【函数】

int swap(int x,int y){  
int t=x;

x=y;

y=t;}

这种机制是不可能完成x与y的交换的，想要实现，需要通过指针来进行传递：

int swap(int \*x,int \*y){  
int t=\*x;

\*x=\*y;

\*y=t;}

****实参(argument)：****

  全称为"实际参数"是在调用时传递给函数的参数. 实参可以是常量、变量、表达式、函数等， 无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值， 以便把这些值传送给形参。 因此应预先用赋值，输入等办法使实参获得确定值。

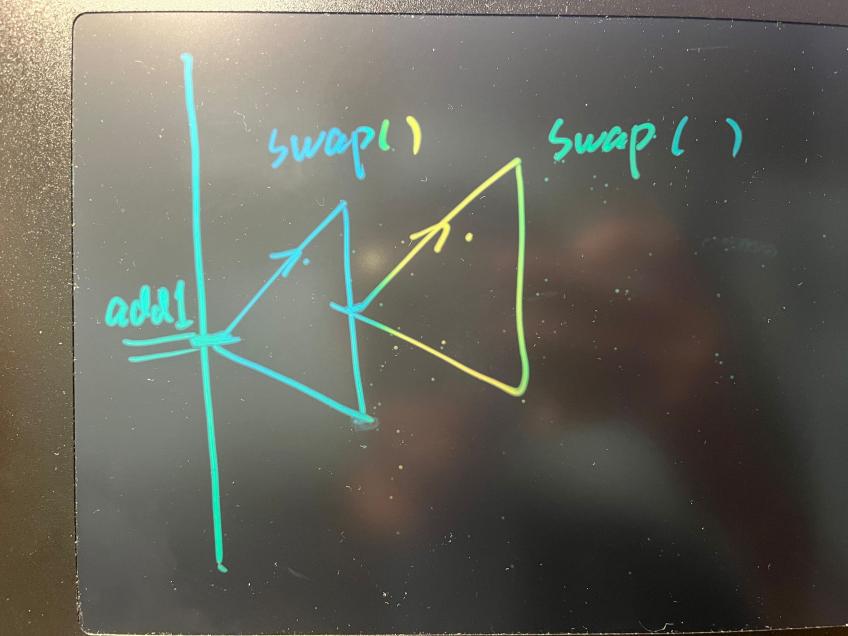
****形参(parameter)：****

全称为"形式参数" 由于它不是实际存在变量，所以又称虚拟变量。是在定义函数名和函数体的时候使用的参数,目的是用来接收调用该函数时传入的参数.在调用函数时，实参将赋值给形参。因而，必须注意实参的个数，类型应与形参一一对应，并且实参必须要有确定的值。

而在第一个程序代码中的(int x,int y)就属于形式参数，只通过改变形参无法传值，而指针操控的则是实参，**函数调用中发生的数据传送是单向的**。 即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。 因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化。

一个函数可以被多条语句调用，而且不知道其调用地址，这样就不知道应该如何return

为解决这个问题，我们应该在每次调用的时候记录其调用的地址，如：add1，add2等，这些地址和函数读取的参数就如同一个个元素一样被压入堆栈当中，当return的时候通过逻辑推理闭合回路可以发现，先调用的地址最后返回，而后调用的地址则最先返回，即先进后出原则，这与堆栈的原理相似。



