C语言

第5讲：函数与程序结构

赵岩

哈尔滨工业大学软件学院

August 20, 2011

目录

目录

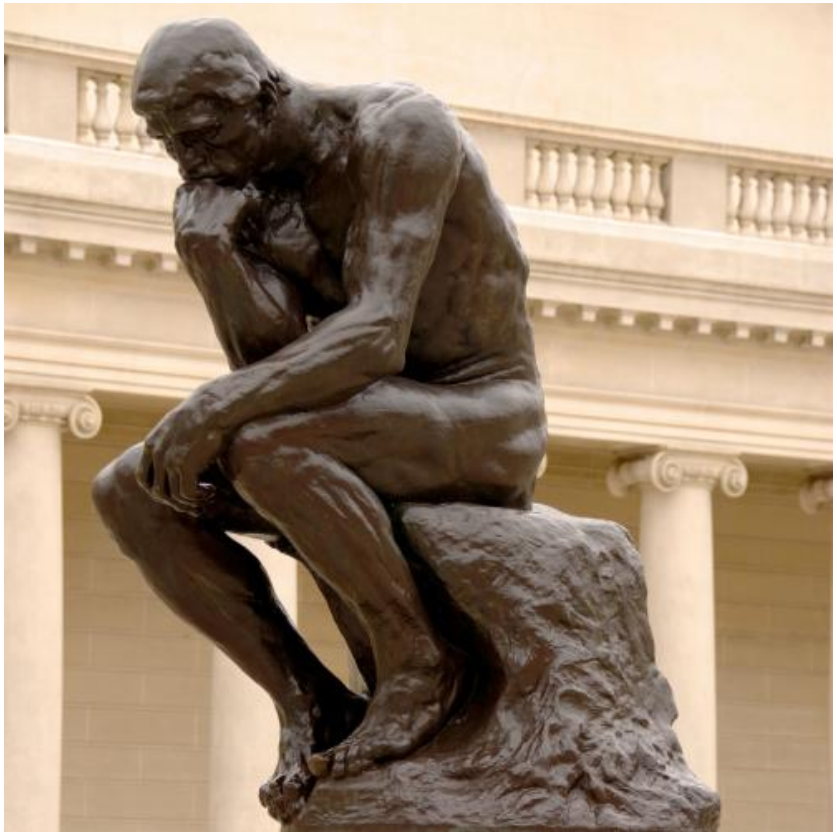
目录

程序设计的艺术

- 如果程序是一个人
  - 算法设计艺术(灵魂)
  - 结构设计艺术(肉体)
  - 语言，只是它的一个外衣

程序设计的艺术

程序设计的艺术



程序设计的结构

- C语言为程序的结构设计提供了两样武器
  - 函数和模块
  - 一个C程序由一个或多个源程序（模块）文件组成
  - 一个源程序文件由一个或多个函数组成
- 函数（function）是结构设计的最基本单位
  - “一个程序应该是轻灵自由的，它的子过程就象串在一根线上的珍珠。”–Geoffrey James的《编程之道》

模块的设计原则

- 模块分解的原则
  - 保证模块的相对独立性
  - 高聚合、低耦合
- 模块的实现细节对外不可见
  - 外部：关心做什么
  - 内部：关心怎么做

模块的设计原则

- 设计好模块**接口**
  - 接口是指罗列出一个模块的所有的与外部打交道的变量等
  - 定义好后不要轻易改动
  - 在模块开头（文件的开头）进行函数声明
- 开发人员各司其职
  - 按模块分配任务，职责明确
  - 并行开发，缩短开发时间

函数间的关系

- 函数生来都是平等和互相独立的，没有高低贵贱之分
- 一个函数并不属于另外一个函数，但是一个函数可以调用另外一个函数
  - main()稍微特殊一点点
  - C程序的执行从main函数开始，调用其他函数后流程回到main函数，在main函数中结束整个程序运行。

函数的分类

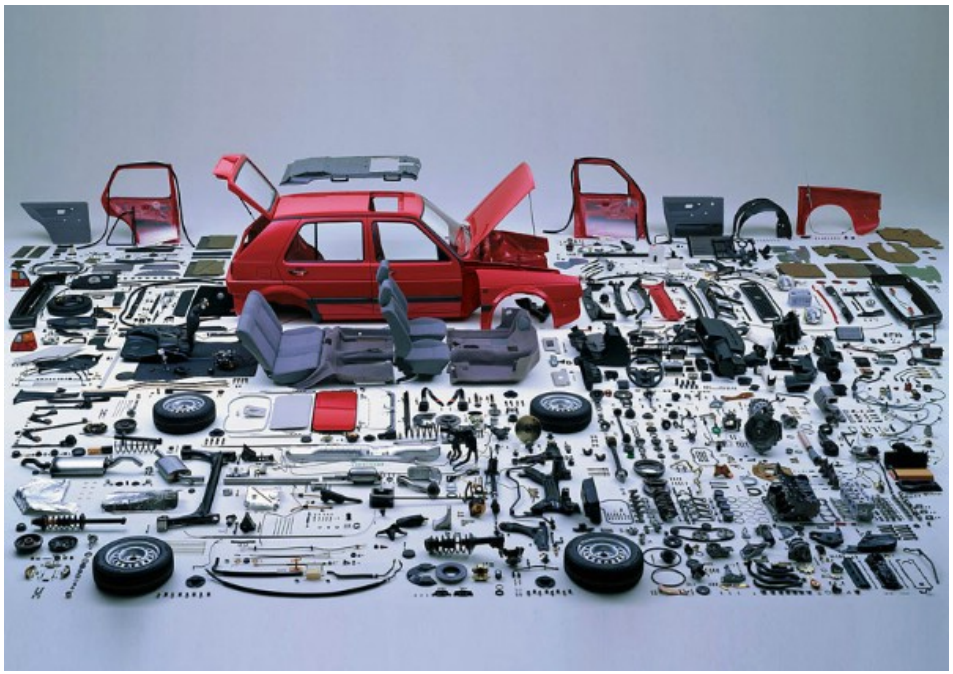
- 库函数
  - ANSI/ISO C定义的标准库函数
    - 如：printf, scanf, sin, cos, ...
    - 符合标准的C语言编译器必须提供这些函数
  - 第三方库函数
    - 由其它厂商自行开发的C语言函数库。不在标准范围内，能扩充C语言的功能（图形、网络、数据库等）
    - Exceptions In C http://adomas.org/excc/
    - Libtc http://libtc.sourceforge.net/
- 自定义函数
  - 自己编写的函数
  - 包装后，也可成为函数库，供别人使用(成为第三方库函数)

