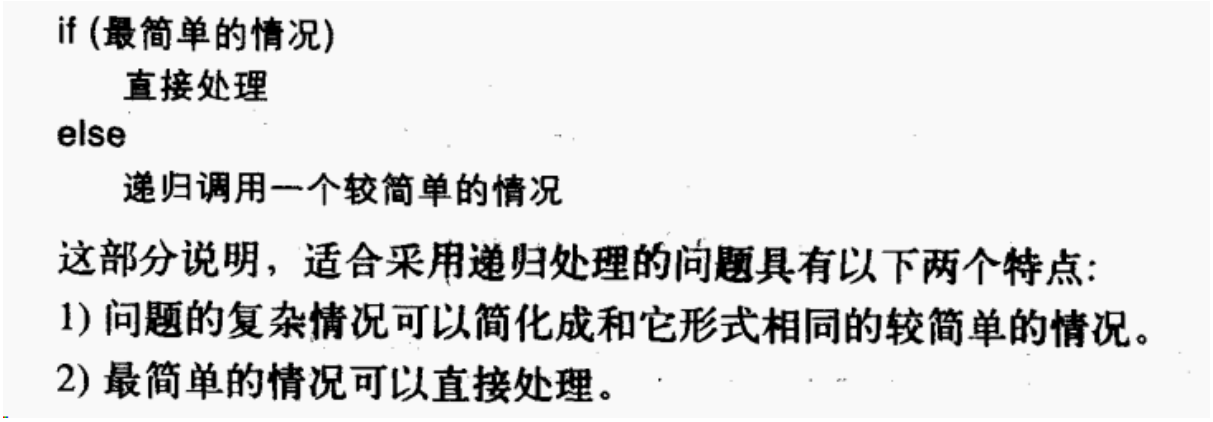
Chapter 4 RECURSION

递归 recursion

Def: a function is recursive if it includes a call to itself

- 核心：以最简化形式的解决为递归结束



Consider recursion when

1. **simplest case(s) can be solved directly;**
2. **complex cases can be solved in terms of simpler cases of the same form.**

- 递归公式（用不断简化的形式去连续定义
- 递归n次 有n个变量（即空间体）空间复杂度的量级函数为O(n)（时间复杂度也为O(n)）
- 对于只有循环的程序 空间复杂度为常量；若有动态存储分配 则不为常量；若是递归 每一次递归执行都会产生一个新的函数执行体（空间体）

In general, worsttime(n) depends on only two things:

1. **the number of loop iterations as a function of n;**
2. **the number of recursive calls as a function of n.**

Example 阶乘 factorial - 递归过程

```
// Example: factorial 阶乘
long factorial(int n)
{
    if (n == 0 || n == 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

factorial递归分析

- 递归调用次数：n-1
- 每次递归调用中需要执行一个常数时间的if语句 → worstTime(n)与n呈线性关系
- 每次递归调用时都需要保存返回地址和变元的拷贝 → worstSpace(n)与n呈线性关系

迭代 iteration

Example 阶乘 factorial - 迭代过程

```
// Example: factorial 阶乘
long factorial(int n)
{
    int product = n;
```

```
    if (n == 0)
        return 1;
    for (int i=n-1; i>1; i--)
        product = product * i;
    return product;
}
```

factorial迭代分析

- 迭代循环次数：n-1
- worstTime(n)与n呈线性关系
- 迭代函数的跟踪始终只需要使用三个变量（product，n，i）→ worstSpace(n)为常数

Example 二进制转换 Converting from Decimal to Binary

```
// Precondition: n >= 0.
// Postcondition: The binary equivalent of n has been printed

void writeBinary (int n)
{
     if (n == 0 || n == 1)
           std::cout << n;
     else
     {
           writeBinary (n / 2);
           std::cout << n % 2 << std::endl;
     } // else
} // writeBinary
```

- 分析：走完所有递归（即到n=0或n=1时）后再一个个跳出循环，输出n%2的值
- worstTime(n)与n呈对数关系
- 此处的worstSpace正比于递归调用 于是与n也呈对数关系

Example 汉诺塔问题

（详细代码分析见ppt）

- worstTime(n)：O(2^n)

复杂的递归方法 - 回溯算法 backtracking p66

定义：回溯通过按序选定位置来尝试达到终点，在过程中遇到无法到达终点的路径则可通过回溯重新返回。

- 每一节点尝试成功的结果可返回上一节点尝试成功的结果

  最简形式：起点即为终点

Example BACKTRACKING

转换成递归问题：策略下顺序遍历的每一个点即为递归调用，不符合则为返回（上一层的递归）

为每一个位置设置一个特殊的实现机制：位置坐标；尝试次数（使用迭代器记录次数）

框架：Application.h (methods such as valid, record, done, undo)

valid - 判断位置是否有效

record - 标记

done - 判断是否到达终点

undo - 取消位置

