关于学习习惯

- 关于scanf的问题
  – 旧事再重提
- “上课没所谓，反正我自学”
  – 动手课的“两面性”（技巧与思想）
  – “参考书”就是拿来参考的
  – 搜索引擎也是拿来参考的

Introduction to C Programming

Jichang Zhao

jichang@buaa.edu.cn

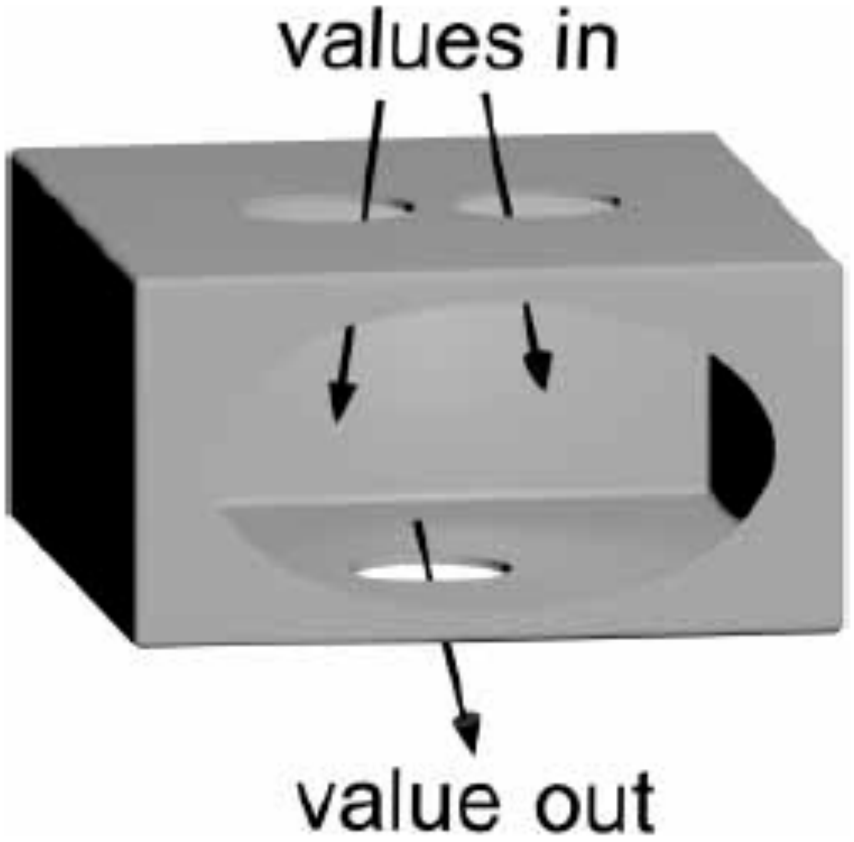
Using Functions (Part II)

Objectives

- Variable Scope
- Variable Storage Class
- Nested Function Calling
- Recursion
- 本章参考教材的错误比较多（混用局部变量和全局变量）

Start from Structural Programming

- Isolation



Variable Scope

- If variables created inside a function are available only to the function itself, they are said to be **local** to the function, or **local variables**
- **Scope** is the section of the program where the variable is valid or “known”
  - 注意：从变量声明之后的行算起

Variable Scope

- A variable with a **local scope** has had storage set aside for it by a declaration statement made within a function body
  - 类型决定大小
- A variable with **global scope** is one whose storage has been created for it by a declaration statement **located outside any function**
  - termed as **global variable**

Varaible Scope

- #include<stdio.h>
- int firstnum;//global variable
- void valfun();
- int main(void)
- {
  - int secnum;//local variable in main
  - firstnum=10;//can be known to main
  - ….
- }
- void valfun()
- {
  - int secnum;//local variable in valfun
  - firstnum=20;//can be known to valfun
- }

Variable Scope

- If a variable that is not local to the function is used by the function, the program **searches the global storage areas for the correct name**
- The scope of a variable **does not influence** the data type of the variable
- **Demo：display.c**