函数的好处

- 分而治之
  - 把较大任务分解成若干个较小的任务，并提炼出公用任务
- 复用
  - 可以在其他函数的基础上构造程序，而不需要从头做起
  - 是高质量软件开发的基本方法之一
- 信息隐藏
  - 设计得当的函数可以把具体操作细节对外隐藏，保证其安全

函数的好处

- 如把编程比做制造一台机器，函数就好比其零部件
  - 可将这些“零部件”单独设计、调试、测试好，用时拿出来装配，再总体调试。
  - 这些“零部件”可以是自己设计制造/别人设计制造/的标准产品



不用函数的缺点

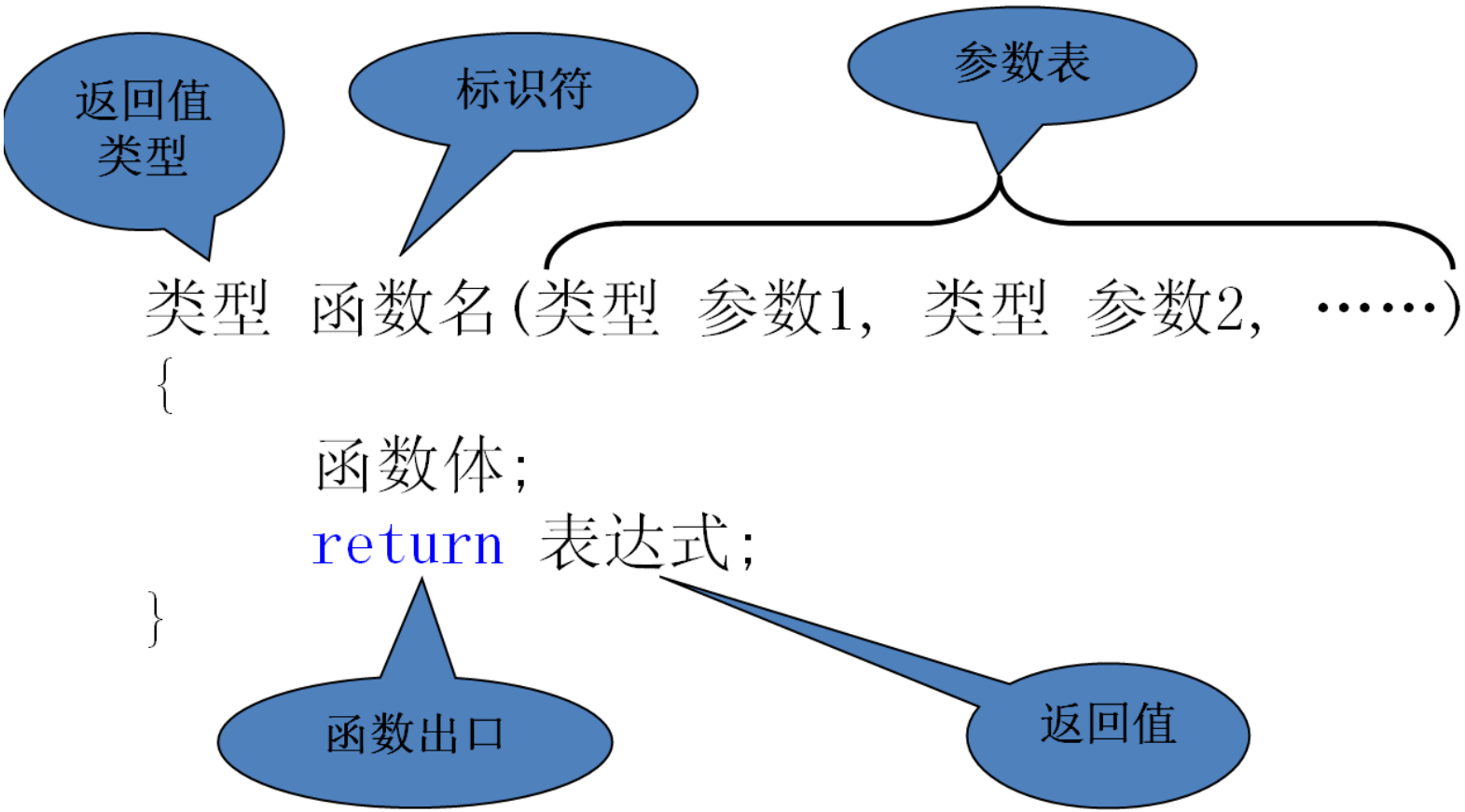
- 计算$\frac{n!}{(n-m)!}$

```
1       int n = 4, m = 2;
2       int i,result;
3       int Pn = 1, Pnm = 1;
4       for(i = 1; i <= n; i++)
5           Pn = Pn * i;
6       for(i = 1; i <= n - m; i++)
7           Pnm = Pnm * i;
8       result = Pn / Pnm;
9       printf("result = %d\n", result);
```

