aardio 范例: 结构体二级指针

```
//结构体二级指针
var code = /****
    #include <stdlib.h>

    typedef struct {
        int x;
        int y;
    }  POINT;

    __declspec(dllexport) void copyPoints(int size,POINT ** pointsArray )
    {
        for(int i=0;i<size;i++){
            for(int j=0;j<2;j++){
                pointsArray[i][j].x  = i*4+j*2+1;
                pointsArray[i][j].y  = i*4+j*2+2;
            }
        }
    }

    __declspec(dllexport) void copyNumbers(int size,double ** numbers )
    {
        for(int i=0;i<size;i++){
            for(int j=0;j<2;j++){
                numbers[i][j] = i*2+j+0.5;
                numbers[i][j] = i*2+j+0.5;
            }
        }
    }

    //下面我们看一看为什么不用 raw.argsPointer 会更简单
    typedef struct {
        double items[2];
    }  NUMBERS;

    void copyNumberArray(int size,NUMBERS * numbers )
    {
        for(int i=0;i<size;i++){
            for(int j=0;j<2;j++){
                numbers[i].items[j] = i*2+j+0.5;
                numbers[i].items[j] = i*2+j+0.5;
            }
        }
    }
****/

import tcc;
var c = tcc();
c.compile(code);

import console;
import raw.argsPointer;

class POINT_ARRAY{
    struct items[2] = { ::POINT() }
}

//第一种用法，直接获取参数列表指针
var points = raw.argsPointer({ POINT_ARRAY(), POINT_ARRAY(), POINT_ARRAY() })
c.copyPoints(/* int */#points,/* POINT** */points ) //调用 C 函数;

//freePtr 释放内存指针,并会刷新 API 返回的数组值，也可以调用 points.updateArray() 刷新
console.dumpJson( points.freePtr() );

//第二种用法，通过回调获取参数列表指针
import raw.argsPointer;
var numbers = raw.argsPointer({ {double items[2]}, {double items[2]}  },
    function(argsPtr,args){
        //argsPtr 是 double 类型二维数组参数的指针,args 是绑定该指针的数组
        c.copyNumbers(/* int */#args,/* double** */argsPtr ) ;
    }
)

//输出外部 API 更新的数组值
console.dumpJson( numbers );
```

```
//下面我们看一看为什么不用 raw.argsPointer 会更简单
class NUMBERS{
    double items[2]
}

var numbers = { struct items[2] ={ NUMBERS() }  }
c.copyNumberArray(2,numbers) ;

console.dumpJson( numbers.items );
console.log("不用 raw.argsPointer 代码少了十倍可读性也更好，实现的功能完全一样。");
console.log("aardio 不是 C/C++，不要机械地套用 C/C++ 的用法。");
console.pause();


/**details(实现原理)
aardio 中的对象比静态语言使用的原始数据要复杂得多，
例如 aardio 结构体与静态语言的原始结构体在内存中的结构实际是不一样的。

aardio 对象之所以能作为参数用于调用静态语言实现的 API 函数，
是因为 aardio 在其中作了隐式的转换，打破了动/静态语言之间原本不可逾越的鸿沟。
这很方便，但这种无感的自动转换也会让我们产生错觉,让我们忘记了 aardio 并不是 C/C++，
也忘记了不可逾越的鸿沟仍然实际存在。

这种鸿沟的存在有其必要性 —— 正如火车不能允许任意乘客在任意时间下地用双脚原生地奔跑。
对于动态语言 —— 不能百分百完美地套用静态语言的写法和习惯，aardio 支持原生类型的目的
也只是为了调用和利用静态语言的能力而并不是让自己变成静态语言。

以结构体为例：

1、aardio 结构体作为调用 API 的参数时会分配一块临时的内存，
并将 aardio 结构体的值复制过去，然后将该内存的指针作为调用 API 的参数，
在调用 API 结束后再将内存中新的值同步到 aardio 结构体，然后立即释放临时内存，
释放临时内存是立即操作，而非等待垃圾回收器操作。

raw.argsPointer 的回调用法就是基于这个原理，
利用了一个中间 API 函数在被调用时回调 aardio 函数以拦截到原始参数指针。

2、当以 aardio 结构体对象作为参数调用 raw.buffer,raw.realloc 等分配内存的函数时，
也会将结构体的值复制过去，这时候 raw.buffer,raw.realloc 得到的原始指针里存储的就是
原始结构体，所以这些指针可以直接用于 API 的原始数据类型指针 —— 而不再需要转换。
使用 raw.convert() 函数 可以将指针指向的内存再次同步到 aardio 结构体。

raw.argsPointer 的非回调用法就是基于这个原理，但简化了分配内存、同步数据的操作。

注意普通结构体可以任意嵌套数组,与静态API函数兼容且用法更简洁，
如无特殊原因,一般不必要用到 raw.argsPointer 。
end details**/
```

Markdown 格式