*/
  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*/
}
```



```
#include <stdio.h>
#include <math.h>
extern double x; /* 引用全局变量 */
main( )
{ int m, subroot(double);
  double eps;
  x=1.0; /*给全局变量赋初值*/
  eps=0.000001; /*给局部变量赋初值*/
  m=subroot(eps);
  printf("m=%d\n", m); /*输出迭代次数，函数返回值*/
  printf("x=%f\n", x); /*输出方程根，全局变量*/
}
double f(double x) /* 计算迭代值的函数 */
{ return 1.0+atan(x); /* 这里的x是形参，是局部变量 */
}
```

程序的运行结果为:

x=2.132268

m=10

x=2.132268

- 全局变量的有效范围是从定义变量的位置开始到本源文件结束。

- 如果在上述程序中，将全局变量x的定义放在函数subroot()的后面，main()的前面，则应在程序开头处加上全局变量的引用说明：

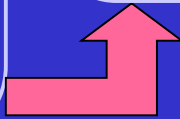
extern double x;

extern double x; 不是定义变量，只是说明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。

程序也可变为：

```
#include <stdio.h>
#include <math.h>
extern double x; /*引用全局变量*/
int subroot(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));
  if (m>100) printf("FAIL!\n");
  printf("x=%f\n",x);
  return(m); /*返回迭代次数*/
}
```

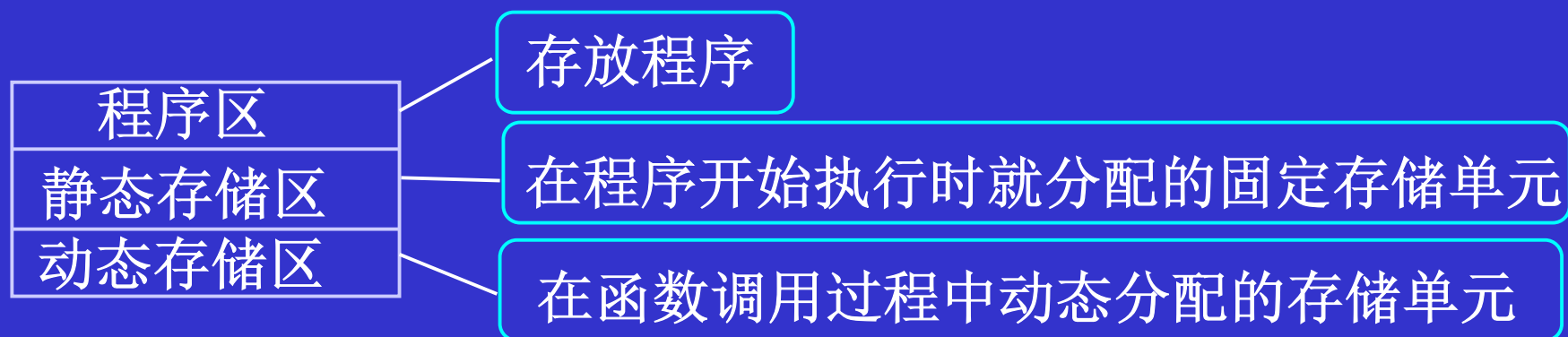
```
#include <stdio.h>
#include <math.h>
double x; /*定义全局变量*/
main( )
{ int m,subroot(double);
  double eps;
  x=1.0;
  eps=0.000001;
  m=subroot(eps);
  printf("m=%d\n",m);
  printf("x=%f\n",x);
}
double f(double x)
{ return 1.0+atan(x); }
```



- 全局变量的有效范围是从定义变量的位置开始到本文件结束。
- 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**）。
- 利用全局变量可以实现各函数之间的数据传递。但是要指出，除非十分必要，一般不提倡使用全局变量，其原因有以下几点：

- ① 由于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，即使实际正在执行的函数中根本用不着这些全局变量，它们也要占用存储空间。
- ② 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。
- ③ 在函数中使用全局变量后，使各函数模块之间的互相影响比较大，甚至产生“副作用(side effect)”，从而使函数模块的“内聚性”差，而与其他模块的“耦合性”强。
- ④ 在函数中使用全局变量后，会降低程序的清晰度，可读性差。

8.2.3 动态存储变量与静态存储变量



变量的存储类型

- 一是数据类型:

如整型(int),实型(float),字符型(char),双精度型(double)等。

- 二是数据的存储类型:

分为自动类型(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。

函数中局部变量默
认为是auto类型

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

静态变量

用static说明的局部变量或全局变量

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，在下次调用时仍为上次调用结束时的值。

注意：

- ① 形参不能定义成静态存储类型。
- ② 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，而对自动变量赋初值是在调用时进行的，每次调用将重新赋初值。如例8-5所示。
- ③ 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，但在定义自动变量时若不赋初值，则其初值为随机数。
- ④ 由于局部静态变量有“副作用”，造成多次运行函数的结果之间有关联效应，若无多大必要，尽量不用局部静态变量。
- ⑤ 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。或者换一个说法：一个文件中定义的外部静态变量，不能被本文件之外的任何函数访问。

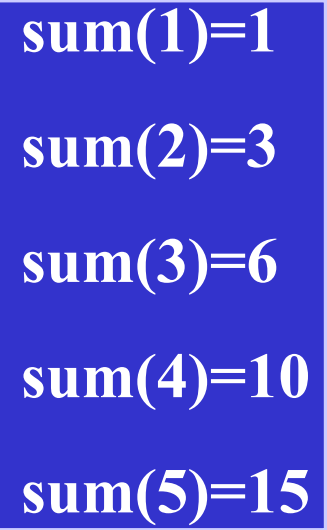
【例8-5】 设有如下C程序：

```
#include <stdio.h>

int ksum(int n)
{ static int x=0;
  x=x+n;
  return(x);
}

main( )
{ int k, ksum(int);
  for (k=1; k<=5; k++)
    printf("sum(%d)=%d\n",k,ksum(k));
}
```

本例中的x是局部静态变量。



```
sum(1)=1
sum(2)=3
sum(3)=6
sum(4)=10
sum(5)=15
```

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

```
int ksum(int n)
```

```
{  int x=0;
```

```
  x=x+n;
```

```
  return(x);
```

```
}
```

```
main( )
```

```
{ int k, ksum(int);
```

```
  for (k=1; k<=5; k++)
```

```
    printf("sum(%d)=%d\n",k,ksum(k));
```

```
}
```

sum(1)=1

sum(2)=2

sum(3)=3

sum(4)=4

sum(5)=5

extern外部变量

用extern说明的变量

- 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。但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。

extern全局变量有以下几种用途:

- 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访问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。

- 下列程序可以实现两个变量值交换：