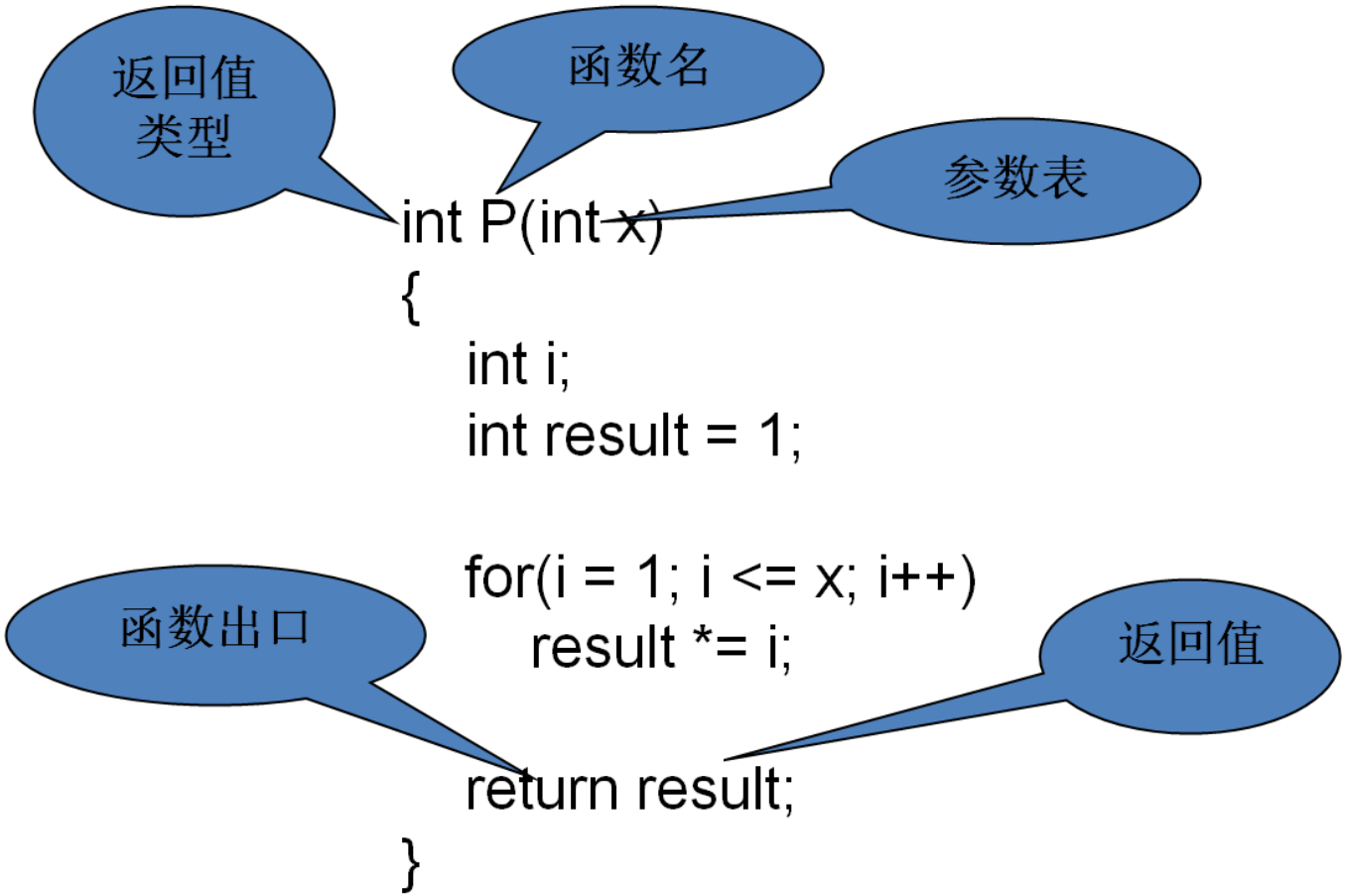
函数的实现

```
1  int P(int x){
2      int i,result=1;
3      for(i=1;i<=x;i++)
4          result*=i;
5      return result;
6  }
7  int main(){
8      int n = 4, m = 2, Pn = 1, Pnm = 1;
9      Pn = P(n);
10     Pnm = P(n-m);
11     printf("result = %d\n", Pn/Pnm);
12     return 0;
13 }
```

函数定义



求x!函数


返回值
类型
函数名
参数表
int P(int x)
{
int i;
int result = 1;
for(i = 1; i <= x; i++)
result *= i;
函数出口
返回值
return result;
}

函数的注释

- 注释的格式

```
1  /* 函数功能：实现××××功能
2       函数参数：参数1，表示×××××
3                 参数2，表示×××××
4       函数返回值：×××××
5  */
6  返回值类型 函数名(参数表)
7  {
8       函数体
9       return 表达式;
10 }
```

用函数名取代函数的注释

- 注释是否需要呢？

```
1  /*
2      函数功能：计算平均数
3      函数入口参数：整型x，存储第一个运算数
4                    整型y，存储第二个运算数
5      函数返回值：平均数
6  */
7  int GetAverage(int x, int y){
8      int result;
9      result = (x + y) / 2;
10     return result;
11 }
```

函数命名

- 在Linux/Unix平台
  - 习惯用function_name
- 本书采用Windows风格函数名命名(驼峰方式)
  - 用大写字母开头、大小写混排的单词组合而成
  - FunctionName
- 变量名形式
  - “名词”或者“形容词+名词”
  - 如变量名oldValue与newValue等
- 函数名形式
  - “动词”或者“动词+名词”（动宾词组）
  - 如函数名GetMax()等

函数的参数

- 如果没有函数参数，则应该用void注明
- 形参(形式参数):
  - 在定义函数时，函数名后面括号中的变量名
- 实参(实际参数):
  - 在主调函数中调用一个函数，调用函数名后面括号中的参数(或表达式)

形参和实参

```
1  int GetAverage(int x, int y)
2  {
3      int result = (x + y) / 2;
4      return result;
5  };
6  main()
7  {
8      int i =2,j = 4;
9      GetAverage(i,j);
10     GetAverage(2,4);
11 }
```

函数返回值

- 不需要返回值
  - 则应该用void定义返回值类型
  - 返回值为void的函数，可以不用return返回
- 需要返回值
  - 返回值类型与return语句配合
  - 能作为表达式的一部分
  - **d=max(max(a,b),c);**
- 当函数执行到return语句时，就中止函数的执行，返回到调用它的地方

函数声明

- 调用一个函数之前，先要对其返回值类型、函数名和参数进行声明（declare）
  - 不对函数进行声明是非常危险的
  - 函数定义也有声明函数的效果
- 声明时不要省略参数以及返回值的类型
- 函数声明的若干位置
  - 函数定义在使用前
  - 函数声明在使用前
  - 函数声明在头文件，使用时包含同文件

函数声明位置1-函数定义在使用前

```
1  int GetAverage(int x, int y)
2  {
3      int result = (x + y) / 2;
4      return result;
5  };
6  main()
7  {
8      int i =2,j = 4;
9      GetAverage(i,j);
10 }
```

函数声明位置2-函数声明在使用前

```
1  int GetAverage(int x, int y);
2  main()
3  {
4      int i =2,j = 4;
5      GetAverage(i,j);
6  }
7
8  int GetAverage(int x, int y)
9  {
10     int result = (x + y) / 2;
11     return result;
12 }
```

函数声明位置3-函数声明在头文件中。

```
1  #include "yanmath.h"
2  main()
3  {
4      int i =2,j = 4;
5      GetAverage(i,j);
6  }
7  int GetAverage(int x, int y)
8  {
9      return (x + y) / 2;
10 }
```

