CS 1502H 程序设计思想与方法 (C++)

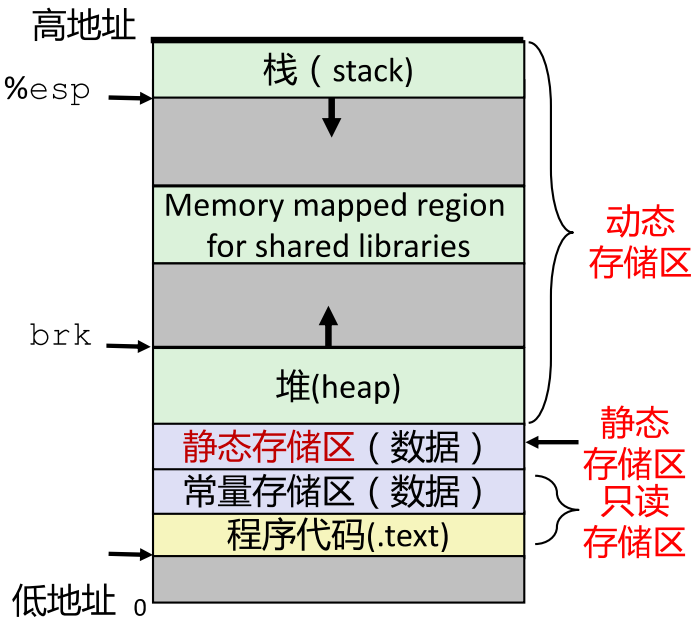
Chapter 6. 函数-3

陈东尧
上海交通大学
2025年秋季

内容回顾：变量的作用域和存储类型

- 全局变量
  - 静态全局变量 （只限本文件使用）
  - 外部变量 （非静态外部变量允许其他文件引用）
- 局部变量
  - 静态局部变量（离开函数，值仍保留）
  - 动态局部变量（自动变量） （离开函数，值就消失）

内容回顾：变量的作用域和存储类型



一个进程的虚拟内存地址空间

```
void a();     void b();
void c();     void d(int);
int main()
{ int x=6;
  cout <<"x in main is " << x << endl;
  a(); b(); c(); d(x);  x++;
  a(); b(); c(); d(x);
  cout << "x in main is " << x << endl; return 0;}
void a()
{ int x=25; cout << "x in a is "<< x << endl; }
void b()
{ static int x=50;
  cout << "x in b is "<<  x++ << endl; }
void c()
{ extern int x;
   x*=10; cout << "x in c is "<< x << endl; }
int x=2;
void d(int x)
{ cout << "x in d is " << ++x << endl; }
```

结果：

```
x in main is 6
x in a is 25
x in b is 50
x in c is 20
x in d is 7
x in a is 25
x in b is 51
x in c 200
x in d is 8
x in main is 7
```

无须解释
局部变量
静态局部变量
外部变量（引用了在本函数外的全局变量）
main中变量的值传递
局部变量
静态局部变量，在代码生存周期中不会第二次初始化
外部变量
main中变量的值传递
main中变量

本章大纲

- 函数的用途和定义
- 函数的使用
- 变量的作用域与存储类别
- 函数参数—数组和默认值
- 内联、重载、模版、递归

数组作为函数的参数

- 设计一函数，统计10位同学的平均成绩，返回值是平均成绩。
- 设计考虑：如何传递参数？
  - 参数是10位同学的考试成绩，可以用10个整型数来表示，所以有10个整型的形式参数；
  - 一组同类数据可以用一个数组来描述，所以参数也可以是一个10个元素的整型数组；
- 第二种方法更加简练；

```
int average (int array[10])
{   int i, sum = 0;
    for (i = 0; i < 10; ++i)
        sum += array[i];
    return sum / 10;
}
```

Average函数的使用

```
int main()
{   int i, score[10];
    cout << "请输入10个成绩：" << endl;
    for ( i = 0; i < 10; i++)  cin >> score[i];
    cout << "平均成绩是：" << average(score)
         << endl;
    return 0;
}
```

注意：形式参数是数组，实际参数也是一个数组

一个有趣的现象

```
int main()
{   int i, score[10];
    cout << "请输入10个成绩： " << endl;
  for ( i = 0; i < 10; i++)  cin >> score[i];
    cout << “平均成绩是： ” <<
          average(score) << endl;
    cout << score[3];
    return 0;
}
```

```
int average (int array[10])
{   int i, sum = 0;
    for (i = 0; i < 10; ++i)
       sum += array[i];
    array[3] = 90;
    return sum / 10;
 }
```

程序运行结束后，score[3]中是什么？

注意事项⚠：数组参数的传递机制

- C++语言规定，数组名是数组的起始地址；
- 参数传递时，实际参数是数组名，形式参数也是数组名；
- 按照值传递，当用实际参数score调用函数 average时，是用score 初始化形式参数数组array。如 score 的首地址为1000，在函数中形参数组array的首地址也为1000。
- 形式参数和实际参数是同一数组！

注意事项⚠：数组参数的传递机制（续）

- 在函数中并没有定义新的数组
- 对形式参数数组指定规模是没有意义的，形式参数数组不需要指定大小，所以方括号中为空
- 函数如何知道数组的规模？用另一个整型参数表示
- 总结：数组传递需要两个参数
  - 数组名
  - 数组规模

默认参数

- 对于某些函数，程序往往会用一些固定的值去调用它。例如对于以某种数制输出整型数的函数print：

  void print(int value, int base);

  在大多数情况下都是以十进制输出，因此base的值总是为10。

- C++在定义或声明函数时可以为函数的某个参数指定默认值。
- 当调用函数时没有为它指定实际参数时，系统自动将默认值赋给形式参数。例如，可以将print函数声明为

  void print(int value, int base=10);

  调用print(20) 等价于 print(20, 10)

带有默认参数的函数的使用

C++在说明函数原型时，可以为一个或多个参数指定缺省值。调用此函数时，若缺省某一参数，C++自动以缺省值作为此参数的值。如：

```
int special(int x=2, float y=1.5);
```

调用时可用：

```
special(5, 3.2);   //x=5; y=3.2
special(6);          //x=6; y=1.5
special( );          //x=2; y=1.5
```

注意事项⚠：带有默认参数的函数使用注意事项

- 缺省参数无论有几个，都必须放在参数序列的最后；
- 在函数调用时，若某个参数省略，则其后的参数皆应省略而取其缺省值。
- 对参数默认值的指定只有在函数声明处有意义。因为函数的默认值是提供给调用者使用的。
- 因此编译器禁止声明和定义时同时定义缺省参数值,可以利用声明来随意更改默认参数的值。
- 在不同的源文件中，可以对函数的参数指定不同的默认值。
  - 例如对于上面的print函数，如果输出的大多是十进制数，那么在对应的源文件中可以指定base的默认值为10。
  - 经常要以二进制输出，那么在对应的源文件中可以指定默认值是2。

