

Function

董洪伟

<http://hwdong.com>

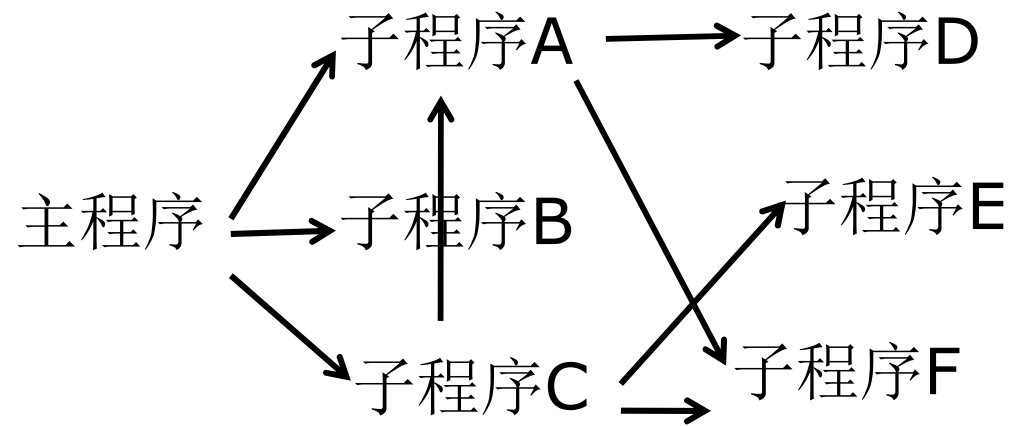
过程抽象-函数

- 早期一个大程序包含很多重复的代码，带来了许多不便：重复劳动、增加程序长度、维护麻烦且易造成代码间不一致、大量重复代码不利于程序理解等
- 功能抽象-子过程：将重复代码抽出来，并起一个名字。
- 带名字的子过程带来了许多方便：提高代码重用率、缩短程序长度、易于维护、易于理解等
- 例子：C语言的函数库. `printf`

子程序-函数

- 复杂的程序由一组子过程（函数）构成。
- 程序设计方法：功能分解、功能复合
- 功能分解：将程序的功能分解为一系列子功能，每个子功能又分解为更小的子功能，...，从而形成一种自顶向下、逐步精华的设计过程。
- 功能复合：将已有的(子)功能组合成更大的(子)功能
- 过程抽象：一个子程序代表一种功能，子程序的使用者只要知道这个子程序(What to do)，不需要知道它内部(How to do)

子程序-函数



子程序-函数

- 子程序的输入数据来自哪里？其结果又送到哪里？
- 子程序从它的调用者那里得到数据，并将结果返回给调用者。-调用者和被调用者之间的数据传递问题。
- 数据传递方式一：使用全局变量。传递很方便，但有两个不利因素：破坏程序独立性，不利于维护、调式等；许多子程序共享全局变量，带来安全性问题。

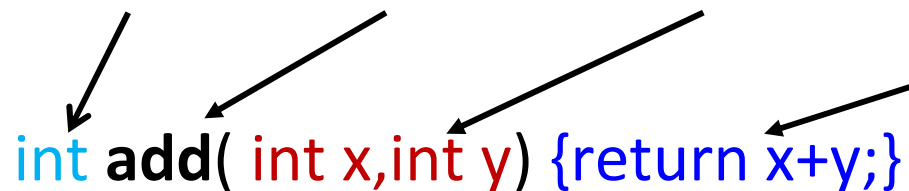
子程序-函数

- 数据传输方式二：参数传递和返回值机制。
- 调用者以参数的形式将数据传给被调用者，子程序将结果通过专门的返回机制返回给调用者
- 较常见的参数传递方式：值传递和引用传递
- 值传递：将调用者的实际参数的拷贝传给被调用者的形式参数。形式参数和实际参数相互独立、互不影响。可防止形式参数的修改影响实际参数。
- 引用传递：将实际参数的内存地址传给被调用者。形式参数和实际参数就是同一个内存。

函数

- 现代程序设计语言几乎都提高了支持子程序的语言成分，函数就是C/C++语言提供的用于实现子程序的语言成分。
- 注意：C/C++语言中的函数不是数学上的函数，是指的一段有名字的程序代码块。
- 函数定义格式：

返回类型 函数名 (形式参数表) 函数体



`int add(int x, int y) {return x+y;}`

The diagram illustrates the components of the function definition `int add(int x, int y) {return x+y;}`. Four arrows point from labels above to specific parts of the code: an arrow from '返回类型' (Return Type) points to 'int'; an arrow from '函数名' (Function Name) points to 'add'; an arrow from '(形式参数表)' (Formal Parameter List) points to '(int x, int y)'; and an arrow from '函数体' (Function Body) points to '{return x+y;}'.