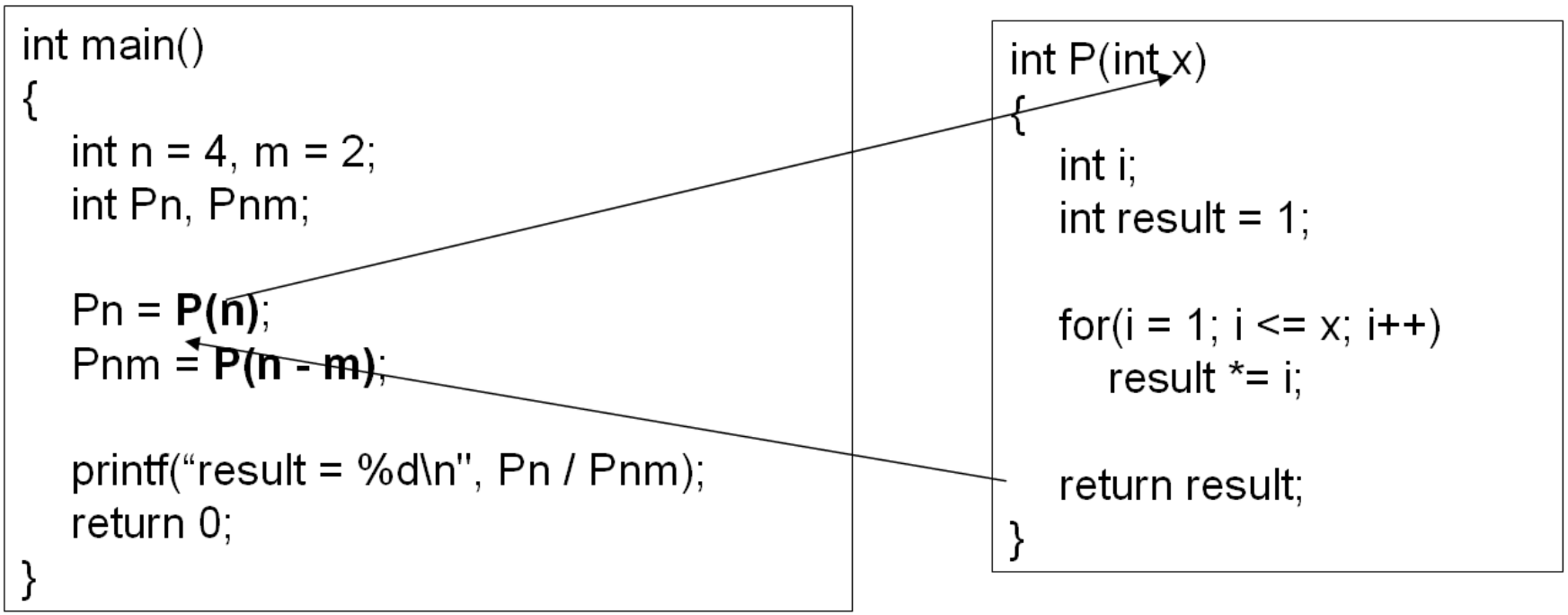
问题

*printf*函数定义在什么地方？

函数的调用

- 单向值传递
- 调用函数时提供的表达式（实参）和该函数的形参必须匹配
  - 数目一致
  - 类型一一对应（否则会发生自动类型转换）
  - 表达式的值赋值给对应的参数

函数调用示意

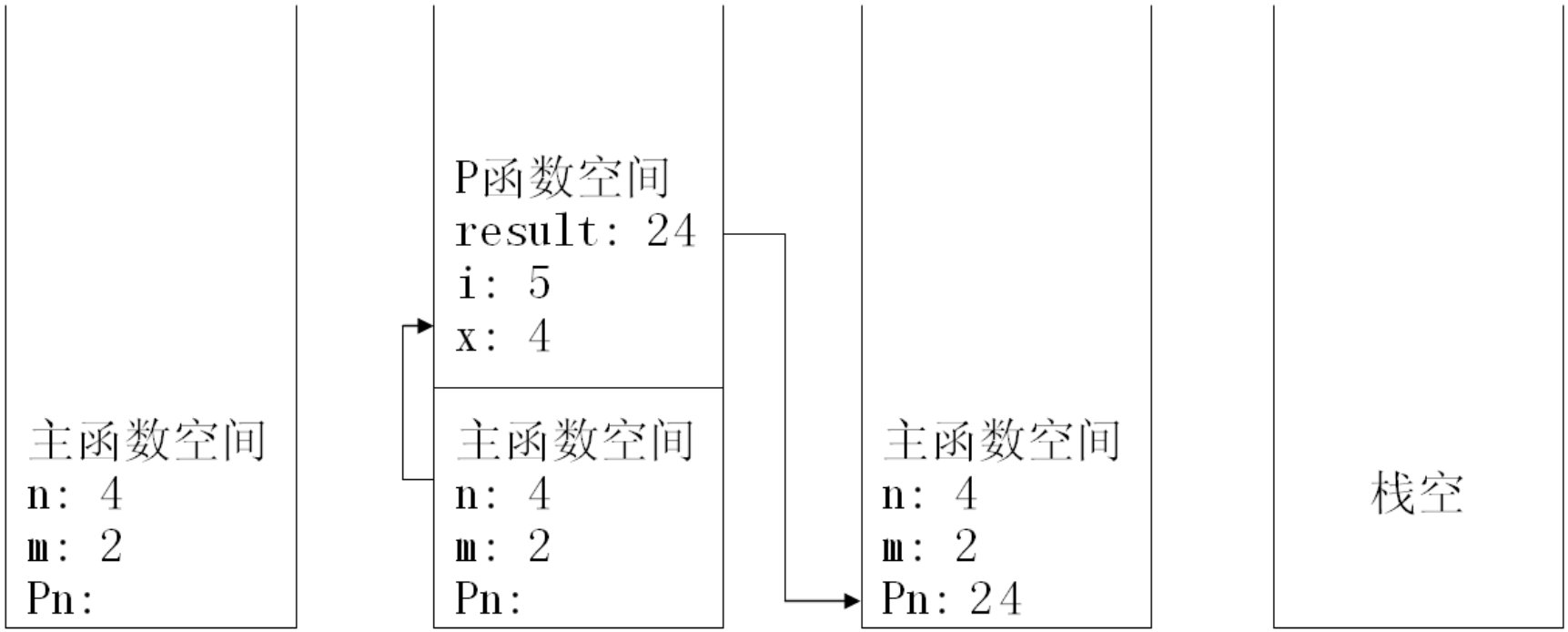

int main()
{
int n = 4, m = 2;
int Pn, Pnm;
Pn = P(n);
Pnm = P(n - m);
printf("result = %d\n", Pn / Pnm);
return 0;
}
int P(int x)
{
int i;
int result = 1;
for(i = 1; i <= x; i++)
result *= i;
return result;
}

函数与栈

Demo 1: stack and function

```
1  int P(int x){
2      int i, result = 1;
3      for(i = 1;i<=x;i++){
4          result*=i;
5      }
6      return result;
7  }
8  int main(){
9      int n = 4, m = 2;
10     int Pn;
11     Pn = P(n);
12     return 0;
13 }
```