课堂练习

- 下列说法错误的是（ ）。

  A. 变量的作用域是指变量的作用范围，即在程序中可以被读写访问的区域，它取决于变量被定义的位置。

  B. 局部变量与全局变量同名时，全局变量隐藏局部变量，即全局变量起作用，局部变量不起作用。

  C. 形参也是局部变量，形参变量和实参变量的作用域是不同的，因此形参变量和实参变量同名时，二者互不干扰。

  D. 只要同名的变量出现在不同的作用域内，二者互不干扰，编译器有能力区分不同作用域中的同名变量

课堂练习

有如下的函数定义：

```
int Xfun(int x)
{
    int y = x;
    {int x = 10; y += x;}
    return x + y;
}
```

通过表达式Xfun(5)调用该函数，则得到的返回值是：（ ）

A. 20　B. 10　C. 15　D. 25

本章大纲

- 函数的用途和定义
- 函数的使用
- 变量的作用域与存储类别
- 函数参数—数组和默认值
- 内联、重载、模板、递归

函数的高阶使用技巧

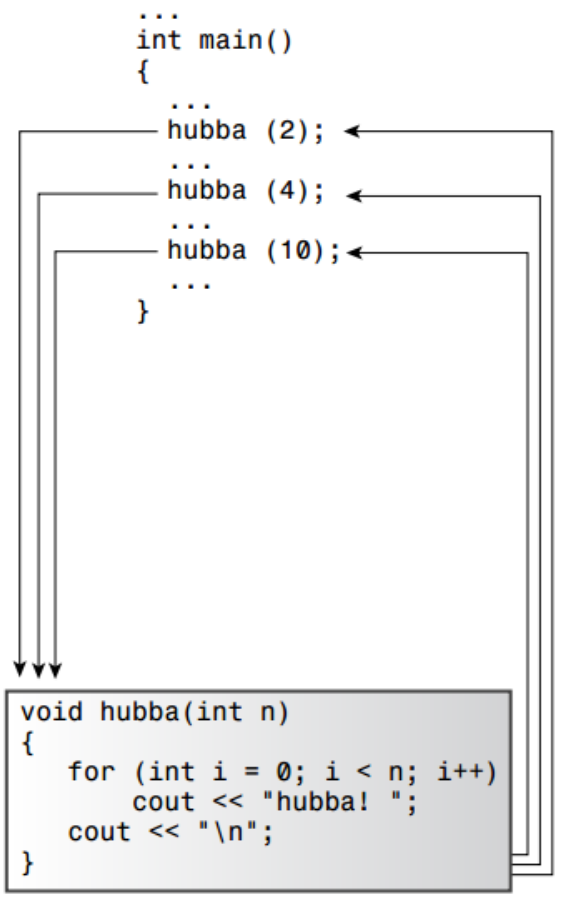
- 内联
- 重载
- 模板
- 递归

内联函数

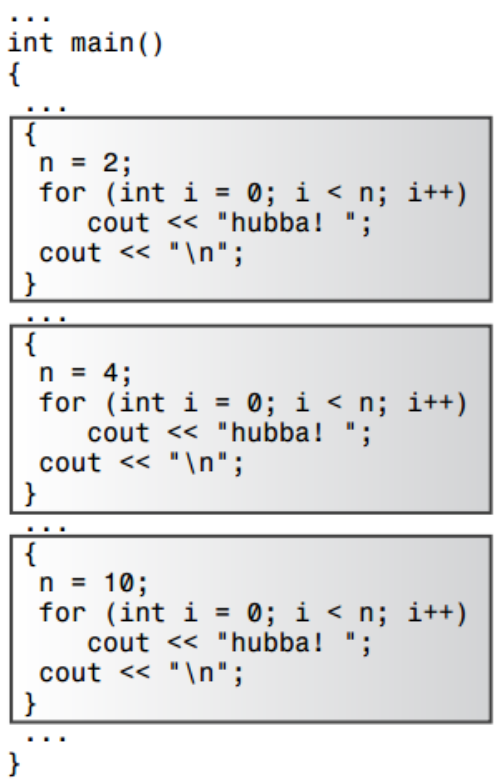
- 函数调用存在开销，需要创建函数的帧
- 内联函数可用于提高执行效率
- 在调用内联函数时，编译器直接用内联函数的代码替换函数调用，于是省去了函数调用的开销

内联函数的本质

- 内联函数用内联代码来替换函数调用



常规函数将程序流程转到被调用的函数



内联函数用内联代码替换函数调用

内联函数示例

- 内联函数的定义：在函数头部前加保留词inline

```
#include<iostream>
inline int cube(int s)
{  return s*s*s; }
int main()
{     int n;
      cin >> n;
      cout << cube(n) << endl;
      return 0;
}
```

注意事项⚠：内联函数使用注意事项

- 内联函数必须定义在被调用之前
- 内联以代码复制为代价，省去了函数调用的开销，提高函数的执行效率。如果相比于执行函数体内代码的时间，函数调用的开销可以忽略不计，那么效率的收获会很小
- 内联函数不能递归

注意事项⚠：内联函数使用注意事项

- 内联函数必须定义在被调用之前
- 内联以代码复制为代价，省去了函数调用的开销，提高函数的执行效率。如果相比于执行函数体内代码的时间，函数调用的开销可以忽略不计，那么效率的收获会很小
- 内联函数不能递归

以下情况不宜用内联:

- 如果函数体代码比较长，内联将导致内存消耗代价较高。
- 如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。

函数的高阶使用技巧

- 内联
- 重载
- 模板
- 递归

重载函数（Function overloading）

- 在传统的C语言中，不允许出现同名函数。当要求写一组功能类似、参数类型或参数个数不同的函数时，必须给它们取不同的函数名
- C++中允许参数个数不同、参数类型不同或两者兼而有之的两个以上的函数取相同的函数名
- 函数重载的目的是提高函数的易用性。这是面向对象编程的特点之一

重载函数（Function overloading）

- 例如某个程序要求找出一组数据中的最大值，这组数据可能有2-5个数据。
  - 那么我们须写四个函数：求两个值中的最大值、求三个值个中的最大值、求四个值中的最大值和求五个值中的最大值。
  - 我们必须为这四个函数取四不同的函数名，例如：max2, max3, max4和max5。
- 写四个函数的原因是：在C语言中，不允许出现同名函数。当要求写一组功能类似、参数类型或参数个数不同的函数时，必须给它们取不同的函数名。

重载函数（Function overloading）

- 使参数个数不同、参数类型不同或两者兼而有之的两个以上的函数取相同的函数名。
- 函数重载的目的是提高函数的易用性。
- 如

```
int max(int a1, int a2);
int max(int a1, int a2, int a3);
int max(int a1, int a2, int a3, int a4);
int max(int a1, int a2, int a3, int a4, int a5);
```

重载函数的实现

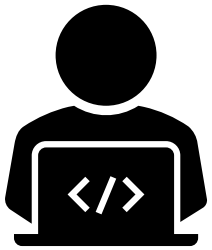
- 由编译器确定某一次函数调用到底是调用了哪一个具体的函数。这个过程称之为绑定（binding，又称为联编或捆绑）。
- 如何绑定？
  - 编译器首先会为这一组重载函数中的每个函数取一个不同的内部名字。
  - 当发生函数调用时，编译根据实际参数和形式参数的匹配情况确定具体调用的是那个函数，**将这个函数的内部函数名取代重载的函数名**。