```
// 后置条件: 返回这个利用输入或赋值生成的
//           Application的初始状态以及
//           起始位置。
Position generateInitialState( );

// 后置条件: 如果pos在通往目的地的路上就返回真。
//           否则将返回假。
bool valid (const Position& pos);

// 前置条件: pos代表了一个有效位置。
// 后置条件: pos被记录成一个有效位置。
void record (const Position& pos);

// 后置条件: 如果pos是这个应用的最后一个
//           位置就返回真，否则
//           返回假。
bool done (const Position& pos);

// 后置条件: pos被标记为不在通往目的
//           地的路径上。
void undo (const Position& pos);
```

```
class Application
{
    friend ostream& operator<<(ostream& stream,Application& a

    public:
    Position generateInitialState();

    bool valid(const Position& pos);

    // ……（省略）
```

```
    class Iterator
    {
        public:
        // 后置条件: 这个Iterator被初始化。
        Iterator ( );

        // 后置条件: 用pos进行这个Iterator的初始化。
        Iterator (const Position& pos);

        // 前置条件: 这个Iterator可以从这个位置前进。
        // 后置条件: 返回这个Iterator的当前位置，并将
        //           这个Iterator前进到
        //           下一个位置。
        Position operator++ (int);

        //后置条件: 这个Iterator再也不能前进了。
        bool atEnd( );
        protected:
        void* fieldPtr;     // 以后解释
    };//类Iterator
};// 类Application
```

※核心掌握：itr后置自增的作用：产生新位置的同时返回当前位置，使得能够维持并记录当前位置的计数与状态

void* fieldPtr：任何类型的指针都可以分配给void指针【声明在头文件application.h中】

struct itrFields：指针结构【声明在源文件Maze.cpp中】

👇通过在Application.h把fieldPtr定义成void指针和在Maze..cpp中生命struct itrFields，解决头文件不能包含专门的迷宫应用信息但又需要定义类的字段的问题，使得fieldPtr为Iterator里的一个字段，对于其每个实例都有一个fieldPtr的拷贝。

```
// Maze.cpp
struct itrFields
{
    int row,
            column,
            direction; //0=north, 1=east, 2=south, 3=west
//为0时方向向北，为1时方向向东
};

// Application.h
Application::Iterator::Iterator(Position pos)
{
```

```
    itrFields* itrPtr = new itrFields;
    itrPtr -> row = pos.getRow();
    itrPtr -> column = pos.getColumn();
    itrPtr -> direction = 0;
    fieldPtr = itrPtr;
} // 构造器
```

```
class BackTrack
{
    protected:
        Application app;

    public:
        BackTrack(const Application& app);

        bool BackTrack:::tryToSolve (Position pos) {
                bool success = false;
            Application::Iterator itr (pos);
            while (!success && !itr.atEnd())
                        {
                        pos = itr++; // pos返回当前的itr位置 itr
                        // 使得在处理下一个位置之前保留当前位置的信息
                if (app.valid (pos))
                        {
                                app.record (pos);
                    if (app.done (pos))
                            success = true;
                    else
                    {
                                        success = tryToSolve
                          if (!success)
                                app.undo (pos);
                    } // not done
                     } // a valid position
                } // while
                return success;
        } // method tryToSolve
```

课堂笔记

不会走回头路：因为record不再为1，会改变数值

区分回退undo和前进方向itr++

回溯是不成功的返回

app的作用：把算法和实际应用联系到一起

迭代器itr真正指向的结构

没有强制类型转换的原因：未对指针作吸取或是引用

汇编语言 - 系统栈与过程调用

- 递归次数过多导致栈中数据溢出
- eax寄存器 用于传参
- call 先让下一条语句的地址入栈
- 进入下一栈帧
- bp总是表示栈帧的开始

汉诺塔的时间复杂度为O(2^n) 空间复杂度为O(n)

二分查找 Binary Search

Def: the size of the subarray searched is divided by 2 until success or failure occurs

每次用中点元素和查找元素比较，通过二分查找可以直接省略不符合的一半元素，从而节省效率。

时间复杂度为O(logn)

排列算法 - 查找字符串

123

132

213

231

312

321

不断交换每一位置（通过设置起始位置的参数实现）

①外面嵌套一个函数外壳 真正的递归函数为permute

②使用默认参数

```
void permute(const string& s)
{
    rec permute(s,0);
}
```

log(!n)