- 函数
  - 模块化程序设计
    - 函数与模块化
      - 实例1
      - 实例2
      - C程序的一般结构
  - 自定义函数
    - 函数定义
      - 函数定义的形式
      - 参数列表
      - 函数返回值
    - 函数原型
      - 函数定义起函数声明的作用
      - 函数原型
      - 函数调用
      - 函数调用的形式
      - 函数调用的执行过程
      - 实参的求值顺序
  - 变量的储存类型
    - 作用域与生存期
      - 作用域
      - 生存期
    - 自动变量
    - 外部变量
    - 静态变量
      - 静态局部变量
      - 静态全局变量
  - 递归
    - 递归概述
    - 递归算法分析
    - 递归函数设计
    - 经典递归程序设计
      - 汉诺塔问题
      - 约瑟夫问题

函数

模块化程序设计

模块化程序设计就是采用"**自顶向下逐步细化**"的方式,把一个大的程序按照功能化为若干相对独立的模块,各个模块完成一个确定的功能,在这些模块之间建立必要的数据联系.

函数与模块化

子程序在程序编制上相互独立,但在数据处理上又相互联系.完成总任务的程序由一个主程序和若干子程序组成,主程序起着**任务调度**的总控作用,每个子程序各自完成单一的任务.
其优点有:

- 子程序代码公用;
- 程序结构模块化,可读性,可维护性和可扩展性强;
- 简化程序控制流程.

实例1

以下面函数为例:

```c
#include <stdio.h>
// 函数声明或函数原型
int add(int a, int b);
int subtract(int a, int b);
int multiply(int a, int b);
int divide(int a, int b);
int main() {
    int num1, num2;
    char op;
    printf("Enter an expression (e.g., 2 + 3): ");
    scanf("%d %c %d", &num1, &op, &num2);
    switch(op) {
        case '+':
            printf("Result: %d\n", add(num1, num2));//函数调用
            break;
        case '-':
            printf("Result: %d\n", subtract(num1, num2));//函数调用
            break;
        case '*':
            printf("Result: %d\n", multiply(num1, num2));//函数调用
            break;
        case '/':
            if (num2 != 0) {
                printf("Result: %d\n", divide(num1, num2));//函数调用
            } else {
                printf("Error: Division by zero\n");
            }
            break;
        default:
            printf("Error: Invalid operator\n");
    }
    return 0;
}
//函数定义
int add(int a, int b) {
    return a + b;
}
int subtract(int a, int b) {
    return a - b;
}
int multiply(int a, int b) {
    return a * b;
}
```

```
int divide(int a, int b) {
    return a / b;
}
```

上述程序包括多个模块，每个模块执行一个特定的功能，例如执行加法、减法、乘法和除法.
main函数中调用了add等函数,因此main是**调用函数**,add是**被调用函数**.
标识符add出现在:函数原型,函数调用和函数定义中.

1. 先看函数**定义的的头部**: `int add(int a, int b) //无分号` .
   函数接收两个整型参数,并包含名字为a,b的**int类型变量声明**,变量a,b称为**形式参数**.
2. 函数调用add(num1, num2)把num1和num2的值传递给a和b,称函数调用传递的值称为**实际参数**.
3. 函数体部分是一个黑盒子,里面**定义的一切变量**都是局部的,对调用函数main来说是不可见的.最后return语句把函数运行的结果返回给了调用函数.
4. 看函数的**声明语句**: `int add(int a, int b); //有分号` .
   它是**函数声明语句,又称为函数原型**,它说明函数add接收了两个int类型的参数,返回一个int类型的值.
   编译器在调用函数add前看到这个原型,就可以根据原型中的描述对**实际参数的个数和类型**进行检验,以确保数量和类型上一致.
5. 函数之间可以根据**参数传递和返回值**进行通信.调用函数将实际参数传递给被调用函数的形式参数,被调用函数通过**return语句**将函数的返回值传给调用函数.

实例2

```
#include<stdio.h>
#include<stdlib.h>/* 标准函数库的头文件 */
#include<time.h>/* 日期和时间函数库的头文件 */
#define MAX_NUMBER 1000/* 被猜数的最大值 */
void guessNum(int);
int getNum(void);
int main()
{
    char choice='y';
    int magic;
    printf("This is a guessing game\n");
    srand(time(NULL));/* 用系统时间初始化随机数发生器 */
    for(;choice=='y'||choice=='Y';)
    {
        printf("A magic number between 1 and %d has been chosen.\n",MAX_NUMBER);
        magic=getNum();
        guessNum(magic);
        printf("Play again?(Y/N)\n");
        scanf("%1s",&choice);
    }
    return 0;
}
int getNum(void)
{
    int numberToGuess;
    numberToGuess=rand();/* 调用标准库函数rand产生一个随机数 */
    numberToGuess=numberToGuess%MAX_NUMBER+1;
    return numberToGuess;
}
void guessNum(int numberToGuess)
{
    int guess;
    int attempts;
    do {
        printf("Guess the number (between 1 and 1000): ");
        scanf("%d", &guess);
        attempts++;

        if (guess > numberToGuess) {
            printf("Too high! Try again.\n");
        } else if (guess < numberToGuess) {
```

```
            printf("Too low! Try again.\n");
        } else {
            printf("Congratulations! You guessed the number %d in %d attempts.\n", numberToGuess
        }
    } while (guess != numberToGuess);
    return;//返回控制但不返回值,事实上这个转移语句可以不用
}
```