Overloading 示例

```
void print(int i) {
cout << " Here is int " << i << endl;
}
void print(double f) {
cout << " Here is float " << f << endl;
}
void print(char const *c) {
cout << " Here is char* " << c << endl;
}

int main() {
print(10);
print(10.10);
print("ten");
return 0;
}
```

```
Here is int 10
Here is float 10.1
Here is char* ten
```



代码演示

Chapter6-function-overloading.cpp

函数的高阶使用技巧

- 内联
- 重载
- 模板
- 递归

函数模板

- 如果一组重载函数仅仅是参数的类型不一样，程序的逻辑完全一样，那么这一组重载函数可以写成一个函数模板。
- 所谓的函数模板就是实现类型的参数化（泛型化），即把函数中某些形式参数的类型定义成参数，称为模板参数。
- 在函数调用时，编译器根据实际参数的类型确定模板参数的值，生成不同的模板函数。

函数模板的定义

一般的定义形式

template<模板形式参数表>
返回类型 函数名(形式参数表)
{
　　//函数定义体
}

- 模板形式参数表可以包含基本数据类型，

```
template <typename T>
T myMax(T x, T y)
{
　　return (x > y)? x: y;
}
```

函数模板的定义

一般的定义形式

```
template<模板形式参数表>
返回类型  函数名(形式参数表)
{
        //函数定义体
}
```

- 模板形式参数表可以包含基本数据类型，也可以包含类类型（需加前缀class）

```
template<class T>
T myMax(T a, T b)
{ return a>b ? a : b;  }
```

函数模板的实例化

- 根据实际参数确定模板参数的值;
- 将模板参数的值代入函数模板，形成一个真正的函数。

如:

- maxNum = myMax(3, 7);
- maxChar = myMax(‘z’, ‘a’);
- maxDouble = myMax(3.5, 4.6);

函数模板的定义

一般的定义形式

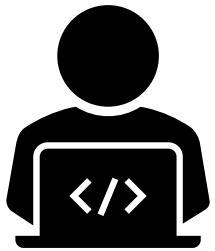
template<模板形式参数表>

返回类型 函数名(形式参数表)

{

//函数定义体

}



代码演示

Chapter6-function-template.cpp

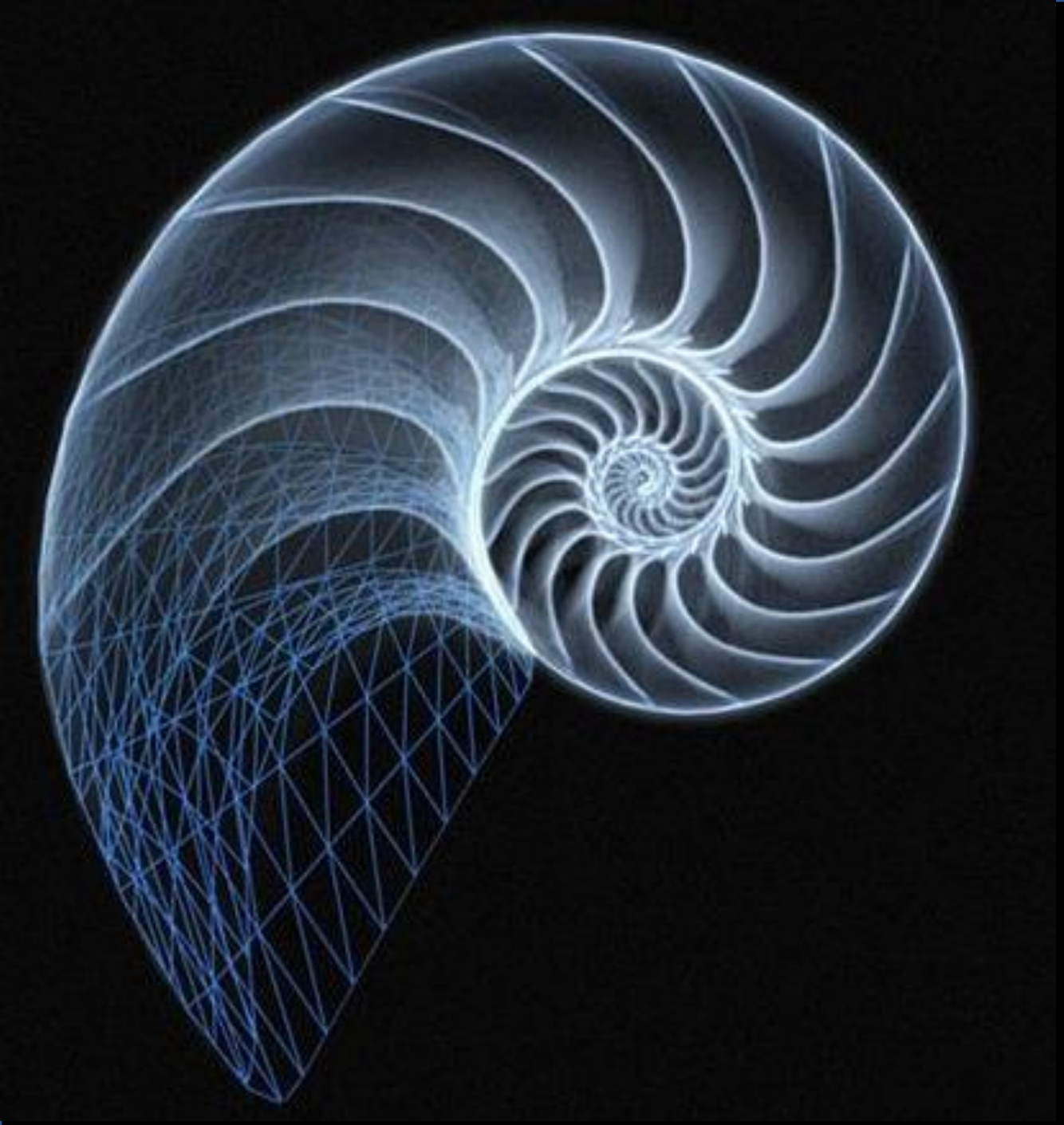
```
// One function works for all data types. This would work
// even for user defined types if operator '>' is overloaded
template <typename T>
T myMax(T x, T y)
{
        return (x > y)? x: y;
}

int main()
{
      cout << myMax<int>(3, 7) << endl; // Call myMax for int
      cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double
      cout << myMax<char>('g', 'e') << endl; // call myMax for char

  return 0;
}
```