When to Use Global Declarations

- The scoping rules for **symbolic constants and function prototypes** are the same as for variables
- When a symbolic constant has a general meaning that is applicable throughout an application, **it makes good programming sense to declare it globally at the top of a source code file**
- **Coding a function prototype as a global makes sense when the function is used by a number of other functions in a source code file**
  - Doing so avoids repeating the prototype within each of the functions that will call it

Misuse of Global Variables

- Except for symbolic constants and prototypes, global variables should **almost never be used**
  - **注意：要确保给全局变量起有完整意义的名字**
- **By making a variable global, you instantly destroy the safeguards C provides to make functions independent and insulated from each other**
- Using global variables can be especially **disastrous** in large programs with many user-created functions
  - Because a global variable can be accessed and changed by any function following the global declaration， **it is a time-consuming and frustrating task to locate the origin of an erroneous value**

Variable Storage Class

- In addition to the space dimension represented by its scope, variables also have a time dimension
  - Called the variable’s “**lifetime**”
- **Where and how long a variable’s storage locations are kept before they are released** can be determined by the **storage class** of the variable
  - `auto`, `static`, `extern`, and `register`

Local Variable Storage Classes

- Local variables can only be members of the `auto`, `static`, or `register` storage classes
  - `auto` is the default class used by C
- The term auto is short for **automatic**
- Storage for automatic local variables is **automatically reserved** each time a function declaring automatic variables is called
- As long as the function has not returned control to its calling function, all automatic variables local to the function are "**alive**"; that is, storage for the variables is available

Local Variable Storage Classes

- A local variable declared as static causes the program **to keep the variable and its value even when the function that declared it is done**
  - **Once created, local static variables remain in existence for the life of the program**
- **Static variables are not initialized at run-time**
  - The initialization of static variables is done only once, when the program is first compiled
  - Some compilers initialize local static variables the first time the definition statement is executed rather than when the program is compiled

Local Variable Storage Classes

- Register variables have the same time duration as automatic variables
  - `register int time;`
- Registers are high-speed storage areas physically located in the computer’s processing unit
- Application programs rarely, if ever, should use register variables
- Variables declared with the register storage class are automatically switched to auto if the compiler does not support register variables or if the declared register variables exceed the computer’s register capacity
  - **No & operation**

Global Variable Storage Classes

- Global variables are created by declaration statements external to a function
  - **They exist until the program in which they are declared is finished executing**
- Global variables are declared `static` or `extern`
  - `extern int sum;`
  - `static float yield;`
  - `float interest;`
- The purpose of the `extern` storage class is to **extend the scope of a global variable declared in one source code file into another source code file**

补充：全局变量的存储类

- 全局变量的存储类
  - 全局变量不能够被声明为在程序正在执行时创建和销毁的auto或register存储类
  - 全局变量可能声明为static或extern，但不能同时声明为这两种
  - `extern int sum;`
  - `extern float price;`
  - `static float yield;`
  - 全局static和extern类既影响变量的作用域，又影响其生命周期
  - 与静态局部变量类似，所有静态全局变量在编译时会被初始化为0（如果程序员没有明确初始化）

补充：extern示例

```
…
int price;
float yield;
static float coupon;
…
int main()
{
…
}
extern float interest;
int func1()
{
…
}
int func2()
{
…
}
```

```
…
float interest;
extern int price;
…
int main()
{
…
}
int func3()
{
extern float yield;
…
}
int func4()
{
…
}
```

补充：extern示例

- 明确包含extern的声明语句不同于其他声明语句
  - 并非通过保留一个新的存储区域而导致一个新变量的创建
  - 仅通知编译器该变量已存在，可以使用
- extern变量的初始化由最初的全局变量的声明完成(多次声明，一次定义)
- extern声明时进行初始化是不建议的(可能产生编译错误或warning)

```
#include<stdio.h>//test.c
int firstnum;
static int secnum;
extern int thirdnum=3; // warning…
int main()
{
        printf("fn=%d\n",firstnum);
        printf("sn=%d\n",secnum);
        printf("tn=%d\n",thirdnum);
        return 0;
}
```

```
int thirdnum; // test2.c
```