1. rand是**随机数发生器**,该发生器从**种子(一个无符号整型数)的初始值**开始用**确定的算法**(线性同余法)产生随机数.显然通过种子产生第一个随机数后,后续的随机序列就确定了,这种随机数称为**伪随机数**.可见随机数的产生依赖于种子,要产生不同的随机数就要改变种子,称为**初始化随机数发生器**,由函数**srand**来实现.
2. 函数原型 `int getNum(void);` 没有输入参数,不接受任何参数,因为它不需要来自调用函数的任何信息.其返回值类型是int,和调用函数之间的通信是单向的.
3. rand函数是stdlib.h中的函数,它返回一个不大于RAND_MAX的随机整数(一个符号常量);
4. 函数原型 `void guessNum(int numberToGuess);` 有一个int类型的输入参数,无返回值.
5. 函数time是自1970年1月1日以来经历的秒数,将其赋值给种子变量.time原型在头文件time.h中
   如果程序在一秒内多次调用 `time(NULL)`，可能会导致 `srand(time(NULL))` 在同一秒内被多次调用，从而导致随机数种子重复，最终产生相同的随机数序列。

为了避免这种情况，可以采取以下几种方法之一：

- **延迟调用**：在调用 `time(NULL)` 之前添加一定的延迟，以确保两次调用之间的时间差超过一秒钟。

```
#include <unistd.h>
// 在调用 time(NULL) 之前添加延迟
sleep(1);
srand(time(NULL));
```

- **使用更高分辨率的时间函数**：如果可用的话，可以使用更高分辨率的时间函数，例如 `gettimeofday()` 或 `clock_gettime()`，以获取微秒级别的时间戳。

```
#include <sys/time.h>
struct timeval tv;
gettimeofday(&tv, NULL);
srand(tv.tv_sec * 1000000 + tv.tv_usec);
```

- **使用更复杂的种子生成方法**：如果你需要更高级的随机性和安全性，可以考虑使用密码学安全的伪随机数生成器，例如 `arc4random()` 或 `random()`。这些函数会自动处理种子问题。

```
#include <stdlib.h>
// 使用 random() 函数生成种子
srand(random());
```

选择哪种方法取决于你的具体需求和可用的函数。如果你只是需要简单的随机数生成，那么第一种方法通常就足够了。

C程序的一般结构

1. C程序只有一个main函数,程序的执行总是从main开始.除了main之外的函数分两类:
   i. 由系统提供的标准函数,又称为**库函数**,用户程序只需包含相应的头文件即可直接调用;
   ii. **自定义函数**.
2. 组成C程序的各个函数可以放在一个源文件中,也可以编辑成多个C源文件,每个C源文件可以有0个(仅有一些说明,如定义一些全局变量)或多个函数.
3. 各个C源文件中要用到的一些外部变量说明,枚举类型声明,结构类型声明,函数原型,编译预处理指令等可以编辑成一个.h**头文件**,然后在每个源文件中包含该头文件.
4. 每个源文件可以单独编译生成目标文件,组成一个C程序的所有源文件都被编译后,由连接程序将各目标文件中的目标函数和库函数装配成一个可执行程序.

自定义函数

程序要用自定义函数实现功能要做三件事:

1. 按语法规则编写完成任务的函数,即**函数定义**;
2. 在有些情况下,调用函数之前要**函数声明**,即**函数原型**;
3. 调用函数.

函数定义

函数定义的形式

```
类型名  函数名(参数列表)
{
    声明部分
    语法部分
}
```

类型名 函数名(参数列表) 是函数头部,由类型名,函数名和参数列表组成.==函数体包含声明部分和语法部分.

参数列表

参数列表说明**函数入口参数的名称,类型和个数**,它是一个用逗号分隔的变量声明表,其中的变量名称为**形式变量**.

- 有参函数.
  参数列表里有参数,每个形参必须声明其类型,不能像普通变量声明那样来声明同类型参数.
  错误的函数体: `int fun(int x,y)` ,其中y没有指定类型.
- 无参函数.
  参数列表是空的或说明是void.
  函数头可以是 `int fun()` 或 `int fun(void)` .