```
#include <stdio.h>
main( )
{          int x,y; /*x与y定义为外部变量*/
  void swap( );
  scanf("%d%d",&x,&y);
  swap( );
  printf("x=%d,y=%d\n",x,y);
}
int x,y;
void swap( )
{ int t;
  t=x; x=y; y=t;
  return;
}
```

由于在主函数中用 **extern** 说明了变量 **x** 和 **y** 是外部变量，因此，后面定义的全局变量也适用于主函数。

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，后面定义的全局变量不适用于主函数。**swap** 不能将主函数中的 **x** 和 **y** 的值交换。

- 在下列程序中，两个函数分别存放在两个文件中：

```
/* file1.c */
```

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

```
int x,y;
```

```
main( )
```

```
{ void swap( );
```

```
    scanf("%d%d",&x,&y);
```

```
    swap( );
```

```
    printf("x=%d,y=%d\n",x,y);
```

```
}
```

```
/* file2.c */
```

```
extern int x,y;
```

```
void swap( )
```

```
{ int t;
```

```
    t=x; x=y; y=t;
```

```
    return;
```

```
}
```

其中在主函数所在的文件file1.c中定义了全局变量x与y，在函数swap()所在的文件file2.c中将x与y说明为外部变量，此时，在函数swap()中就可以使用文件file1.c中定义的全局变量x与y。

- 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

特别需要说明的是，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。别的编译系统不一定这样。

- 下列程序是**错误**的，编译链接时会产生链接错误：

```
/* file1.c */
static int x,y; /*x与y是只适用于本文件的静态全局变量*/
#include <stdio.h>
main()
{void swap( );
  scanf("%d%d",&x,&y);
  swap( );
  printf("x=%d,y=%d\n",x,y);
}
```

```
/* file2.c */
extern int x,y;
/*实际上x,y根本未定义*/
void swap( )
{ int t;
  t=x; x=y; y=t;
  return;
}
```

1>正在链接...

1>LINK : 没有找到 C:\file\file1\Debug\file1.exe 或上一个增量链接没有生成它；正在执行完全链接

1>file2.obj : error LNK2001: 无法解析的外部符号 _y

1>file2.obj : error LNK2001: 无法解析的外部符号 _x

1>C:\file\file1\Debug\file1.exe : fatal error LNK1120: 2 个无法解析的外部命令

1>生成日志保存在“file://c:\file\file1\file1\Debug\BuildLog.htm”

1>file1 - 3 个错误，1 个警告

```
/* file1.c */
```

```
static int x,y; /*x与y是只适用  
于本文件的静态全局变量*/
```

```
/* file2.c */
```

```
extern int x,y;  
/*实际上x,y根本未定义*/
```

- ✓ 这是因为, 在主函数所在的文件file1.c中static int x,y;定义的是静态全局变量x与y, 只适用于本文件, 其它文件不能引用和访问。
- ✓ 而在函数swap()所在的文件file2.c中extern int x,y;企图将它们说明为外部变量而访问文件file1.c中全局变量x,y, 这是不可能的。
- ✓ 在编译器看来文件file1.c中通过extern引用的全局变量x,y最终没有找到定义之处, 因此链接时出现致命错误信息: 两个无法解析的外部符号 x和y。
- ✓ 也不能用: extern static int x,y; 进行说明。

● 若程序改为:

```
/* file1.c */
static int x,y; /*x与y是只适用于本文件的静态全局变量*/
#include <stdio.h>
main( )
{void swap( );
 scanf("%d%d",&x,&y);
 swap( );
 printf("x=%d,y=%d\n",x,y);
}
```

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

编译没有错误，但上述程序没有实现主函数的两个变量值的交换。这是因为，在这个程序中，函数swap()所在的文件file2.c中定义了两个全局变量x与y。主函数所在的文件file1.c中定义了只适用于本文件的两个静态全局变量x与y，因为函数swap()与主函数在不同的文件中，函数swap()不能访问他们，swap()访问的是本文件中定义的全局变量x与y。因此，全局变量虽然名字相同，但不是同一个变量，互不相干，不会产生编译错误。

C变量的作用域(scope)（由大到小）分为：

- 全局作用域 全局变量
- 文件作用域 静态全局变量
- 函数作用域 函数中形参及定义的(静态)局部变量
- 复合语句作用域 复合语句中定义的(静态)局部变量

全局作用域

文件作用域

函数作用域

复合语句作用域



C变量的生命周期（lifetime）：

- (静态)全局变量与静态局部变量

在程序运行过程中一直存在，存放在静态存储区，直到程序运行结束此类变量的生命周期才会结束。

- 函数形参和函数中定义的局部变量

仅限在此函数调用执行过程中存在，当此函数执行结束，函数形参和函数中定义的局部变量的生命周期也随之结束，形参和局部变量所占内存（栈空间）也会被释放。如果重复执行此函数，此类变量会重复产生和消亡。

- 复合语句中定义的局部变量

仅限在复合语句执行过程中存在，当{}执行结束，此类局部变量的生命周期也随之结束，此类局部变量所占内存（栈空间）也会被释放。如果重复执行此复合语句，此类变量会重复产生和消亡。

8.2.4 内部函数与外部函数

内部函数

静态函数

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如下：

static 类型标识符 函数名(形参表)

目的：缩小某函数的作用域，仅限本文件内部使用。内部函数属于文件作用域。

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如下：

[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

【例8-6】 设有如下C程序:

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

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

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

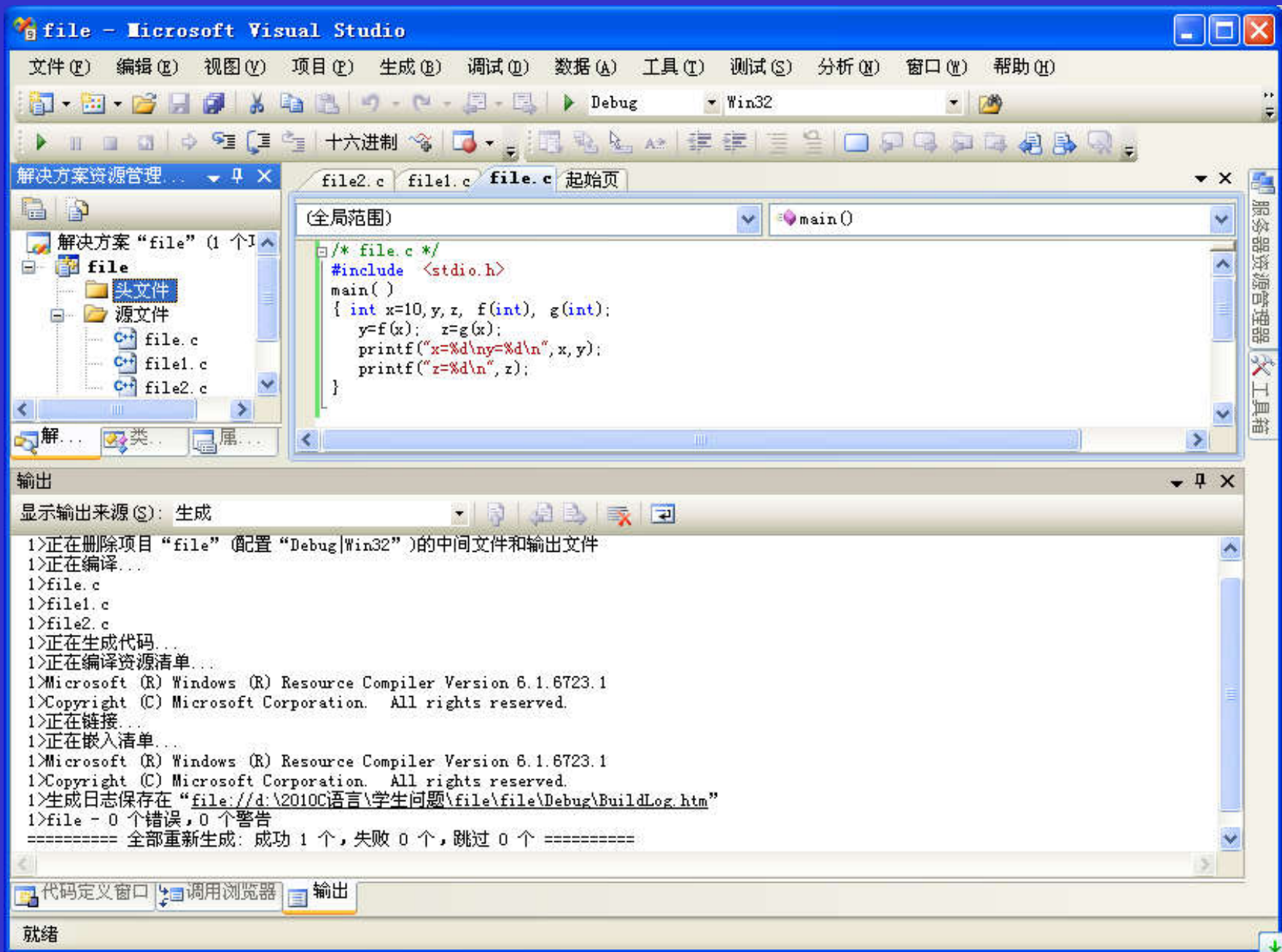
建立一个项目，将这3个文件插入，编译链接，输出结果为

x=10

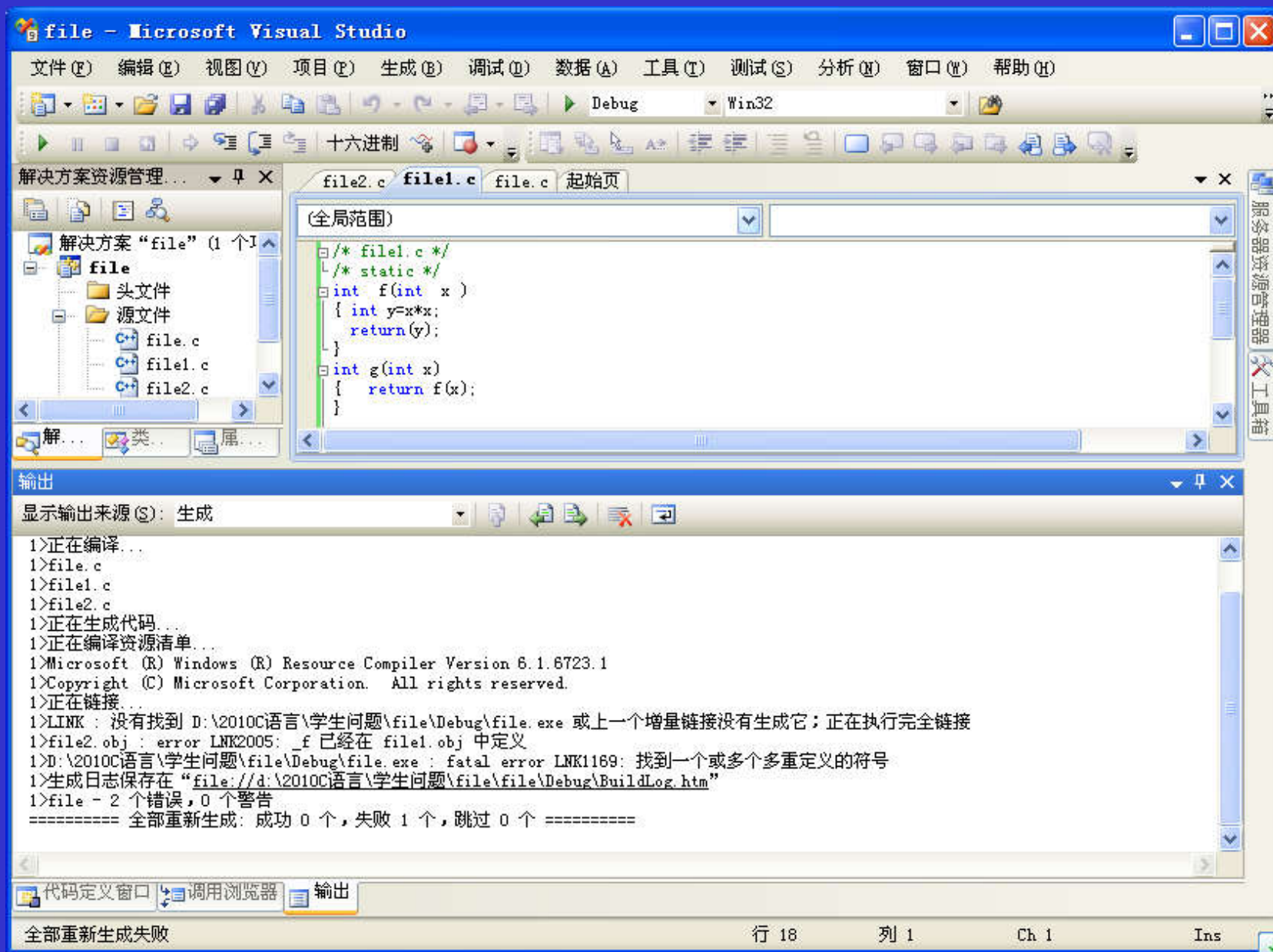
y=1000

z=100

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，因此编译连接时不会出现函数重名错误。



将file1.c中的static去掉,编译链接时就会出现函数重复定义的错误



8.3 函数（模块）的递归调用

递归的基本思想

- 为了降低问题的复杂程度(如问题的规模等), 可以将问题逐层分解简化, 最后归结为一些最简单的问题, 这个过程称为递归(recursion)。
- 这种将问题逐层分解简化的过程, 实际上并没有马上对问题进行求解, 而只是当解决了最后那些最简单的问题后, 再沿着原来分解的逆过程逐步进行综合最终解决复杂问题。
- 递归法与递推法方向相反, 递推法是利用递推公式从最简单问题入手, 从小到大逐步推进, 最终得到某问题的最终解。
- 递归法能解决递推法所能解决的所有问题, 但反过来, 递归法能解决更复杂的问题, 很多问题是递推法所不能解决的。

【例8-7】 编写一个C函数，对于给定的参数n，依次打印输出自然数1~n。