函数

- 除void(空类型)外，函数都必须用return语句返回返回值类型的数据给调用者。
- 例：编写一个求n!的函数

```
int factorial(int n){  
    int f=1;  
    for(int i=2;i<=n;i++) f *=i;  
    return f;  
}
```


练习

- 编写一个函数根据调用者提供的成绩分数显示分数等级(优秀、良好、及格、不及格)信息。

```
void display(double score){  
    if(score>=90) cout<<"优秀"<<endl;  
    else if(score>=80) cout<<"良好"<<endl;  
    else if(score>=70) cout<<"中"<<endl;  
    else if(score>=60) cout<<"及格"<<endl;  
    else cout<<"不及格"<<endl;  
}
```

函数调用

- 通过提供**函数名和合适的参数**调用一个函数。

函数名(实际参数列表);

- 例：求阶乘函数的调用

```
int main(){  
    int x; cout<<"请输入一个整数：";  
    cin>>x;  
    cout<<"\nFactorial of "<<x<<" is "  
        <<factorial(x)<<endl;  
}
```

函数的参数传递-值参数

- 实际参数的值拷贝给形式参数。形式参数和实际参数有各自独立的存储空间。

```
void swap(int x,int y){int t =x,x= y;y = x;}
```

```
int main(){  
    int a = 3,b = 10;  
    swap(a,b);  
    cout<<"a="<<a<<","b="<<b<<endl;  
}
```

函数的参数传递-引用参数

- 引用传递：即形式参数名引用的就是实际参数，实际参数名和形式参数名指向的都是同一个内存地址

```
void swap(int &x,int &y){int t =x,x= y;y = x;}
```

```
int main(){  
    int a = 3,b = 10;  
    swap(a,b);  
    cout<<"a="<<a<<"",b="<<b<<endl;  
}
```

函数的参数传递-引用参数

- 引用参数用于传回结果。

```
void add(const int x,const int y,int& sum)
{sum = x+y;}
```

```
int main(){
    int a = 3,b = 10,c;
    add(a,b,c);
    cout<<a<<"+"<<b<<"="<<c<<endl;
}
```

全局变量、局部变量

- 定义在函数外部的变量，称为全局变量；定义在函数内部的变量，称为局部变量。

```
int x = 3; //全局变量
```

```
int main(){
```

```
    int y = ++x; //局部变量
```

```
    int x = 12; //局部变量 覆盖了全局变量
```

```
    cout<<++x<<endl;
```

```
    return 0;
```

```
}
```

全局变量、局部变量

- 变量使用前需要见到它的声明，定义可以在使用语句后面。

```
extern int x;
void f(){
    extern int y;    y =x;
    int y = 9;
}
int x = 0 ;
void main(){
    cout<<"x="<<x<<" y"=<<y<<endl;
}
```

变量的存储位置与生存期

- 程序的内存可分为：静态数据区(data)、代码区(code)、堆栈区(stack)和堆区(heap)。
- 变量就是一块内存，但可能具有不同的存储分配。程序的静态数据区存储全局变量、常量，代码区存储程序的指令，即全部函数代码，堆栈区存储局部变量、函数的形式参数、函数调用时有关信息（返回地址），堆区用于动态分配内存。
- 因此，全局变量从程序执行就分配内存，程序执行结束后才销毁；而局部变量随函数的执行而出现，随函数结束而销毁

变量的存储修饰符

- 局部变量可以加 **auto**、**static**、**register**指示其存储位置和生存期。

```
void f(){  
    auto char c;  
    static double f; register int i;  
}
```

- **auto**局部变量具有自动生存期，可以省略**auto**函数调用时产生，函数调用结束时销毁
- **static**变量,函数第一次执行遇到时初始化，程序结束时才销毁。

变量的存储修饰符

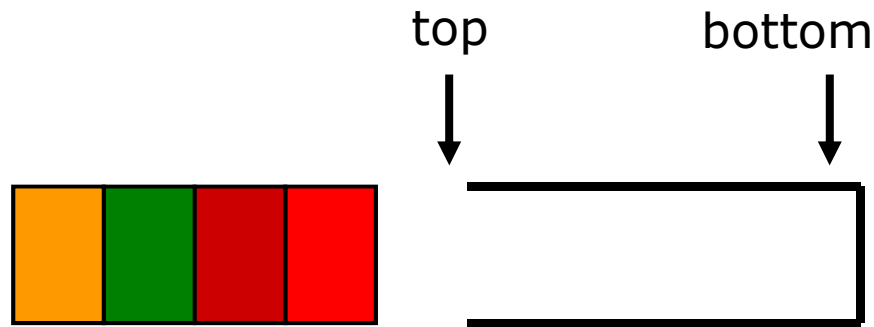
- **Register**建议编译器将该局部变量存储在寄存器中，以提高执行速度。但编译器不一定这么做，因为寄存器很少。

```
#include <iostream>
using namespace std;
int z = 0;
void f(){
    int x = 0;
    static int y = 0;
    x++;y++;z++;
    cout<<"x="<<x<<" ,y="<<
        <<y<<" ,z="<<z<<endl;
}
```

```
int main(){
    f();
    z++;
    f();
    return 0;
}
```