函数返回值

函数名前的类型名说明函数返回值的数据类型,简称**函数类型**.
函数必须明确指定返回值类型.

- 有返回值函数.
  - 执行完后送出一个数值给调用者,提出称为函数返回值,该值是通过return语句送出的.因此==有返回值的函数必须使用**带表达式的return语句**,return中表达式的值就是函数的返回值.
  - 一般来说,表达式值的类型应该与函数定义时指明的类型一致.如果两者不一致,对于基本类型,==将表达式的值的类型**自动转换为函数的类型**.
  - 对于**指针类型**,必须使用**强制类型符**将表达式值的类型显示转换为函数的类型.
  - 对应函数返回的值,程序可以试用它,也可以不使用它.
- 无返回值的函数.
  - 无返回值的函数可以使用不带表达式的return语句: `return;` ,其仅将控制返回到调用处,但不返回值.也可以不用return语句,如果不用则执行的函数结束时控制返回,称为**离开结束**.

- return后面只能跟一个表达式,函数只能返回一个值到调用处.如果要返回多个值,可以使用**外部变量和指针参数**间接返回.

函数原型

和变量一样,函数也应该**先声明后使用**,声明函数的方法有两种:

- 给出完整的**函数定义**;
- 提供**函数原型**,或两者都给出.

函数定义起函数声明的作用

如果函数定义**出现在函数调用之前**,那么函数定义起函数声明的作用.如果函数定义出现在函数调用之后,那么在**调用前就必须写上原型声明**.
有了函数原型信息,编译器就也可以检查函数调用是否与原型声明一致.如果参数类型不匹配,则编译器会根据形参类型转换实参类型,再传递给形参.

函数原型

如果函数定义出现在函数调用之后,或者被调用函数在其他文件中定义,则必须在函数调用前给出函数原型.函数原型一般形式为: `类型名 函数名(参数列表);`

- 函数原型是声明语句;
- 形参名可以省略;
- 对于无参函数,不同于函数定义的函数头,函数原型的参数类型表必须指定为void,即 `int fun(void);` .
- 函数原型可以位于调用函数体内,但此时如果其他函数也要调用它,就要在自己的函数体内也写上函数原型.
- 为了重复利用C语言的类型检索能力,在程序中应该包含所有函数的函数原型.库函数的函数原型在相应头文件中,因此要**用#include预处理指令包含相应头文件**.

函数调用

调用时,函数名后面的参数表中的表达式称为**实际参数**,将实参传递给形参.

函数调用的形式

`函数名(实参列表)` 实参和形参的数目,次序和类型应该一致.函数调用的执行过程是计算实参表达式的值,然后执行函数调用.对于无参函数,调用时实参列表为空: `函数名()` .
函数调用在程序中起一个表达式或语句的作用.
`while(putchar(getchar())!='#')`:
调用getchar函数→以getchar函数的返回值作为实参调用putchar函数→将putchar的返回值与'#'比较.

函数调用的执行过程

1. 遇到函数调用时,系统先**为每个形参分配存储单元,并计算实参表达式的值**,把实参的值赋值给形参;
2. 将执行的**控制转移到被调用函数的第一条执行语句处**开始执行,直到遇到下一个return语句;
3. 执行return语句或离开结束时,**控制返回到调用处,送回返回值**.

实参的求值顺序

C语言参数传递的方式是值传递.要计算每个**实参表达式的值**,再将实参的值单项传递给相应的形参.实参与形参各自有着**不同的储存单元**,形参接收的是实参的值,被调用函数对形参变量值的修改不会影响实参变量的值.
传地址调用是将变量的地址传递给函数,函数既可以**使用**,又可以**改变**实参变量的值. `scanf("%d",&x);` 就是传地址的例子.通过传递x的地址(&x),被调用函数可以改变实参表达式中变量x的值.

变量的储存类型

C语言的变量有两种属性:**数据类型和存储类型**.

- 数据类型决定变量==**存储空间的大小,数据的取值范围,数据的操作运算**==.
- 存储类型决定==**变量的作用域,存储方式,生存期和初始化方式**==.
    - 作用域是指变量的空间有效性;
    - 存储方式是指何处给变量分配存储单元;
    - 生存期是指变量在内存中存在的时间,是变量的**时间有效性**;
    - 初始化方式是指在定义变量时如果未显示初始化,是否有默认初值;如果显示初始化,赋初值的操作如何执行(执行一次还是多次).