函数的高阶使用技巧

- 内联
- 重载
- 模板
- 递归

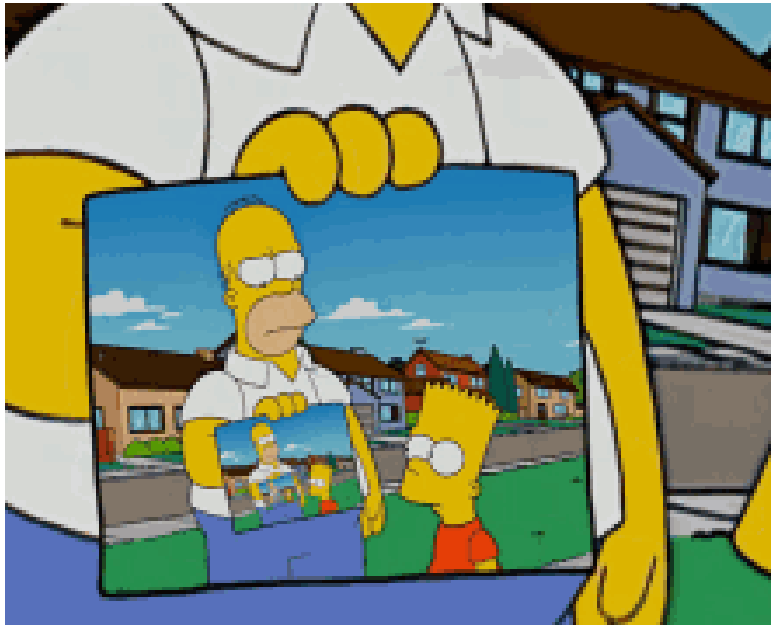


递归

Recursion

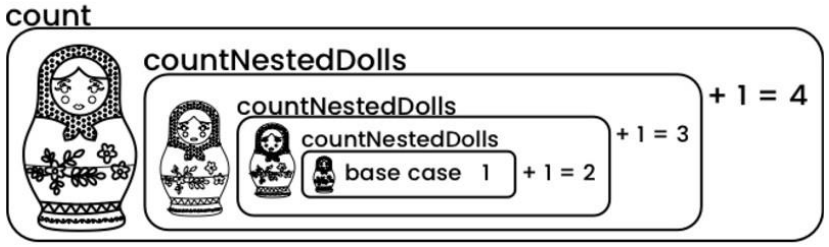
什么是递归

- 是指在函数的中使用函数自身的方法



递归在生活中非常广泛

镜中之镜



数套娃的数量?

