应网友之邀，简单介绍我的五子棋 xl 的思路。
我的五子棋原想起名星浪五子棋，xl 是缩写，后来曾以 hawk 为名发表过。
程序的引擎部分主要采用了以下技术：

1 置换表
2 历史启发
3 alpha-beta 或 MTD(f)
4 增量式棋盘
5 增量式估值
6 真假禁手识别
7 VCF 扩展

1,2,3大家很熟悉，就不说了，下面简单介绍4，5，6：

[棋盘结构]：

标准五子棋棋盘是15*15=225，为提高效率， 我用了256的数组基本棋盘，因为是最基本的棋盘，所以称之为第一层。其中的16行和16列没用到。

```
static uint8_t    _layer1[256];
```

这样就能用一个字节的 unsigned char 做索引，称之为 Pos．低四位是 x，高四位是 y．下面三个个宏可以快速在 x,y 坐标与一个字节的索引之间相互转换。

```
#define POSX(p) (uint8_t)((p)&0x0f)
#define POSY(p) (uint8_t)((p)>>4)
#define MAKEPOS(x,y) (uint8_t)(((y)<<4)+(x))
```

容易想到，可以用3个数字分别表示 黑子，白子和空位：

```
#define White 0
#define Black 1
#define Empty 2
```

由于经常要在黑方和白方之间转换，所以定义了 OPP 宏

```
#define OPP(side) (uint8_t)((~side) &0x01)
```

还需要经常遍历棋盘，以下两个宏可完成这个工作：

```
#define ESB(p) for(p=0;p<=(BW-1)*16+BW ;p++) {if(POSX(p)>=BW) continue;
#define ESE() }
```

这是程序中的一个实例：

```
ESB(p);
switch(_layer1[p])
{
case Black: TRACE("[ X ]\t");break;
case White: TRACE("[ O ]\t");break;
.....
}
ESE()
```

展开后是这样

```
for(p=0;p<=(BW-1)*16+BW ;p++)
{
	if(POSX(p)>=BW)
		continue;
	switch(_layer1[p])
	{
	case Black: TRACE("[ X ]\t");break;
	case White: TRACE("[ O ]\t");break;
	.....
	}
}
```

光有基本棋盘还不能判断棋形,五子棋有四个方向，所以我们还要用4个数组来储存这四个方向的棋形，而且黑白双方要分开放，所以这是个三维数组，称之为第二层

```
static uint8_t    _layer2[2][4][256];
```

假如在 H8点放了一个黑子， 那么 I8再落黑子就是2连， 记做 _layer[Black][横向][I8] = 2连
四个方向按"横竖撇捺"来排序， 0是横向.
单线各个棋形已经定义成了 FSP_*系列宏， 如 FSP_d4p，d 表示被堵了一半， p 指 plus，表示除了冲四外，还有附加手段,见下图

XO_OaO   --- a 冲一次，还可再往外冲，所以是 plus

四个方向分开的棋形最终要还是要合起来分析， 用另一个数组表示， 称之为第三层：

```
static uint8_t    _layer3[2][256];
```

各棋形也定义成了 FMP_*系列宏， 如 FMP_43 表示 四三

这样棋盘就基本完整了， 下面介绍算法。
把四个方向棋形合成一个很简单， 一个3就是三，两个3就是三三，查表即可得到。

而把基本棋盘变为4个方向的基本形相对较难，对此我用了以查表为基础的算法：

单个方向不管有多少格， 最终总是成五，即使成六也是包含了五(当然还有长连等例外，特殊处理)， 所以"五"可作为最小查表单位

```
OOaOb
```

上图在 a,b 两个空位是什么形？
答：应这么查：

首先把这个图转成索引，容易想到 1表示有棋子，0表示没有， 上图即 11010 = 26，查表得

```
_xl_gene[26] = {0,0,7,0,5};
```