基于栈的函数调用实现

- 函数调用是通过“**栈**”这种数据结构实现的。栈是一种元素个数可变的**先进后出**的线性数据结构，而**队列**则是**先进先出**的线性数据结构。



基于栈的函数调用实现

```
void f1(int x1){  
    int a1;  
    //...  
}  
void f2(int x2){  
    int a2;  
    //...  
    f1(1);  
    //...  
}
```

```
void f3(int x3,int x4)  
{  
    int a3;  
    //...  
}  
int main(){  
    int a;  
    f2(2);  
    f3(3,4);  
    return 0;  
}
```

基于栈的函数调用实现

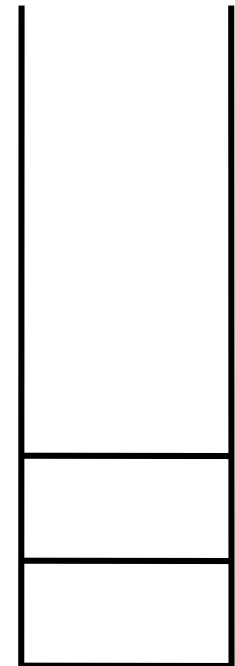
```
void f1(int x1){
    int a1;
    //...
}
void f2(int x2){
    int a2;
    //...
    f1(1);
    //...
}
```

```
void f3(int x3,int x4)
{
    int a3;
    //...
}
int main(){
    int a;
    f2(2);
    f3(3,4);
    return 0;
}
```

刚进入main函数：

Top → a

main返回地址



基于栈的函数调用实现

```
void f1(int x1){  
    int a1;  
    //...  
}  
void f2(int x2){  
    int a2;  
    //...  
    f1(1);  
    //...  
}
```

```
void f3(int x3,int x4)  
{  
    int a3;  
    //...  
}  
int main(){  
    int a;  
    f2(2);  
    f3(3,4);  
    return 0;  
}
```

进入f2函数, 调用
f1前:

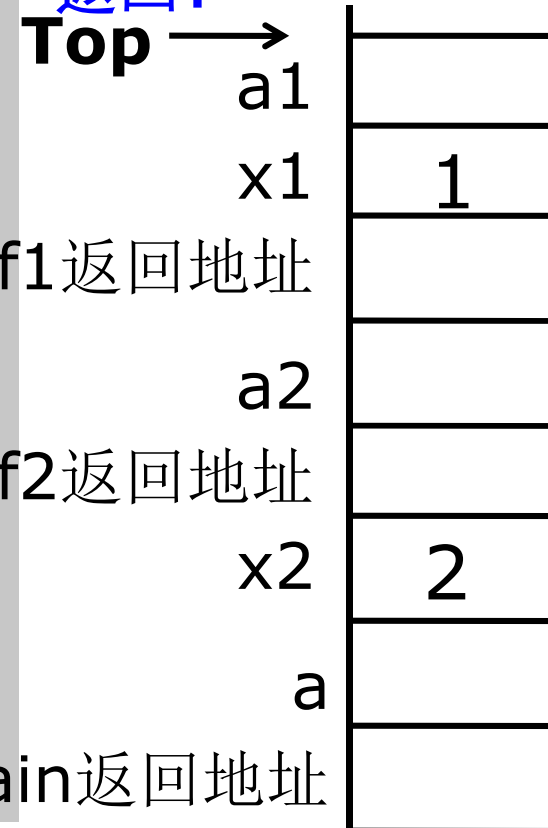


基于栈的函数调用实现

```
void f1(int x1){  
    int a1;  
    //...  
}  
void f2(int x2){  
    int a2;  
    //...  
    f1(1);  
    //...  
}
```

```
void f3(int x3,int x4)  
{  
    int a3;  
    //...  
}  
int main(){  
    int a;  
    f2(2);  
    f3(3,4);  
    return 0;  
}
```

进入f1函数, 但没
返回:



基于栈的函数调用实现

```
void f1(int x1){  
    int a1;  
    //...  
}  
void f2(int x2){  
    int a2;  
    //...  
    f1(1);  
    //...  
}
```

```
void f3(int x3,int x4)  
{  
    int a3;  
    //...  
}  
int main(){  
    int a;  
    f2(2);  
    f3(3,4);  
    return 0;  
}
```