作用域与生存期

作用域

1. 作用域是指**标识符(变量名或函数名)起作用的有效范围**,也就是正文中**可以使用该标识符**的那部分程序段.变量的作用域可以是**代码块范围或文件范围**.
2. 变量的作用域**由变量的定义位置决定**，在不同位置定义的变量，它的作用域是不一样的。
3. 在代码块中定义的变量,其作用域仅局限于该代码块,代码块作用域变量也被称为**局部变量**.函数的形式参数也是局部变量.
   E.G.

```
int addDigits(int x)/* x的作用域开始 */
{
    int sum=0;/* sum的作用域开始 */
    while(x){
        int a;/* a的作用域开始 */
        a=x%10;
        sum+=a;
        x/=10;
    }/* a的作用域结束 */
    return sum;/* x和sum的作用域结束 */
}
```

1. 在所有函数之外定义的变量具有**文件作用域**,其作用范围从定义它们的位置开始,一直到文本结束.由于它们可以在多个函数中使用,文件作用域变量又称为**全局变量**.

生存期

1. 变量的生存期是指一个变量在程序执行过程中的有效期,即能在内存中存在多久,它**由变量的存储方式决定**.
2. 存储方式是指为变量分配使用内存空间的方式,可以分为**静态存储方式**和**动态存储方式**,对应有**静态生存期**和**动态生存期**.
3. 一个程序将操作系统分配给其的内存空间分为四个区域:
    i. **栈区(动态数据区)**:存储空间在程序运行期间由编译器**自动分配释放**;
    ii. **堆区**:由程序员==**调用malloc等函数**主动申请,需使用free函数来释放==,若不释放,程序结束时可能由系统回收;
    iii. **全局区(静态数据区)**:存储空间是在编译时分配,在整个程序执行期间**静态区中的数据一直存在**,程序结束后由系统释放.
    iv. **常量区**:**常量字符串**放在这里,程序结束后由系统释放.常量区的数据是只读的，不允许被修改;
    v. **程序代码区**:存放程序的二进制代码.
4. 静态存储方式是指程序运行之前,系统就为变量在全局区分配存储单元,并一直保持不变,直到程序结束.**全局变量就是这种存储方式**.
5. 动态存储方式是在程序运行过程中,执行到变量所在代码块时,系统为变量**在栈区分配内存**,退出块时立即释放内存空间.**函数的形参属于此类存储方式**,在函数被调用时,给形参在栈上分配存储空间,调用完毕立即释放.一个函数如果被多次调用则反复分配,释放形参变量的存储单元.
6. 静态存储变量在程序执行过程中**始终占据着固定的存储空间**,其生存期是整个程序的执行时间;而动态存储变量在程序执行过程中**动态分配和释放**,生存期是代码块执行时间.
7. 生存期和作用域是在**时间和空间**这两个不同的角度描述变量有效性.全局变量具有静态生存期,而局部变量既可以由静态生存期,又可以有动态生存期.

自动变量

- 局部变量的默认存储类型是auto,称为**自动变量**.运行省略关键字auto.函数内部的下列三条语句完全等价: `auto int x;`；`int x;`；`auto x;` .
- 自动变量的**作用域是块范围**,局限于定义他的代码块.
- 自动变量有**动态生存期**,只存在于**该块的执行期间**.从块内定义之后直到该块结束有效.程序进入块时,编译器为自动变量在**栈区**内分配内存(由此可以判断自动变量是**动态存储方式**),退出块时,释放分配给自动变量的内存空间,**变量的值就丢失了**.重新进入块,编译器会为自动变量再次分配空间,但原先的值就没有了.
- 自动变量**没有默认初值**,如果定义时没有显示初始化,则其初值是不确定的.如果有显示初始化,每次进入块时都要执行一次赋初值操作.
- 代码块可以多重嵌套,每个块都可以定义自己的变量.外层块变量在内层块中是有效的,这称为**作用域的嵌套**.但是当内层变量和外层变量同名时,就引入了**可见性**的概念:**在内层中,外层的同名变量暂时失去了可见性,即被屏蔽了**.