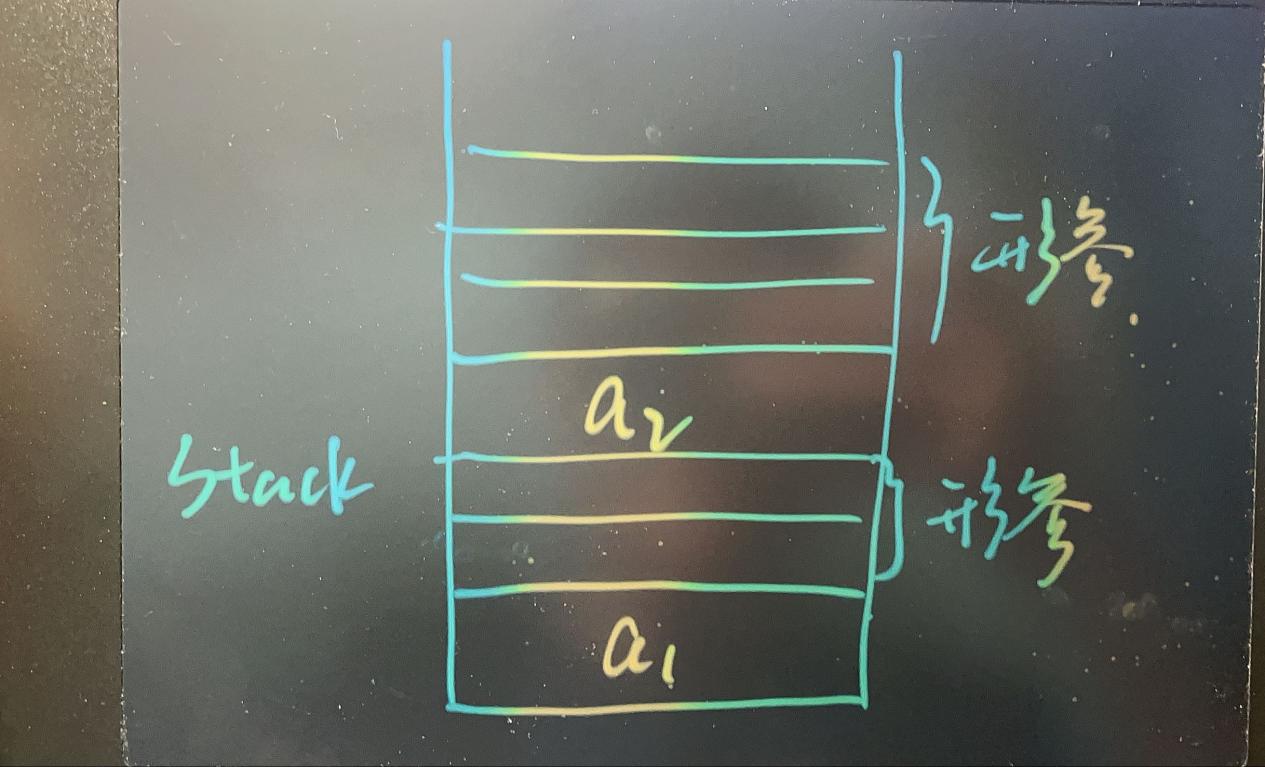
也就是说，每次调用函数的时候，我们就把调用的地址压入栈中，在return时再pop出来即可。

要注意的是，虽然堆栈中不仅存放着地址，还存放着形参，但是编译器可以自动根据参数的数量和类型进行筛除，所以最后pop出来的依然是address。函数中的局部变量，比如例子中swap（）里定义的int t，也要放在堆栈之中。

ERROR:Stack Overflow 栈满溢出，这样就无法保证顺序的正确性，同时引发丢失数据等情况，因此不要一次性调用大量函数，函数中也不要有过多的形参。

从逻辑图中可以看出，递归时，由于调用的是一个函数，那常常是在同一语句中调用，所以此时address就往往是相等的，add1可以等于add2，但在Hanoi程序中，由于main函数在不同位置进行了调用，虽然是递归类型，但两个地址并不相同。

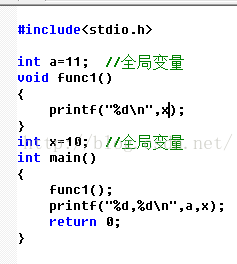
要注意下图中，return a1的话，a1中的值完全是调用语句的下一条语句的地址，是内存自动分配的，而并非返回值赋给它的，切记！



【作用域】

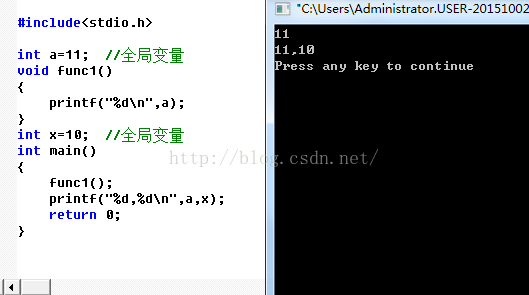
****局部变量（Local Variable）****：定义在函数体内部的变量，作用域仅限于函数体内部。离开函数体就会无效。再调用就是出错。