查看头文件， 有以下宏：

```
#define FSP_d4              5
#define FSP_4              7
```

把这个结果套上去， 得 a 是 FSP_4活四， b 是 FSP_d4冲四

那么长一点的图如何处理？
答：分成数段来查，然后合并， 如下图：

```
00_O__O_
[ a ]
[ b ]
  [ c ]
   [ d ]
```

查表得

```
[a]  OO_O_  =  {0,0,7,0,5}
[b]  O_O__  =  {0,4,0,4,3}
[c]  _0__O  =  {3,0,4,4,0}
[d]  O__O_  =  {0,4,4,0,3}
```

合并时取数值大的

```
{0,0,7,0,5}
__{0,4,0,4,3}
____{3,0,4,4,0}
______{0,4,4,0,3}
{0,0,7,0,5,4,0,3}
```

检验一下， 这个结果是对的

问题来了，上面是都是两边没有堵住的情况， 如果有一边被堵住呢？ 或者两边都被堵住呢？
答：把表扩大4被， 分别储存"不堵"，"左堵"，"右堵"和"两边堵" 四种情况，索引上新增的第6,7位分别是"右堵标记"和"左堵标记"：

```
XOOaOb_
[   ]

_xl_gene[10 11010]  =  {0,0,5,0,5}
```

a,b 都是冲四

那么，中间有对方的棋子呢？
答：无需考虑这种情况， 单看这五个格已不可能出棋。

```
OOXOOaOb
[ a ]
[ b ]
  [ c ]
   [ d ]
```

a,b,c 不用考虑， 单查 d 即可

其它注意事项：
1 注意同一条线上的44
2 注意长连

[秘籍] 在查表时， "o_" 等价于左堵的"x"，同理"_O"等价于右堵， 如下图

```
XXXXXX__XXa__XXXXXX
```

黑 a 不是活三，而是左右被堵的形， 诸位可自行验证。

把基本棋盘变为4个方向的基本形的算法介绍完了， 很明显，这个算法很慢，所以我在程序中预先生成一些中间结果，以提高计算速度：

```
uint8_t *_cache[2][BW + 1];
```

取名 cache，但它并不是高速缓存， 叫预生成表更确切些，有了这个表，每次计算棋形只需查一下就行了，速度自然大幅度提高。

基本棋盘介绍完了，下面介绍"增量式棋盘"的概念。 实际上"增量式棋盘"就是上面介绍的 _layer2和

_layer3.
因为估值时要知道各个点的棋形， 每次都计算将很慢，所以用了增量式计算的办法， 每次添加或移走棋子，都立即更新_layer2和_layer3，保证它们永远是最新。

[增量式估值]

增量式棋盘是增量式估值的基础， 原理是把估值用到的一些东西(比如冲三的个数)做成增量式的， 化整为零， 大大减轻估值的负担。

[真假禁手识别]

假禁手其实就是假三组成的禁手， 所以关键是如何判断假三。显然，假三就是表面上看来是三，但实际上不能成四的形。
这里"表面上看来是三"可以直接从_layer2中得到， 而在那一点落一子， 就能看出它到底能不能成四。
但是， 光这样判断是不完整的， 五子棋有高级禁手的情况：两个相关的三， 其中一个是不是假三取决于另一个是不是假三.
这种情况实际对局中时有出现，所以需要对假三进行递归判断。进一步还有利用禁手解禁手的下法，这就是搜索选点的问题了。

[VCF 扩展]

类似于象棋的连将胜， 五子棋有 VCF(冲四连胜)。幸运的时，用置换表可以很好地解决 VCF 问题， 许多上百步的局面都可瞬间解出。
但即使如此，一些几十分钟也解不开的局面还是存在的，所以对 VFC 扩展要有一定的限制，否则碰到那些局面时程序就死在那里了。

作为 VCF 的延伸， 更像象棋残局的 VCT 是根难啃的骨头， 经常一个局面算几十分钟也没有结果，所以 xl 里没有 VCT 扩展。看前面的贴子里
有讨论活三是否该估值的问题， 其实活三问题是 VCT 问题的一种， 如果不能算尽棋变法， 完全的"活三不估值"将无法实现。

就写这些了， 如有错误请指正。