F1返回到f2, f2未结束:

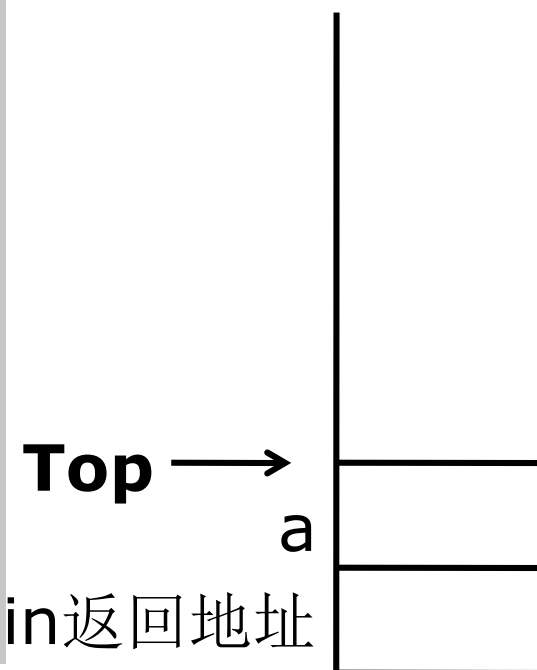


基于栈的函数调用实现

```
void f1(int x1){  
    int a1;  
    //...  
}  
void f2(int x2){  
    int a2;  
    //...  
    f1(1);  
    //...  
}
```

```
void f3(int x3,int x4)  
{  
    int a3;  
    //...  
}  
int main(){  
    int a;  
    f2(2);  
    f3(3,4);  
    return 0;  
}
```

F2结束，未进入f3时：



基于栈的函数调用实现

```
void f1(int x1){  
    int a1;  
    //...  
}  
void f2(int x2){  
    int a2;  
    //...  
    f1(1);  
    //...  
}
```

```
void f3(int x3,int x4)  
{  
    int a3;  
    //...  
}  
int main(){  
    int a;  
    f2(2);  
    f3(3,4);  
    return 0;  
}
```

进入f3, f3未结束:



基于栈的函数调用实现

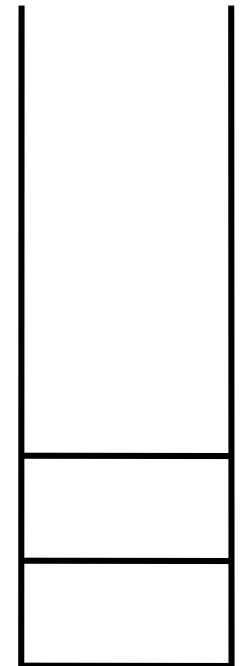
```
void f1(int x1){  
    int a1;  
    //...  
}  
void f2(int x2){  
    int a2;  
    //...  
    f1(1);  
    //...  
}
```

```
void f3(int x3,int x4)  
{  
    int a3;  
    //...  
}  
int main(){  
    int a;  
    f2(2);  
    f3(3,4);  
    return 0;  
}
```

F3结束, main未结束:

Top → a

main返回地址



基于栈的函数调用实现

```
void f1(int x1){  
    int a1;  
    //...  
}  
void f2(int x2){  
    int a2;  
    //...  
    f1(1);  
    //...  
}
```

```
void f3(int x3,int x4)  
{  
    int a3;  
    //...  
}  
int main(){  
    int a;  
    f2(2);  
    f3(3,4);  
    return 0;  
}
```

main结束:

标识符的作用域

- 程序中的每个实体都要定义，包括常量、变量、函数、类、对象以及语句标号等。定义它们时需要给它们起名字。
- 不同的实体一般要取不同名字才能区分它们、不产生冲突。
- 但如果不同的人编写的不同函数内，对某个对象(变量)起了同样的名字，会不会冲突呢？
- 答案：不冲突！C++给每个标识符都规定了作用域，不同作用域的对象名字相同是允许的。标识符只存在于它定义的作用域中。

标识符的作用域

- 局部作用域、全局作用域、文件作用域、函数作用域、函数原型作用域、类作用域、名字空间作用域
- 局部作用域：函数定义或复合语句定义的程序块。
函数的局部变量、形式参数等

```
void f(int y){  
    y = 3;  
    x = 2; //未定义  
    int x; //x从这里开始  
    x = y;  
}
```

```
void g(double y){  
    double x = 3.;  
    f(1); //OK, 函数名在全局作用域  
    //...  
}
```

标识符的作用域-全局作用域

- 程序包含的所有源文件范围。有全局函数、全局变量、全局类型。可以在程序的任何地方使用它们，程序结束时才不存在。
- 文件A中假如有一个全局变量(如 `int x`)，要在文件B中使用它，可以在文件B使用它之前声明一下。