****全局变量（Global Variable）:****定义：所有的函数外部定义的变量，它的作用域是整个程序，也就是所有的源文件，包括.c和.h文件。



提示的错误是 变量 x 没有定义，原因是：

C语言代码是从前往后依次执行的，由于 x 定义在函数 func1() 之后，所以在 func1() 内无效；修改程序如下：



****可以看出，输出正确的答案，所以全局变量必须定义在我们需要用它的函数体之前。****

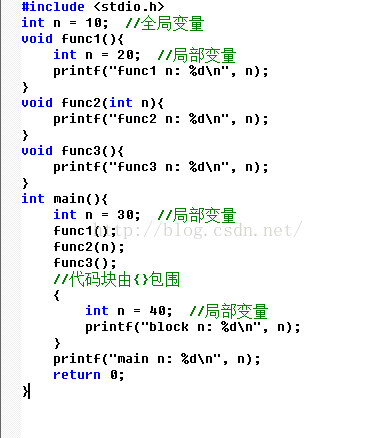
****注意事项：****

****①在main函数中定义的变量也是局部变量，只能在main函数中使用；同时，main函数中也不能使用其他函数中定义的变量。main函数也是一个函数，与其它函数地位平等。****

****②形参变量、在函数体内定义的变量都是局部变量，实参给形参传值的过程也就是给局部变量赋值的过程。****

****③可以在不同的函数中使用相同的变量名，它们表示不同的数据，分配不同的内存，互补干扰，也不会发生混乱。****

****④在语句块（由一对{}包含的若干条语句）中也可以定义变量，它的作用域只限于当前的语句块。****



判断一下打印出来的五个值分别应该是多少：  
①先调用fun1()，在fun1()中，有局部变量n,所以输出的结果是 n:20;

②对于fun2()，使用了形参，形参的作用范围也是整个函数内（实参给形参传值的过程也就是给局部变量赋值的过程），所以输出的结果是 n:30;

③对于fun3()，没有局部变量，所以全局变量定义的 n 对所有的函数都可见。所以输出的结果是 n:10;

④根据局部变量注意事项第四条。所以输出的结果是 n:40;

⑤3C语言规定，只能从小的作用域向大的作用域中去寻找变量，而不能反过来，使用更小的作用域中的变量。对于 main() 函数，即使代码块中的 n 离输出语句更近，但它仍然会使用 main() 函数开头定义的 n，所以输出结果是 30。

（趋向于选择大作用域）

【生命周期】

生命周期指的是指一个组件从创建 -> 运行 -> 销毁的整个阶段，强调的是一个时间段

全局变量：extern static，在程序启动时存在，结束后回给系统

静态局部变量（前加static）与全局变量相同，都不存在堆栈里，因此调用时不会存档，如果在调用的过程当中在定义处进行了对其值的改变，那么整个程序中他的值也都会发生改变。

【单文件与多文件的转化】

1. 预编译阶段

将main.c->main.c’通过预编译将源文件转变为纯纯可读的C文件；

1. 编译阶段

正式编译阶段，将main.c’转变为main.obj，也就是要进行加工处理的目标文件；

1. Link阶段

Stdio.lib（库文件）与目标文件一起被加工打包成main.exe，这个文件也就是我们可以直接操作的文件，一般来说文件持有者给予的应用文件（可以正常使用，但是没有查看库文件的权利）,就是以.lib文件实现共享的

如果在Link的文件中出现了多个main函数

【定义与声明】

声明可以声明多次，但定义却只能定义一次，内存是在定义时分配的。

声明一般都放在头文件中，但不要定义，因为每次调用头文件时都会重复定义导致出错。

extern int x;x=100;在这里extern代表声明的是一个外部变量

【C语言实现】

main.c  
  
#include <assert.h>  
  
#include "stack.h"  
  
int main() {  
    Stack \*stack = createStack(100);  
    push(stack, 0);  
    push(stack, 1);  
    push(stack, 2);  
    int x = pop(stack);  
    assert(x == 2);  
  
    return 0;  
}

stack.c  
  
#include <stdlib.h>  
#include "stack.h"  
  
int pop(Stack \*stack) {  
    return stack->data[--stack->sp];  
}  
  
void push(Stack \*stack, int item) {  
    stack->data[stack->sp++] = item;  
}  
  
Stack \*createStack(int size) {  
    Stack \*stack = (Stack \*) malloc(sizeof(Stack));  
    stack->sp = 0;  
    stack->data = (int \*) malloc(sizeof(int) \* size);  
    return stack;  
}

stack.h  
  
#ifndef STACK2\_STACK\_H  
#define STACK2\_STACK\_H  
  
typedef struct {  
    int \*data;  
    int sp;  
} Stack;  
  
Stack \*createStack(int size);  
  
void push(Stack \*stack, int i);  
  
int pop(Stack \*stack);  
  
#endif //STACK2\_STACK\_H