甚至电影创作中

Lets Play a Game! 汉诺塔（Hanoi Tower）

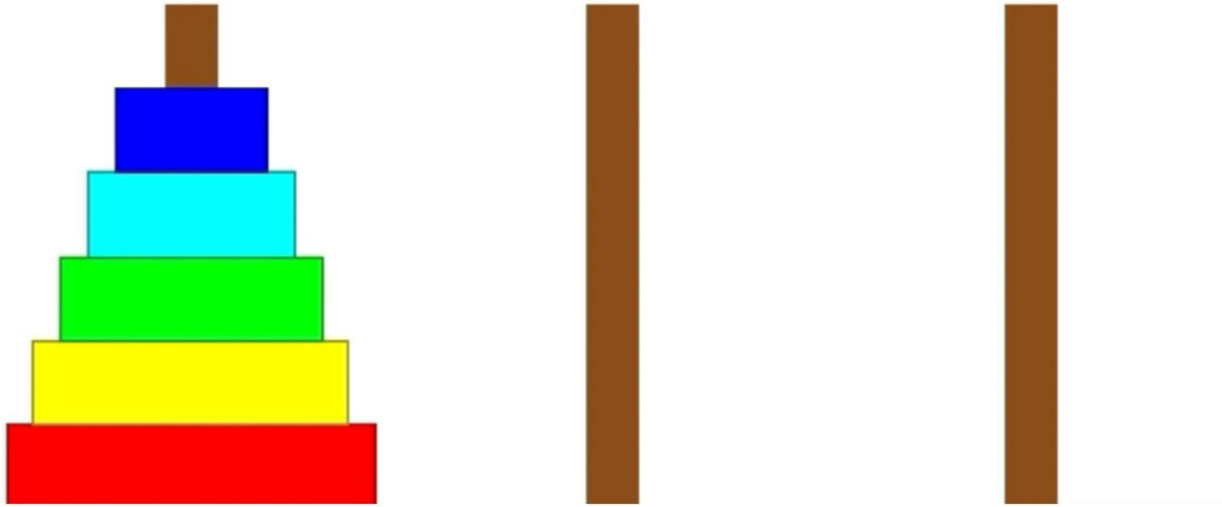
目标：

将最左的盘子全部移到最右

规则：

每次只能移动一个盘子

不允许大盘子放在小盘子上



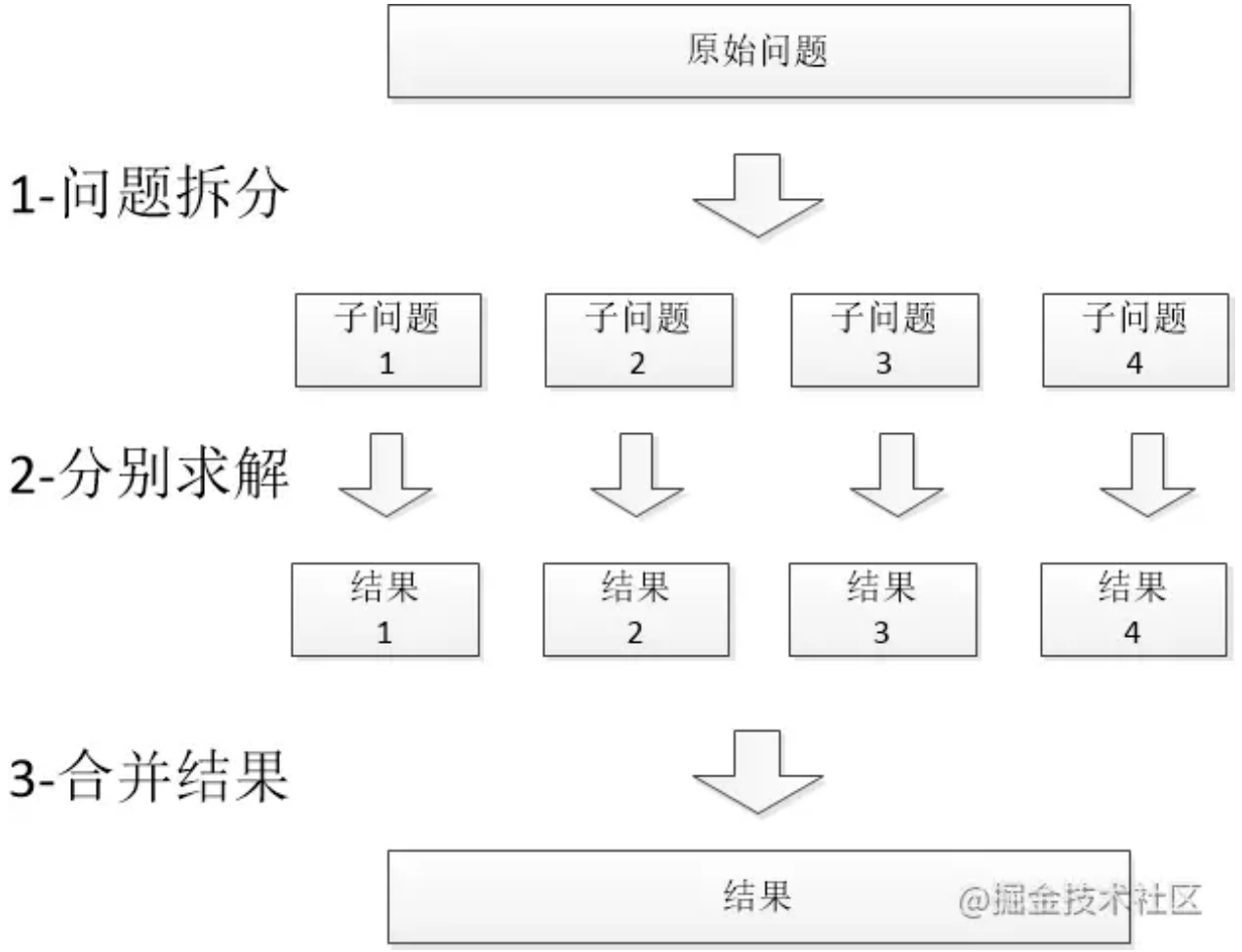
如何求解？

递归

- 分而治之（divide and conquer）思想。又称为分治法
- 递归程序设计：将一个大问题简化为同样形式的较小问题。
  - 在一个递归求解中，分解的子问题与最初的问题具有一样的形式

分而治之

- 例如计算斐波那契数列



递归

- 递归调用：在一个函数中直接或间接地调用函数本身。
- 递归函数的使用条件：
  - 必须有递归终止的条件
  - 函数有与递归终止条件相关的参数
  - 在递归过程中，决定终止条件的参数有规律地递增或递减

递归的标准模式

- 有可对函数的入口进行测试的基本情况

```
if（条件）
    return (不需要递归的简单答案);
else
    return (递归调用同一函数)；
```

分析阶乘（1）- 明确问题及终止条件

- 函数自己call自己
- 想想阶乘的递归写法

递归终止条件

$$n! = \begin{cases} 1 & (n = 0,1) \\ n \cdot (n-1)! & (n > 1) \end{cases}$$

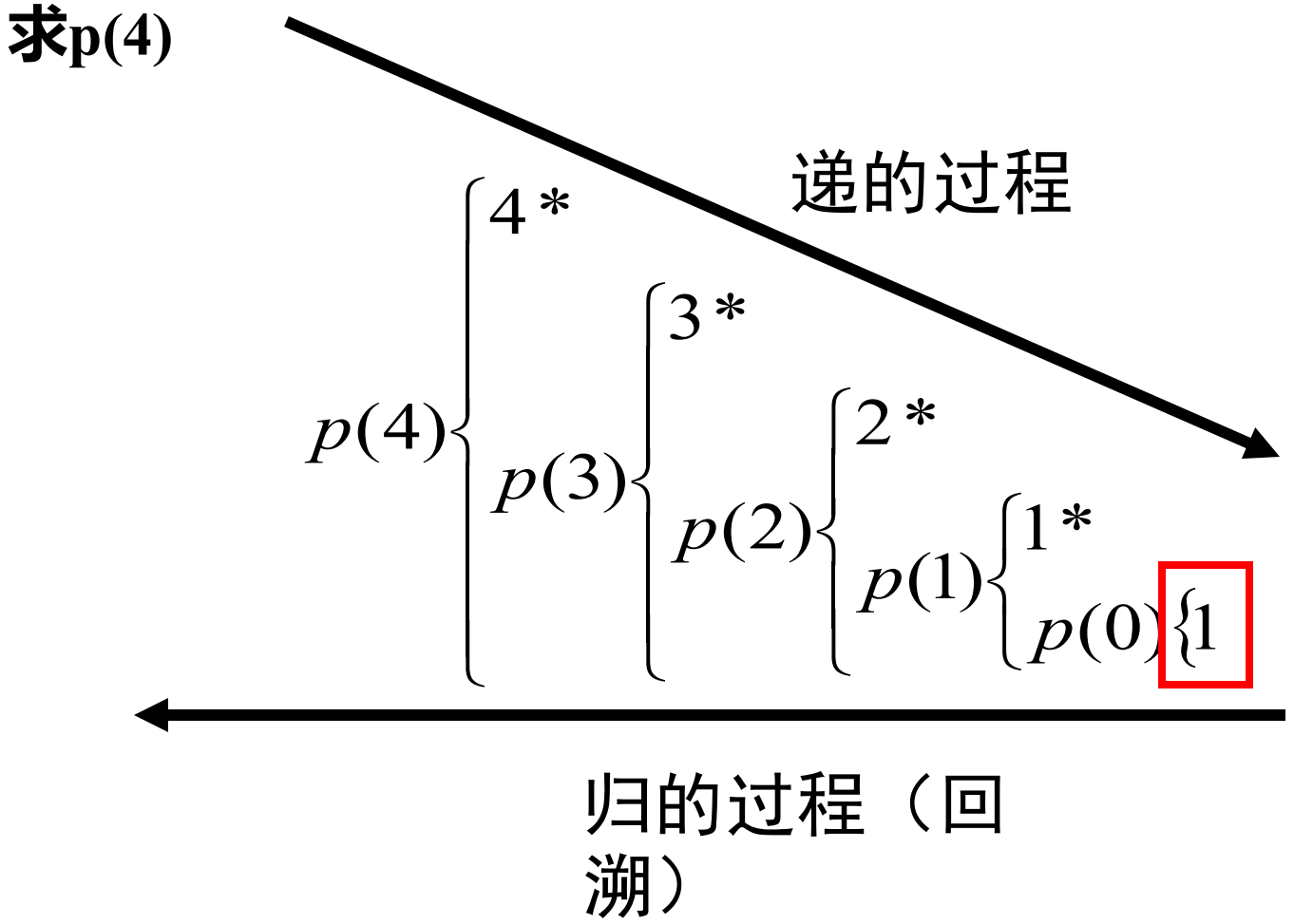
```
factorial(3)
3 * factorial(2)
3 * (2 * factorial(1))
3 * (2 * (1 * factorial(0)))
3 * (2 * (1 * 1))
3 * (2 * 1)
3 * 2
6
```

引用的例子：https://juejin.cn/post/6954905395655606286

分析阶乘（2）- 递归过程分析

- 函数自己call自己
- 想想阶乘的递归写法

$$n! = \begin{cases} 1 & (n = 0,1) \\ n \cdot (n-1)! & (n > 1) \end{cases}$$



分析阶乘（3）- 代码表述

- 函数自己call自己
- 想想阶乘的递归写法

递归终止条件

$$n! = \begin{cases} 1 & (n = 0,1) \\ n \cdot (n-1)! & (n > 1) \end{cases}$$

```
long  p(int n)
{
      if (n == 0) return 1;
      else return n * p(n-1);
 }
```

```
long  p(int n)
{
      if (n == 0) return 1;
      else return n * p(n-1);
 }

int main(){
            int res = p(10);
            return 0;
}
```