`extern int x;`

- 文件中声明的全局变量一定要在某个文件中定义过！

标识符的作用域-文件作用域

- 全局标识符前加“**static**”关键字的，该标识符只能存在于该文件作用域中。其他文件不能访问。注意：不同于局部静态变量，那里的**static**是为了指定变量采用静态存储分配。

标识符的作用域-名字空间作用域

- 为防止全局名字冲突，C++引入名字空间(namespace)，不同名字空间的名字互不冲突

```
namespace A{  
    void f(){}  
}  
namespace B{  
    void f(){}  
}  
void f(){}
```

```
int main(){  
    f();  
    A::f();  
    B::f();  
    return 0;  
}
```

```
using namespace A;  
int main(){  
    f();//全局f还是A::f?  
    B::f();  
    return 0;  
}
```

标识符的作用域-无名名字空间

- 无名名字空间的变量只在该源文件中使用，类似于文件的静态(static)全局变量。

```
namespace{  
    int x,y;  
}
```

相当于

```
static int x,y;
```

递归函数：调用自身的函数

- 编写程序求第n个fibonacci数。

$$\text{fib} = \begin{cases} 0 & (n=1) \\ 1 & (n=2) \\ \text{fib}(n-2) + \text{fib}(n-1) & (n>3) \end{cases}$$

```
int fib(int n){  
    if(n==1) return 0;  
    if(n==2) return 1;  
    return fib(n-2)+fib(n-1);  
}
```

递归函数：调用自身的函数

- 一个问题可以分解为若干个子问题，而子问题的性质和原问题是一样的，只是规模不一样。
- 递归采用“分而治之”的方法解决问题。
- 例如：求 $n!$ 的问题等价于求 $(n-1)!$ 和 n 的积。即 $n! = n * (n-1)!$

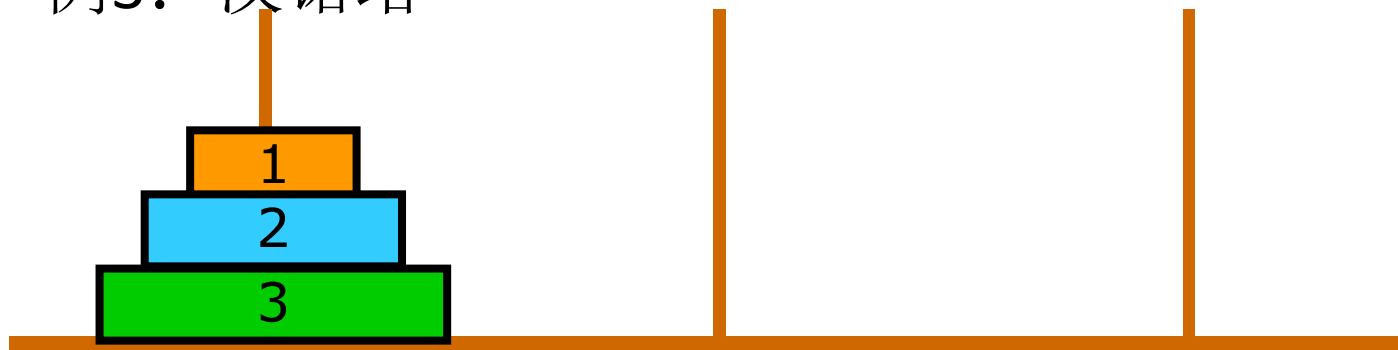
```
int f(int n){  
    if(n==0) return 1;  
    return n*f(n-1)  
}
```

递归与循环

- 一些问题可能用递归更为自然，而循环则比较复杂
- 递归是一种函数调用，可能受到程序的堆栈空间限制，且函数调用可能需要开销比较大。
- 递归往往包含问题的分解与综合，而不仅仅是分解。即需要对小问题的解进行综合加工：如 $\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1)$