```
#include <stdio.h>
void wrt(int n)
{ int k;
  for (k=1; k<=n; k++)
    printf("%d\n",k);
}
```

如果加上一个附件条件：不希望函数中出现任何循环语句，该如何实现？

```
#include <stdio.h>
void wrt1(int n)
{ if (n>1)
    wrt1(n-1);
  printf("%d\n",n);
}
```

递归方法实现

说明:

1. 在递归函数wrt1()中, n是函数的形参。
2. 在执行函数wrt1()时, 先判断形参n的值是否大于1, 如果是, 则将形参值减1(即 $n-1$)后作为新的实参再次调用函数wrt1();
3. 再次调用函数wrt1()时, 又需判断形参的值(此时已变为 $n-1$)是否大于1, 如果大于1, 则又将形参值减1(即 $n-2$)后作为新的实参再次调用函数wrt1();
4. 这个过程一直进行下去, 直到函数wrt1()的形参值等于1, 不再执行if后的wrt1()调用, 而打印出1。
5. 此时, 由于在先前各层的函数调用中, 函数wrt1()实际上没有执行完, 即各层中的形参值还没有被打印输出, 这就需要逐层返回, 以便打印输出各层中的输入参数2, 3, ..., n。

说明（续）：

6. 为此，在递归函数的执行过程中，需要用栈操作来记忆各层调用中的参数及其返回地址，以便在逐层返回时从上次暂停的位置恢复这些参数继续进行执行。

7. 具体来说，在函数wrt1()开始执行后，随着各次的递归调用，逐次入栈记忆各层调用中的输入参数n, n-1, n-2, ..., 2, 1及返回地址，在逐层返回时，又依次出栈（后进先出，按入栈的相反次序）将这些参数打印输出。

8. 在程序设计中，递归是一个很有用的工具。对于一些比较复杂的问题，设计成递归算法后程序结构清晰，可读性也更强。

1
wrt1返回地址
2
wrt1返回地址
.....
n-3
wrt1返回地址
n-2
wrt1返回地址
n-1
wrt1返回地址

- 自己调用自己的过程称为**自递归调用过程**。
- 在C语言中，自己调用自己的函数称为**递归函数**。

直接递归

指直接调用函数本身。例8-7中的函数wrt1()就是一个**直接递归函数**，或称为**自递归函数**。

间接递归

指函数通过调用别的函数调用自身。例如，两个函数之间的相互调用关系就属于间接递归调用。

```
int f1(int x)
{ int y, z;
  ...
  z=f2(y);
  ...
  return(z*z);
}
```

```
int f2(int x)
{ int a, b;
  ...
  b=f1(a);
  ...
  return(3*b);
}
```

函数f1()中调用函数f2()，而在函数f2()中调用函数f1()，因此，函数f1()实际上是通过函数f2()来间接调用了自身，这种递归调用称为间接递归调用。

- 如果将打印语句放在递归语句之前：

```
#include <stdio.h>
void wrt1(int n)
{ printf("%d\n",n);
  if (n>1)
    wrt1(n-1);
}
```

- 输出结果将是：n，n-1，.....，2，1。因此在编写递归程序时一定要注意这个递归与操作的时序问题。

```
#include <stdio.h>
void wrt1(int n)
{ if (n>1)
  wrt1(n-1);
  printf("%d\n",n);
}
```

- 注意：上面的程序中的printf语句前没有else，就是说，这个printf语句对于任意n都要执行一次，不同于一般的递归程序中的if语句二择一（不能省略else语句）。

- 编写递归程序的关键是，对于一个问题，要找出其递归关系和初始值。方法之一是利用归纳法，把一个问题归纳总结出递归式，加上初始条件，从而编写出递归函数。
- 对于单变量的递归问题 $f(n)$ ，基本步骤是：
 - (1) 当 $n=1$ 或 0 时，可以得到 $f(n)$ 的值；
 - (2) 假设 x 小于等于 $n-1$ 时，都可以得到 $f(x)$ 的值；
 - (3) 则对于 n ，找出 $f(n)$ 与 $f(n-1)$, $f(n-2)$,的关系式： $f(n)=F(f(n-1), f(n-2),)$;
 - (4) 开始编写递归程序。
- 对于多变量的递归问题 $f(m,n)$ ，基本步骤得视实际情况而定。

【例8.8】上楼梯问题。用递归方法编写函数 $f(n)$ ：一共有 n 个台阶，某人上楼一步可以跨1个台阶，也可以跨2个台阶，问有多少种走法。详细解释思路和算法。

分析：

(1) 当 $n=1$ 时，共1种走法； $f(1)=1$ ；

(2) 当 $n=2$ 时，可以一步一个台阶，也可以一步走完2个台阶，共2种走法； $f(2)=2$ ；

(3) 假设已经知道 $n-1$ 时的走法（当然也知道 $n-2$ 时的走法），那么当 n 时，可以归结为两种情况：

① 一步1个台阶，剩 $n-1$ 个台阶；

② 一步2个台阶，剩 $n-2$ 个台阶。

因此得到递归式： $f(n)=f(n-1)+f(n-2)$

```
#include <stdio.h>
int f(int n)
{   if (n==1)
        return 1;
    else if (n==2)
        return 2;
    else
        return f(n-1)+f(n-2);
}
main( )
{   int m,n;
    scanf("%d", &n);
    printf("T=%d\n", f(n));
}
```

运行结果:

3

T=3

请按任意键继续...

10

T=89

请按任意键继续...

30

T=1346269

请按任意键继续...

40

T=165580141

请按任意键继续...

【例8.9】全组合问题。用递归方法编写函数：从1, 2,,n的n个整数中取k个的全组合（有 C_n^k 种组合）的个数，详细解释思路和算法。

分析：这属于多变量的递归问题 $c(n, k)$

（1）当 $k=1$ 时，即从 n 个数中取1个，很显然，是这 n 个数的每1个，共 n 种取法；

（2）当 $n=k$ 时，即从 k 个数中取 k 个，很显然，是这 k 个数的全部，共1种取法；

（3）假设已经知道如何从 $n-1$ 个数中取 k 个的方法（当然也知道从 $n-1$ 个数中取 $k-1$ 个的方法），那么当从 n 个中取 k 个时，可以归结为两种情况：

① 从前 $n-1$ 个数中取 k 个；

② 先从前 $n-1$ 个数中取 $k-1$ 个，再加上第 n 个数，一共取 k 个。