递归的好处

- 作为处理问题的工具，递归技术是一种非常有力的工具。
  - 利用递归不但可以使得书写复杂度降低
  - 而且使程序看上去更加美观

递归的实践1：求n的k次幂

1. 终止条件是什么？
2. 递归过程是怎样的？
3. 如何用程序来表述？

递归的实践1：求n的k次幂

- 大家来写一下吧

```
int nPowerK(int n, int k){
    if (k == 0) {
        return(1);
    } else {
        return( n * nPowerK( n, k - 1));
    }
}

int main(){
    cout<<"5 power 3 is: "<<nPowerK(5,3)<<endl;
}
```

递归的实践2：斐波那契数列

n	0	1	2	3	4	5	6
F(n)	0	1	1	2	3	5	8

1. 终止条件是什么？
2. 递归过程是怎样的？
3. 如何用程序来表述？

递归的实践2：斐波那契数列

- 终止条件
  - 0 和 1的情况

- 寻找通项

递归的实践2：斐波那契数列

- 终止条件
  - 0 和 1的情况

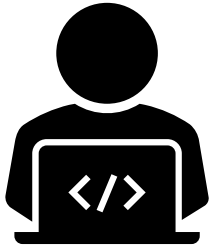
- 寻找通项

$$\begin{cases} F_n = F_{n-1} + F_{n-2} \\ F_0 = 0 \text{ and } F_1 = 1. \end{cases}$$

递归的实践2：斐波那契数列

- 通项

$$\begin{cases} F_n = F_{n-1} + F_{n-2} \\ F_0 = 0 \text{ and } F_1 = 1. \end{cases}$$



代码演示

chapter6-Recursion-fibonacci.cpp

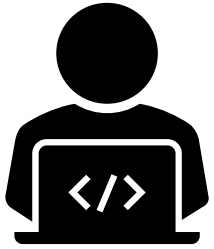
```
int fib(int n){
    if (n <= 1)
        return n;

}
```

递归的实践2：斐波那契数列

- 通项

$$\begin{cases} F_n = F_{n-1} + F_{n-2} \\ F_0 = 0 \text{ and } F_1 = 1. \end{cases}$$



代码演示

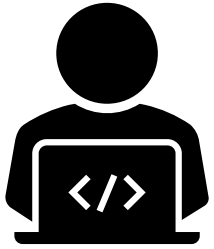
chapter6-Recursion-fibonacci.cpp

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

递归的实践2：斐波那契数列

- 通项

$$\begin{cases} F_n = F_{n-1} + F_{n-2} \\ F_0 = 0 \text{ and } F_1 = 1. \end{cases}$$



代码演示

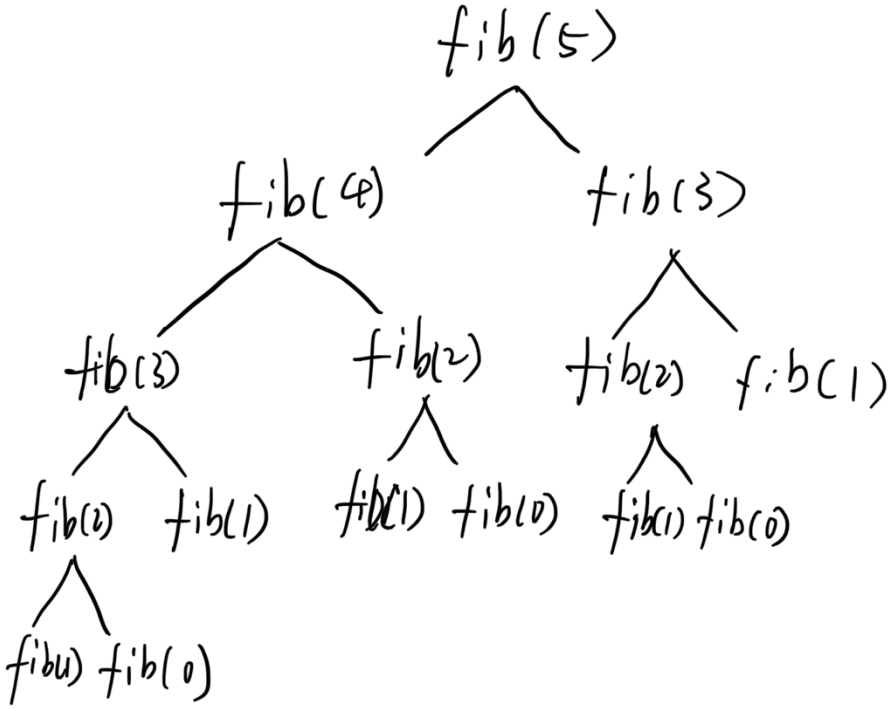
chapter6-Recursion-fibonacci.cpp

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

int main (){
    int n = 9;
    cout << fib(n)<<endl;
    return 0;
}
```

拆解Fib函数

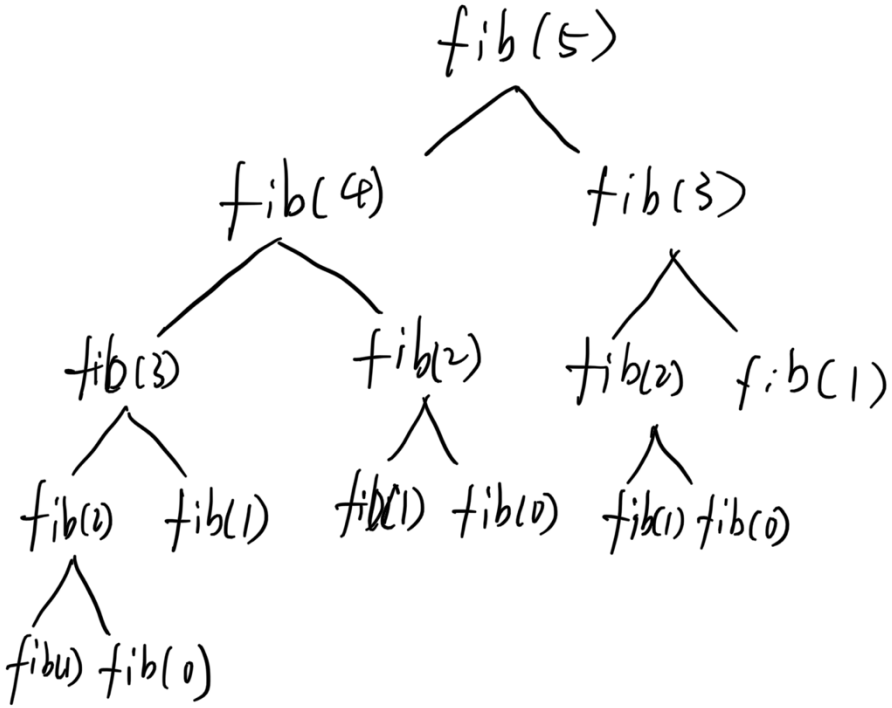
- 有哪些问题？
- 如何优化？

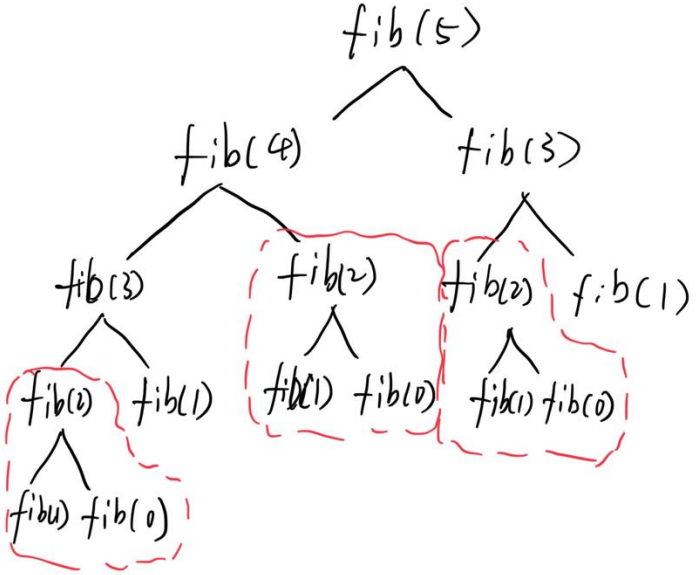


【Fib函数的各种写法，涵盖有各种复杂度】https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/

拆解Fib函数

- 有哪些问题？
- 如何优化？





有些步骤重复计算了

【Fib函数的各种写法，涵盖有各种复杂度】https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/

斐波那契函数的递归与迭代实现

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

int main (){
    int n = 9;
    cout << fib(n)<<endl;
    return 0;
}
```

```
int fib(int n){
int a = 0, b = 1, c, i;
if( n == 0)
  return a;
for(i = 2; i <= n; i++)
{
    c = a + b; //f(n) = f(n-1)+f(n-2)
    a = b; //f(n-2) = f(n-1)
    b = c; //f(n-1) = f(n)
}
return c;
}
```