函数调用栈



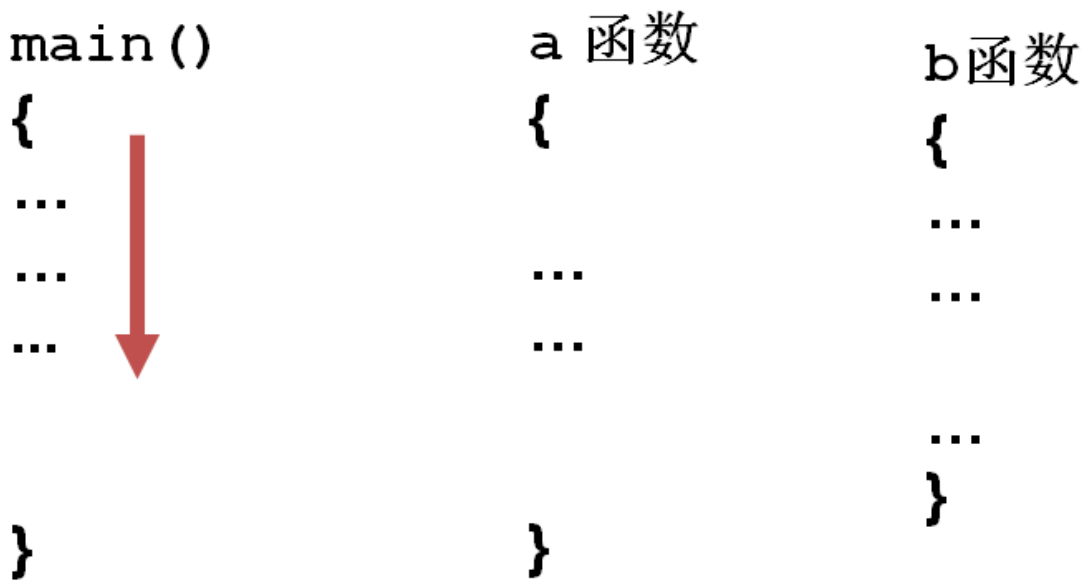
1. 主函数被调用
2. P函数被调用
3. P函数调用结束，返回值传递给Pn
4. 主函数结束

使用函数注意事项

- 每个函数只完成一个功能（包括main()）
  - 对函数的功能可以用不含连词的一句话描述
- 函数不能过长
  - 1986年IBM在OS/360的研究结果：大多数有错误的函数都大于500行
  - 1991年对148,000行代码的研究表明：小于143行的函数比更长的函数更容易维护