补充：链接（第三个维度）

- 链接
  - 为链接器服务
  - 变量的链接确定了程序的不同部分可以共享此变量的范围
  - 具有外部链接的变量可以被程序中的几个（或全部）文件共享
  - 具有内部链接的变量只能属于单独一个文件
  - 无链接的变量属于单独一个函数，而且根本不能被共享
- 示例
  - `int i;  //`静态存储期限，文件作用域，外部链接
  - `void f(void)`
  - `{`
    - `int j;  //`自动存储期限，块作用域，无链接
  - `}`

补充：链接

- static int i;//静态存储期限，文件作用域，内部链接
- void f(void)
- {
  - static int j;//静态存储期限，块作用域，无链接
- }

补充：存储类规则

- 自动变量和寄存器变量总是局部变量
- 局部变量的作用域也称之为块作用域
- 形式参数具有与局部变量一样的性质
  - 区别在于调用的时候通过实参自动初始化
- 全局变量（外部变量）是静态存储类型，且其作用域为文件作用域
- extern对外部链接变量作用域的扩展（始终具有静态存储期限）
  - 到另一个文件
  - 到另一个文件内的函数（块作用域）
- 除静态变量（局部或全局）外，所有变量在每次进入作用域时均被初始化
- 静态局部变量的最后值在下一次函数调用时是可获得的
  - 可以用于函数执行次数的计数
  - Demo slv.c
- 参考教材p263 编程注解翻译有误

补充：函数的存储类型

- 仅包括extern和static
  - extern：外部链接，允许其他文件调用此函数
  - static：内部链接，只能在定义函数的文件内部调用此函数
  - 如果不指明存储类型，默认为外部链接
- 当声明不打算被其他文件调用的函数时，建议使用static
  - 更容易维护
    - 文件之外不可见
  - 减少了“名字空间污染”
    - 内部链接，其他文件中可以重新使用这些名字
  - Demo

Nested Function Calling

- The definition of function cannot be nested
- **However, function calling can be nested**
  - We use printf() function extensively in functions

Recursion

- Functions that call themselves are referred to as **self-referential** or **recursive** functions
- When a function invokes itself, the process is called **direct recursion**
- A function can invoke a second function, which in turn invokes the first function; this type of recursion is referred to as **indirect** or **mutual recursion**

Mathematical Recursion

- The definition for n! can be summarized by the following statements
  - `0! = 1`
  - `n! = n * (n-1)! for n >= 1`
- This definition illustrates the general considerations that must be specified in constructing a recursive algorithm
  1. What is the first case or cases?
  2. How is the *n*th case related to the *(n-1)* case?