```
#include <stdio.h>
int combine(int n, int k)
{ if (n > k && k > 1) /* 当n大于k并且k大于1时，递归 */
    return combine(n-1, k-1) + combine(n-1, k);
    /* 递归后，把n个元素取k个的组合问题，变成了n-1个中取k-1
    个、n-1个中取k个的组合问题, 这样递归下去，前一个会变成
    k等于1的问题，后一个会变成n等于k的组合问题*/
    else if (k == 1) /* 若k为1, 则取其中每个元素作为一个组合, 共n种 */
        return n;
    else if (n == k) /* 若n等于k, 则将这n个元素作为一个组合, 共1种 */
        return 1;
}

void main( )
{ int  n,k,c;
    printf("Input n and k:");
    scanf("%d%d", &n, &k);
    c = combine(n, k);
    printf("C(%d,%d)=%d\n", n, k, c);
}
```


运行结果:

Input n and k:5 3

$C(5,3)=10$

请按任意键继续...

Input n and k:36 7

$C(36,7)=8347680$

请按任意键继续...

Input n and k:33 7

$C(33,7)=4272048$

请按任意键继续...

Input n and k:49 6

$C(49,6)=13983816$

请按任意键继续...

```

#include <stdio.h>
int cc=0; /* 统计递归函数被调用的次数 */
int combine(int n, int k)
{
    cc++;
    if (n > k && k > 1) /* 当n大于k并且k大于1时，递归*/
        return combine(n-1, k-1) + combine(n-1, k);
    /* 递归后，把n个元素取k个的组合问题，变成了n-1个中取k-1
       个、n-1个中取k个的组合问题, 这样递归下去，前一个会变成
       k等于1的问题，后一个会变成n等于k的组合问题*/
    else if (k == 1) /* 若k为1, 则取其中每一个元素作为一个组合*/
        return n;
    else if (n==k) /* 若n等于k，则将这n个元素作为一个组合*/
        return 1;
}

void main( )
{
    int n,k,c;
    printf("Input n and k:");
    scanf("%d%d", &n, &k);
    c = combine(n, k);
    printf("C(%d,%d)=%d called=%d\n", n, k, c, cc);
}

```

运行结果:

Input n and k:5 3

$C(5,3)=10$ called=11

请按任意键继续...

Input n and k:33 7

$C(33,7)=4272048$ called=1812383

请按任意键继续...

Input n and k:36 7

$C(36,7)=8347680$ called=3246319

请按任意键继续...

Input n and k:49 6

$C(49,6)=13983816$ called=3424607

请按任意键继续...

【例8.10】上楼梯问题的拓展。用递归方法编写函数 $f(n,m)$ ：一共有 n 个台阶，某人上楼可能一步可以跨1个台阶，也可以跨2个台阶，最多一步跨 m 个台阶，问有多少种不同走法。请详细解释思路和算法。

分析：这属于多变量的递归问题 $f(n, m)$

(1) 当 $n=1$ 或者 $m=1$ 时，共1种走法；

(2) 当 $n < m$ 时，一步最多 n 个台阶，因此 $f(n, m) = f(n, n)$

(3) 当 $n = m$ 时，可以一步走 m 个台阶，共1种走法，也可以不一步走 m 个台阶，有 $f(n, m-1)$ 种走法，因此
 $f(n, m) = f(n, m-1) + 1$

(4) 当 $n > m$ 时，可以一步走1个台阶，有 $f(n-1, m)$ 种走法；也可以一步走2个台阶，有 $f(n-2, m)$ 种走法；
.....直到一步走 m 个台阶，有 $f(n-m, m)$ 种走法。
因此 $f(n, m) = f(n-1, m) + f(n-2, m) + \dots + f(n-m, m)$

```
#include <stdio.h>
int f(int n, int m)
{
    if (n==1 || m==1)
        return 1;
    else if (n<m)
        return f(n, n);
    else if (n==m)
        return f(n, m-1)+1;
    else
    {
        int s=0, i;
        for (i=1; i<=m; i++)
            s += f(n-i, m);
        return s;
    }
}
main( )
{
    int m,n;
    scanf("%d%d", &n, &m);
    printf("T=%d\n", f(n, m));
}
```

运行结果:

3 2

T=3

请按任意键继续...

30 3

T=53798080

请按任意键继续...

10 2

T=89

请按任意键继续...

30 4

T=201061985

请按任意键继续...

【例8.11】 苹果摆放问题。用递归方法编写函数 $f(m,n)$ ：把 m 个同样的苹果放在 n 个同样的盘子里，允许有的盘子空着不放，问共有多少种不同的放法？当 $m=7$ ， $n=3$ 时，视5,1,1和1,5,1为同一种放法。请详细解释思路和算法。

分析：这属于多变量的递归问题 $f(m, n)$ 。不同的摆放方法可以分为两类：至少有一个盘子空着和所有盘子都不空。分别计算这两类摆放方法的数目，然后把它们加起来。

设 $f(m, n)$ 为 m 个苹果 n 个盘子的摆法数，如果 $n > m$ ，必定有 $n-m$ 个盘子要空着，去掉它们对摆法数不产生影响；即 $n > m$ 时， $f(m, n) = f(m, m)$ 。当 $n \leq m$ 时，不同的摆法可以分成两类：即有至少一个盘子空着，相当于 $f(m, n) = f(m, n-1)$ ；或者所有盘子都有苹果，每个盘子至少要先放一个苹果，即 $f(m, n) = f(m-n, n)$ 。苹果的放法总数等于两者的和，即

$$f(m, n) = f(m, n-1) + f(m-n, n)$$

算法:

- (1)当 $m=1$ 或 $n=1$ 时, 只有1种摆法, 因此 $f(m, n)=1$;
- (2)当 $m=n$ 时, 可以分为两种情况: 一是每个盘子放一个, 只有1种摆法; 二是第 n 个盘子不放, 有 $f(m, n-1)$ 种摆法。因此 $f(m, n)=f(m, n-1)+1$;
- (3)当 $n>m$ 时, 必定有 $n-m$ 个盘子要空着。因此 $f(m, n)=f(m, m)$;
- (4)当 $n<m$ 时, 可以分成两类: 一是至少一个盘子空着, 相当于 $f(m, n)=f(m, n-1)$; 二是所有盘子都有苹果, 每个盘子至少要先放一个苹果, 即 $f(m, n)=f(m-n, n)$ 。总的摆法数等于两者的和, 因此 $f(m, n)=f(m, n-1) + f(m-n, n)$