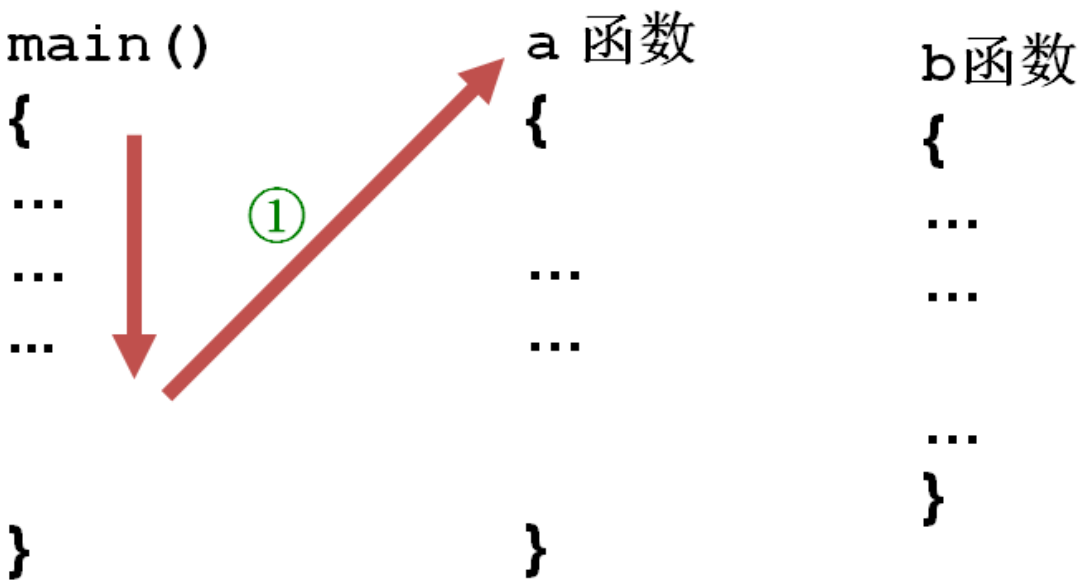
函数的嵌套调用

- 在被调函数中，又调用了函数–嵌套调用



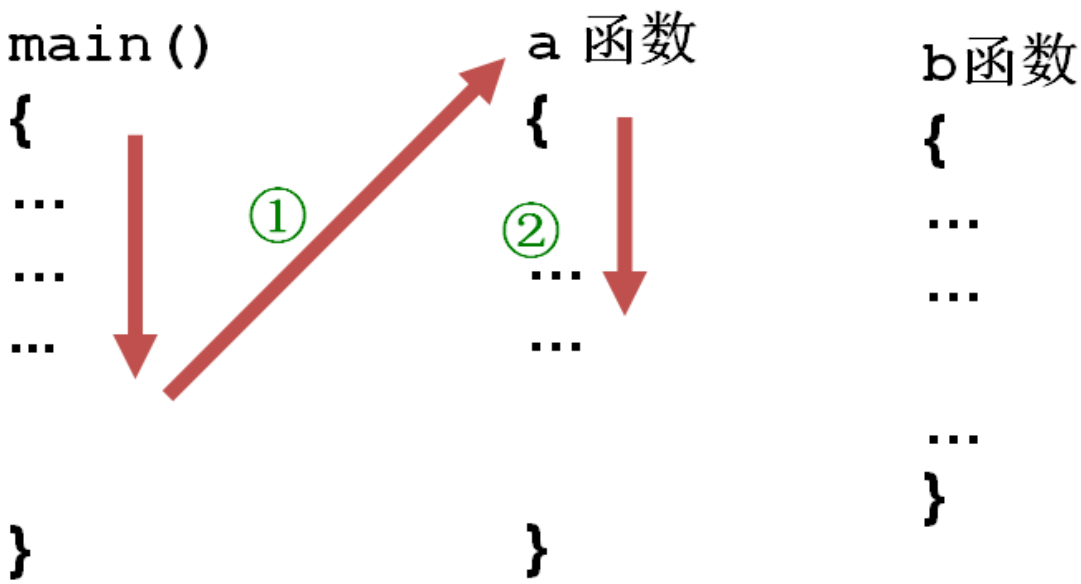
函数的嵌套调用

- 在被调函数中，又调用了函数–嵌套调用



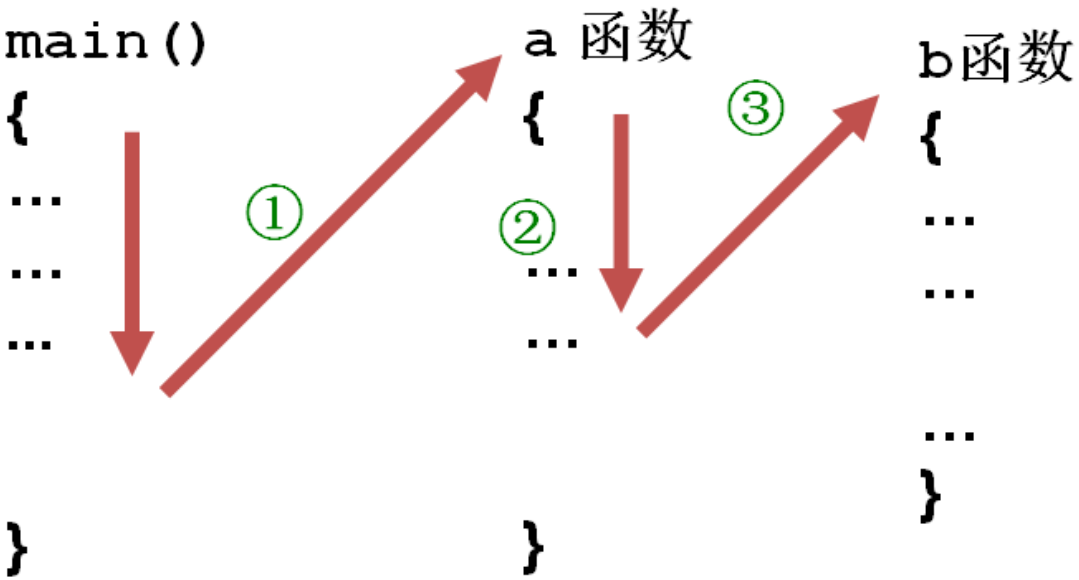
函数的嵌套调用

- 在被调函数中，又调用了函数–嵌套调用



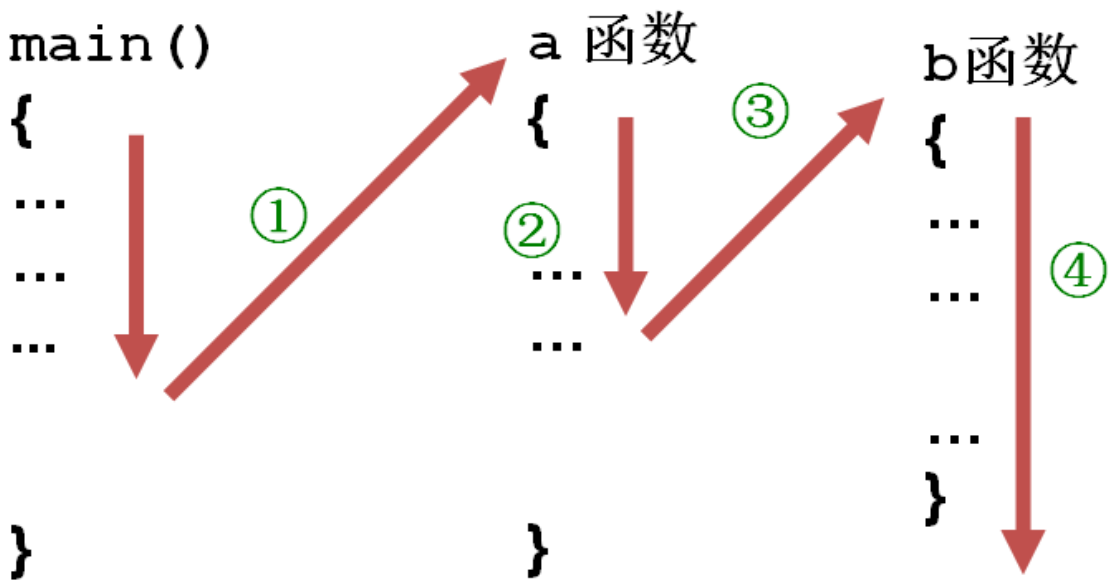
函数的嵌套调用

- 在被调函数中，又调用了函数–嵌套调用



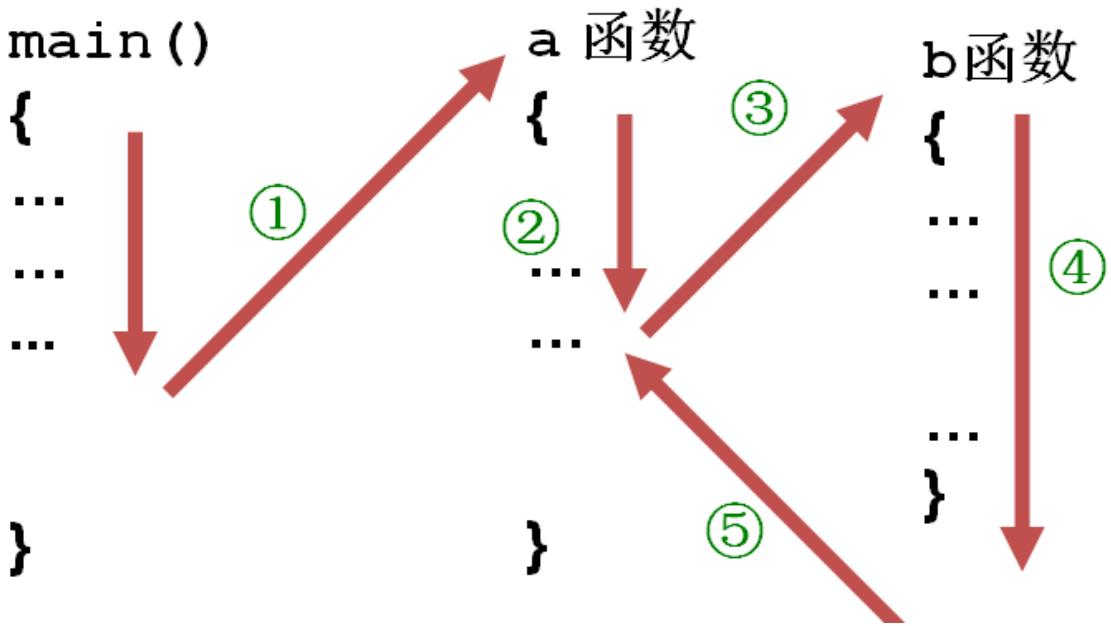
函数的嵌套调用

- 在被调函数中，又调用了函数–嵌套调用



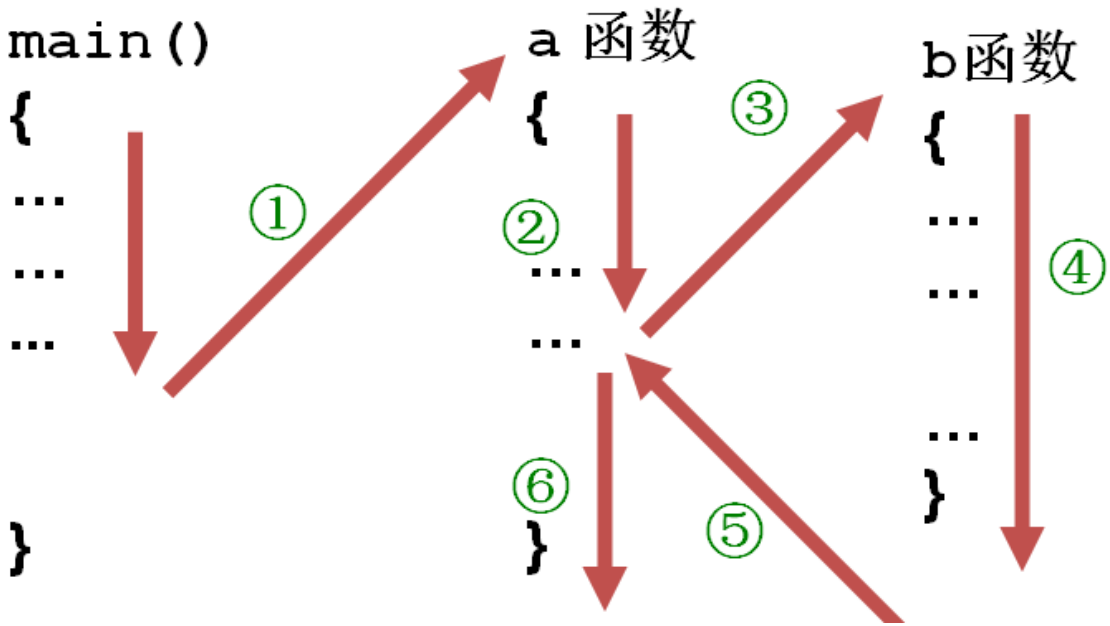
函数的嵌套调用

- 在被调函数中，又调用了函数–嵌套调用



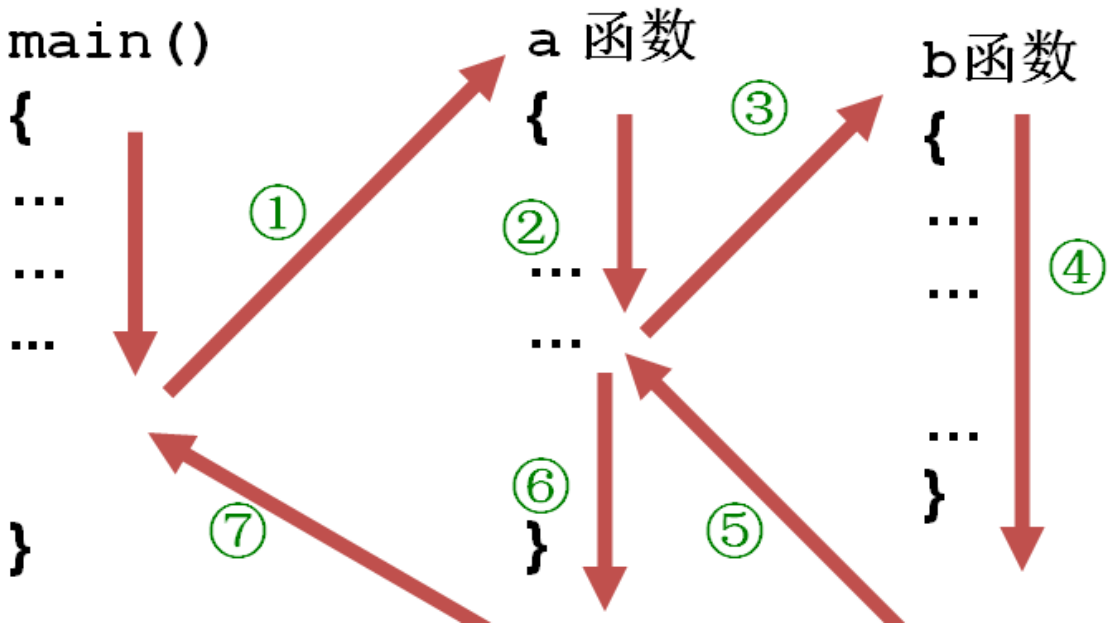
函数的嵌套调用

- 在被调函数中，又调用了函数–嵌套调用



函数的嵌套调用

- 在被调函数中，又调用了函数–嵌套调用



递归的定义

- 函数直接或间接调用自己为递归
- 缺点
  - 递归太多层易导致栈空间溢出
- 优点
  - 简单的思路解决复杂的问题

```
1 unsigned long func(unsigned int n)
2 {
3     if (n == 0)
4         return 1;
5     else
6         return n * func(n-1);
7 }
```

汉诺塔源码

Demo 2: Hanoi source code

```
1  void hanoi(int n,char A,char B,char C)
2  {
3   if(n==1){
4    printf("Move disk %d from %c to %c\n",n,A,C);
5   }
6   else
7   {
8    hanoi(n-1,A,C,B);
9    printf("Move disk %d from %c to %c\n",n,A,C);
10   hanoi(n-1,B,A,C);
11  }
12 }
```

存储类型

- 一个完整的变量说明格式如下：
  - 存储类型　数据类型　变量名
  - static int x , y;
- C程序的存储类型有：
  - auto型（自动变量型）
  - static型（静态变量型）
  - register型（寄存器型）

auto自动变量

- “自动”体现在
  - 缺省的存储类型
  - 进入语句块时**自动申请**内存，退出时**自动释放**内存
- 标准定义格式
  - auto 类型名　变量名;
  - 不初始化时，值是不确定的

自动变量例子

Demo 3: auto variable example

```
1  void Func(void){
2   int times = 1;
3   printf("Func() was called
4               %d times.\n", times++);
5  }
6  main()
7  {
8       int i;
9       for (i=0; i<10; i++){
10          Func();
11      }
12 }
```