递归算法的应用

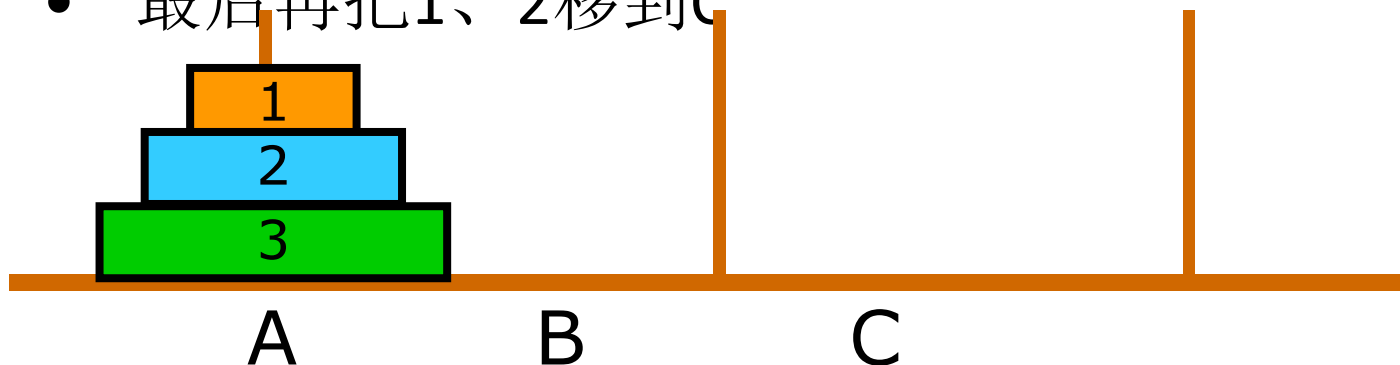
- 例3：汉诺塔



- 每次只允许移动一个盘子
- 必须保证小盘子在大盘子之上
- 如何把所有的盘子从A移到C?

递归算法的应用

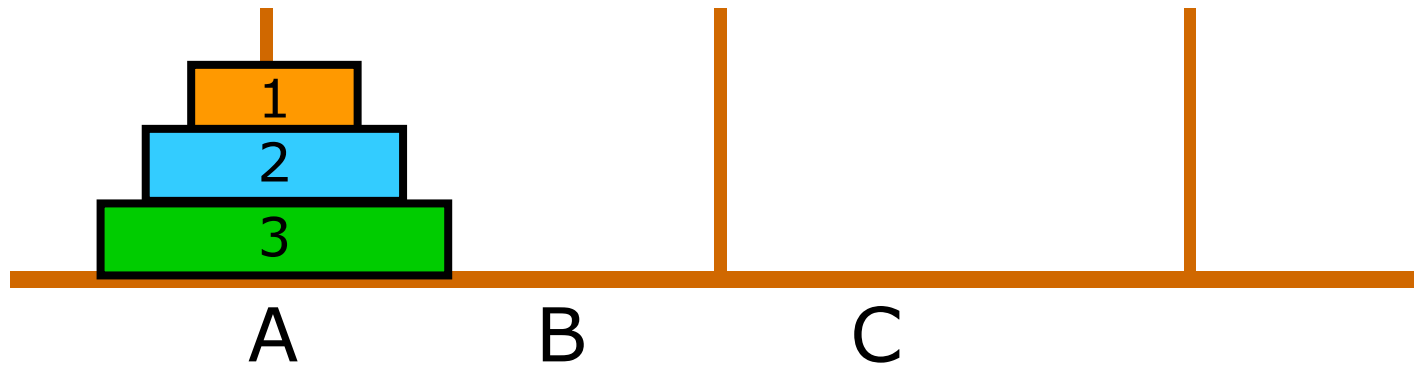
- 这个问题粗看似乎没有有递归的特点
- 但 n 个盘子的移动可分解为 $n-1$ 个盘子和1个盘子的移动问题：
 - 我们把1、2看成一个整体，假设能够把1、2移到B（至于怎么移动这个整体，以后再说）
 - 这时3上面没有盘子，可以直接把3移到C
 - 最后再把1、2移到C



递归算法的应用

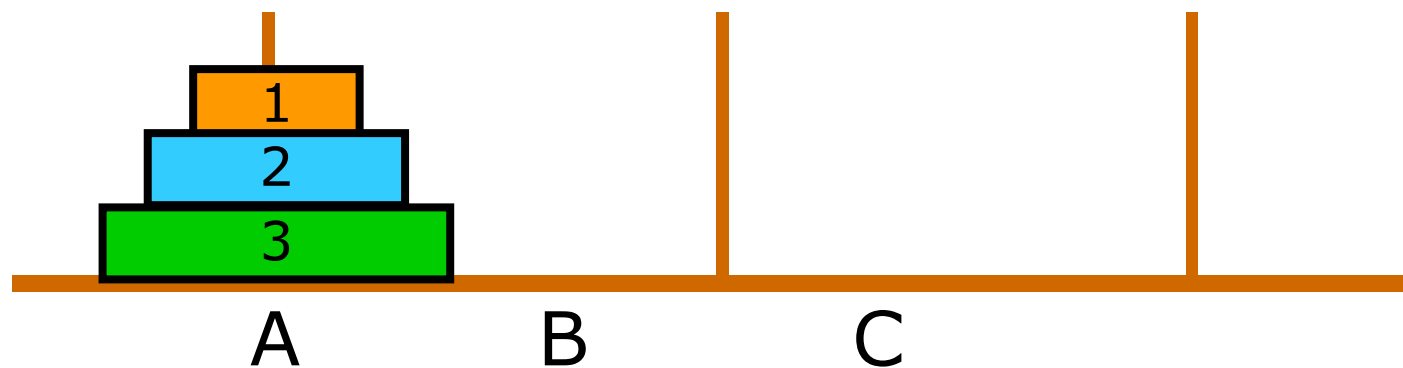
这样移动 n 个盘子的问题就简化为：

- (1)用C柱做过渡,将A柱上的 $n-1$ 个盘子移到B上
- (2)把A柱上最下面的盘子直接移到C柱上
- (3)用A柱做过渡,将B柱上的 $n-1$ 个盘子移到C上