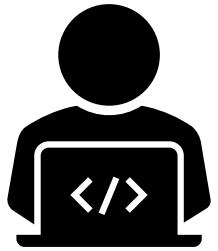
二分查找的递归实现

```cpp
int binarySearch(vector<int> &arr, int low, int high, int x) {
    if (high >= low) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == x)
            return mid;

        if (arr[mid] > x)
            return binarySearch(arr, low, mid - 1, x);

        return binarySearch(arr, mid + 1, high, x);
    }
  return -1;
}
```



代码演示

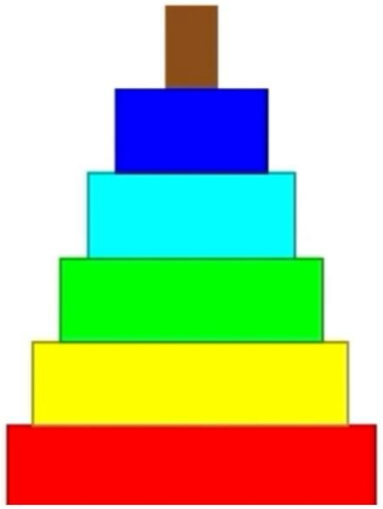
chapter6-binarySearch-recursion.cpp

递归与迭代的转换

- 对于大多数常用的递归都有简单、等价的迭代程序。究竟使用哪一种，凭经验选择。
- 迭代程序复杂，但效率高。
- 递归程序逻辑清晰，但往往效率较低。

对递归函数的每次调用都需要内存空间。由于很多调用的活动都是同时进行的，操作系统可能会耗尽可用的内存，避免在处理过大的n时产生溢出问题。

递归的实践3: 汉诺塔（Hanoi Tower）



A B C

目标：

将最左的盘子全部移到最右

规则：

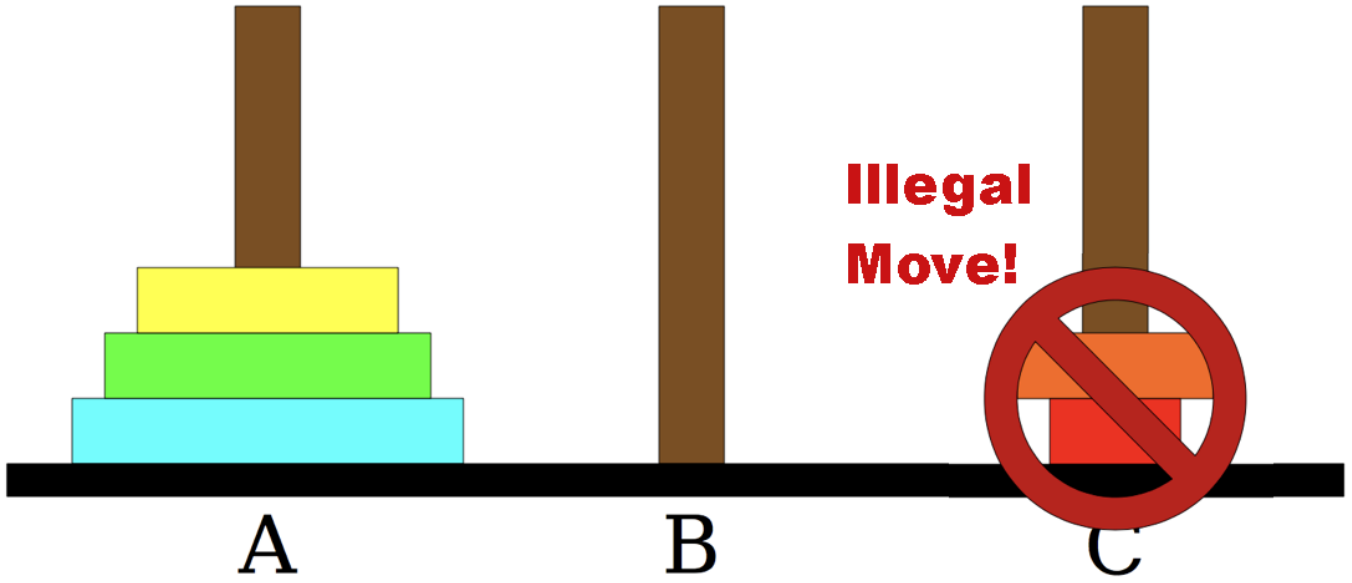
每次只能移动一个盘子

不允许大盘子放在小盘子上

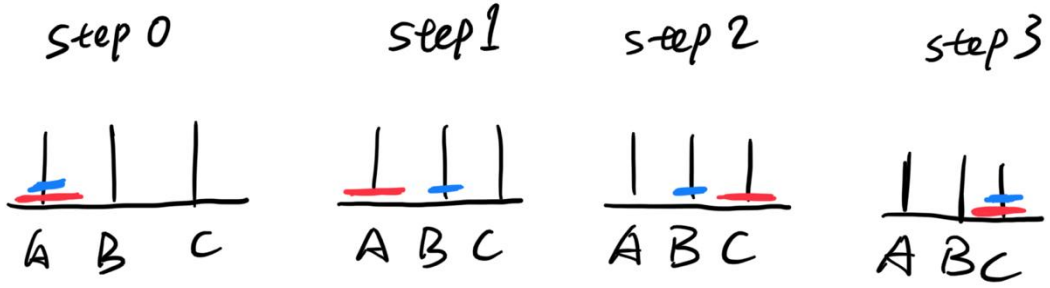
https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1206/lectures/intro-to-recursion/