寄存器变量

- 寄存器是CPU内的高速但容量有限的存储器
- 使用频率比较高的变量声明为register ，可以使程序更小、执行速度更快
  - register 类型名　变量名;
  - register int i;
- 它只是一个建议
  - 编译器有最终权利决定变量是否是register
  - 编译器的决定往往更科学，不需要我们人为制定。

静态变量

- 静态变量自动初始化为0
- 变量定义为static，则变量一直保持，不会伴随函数调用时的堆栈消失

Demo 4: static variable example

```
1 main()
2 {
3       static int i;
4       printf("%d",i); /* output 0 */
5 }
```

静态变量例子

Demo 5: static variable example

```
1  void Func(void)
2  {
3    static int times = 1;   /*静态局部变量*/
4    printf("Func() was called
5                %d times.\n", times++);
6  }
7  main()
8  {
9       for (int i=0; i<10; i++){
10           Func();
11      }
12 }
```

作用域

- 源程序中定义变量的位置及其能被读写访问的范围
  - 局部变量(Local Variable)
  - 全局变量(Global Variable)

局部变量

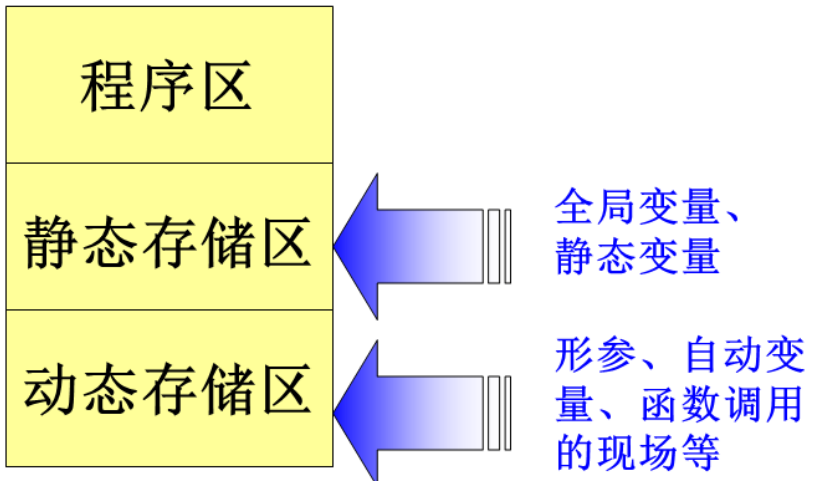
- 局部变量
  - 函数内部定义的变量
  - 在语句块内定义的变量
  - 形参也是局部变量
- 特点
  - 定义时不会自动初始化，除非程序员指定初值
  - 进入语句块时获得内存，仅能由语句块内语句访问，退出语句块时释放内存，不再有效
  - 并列语句块各自定义的同名变量互不干扰

全局变量

- 全局变量
  - 在所有函数之外定义的变量
  - 从程序运行起即占据内存，程序运行过程中可随时访问，程序退出时释放内存
- 特点
  - 在程序中定义它的位置以后都有效
  - 在定义点之前或在其他文件中引用，应该进行如下声明：extern 类型名　变量名;
  - 使函数之间的数据交换更容易，也更高效
  - 但是并不推荐使用，因为谁都可以改写全局变量，封装性差。

存储类型总结

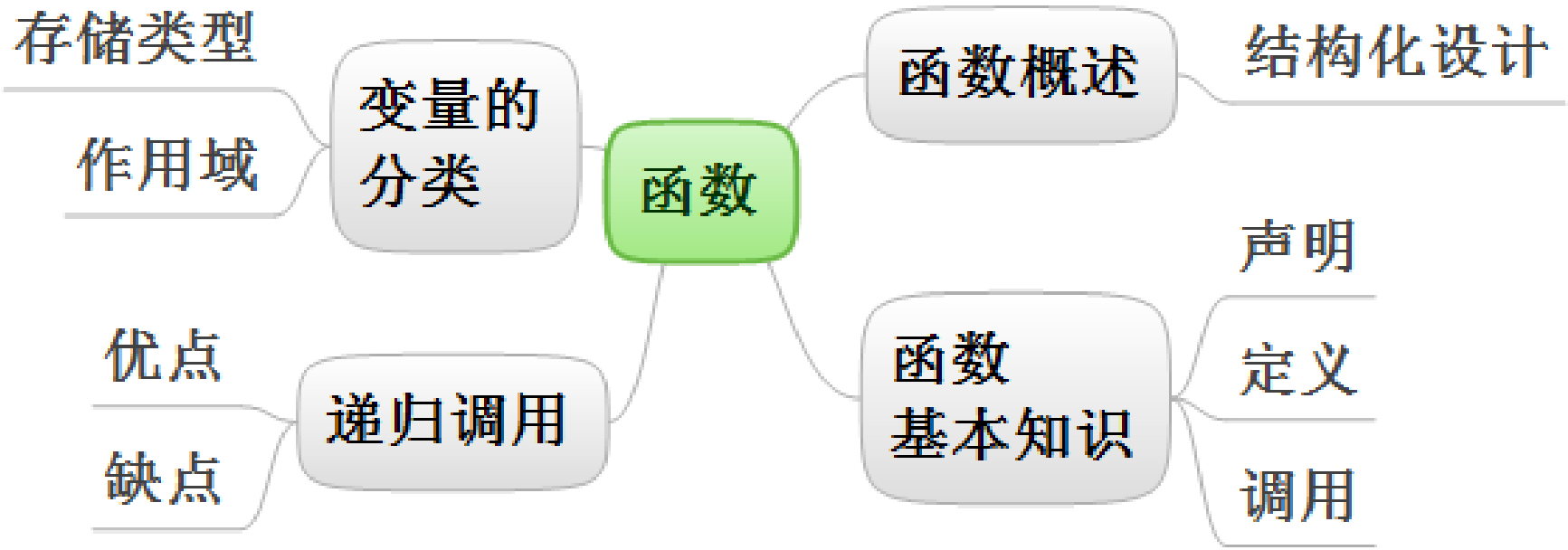
- 静态存储区中的变量
  - 与程序“共存亡”
- 动态存储区中的变量
  - 与语句块“共存亡”
- 寄存器中的变量
  - 同动态存储区



assert

- 名唤：断言
- assert(int expression)
  - expression为真，无声无息；
  - expression为假，中断程序。
- 用来测试某不可能发生的状况确实不会发生
  - Debug版有效
  - Release版失效
- 只为调试程序用，不可作为程序的功能

总结



谢谢大家，欢迎提问！