递归算法的应用

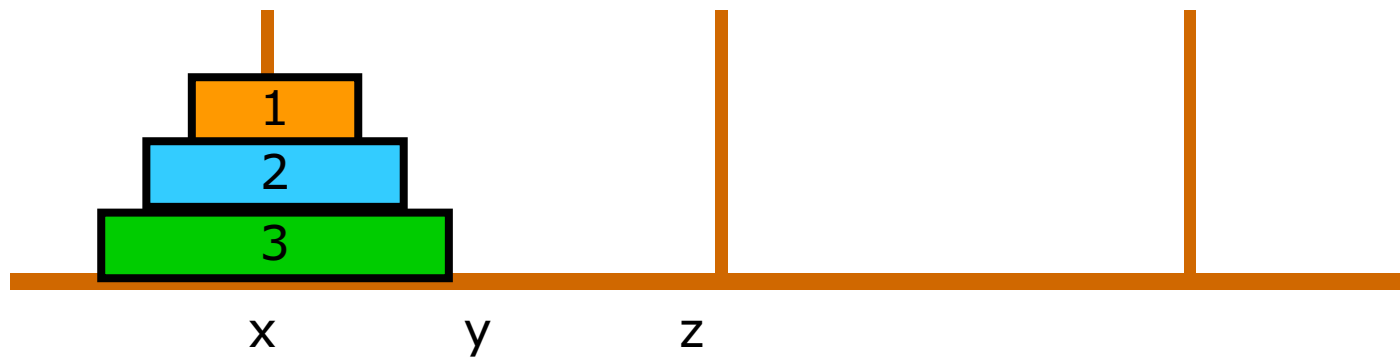
- 移动1,2个盘采用同样的方法
 - 先把上面的1个盘子移到一个过渡柱子上
 - 然后把最下面的1个盘子移到目标柱子上
 - 最后把上面的1个盘子移到目标柱子上\



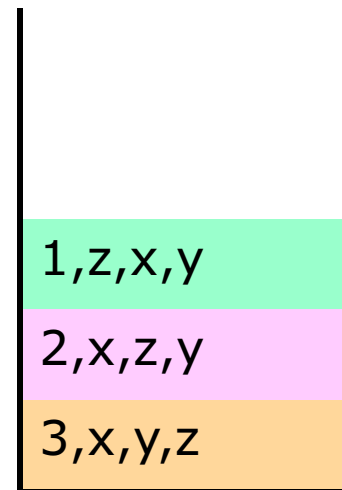
递归算法的应用

- 递归算法

```
void hanoi
(int n, char x, char y, char z){
    if (n==1)  move(x, 1, z);
    else {
        hanoi(n-1, x, z, y);
        move(x, n, z);
        hanoi(n-1, y, x, z);
    }
}
```



```
void hanoi  
(int n, char x, char y, char z){  
    if (n==1)    move(x, 1, z);  
    else {  
        hanoi(n-1, x, z, y);  
        move(x, n, z);  
        hanoi(n-1, y, x, z);  
    }  
}
```



运行栈

函数重载

- C语言要求同一作用域中的函数名不能相同，而C++则允许同一作用域函数名同名，只要其参数列表不同（个数或类型不同）。

```
void print(int i){...}
```

```
void print(double d){...}
```

```
void print(char c){...}
```

```
void print(string s){...}
```

- 同名函数不同参数列表的特性叫做**函数重载**。

函数重载的绑定

- 在编译时刻由编译器根据实际参数和形式参数的匹配情况决定使用那一个函数。
- 编译时刻的绑定称为**静态绑定**。
- 实际参数和形式参数匹配有：精确匹配、提升匹配、标准转换匹配、自定义转换匹配。
- 精确匹配：类型完全一样或只是微不足道的转换。

`print(1);`//精确匹配 `void print(int i)`

`print(1.0);`//精确匹配 `void print(double d)`

`print('a');`//精确匹配 `void print(char c)`

精确匹配

- 无须任何类型转换或只做平凡转换（如数组名到指针、函数名到函数指针、T到`const T`）

提升匹配

- 对实参进行提升转换与形参匹配。
- 整数提升：char到int、bool到int、short到int
- float 到double、double 到long double