Mathematical Recursion

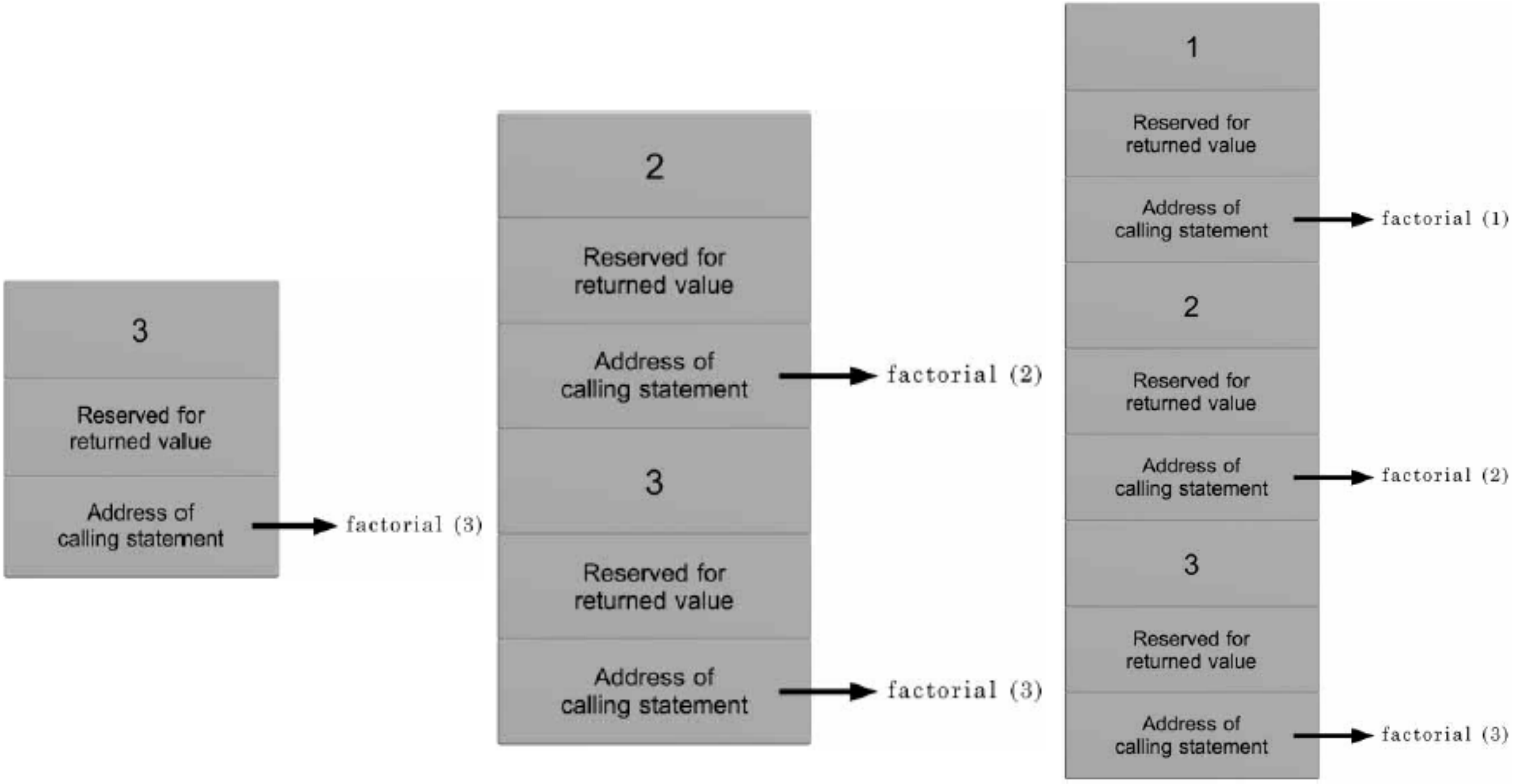
- In pseudocode, the processing required is

```
If n = 0
    factorial = 1
Else
    Factorial = n * factorial(n - 1)
```

- In C, this can be written as

```
int factorial(int n)
{
   if (n == 0)
      return (1);
//这一直接的结果返回是必须的，否则递归无法中止
   else
      return (n * factorial(n-1));
}
```