```
#include <stdio.h>
int f(int m,int n)
{
    if(m==1||n==1)
        return 1;
    else if(m==n)
        return f(m,n-1)+1;
    else if(n>m)
        return f(m,m);
    else
        return f(m,n-1) + f(m-n,n);
}
void main( )
{ int m,n,c;
  printf("Input m and n:");
  scanf("%d%d", &m, &n);
  c = f(m, n);
  printf("C=%d\n", c);
}
```

运行结果:

Input m and n:5 3

C=5

请按任意键继续...

Input m and n:10 4

C=23

请按任意键继续...

Input m and n:7 3

C=8

请按任意键继续...

Input m and n:50 10

C=62740

请按任意键继续...

为了防止运行时递归次数过多引起堆栈溢出错误，需
要把编译系统默认的栈的大小（Size）加大。

如何在 Visual Studio 开发环境中设置栈的大小？

打开“项目”菜单中的“属性”对话框。

单击“链接器”文件夹。

单击“系统”属性页。

修改下列任一属性：

堆栈提交大小

堆栈保留大小

方法是：打开“项目”菜单中“xxx属性”（这里xxx
是你的项目名，本题是test2）对话框，找到“配置属
性”中“链接器”中“系统”页：

这里修改“堆栈保留大小”为500000000（500MB）

file1 属性页

配置(C): 活动(Debug)

平台(P): 活动(Win32)

配置管理器(O)...

通用属性

框架和引用

配置属性

常规

调试

C/C++

链接器

常规

输入

清单文件

调试

系统

优化

嵌入的 IDL

高级

命令行

清单工具

XML 文档生成器

浏览信息

生成事件

自定义生成步骤

子系统

控制台(/SUBSYSTEM:CONSOLE)

堆保留大小

0

堆提交大小

0

堆栈保留大小

0

堆栈提交大小

0

启用大地址

默认值

终端服务器

默认值

从 CD 交换运行

否

从网络交换运行

否

驱动程序

未设置

子系统

指定链接器的子系统。 (/SUBSYSTEM:[type])

确定

取消

应用(A)

8.4 算法举例

1. 梯形法求定积分

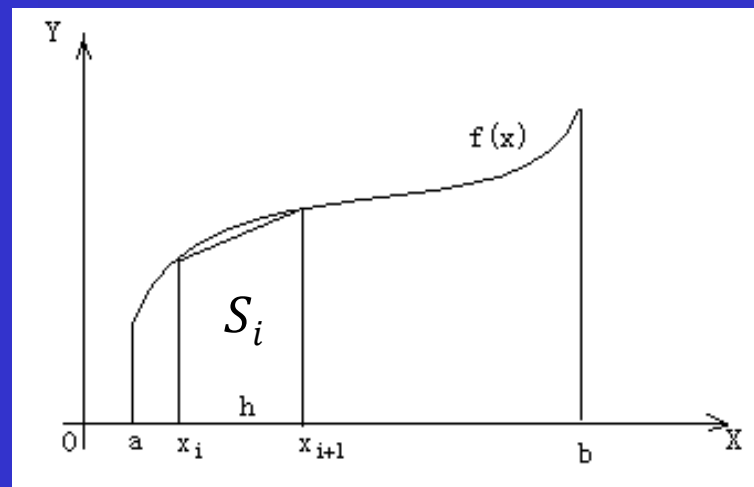
$$S = \int_a^b f(x) dx$$

S 是在区间 $[a, b]$ 内的曲线 $f(x)$ 下的面积，用梯形近似，每一个梯形 S_i ：

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

因此：

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



将梯形法求定积分的功能编写成一个独立的C函数tab(a,b,n), 其中: a为积分的下限, b为积分的上限, n为等分数, 函数返回积分值。

函数程序:

```
#include <stdio.h>
#include <math.h>
double tab(double a, double b, int n)
{   int k;
    double h, x, s, p=0.0, f(double);
    h=(b-a)/n;   /* 步长 */
    s=h*(f(a) + f(b))/2;
    for (k=1; k<n; k++)
    {   x= a+k*h ;
        p += f(x);
    }
    s += p*h;
    return s;
}
```

【例8.12】 用梯形法求积分

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

即 $a=0$, $b=1$, $f(x)=e^{-x^2}$ 。

为了求上面的定积分,只需再编写一个主函数以及计算被积函数值的函数 $f()$ 即可。这两个函数的程序如下:

```
double f(double x)
{ return exp(-x*x);
}
main( )
{ int n; double a=0.0, b=1.0;
  printf("input n: ");
  scanf("%d", &n);
  printf("%16.12f\n", tab(a,b,n));
}
```

将程序和前面的tab函数
一起编译链接，运行结果：

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: 1000

0.746824071499

请按任意键继续...

input n: 5000

0.746824130360

请按任意键继续...

input n: 50000

0.746824132788

请按任意键继续...

input n: 100000000

0.746824132812

请按任意键继续...

input n: 1000000000

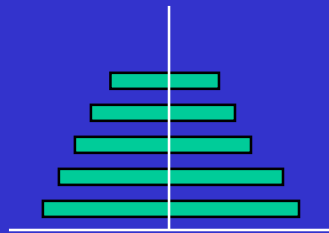
0.746824132812

请按任意键继续...

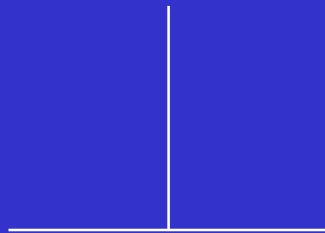
8.4 算法举例

2. Hanoi塔问题

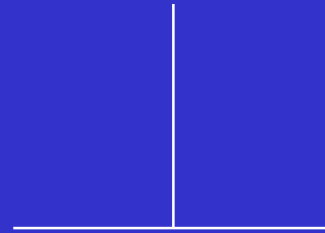
Hanoi塔问题：有 n 个盘子在A处，盘子从大到小，最上面的盘子最小，现在要把这 n 个盘子从A处搬到C处，每次只能搬一个，可以在B处暂存，但任何时候不能出现大的盘子压在小的盘子上面的情况。



A



B

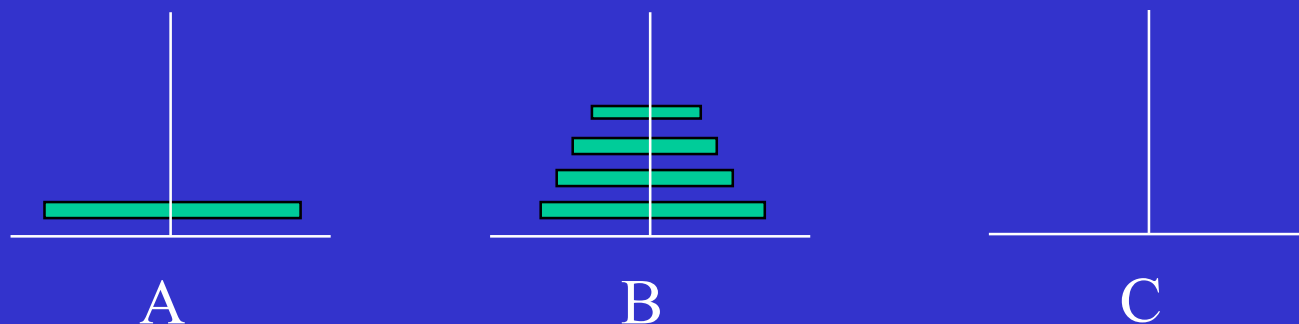


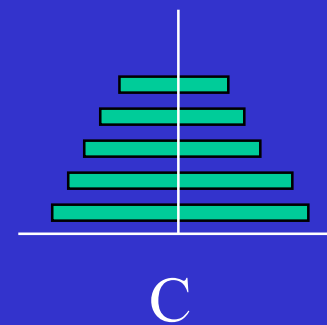
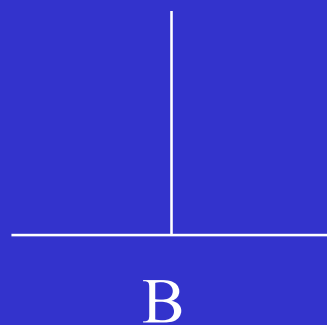
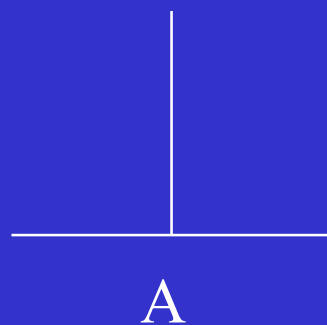
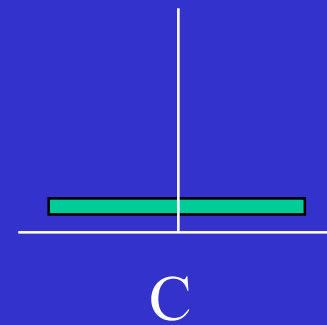
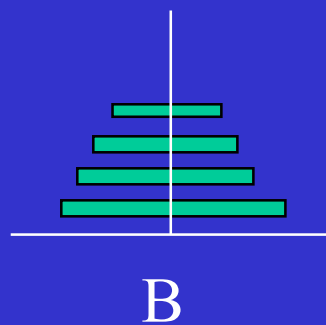
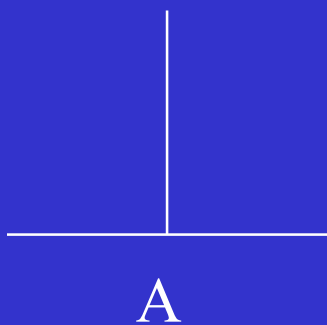
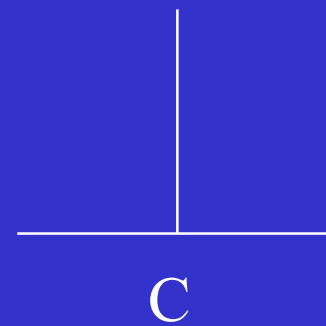
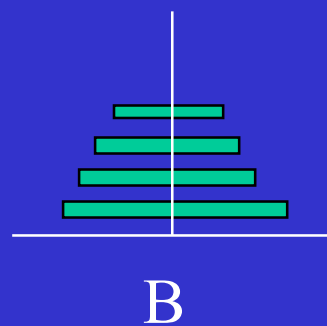
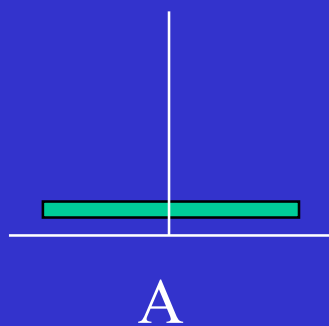
C

● Hanoi塔问题

分析:

- (1) 若 $n=1$, 则可以把盘子直接从A处搬到C处;
- (2) 假设 $n-1$ 时, 知道如何搬;
- (3) 则 n 时, 根据(2)的假设, 可以先把前 $n-1$ 个盘子从A处通过C处暂存搬到B处, 再把第 n 个盘子直接从A处搬到C处, 最后把前 $n-1$ 个盘子从B处通过A处暂存搬到C处, 则完成了全部盘子的搬动。