外部变量

- 全局变量隐含的存储类型是extern,也称为**外部变量**.它是在函数外定义的变量,但在**定义时不使用关键字extern**.
- 外部变量的存储方式是**静态的**,被分配在静态数据区.因此其生存期是**静态生存期**,在程序执行期间外部变量一直占据内存.
- 如果定义时没有初始化,外部变量**默认初始值为0**;如果有初始化,只进行一次赋初值操作.
- 外部变量的作用域是**文件范围**,从变量的定义开始,一直到该源文件结束(可以看出外部文件就是全局变量).在作用域内的函数可以合法地引用该外部变量.不在作用域内的函数(**如定义点前的函数和其他文件中的函数**)也可以引用,但是必须**在引用前用extern对外部变量进行引用性声明**,否则就是不合法引用.extern告诉编译器"此变量是在别处定义的,要在此处引用.
- 因为外部变量可以**全局访问**,因此函数之间传递数据的方式除了用参数外,还可以用外部变量.这就为函数提供了一种可以代替return间接返回值的方法,从而解决函数返回多值的问题.
  E.G.

```c
#include<stdio.h>
void secondToTime(int sec);/* 函数原型 */
int main()
{
    extern int h,m,s;/* 外部变量的引用性说明 */
    int second;
    printf("Please input number:");
    scanf("%d",&second);
    secondToTime(second);
    printf("%d:%d:%d",h,m,s);
    return 0;
}
int h,m,s;/* 外部变量定义性声明，作用域开始 */
void secondToTime(int sec)/* 函数定义 */
{
    h=sec/3600;
    m=(sec-3600*h)/60;
    s=sec%60;
}
/* 外部变量作用域结束 */
```

上面的例子中,被调用函数在外部变量的作用域内,该函数无需用extern作引用性声明;main函数的定义在外部变量定义之前,不在作用域内,extern的**引用性声明**必须有.

外部变量的引用性声明语句可以在函数之外,也可以在函数之内.如果在函数之外,则从声明后到文件结束的所有函数都有效;如果在函数内,则所有使用该外部变量的函数内部都要写一条引用性说明.

局部变量只有定义性声明,没有引用性声明.而对于外部变量的定义性声明 `int sp;` ,声明语句必须在函数之外,这一方面说明变量sp是一个类型为int的外部变量;另一方面系统还会为sp分配4B存储单元.

而声明语句 `extern int sp;` 是对已经定义的外部变量sp做引用性说明,它既可以在函数内又可以在函数外,系统不会为sp分配存储空间,仅在代码中使用它.

外部变量可以与局部变量同名,此时局部变量会屏蔽同名的外部变量,在定义了局部变量的块内,同名的外部变量不可见.

应当避免不必要的外部变量,因为它始终占据内存,会使函数通用性变差等等.

静态变量

全局变量和局部变量(**默认类型是自动变量**)都可以**用关键字static定义为静态变量**.
static用于定义局部变量时,称为**静态局部变量**,它和自动变量(局部变量默认存储类型为auto)有根本性的

区别.
static定义全局变量,称为**静态全局变量或静态外部变量**,它和外部变量的区别仅在于作用域不同.
静态变量(包括局部和全局)和外部变量一样,分配在静态数据区,其生存期为整个程序执行期,默认初值为0.

静态局部变量

1. 静态局部变量的作用域和局部变量一样,是**块作用域**,只作用于定义它的块(因为作用域由变量定义的位置决定).
2. 但静态局部变量被分配在**静态数据区**,他在程序执行期间不会消失,因此它的值具有**记忆性**,当退出块时,它的值能保留下来,以便下次进入块时使用(**用空间换时间**),而自动变量(栈区)的值在退出时都丢失了.
3. 如果定义时静态局部变量有显示初始化,仅执行一次赋初值操作;而自动变量每次进入块时都要执行一次赋初值操作.
   E.G.

```
#include<stdio.h>
long fac(int);
int main()
{
    int i,n;
    long sum=0;
    printf("Input n(n>0):\n");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    sum+=fac(i);
    printf("1!+2!+...+%d!=%ld\n",n,sum);
}
long fac(int n)
{
    static long f=1;/* 静态局部变量 */
    f*=n;
    return f;
}
```

静态全局变量

1. 全局变量在定义时**加static**就成了**静态全局变量**,它和外部变量唯一的区别在于**作用域不同**.
2. 静态全局变量只能作用于定义它的文件(**文件作用域**),而且其作用域与定义位置无关.其他文件中的函数不能使用,可以在定义他的文件内用**extern**对静态全局变量进行引用用性声明，也可以不作引用性声明.

3. 静态全局变量具有**内部链接(internal linkage)**,意味着它们只能在声明它们的源文件中访问，其他源文件无法访问,即使试图用extern声明也不能引用.外部变量具有**外部链接(external linkage)**,意味着它们可以在其他源文件中通过外部链接符号来访问.

```
/* 文件file1.c:实现伪随机数发生器 */
/* 线性同余法是产生伪随机数的经典算法,它通过如下递推关系定义:
X(0) = seed
X(n) = (A*X(n-1)+C) mod M
X(n)是伪随机序列,seed是种子变量,M是模数,也是生成序列的最大周期.参数的选择十分重要 */
#define INITIAL_SEED 17
#define MULTIPLIER 25173
#define INCREMENT 13849
#define MODULUS 32767
static unsigned long seed = INITIAL_SEED; /* 种子 */
/* 产生0~MODULUS的整型随机数 */
unsigned myrand()
{
    seed = (seed*MULTIPLIER+INCREMENT) % MODULUS;
    return seed;
}
/* 用形参x初始化随机数种子 */
void mysrand(unsigned x)
{
    seed=x;
}
```