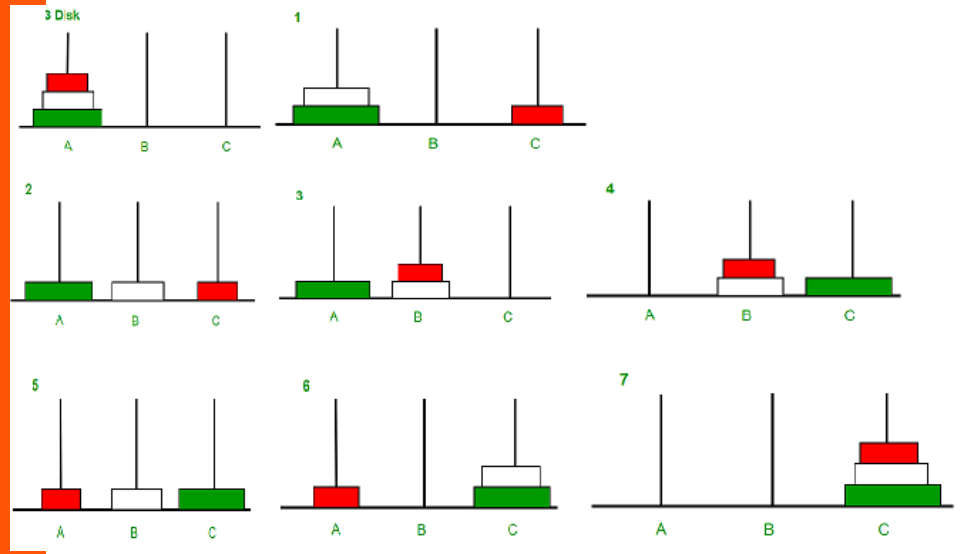
具体规则

- 每次只能移动一个盘子
- 不允许大盘子放在小盘子上



解决思路






```
Input : 2
Output : Disk 1 moved from A to B
         Disk 2 moved from A to C
         Disk 1 moved from B to C

Input : 3
Output : Disk 1 moved from A to C
         Disk 2 moved from A to B
         Disk 1 moved from C to B
         Disk 3 moved from A to C
         Disk 1 moved from B to A
         Disk 2 moved from B to C
         Disk 1 moved from A to C
```

解决思路

- 假设有 A、B、C 三个塔，A 塔有N 块盘，目标是把这些盘全部移到 C 塔。那么：

1. 先把 A 塔顶部的 N-1块盘移动到 B 塔；
2. 再把 A 塔剩下的大盘移到 C；
3. 最后把 B 塔的 N-1块盘移到 C

N-1看作一个整体，这是个子问题

只需挪一个盘

另一个子问题

1. 先把 A 塔顶部的 N-1块盘移动到 B 塔; ← 这是个子问题
2. 再把 A 塔剩下的大盘移到 C; ← 只需挪一个盘
3. 最后把 B 塔的 N-1块盘移到 C ← 另一个子问题

```
#include <iostream>
using namespace std;

void towerOfHanoi(int n, char A_rod, char C_rod, char B_rod){









}

int main()
{
	int n = 4; // Number of disks
	towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
	return 0;
}
```

1. 先把 A 塔顶部的 N-1块盘移动到 B 塔; ← 这是个子问题
2. 再把 A 塔剩下的大盘移到 C; ← 只需挪一个盘
3. 最后把 B 塔的 N-1块盘移到 C ← 另一个子问题

```
#include <iostream>
using namespace std;

void towerOfHanoi(int n, char A_rod, char C_rod, char B_rod){
	if (n == 1){
		cout << "Move disk 1 from rod " << A_rod <<" to rod "<< C_rod<<endl;
		return;
	}




}

int main()
{
	int n = 4; // Number of disks
	towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
	return 0;
}
```

1. 先把 A 塔顶部的 N-1块盘移动到 B 塔; ← 这是个子问题
2. 再把 A 塔剩下的大盘移到 C; ← 只需挪一个盘
3. 最后把 B 塔的 N-1块盘移到 C ← 另一个子问题

```cpp
#include <iostream>
using namespace std;

void towerOfHanoi(int n, char A_rod, char C_rod, char B_rod){
	if (n == 1){
		cout << "Move disk 1 from rod " << A_rod <<" to rod "<< C_rod<<endl;
		return;
	}
	towerOfHanoi(n - 1, A_rod, B_rod, C_rod);


}

int main()
{
	int n = 4; // Number of disks
	towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
	return 0;
}
```

1. 先把 A 塔顶部的 N-1块盘移动到 B 塔; ← 这是个子问题
2. 再把 A 塔剩下的大盘移到 C; ← 只需挪一个盘
3. 最后把 B 塔的 N-1块盘移到 C ← 另一个子问题

```cpp
#include <iostream>
using namespace std;

void towerOfHanoi(int n, char A_rod, char C_rod, char B_rod){
	if (n == 1){
		cout << "Move disk 1 from rod " << A_rod <<" to rod "<< C_rod<<endl;
		return;
	}
	towerOfHanoi(n - 1, A_rod, B_rod, C_rod);
	cout << "Move disk " << n << " from rod " << A_rod <<" to rod "<< C_rod << endl;

}

int main()
{
	int n = 4; // Number of disks
	towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
	return 0;
}
```

1. 先把 A 塔顶部的 N-1块盘移动到 B 塔; ← 这是个子问题
2. 再把 A 塔剩下的大盘移到 C; ← 只需挪一个盘
3. 最后把 B 塔的 N-1块盘移到 C ← 另一个子问题

```cpp
#include <iostream>
using namespace std;

void towerOfHanoi(int n, char A_rod, char C_rod, char B_rod){
	if (n == 1){
		cout << "Move disk 1 from rod " << A_rod <<" to rod "<< C_rod<<endl;
		return;
	}
	towerOfHanoi(n - 1, A_rod, B_rod, C_rod);
	cout << "Move disk " << n << " from rod " << A_rod <<" to rod "<< C_rod << endl;
	towerOfHanoi(n - 1, B_rod, C_rod, A_rod);
}

int main()
{
	int n = 4; // Number of disks
	towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
	return 0;
}
```

总结

- 函数可以将一段完成独立功能的程序封装起来。通过函数名就可执行这一段功能。使用函数可以将程序模块化。
- C++的程序是由一组函数组成。每个程序必须有一个名为main的函数，它对应于一般的程序设计语言中主程序。
- 函数也可以调用自己，这样的函数称为递归函数。

下一章，指针！