```

#include <stdio.h>
int Step=1;
void move(int n, char a, char c)
{   printf("Step %2d: Disk %d  %c ---> %c\n", Step, n,a,c);
    Step++;
}
void Hanoi(int n, char a, char b, char c) /* 递归程序 */
{   if (n>1)
    {   Hanoi(n-1, a, c, b); /* 先将前n-1个盘子从a通过c搬到b */
        move(n, a, c);      /* 将第n个盘子从a搬到c */
        Hanoi(n-1, b, a, c); /* 再将前n-1个盘子从b通过a搬到c */
    }
    else move(n, a, c);      /* 将第1个盘子从a搬到c */
}
main( )
{   int n;
    scanf("%d", &n);
    Hanoi(n, 'A', 'B', 'C');
}

```


运行结果:

2

Step 1: Disk 1 A ---> B

Step 2: Disk 2 A ---> C

Step 3: Disk 1 B ---> C

按任意键继续...

3

Step 1: Disk 1 A ---> C

Step 2: Disk 2 A ---> B

Step 3: Disk 1 C ---> B

Step 4: Disk 3 A ---> C

Step 5: Disk 1 B ---> A

Step 6: Disk 2 B ---> C

Step 7: Disk 1 A ---> C

按任意键继续...

4

Step 1: Disk 1 A ---> B

Step 2: Disk 2 A ---> C

Step 3: Disk 1 B ---> C

Step 4: Disk 3 A ---> B

Step 5: Disk 1 C ---> A

Step 6: Disk 2 C ---> B

Step 7: Disk 1 A ---> B

Step 8: Disk 4 A ---> C

Step 9: Disk 1 B ---> C

Step 10: Disk 2 B ---> A

Step 11: Disk 1 C ---> A

Step 12: Disk 3 B ---> C

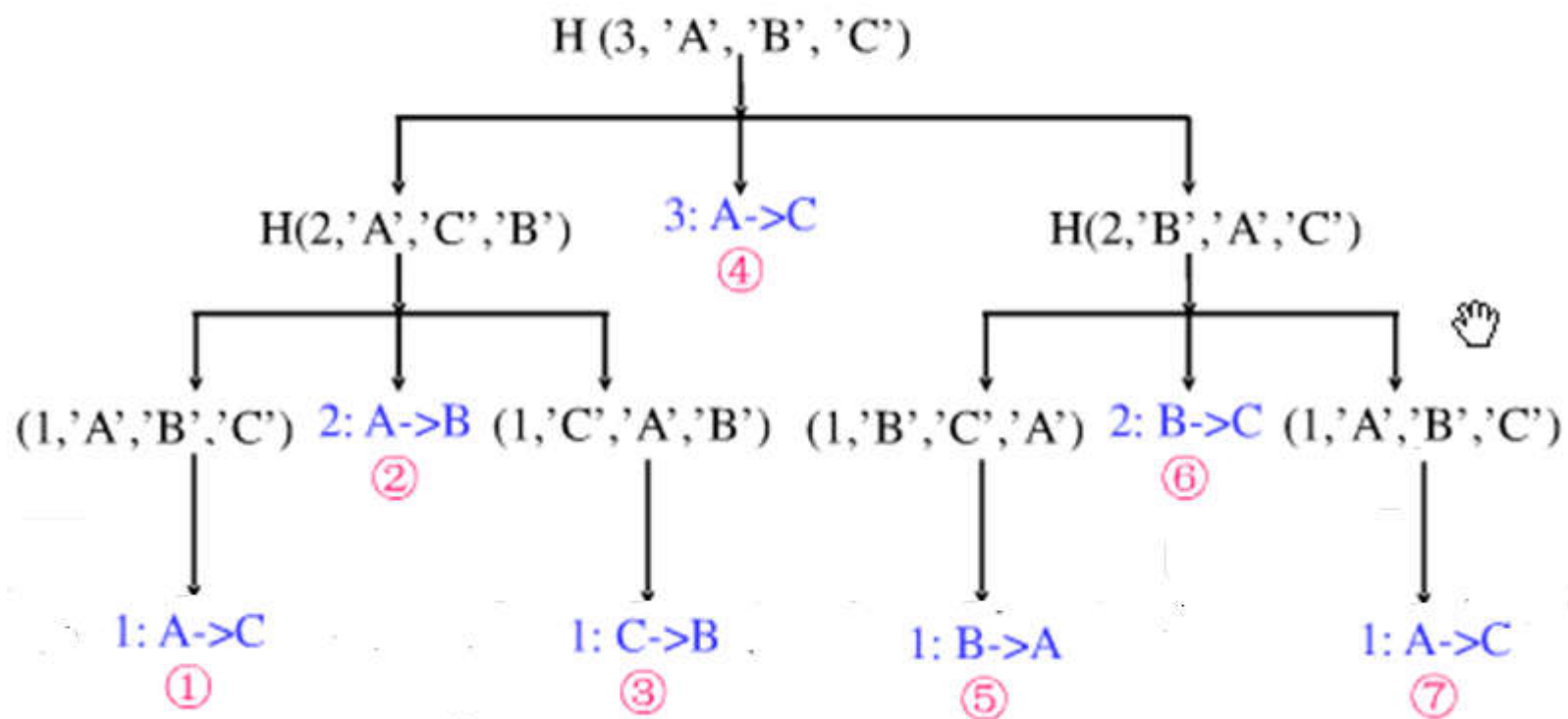
Step 13: Disk 1 A ---> B

Step 14: Disk 2 A ---> C

Step 15: Disk 1 B ---> C

按任意键继续...

执行过程分析: (n=3)



题外话：河内塔(Towers of Hanoi)是法国人M. Claus (Lucas)于1883年从泰国带至法国的，河内为越南的首都，1883年法国数学家 Edouard Lucas曾提及这个故事。

据说创世纪时Benares有一座波罗教塔，是由三支钻石棒（Pag）所支撑，开始时神在第一根棒上放置64个由上至下依由小至大排列的金盘子（Disc），并命令僧侣将所有的金盘子从第一根棒移至第三根棒，且搬运过程中遵守每次只能移动一个、大盘子不能压在小盘子之上的原则，则当金盘子全数搬运完毕之时，此塔将毁损，也是世界末日来临之时。

一共需要搬运多少次？结果分析：

假设n个盘子需要搬动的次数为c：

$$n=1, c=1=2^1-1$$

$$n=2, c=3=2^2-1$$

$$n=3, c=7=2^3-1$$

假定n-1时,需要搬动 $c=2^{n-1}-1$

则n时, $c=2^{n-1}-1 + 1 + 2^{n-1}-1$

$$=2 \times 2^{n-1}-1$$

$$=2^n-1$$

$$n=64 \text{ 时, } c=2^{64}-1=18446744073709551615$$

$$=1.8446744073709551615 \times 10^{19}$$

若每秒搬1次，每年按照365天计算，需要：

$$1.8446744073709551615 \times 10^{19} / (365 \times 24 \times 3600)$$

$$=584942417355 \text{ (年)} \quad \text{大约需要5849亿年!}$$

第7次作业 p.191-192 习题 1, 2, 6, 7, 10, 11

思考探究题：用递归算法解决青蛙过河问题

一条小溪尺寸不大，青蛙可以从左岸跳到右岸，在左岸有一石柱L，面积只容得下一只青蛙落脚，同样右岸也有一石柱R，面积也只容得下一只青蛙落脚。有一队青蛙从尺寸上一个比一个小。我们将青蛙从小到大，用1, 2, ..., n编号。规定初始时这队青蛙只能趴在左岸的石头L上，当然是一个摞一个，小的摞在大的上面。不允许大的摞在小的上面。在小溪中有S个石柱，有y片荷叶，规定溪中的柱子上允许一只青蛙落脚，如有多只同样要求一个摞一个，大的在下，小的在上。对于荷叶只允许一只青蛙落脚，不允许多只在其上。对于右岸的石柱R，与左岸的石柱L一样允许多个青蛙落脚，但须一个摞一个，小的在上，大的在下。当青蛙从左岸的L上跳走后就不允许再跳回来；同样，从左岸L上跳至右岸R，或从溪中荷叶或溪中石柱跳至右岸R上的青蛙也不允许再离开。问在已知溪中有S根石柱和y片荷叶的情况下，最多能跳过多少只青蛙？