标准转换匹配

- 标准转换包括：
 - a) 任何算术类型相互转换:int到double、T*到void*、
 - b)枚举类型到任何算术类型
 - c) 派生类指针可转换为基类指针。

自定义类型转换

- 以后讲

带默认值的形式参数

- `void print(int value ,int base);`
- 有默认值的参数一律紧靠在形参表右边
`void f(int a=1,int b=3,int c=4);//ok!`
`void f(int a=1,int b,int c=4);//bad!`
- 默认参数只能出现在函数声明中。

内联函数

- 函数调用是一种开销：保护调用现场、进行参数传递、执行调用指令、为局部变量分配存储、执行返回指令。
- 对小函数的频繁调用会降低程序效率。
- C++提供了两种解决方法：宏定义和内联函数
- 宏定义

```
#define max(a,b) a>b?a:b
```

```
int main(){ int x=3,y=9;
```

```
    max(x,y); //宏定义直接文字替换成  
              //a>b?a:b
```

```
}
```

内联函数

- 宏定义有一些缺点：重复计算、引起错误
- 内联函数可避免宏定义的缺陷，却具有宏定义一样的高效率。
- 函数定义前加上**inline**关键字，函数就成为内联函数。

```
inline int swap(int x,int y){  
    int t=x;x=y;y=t;}  
}
```

- 编译器对内联函数的调用会用其代码展开，因此效率高。

内联函数

- 编译器不总是展开内联代码，对于包含循环的内联函数或程序代码长的函数，编译器可能只是将它作为普通函数，因此，此时效率就不会提高。

条件编译

- 编译预处理命令不是语言的一部分，只是为了对编译过程进行指导，预处理命令有：#include,#define, 条件编译指令(#ifdef ...)
- 例：

```
#ifdef <宏名> // #ifndef <宏名>
    <程序段1>
#else
    <程序段2>
#endif
```

```
#ifdef UNIX
    ...//UNIX代码
#else
    ...//其他环境代码
#endif
```

条件编译

- 条件编译的另一种格式:

```
#if <常量表达式1> // #ifdef <宏名> // #ifndef <宏名>
    <程序段1>
#elif <常量表达式2> // 表达式非0则编译
    <程序段2>
#else
    <程序段3>
#endif
```

条件编译

- 头文件的预处理保护：重复包含某个定义可能会带来问题，可以用头文件的预处理保护防止一个源文件多次包含一个头文件。

```
#ifndef MODULE1
```

```
#define MODULE1
```

```
....
```

```
#endif
```

- VC200x会自动生成预处理保护头。

条件编译

- 可以利用条件编译帮助程序调试。如

```
#ifdef DEBUG
.....//调式信息
#endif
```
- C++提供了一个宏assert(头文件<cassert>)来表示断言，当表达式为0时，会显示相应表达式、断言的源代码名及行号等诊断信息，并调用库函数abort终止程序的执行。表达式不为0时，则程序继续执行。

条件编译

- 例求n!

```
#include <cassert>
int f(int n){
    assert(n>=0);
    if(n==0) return 1;
    return n*f(n-1);
}
```

条件编译

```
#ifdef DEBUG
```

```
...
```

```
#endif
```

库函数

- C++语言标准库包含了C语言标准库的功能和C++的一些新功能。
- 为使用C++标准库的函数、类、变量，需要包含其声明的头文件。C语言头文件以.h为扩展名，而C++则不带.h。C语言相应头文件的C++对应文件去掉.h，开头加上字符c。如
stdio.h变为cstdio。

例题

- 用递归或循环编写计算 x^n 的值的函数

```
double f(double x,int n){  
    if(n==0) return 1;  
    return x*f(x,n-1);  
}
```

```
double f(double x,int n){  
    double ret = 1.;  
    for(int i=1;i<=n;i++)  
        ret *=x;  
    return ret;  
}
```

练习

- 下述程序结果是什么？

```
#include <iostream.h>
char *str = "global";
void Print (char *str){
    cout << str << '\n';
    { char *str = "local";
        cout << str << '\n';
        cout << ::str << '\n';
    }
    cout << str << '\n';
}
```

```
int main (void)
{
    Print("Parameter");
    return 0;
}
```

练习

- 定义一个内联函数isAlpha: 如果输入字符是字母返回1, 否则返回0.
- 写一个计算Ackermann函数Ack(m,n)值的函数。
Ack(m,n)定义如下($m \geq 0, n \geq 0$):

$Ack(0, n) = n + 1;$

$Ack(m, 0) = Ack(m - 1, 1);$

$Ack(m, n) = Ack(m - 1, Ack(m, n - 1)); \quad (m, n > 0)$