线性同余法可以用来产生伪随机序列的原因在于它具有混合和周期性的特性。这种算法的设计使得生成的序列在统计上表现良好，看起来像是随机的。

以下是线性同余法能产生伪随机序列的几个关键原因：

1. **混合性（Mixing）**：线性同余法通过对先前的随机数进行一系列简单的数学操作（乘法、加法、取模），将先前的随机数的信息混合起来。这样做可以确保每个生成的随机数都至少部分取决于前一个数，从而在一定程度上保持了随机性。
2. **周期性（Periodicity）**：由于取模运算的存在，生成的随机数序列是有限的。换句话说，一旦生成的序列达到了一个特定的点，它将开始重复。这个周期的长度取决于所选用的参数。如果选择的参数合适，周期可能会很长，使得序列的重复性不容易被观察到。
3. **均匀性（Uniformity）**：如果选择参数得当，线性同余法生成的随机数序列可能具有良好的均匀性，即在一定范围内的随机数出现的概率大致相等。

虽然线性同余法具有这些优点，但是需要注意的是，如果参数选择不当或者种子不合适，可能会导致生成的随机数序列出现不均匀分布、周期性较短等问题。因此，在使用线性同余法生成随机数时，需要仔细选择参数，并考虑到需要满足的随机性质。

```c
/* 文件files.c模拟抛硬币 */
#include<stdio.h>
#include<time.h>
#define N 1000
void mysrand(unsigned);
unsigned myrand(void);
int main()
{
    int i,head=0,reverse=0;
    //sleep(0.01);
    mysrand(time(NULL));
    for(i=0;i<N;i++)
    {
        if(myrand()%2)
        head++;
        else reverse++;
    }
    printf("\n%d %d\n",head,reverse);
    return 0;
}
```

产生随机序列需要给定初始种子seed,且种子的初始化由初始化函数单独完成,因此种子是各随机数发生函数共享的变量,要定义在函数外.而且seed只提供给mysrand,myrand等函数使用,不希望任何其他函数访问操作它.因此将函数myrand,mysrand以及他们所操作的种子seed**设计在一个源文件file1.c中**,在定义seed时加上static,使之成为**静态全局变量**,作用域局限在文件file1.c中.main函数设计在另一个源文件file2.c中,seed在main函数中不可用.

外部变量已经可以被函数所共用,但是静态全局变量有其他好处:多个人编写一个程序的不同文件时,可以按照需要命名变量而不比考虑其他文件中的变量是否同名,保证文件独立性.

一个文件中的静态全局变量可以屏蔽其他文件中的同名外部变量.当一个文件中的静态全局变量和另一个文件中的外部变量同名时,如果在同一个文件中访问这个变量,则会优先访问静态全局变量,因为它的**作用域更小**.

递归

递归概述

函数递归是指**函数调用自身**的过程.在C语言中,函数直接调用自己或通过另一函数间接调用自己的方式称为**递归调用**.含有递归调用的函数称为**递归函数**.递归是一种重要的编程技术,常用于解决可以分解为相似子问题的问题,同时要注意递归函数的构造和参数的选取.

```
#include<stdio.h>
int demoRecur(unsigned int x)
{
    putchar(x%10+'0');
    if(x>9) demoRecur(x/10);/* 递归调用 */
    putchar(x%10+'0');/* 逐层返回 */
}
int main()
{
    int num;
    printf("Please input the number:\n");
    scanf("%d",&num);
    demoRecur(num);
    return 0;
}
```

1. 递归函数必须有**终止递归的条件**否则会无限递归,耗尽内存.
2. 每次递归都有一次**返回**,当程序流执行到某次调用的函数结束时,会返回到**前一次递归调用点**接着向下执行.当结束条件不满足时,将一层层递归调用下去;当结束条件满足时,**递归调用逐层返回**.
3. 位于递归调用前的语句的执行顺序和调用顺序相同;位于递归调用后的语句的**执行顺序和调用顺序相反**.
4. 每一次递归调用都使用**自己的私有变量**,包括**函数参数和函数内的自动变量**.同一层的参数和变量值是相同的.