How the Computation is Performed

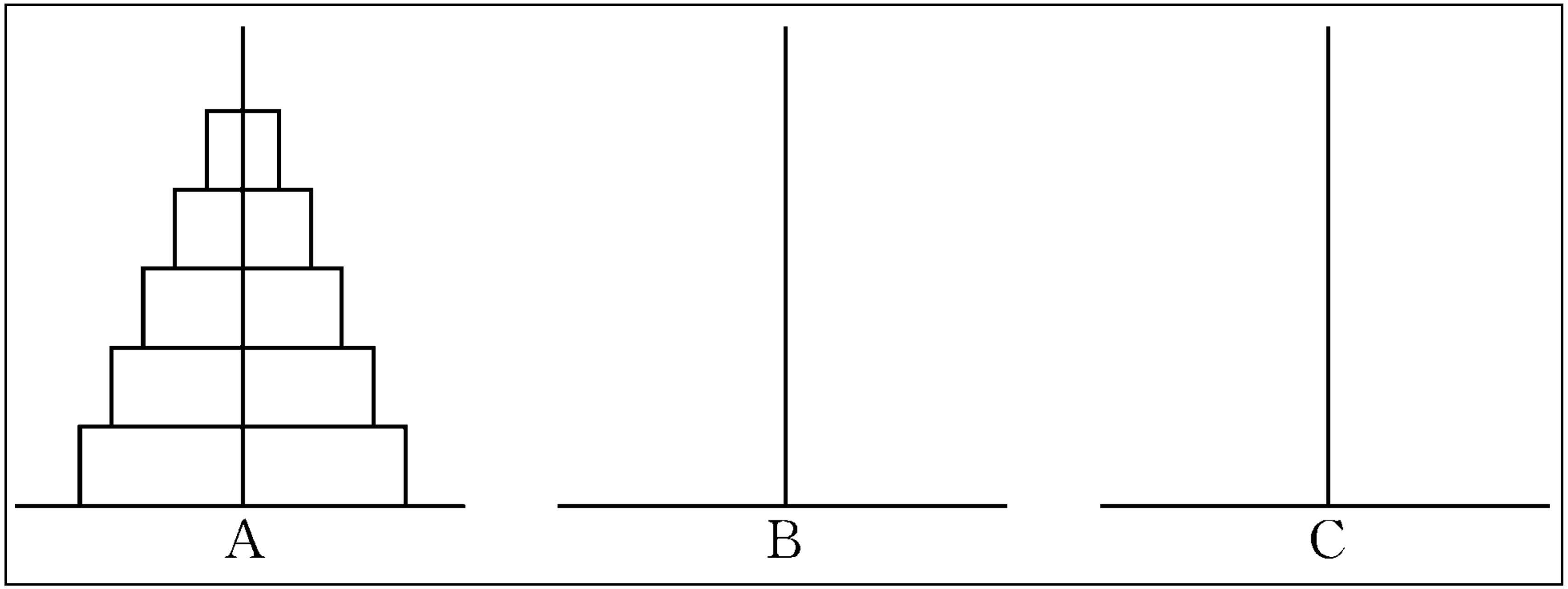

3
Reserved for returned value
Address of calling statement
factorial (3)
2
Reserved for returned value
Address of calling statement
factorial (2)
3
Reserved for returned value
Address of calling statement
factorial (3)
1
Reserved for returned value
Address of calling statement
factorial (1)
2
Reserved for returned value
Address of calling statement
factorial (2)
3
Reserved for returned value
Address of calling statement
factorial (3)

Recursion versus Iteration

- The recursive method can be applied to any problem in which the solution is represented in terms of solutions to simpler versions of the same problem
    - 数学中的迭代求值
- Any recursive function can be written in a nonrecursive manner using an iterative solution
    – If a problem solution can be expressed repetitively or recursively with equal ease, **the repetitive solution is preferable because it executes faster and uses less memory**

```
int factorial(int n)
{
   int fact;
   for(fact = 1; n > 0; n--)
      fact = fact * n;
   return (fact);
} //DEMO fact.c 比较循环与递归
```

Case Study: Hanoi



- Move 5 hanoi stacks from A to C.
- Smaller stack always on top.
- Move one stack each time.
- Print the steps.

Homework

1. 尝试用递归倒序输出一个正整数
2. P285，1（尝试比较一下循环和递归的效率）
3. P285，3
4. P286，6
5. P286，7(题目表述奇怪，本意就是实现两个函数，一个递归，一个循环，来进行回文数字的判读)
6. 《现代方法》(第2版）p153, 7
7. 阅读《现代方法》P328-P333