递归算法分析

递归函数的算法是**递归定义**,为了描述某一状态,必须回到他的上一状态...,这种**用自己来定义自己的方法称为递归定义**.

```
unsigned long long factorial(int n) {
    if (n == 0 || n == 1) {
        return 1; // 阶乘的基本情况
    } else {
        return n * factorial(n - 1); // 递归调用
    }
}
```

- **递归调用的过程**:factorial(n)→n*factorial(n-1)→(n-1)*factorial(n-2)→...→2*factorial(1)→factorial(1)
- **递归调用的返回值**:1→2*1!→3*2!→...→(n-1)*(n-2)!→n*(n-1)!→n!
- 为了保证递归调用正确执行,系统要建立**递归调用工作栈(调用栈、执行栈或递归栈)**,为各层次的调用分配数据存储区.每层递归调用所需的信息构成一个工作记录,其中包括所有实参所有自动变量,以及返回上一层的地址.
- 每进入一层递归调用,就产生一个新的工作记录**压入栈顶**;每退出一层递归调用,就从栈顶**弹出一个工作记录**.
- 调用栈是一个**先进后出（FILO）的数据结构**,它确保了函数调用的正确顺序和局部变量的正确管理.然而,当递归调用的层级过深时,调用栈可能会耗尽内存空间,导致栈溢出错误.因此,在编写递归函数时,需要谨慎考虑递归的终止条件和递归深度,以避免栈溢出的问题.

递归函数设计

```
#include<stdio.h>
int mystrcmp(char s[],char t[])
{
    if(s[0]!=t[0])
    return 1;
    else if(s[0]=='\0')
    return 0;
    else return(mystrcmp(&s[1],&t[1]));
}
```

经典递归程序设计

汉诺塔问题

定义函数 `void move(int n,int a,int b,int c);` 其功能是:将n个盘子通过a借助b移到c.

```c
#include<stdio.h>
#include<stdlib.h>
void move(int,int,int,int);
int main()
{
    int n,a ='A',b ='B',c ='C';
    printf("The problem starts with n disks on Tower A.\nInput n:\n");
    if(scanf("%d",&n)!=1||n<1){
        printf("\nERROR:positive integer not found");
        return -1;
    }
    move(n,a,b,c);
    return 0;
}
void move(int n,int a,int b,int c)
{
    static int cnt=1;
    if(n==1)
        printf("step %d:%c-->%c\n",cnt++,a,c);
    else {
        move(n-1,a,c,b);
        printf("step %d:%c-->%c\n",cnt++,a,c);
        move(n-1,b,a,c);
    }
}
```

我们以 `n=3` 为例，来说明代码的运行过程，包括调用栈中的处理方式。
首先，调用 `move(3, 'A', 'B', 'C')` 。

1. 调用 `move(3, 'A', 'B', 'C')` :
    - 进入 `move` 函数， `cnt` 初始化为 1。
    - 进入 `else` 分支，调用 `move(2, 'A', 'C', 'B')` 。
2. 调用 `move(2, 'A', 'C', 'B')` :
    - 进入 `move` 函数， `cnt` 保留为 1。
    - 进入 `else` 分支，调用 `move(1, 'A', 'B', 'C')` 。
3. 调用 `move(1, 'A', 'B', 'C')` :
    - `n==1` 成立，打印 "step 1: A --> C"。
    - 返回上一级调用。

4. 回到 `move(2, 'A', 'C', 'B')`：
    - 打印 "step 2: A --> B"。
    - 调用 `move(1, 'C', 'A', 'B')`。
5. 调用 `move(1, 'C', 'A', 'B')`：
    - `n==1` 成立，打印 "step 3: C --> B"。
    - 返回上一级调用。
6. 回到 `move(2, 'A', 'C', 'B')`：
    - 打印 "step 4: A --> C"。
    - 调用 `move(1, 'B', 'A', 'C')`。
7. 调用 `move(1, 'B', 'A', 'C')`：
    - `n==1` 成立，打印 "step 5: B --> A"。
    - 返回上一级调用。
8. 回到 `move(3, 'A', 'B', 'C')`：
    - 打印 "step 6: A --> B"。
    - 调用 `move(2, 'C', 'A', 'B')`。
9. 调用 `move(2, 'C', 'A', 'B')`：
    - 进入 `move` 函数，`cnt` 保留为 1。
    - 进入 `else` 分支，调用 `move(1, 'C', 'B', 'A')`。
10. 调用 `move(1, 'C', 'B', 'A')`：

- `n==1` 成立，打印 "step 7: C --> A"。
- 返回上一级调用。

11. 回到 `move(2, 'C', 'A', 'B')`：

- 打印 "step 8: C --> B"。
- 调用 `move(1, 'A', 'C', 'B')`。

12. 调用 `move(1, 'A', 'C', 'B')`：

- `n==1` 成立，打印 "step 9: A --> B"。
- 返回上一级调用。

13. 回到 `move(2, 'C', 'A', 'B')`：

- 打印 "step 10: C --> A"。
- 调用 `move(1, 'B', 'A', 'C')`。

14. 调用 `move(1, 'B', 'A', 'C')`：

- `n==1` 成立，打印 "step 11: B --> C"。

- 返回上一级调用。

15. 回到 `move(3, 'A', 'B', 'C')`：

- 打印 "step 12: A --> B"。
- 调用 `move(2, 'B', 'C', 'A')`。

16. 调用 `move(2, 'B', 'C', 'A')`：

- 进入 `move` 函数， `cnt` 保留为 12。
- 进入 `else` 分支，调用 `move(1, 'B', 'A', 'C')`。

17. 调用 `move(1, 'B', 'A', 'C')`：

- `n==1` 成立，打印 "step 13: B --> A"。
- 返回上一级调用。

18. 回到 `move(2, 'B', 'C', 'A')`：

- 打印 "step 14: B --> C"。
- 调用 `move(1, 'A', 'B', 'C')`。

19. 调用 `move(1, 'A', 'B', 'C')`：

- `n==1` 成立，打印 "step 15: A --> C"。
- 返回上一级调用。

20. 回到 `move(2, 'B', 'C', 'A')`：

- 打印 "step 16: B --> A"。
- 调用 `move(1, 'C', 'A', 'B')`。

21. 调用 `move(1, 'C', 'A', 'B')`：

- `n==1` 成立，打印 "step 17: C --> B"。
- 返回上一级调用。

22. 回到 `move(2, 'B', 'C', 'A')`：

- 打印 "step 18: B --> C"。
- 调用 `move(1, 'A', 'C', 'B')`。

23. 调用 `move(1, 'A', 'C', 'B')`：

- `n==1` 成立，打印 "step 19: A --> B"。

- 返回上一级调用。

24. 回到 `move(3, 'A', 'B', 'C')`：

- 打印 "step 20: A --> B"。
- 返回上一级调用。
  最终，整个递归调用过程结束，汉诺塔问题得到了解决，所有盘子都从柱子 'A' 移动到了柱子 'C'，并且每一步移动的过程都被正确输出。

约瑟夫问题

约瑟夫问题(Josephus Problem)一个经典的数学问题,涉及一个有 n 个人的环,这些人按顺时针方向从 1 到 n 编号.从环中的第一个人开始,沿着环的方向每次数到第 m 个人,然后将该人从环中删除.接着从被删去的人的下一个人开始,继续进行相同的操作,直到环中只剩下一个人为止.问题的目标是找出最后留下的那个人的编号.

假设我们用$f(n,m)$表示当有n个人,每报到第m个人就出列时,最后留下的人在原始序列中的位置.为了推导递推关系,我们可以考虑从小规模问题出发,逐步推导出递推关系.

当$n=1$时,只有一个人,那么他肯定是最后留下的人,位置就是1,即$f(1,m)=1$.

当$n=2$时,有两个人,按照规则,第一个人报数为1,当他出列后,剩下的人就是最后留下的,位置是2,即$f(2,m)=2$.

当$n=3$时,有三个人,我们可以模拟报数的过程,假设报到m的人出列后,剩下的人的位置为x,由于是环形的,所以剩下的人重新从1开始报数,所以剩下的人在出列前的编号为 $(x+m-1)\%3+1$,所以最后剩下的人的位置就是 $f(3,m)=(x+m-1)\%3+1$.现在,我们需要找到x和$f(2,m)$之间的关系,我们知道当$n=2$时,最后留下的人的位置是$f(2,m)=2$,所以 $x=f(2,m)$.将其代入上面的公式,得到$f(3,m)=(f(2,m)+m-1)\%3+1$.
我们可以通过这种方式推导出更大规模的问题的递推关系，一般情况下，有如下的递推关系:

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

这就是约瑟夫问题的递推关系.

```
int josephus(int n, int m) {
    int result = 0;
    for (int i = 2; i <= n; i++)
    {
        result = (result + m) % i;
    }
    return result + 1;
}
int main() {
    int survivor = josephus(N, M);
    printf("The survivor is at position:%d\n", survivor);
    return 0;
}
```

也可以使用数组模拟的递归算法.

```c
#include<stdio.h>
#define N 15
#define M 6
//递归思想
int circle[N]={0}; //0表示在圈内,1表示被移出圈
void josephus(int n,int m,int position) // n个人从position位置开始1~m报数
{
    int count;
    if(n==1) return;
    for(count = 1;count<=m;position++)
    {
        position = position % N; //让数组编号为position的下标圆圈化,也防止下标溢出
        if(circle[position]==0) count++; //在圈内则计数加一
    }
    circle[position-1] = 1; //报数为m的人出圈,模拟删除过程
    josephus(n-1,m,position);
}
int main(void)
{
    int i;
    josephus(N,M,0);
    for(i = 0;circle[i];i++)
        ;
    printf("The survivor is at position:%d\n",i+1);
    return 0;
}
```