PlugCode 1.0alpha Reference Guide

Sergio Davis

19 de abril de 2013

1. What is PlugCode?

PlugCode it is a *dialect* of C programing language, developed to write *plug-ins* of **lpmd** software package. PLugCode it is a *superset* of C, this mean that any C routine it is allowed automatically as a valid PlugCode subroutine.

PlugCode add C facilities as macros, clousures (in a limited way), and allow a very clear ab optimized syntaxis over vector operations, without the poor performance usually related to high level languages like C++. These characteristics are possible because PlugCode is internally tranformed to traditional C code[1] during compile.

```
@slot Evaluate (reader)
{
 Vector * vel = GetArray("vel");
 double ti = 0.0e0;
 for (long i=0;i<size;i++)
 {
  ti += 0.5*M*SquareModule(vel[i]);
 }
 ti *= KIN2EV;
 ti /= ((3.0/2.0)*KBOLTZMANN*totalsize);
 return ti;
}
```

Figura 1: "slot" example wrote at PlugCode.

[1]To be accurate, traditional C means the standard C99, that incorporate comments with `//` and declare allowed inside `for` and `while` cicles.

```
void Evaluate(const RawAtomSet * aset, const RawCell * cell)
{
 double * vel = NULL;
 ASet_GetArrays(aset, NULL, &vel, NULL, NULL, NULL, NULL);
 double ti = 0.0e0;
 long size = ASet_LocalSize(aset);
 long totalsize = ASet_TotalSize(aset);
 for (long i=0;i<size;i++)
 {
  ti += 0.5*M*(pow(vel[3*i],2.0)+pow(vel[3*i+1],2.0)+pow(vel[3*i+2],2.0));
 }
 ti *= KIN2EV;
 ti /= ((3.0/2.0)*KBOLTZMANN*totalsize);
 ASet_ReturnValue(aset, RET_DOUBLE, &ti, 1);
}
```

Figura 2: The same example of figure (1) but in C.

To more details about "slots" en **lpmd** 0.7, refeer to the document "Scheme-Design of LPMD 0.7".

2. Using `Allocate` and `Deallocate`

PlugCode proporciona las funciones `Allocate` y `Deallocate`, como reemplazos de `malloc`, `realloc` y `free` para pedir y liberar memoria dinámica (equivalentes a `new` y `delete` en C++).

Su sintaxis es la siguiente:

`T * Allocate (T, long N)`

pide memoria suficiente para un arreglo de N elementos de tipo T, y devuelve un puntero al comienzo de esa memoria. Es similar a una llamada a `malloc` en C tradicional, con la diferencia que `Allocate` lleva cuenta de la memoria pedida y permitirá evitar muchas fugas de memoria en los *plug-ins*.

`T * Allocate (T, long M, T * pointer)`

reajusta el tamaño del bloque de memoria al que apunta *pointer*, para que acomode ahora M elementos del tipo T. Es similar a una llamada a `realloc` en C tradicional, pero también evita fugas de memoria.

`void Deallocate (T * pointer)`

libera el bloque de memoria al que apunta *pointer*. Es similar a una llamada a `free` en C tradicional, con la diferencia de que está al tanto de si el

bloque de memoria ya fue liberado previamente lo que ayudará a evitar algunos fallos de segmentación producidos por aplicar `free` dos veces.

3. El tipo de datos `Vector`

El tipo `Vector` es una agregación de tres valores reales, de tipo `double`, consecutivos en memoria. Para muchos efectos es equivalente a ser declarado con `typedef double Vector[3]`.

4. El ciclo `VectorLoop`

El ciclo `VectorLoop` se utiliza para realizar operaciones sobre las componentes de un vector ïmplícitamente". Por ejemplo, si a y b son de tipo Vector y queremos hacer que b sea igual a a multiplicado por el escalar f, lo conseguimos con

```
double f = 20.0;
VectorLoop { b = a*f; }
```

que es equivalente a hacer en C tradicional

```
double f = 20.0;
for (int q=0;q<3;++q) { b[q] = a[q]*f; }
```

Es posible usar `Vector` en combinación con `VectorLoop` para asignar las componentes de un vector, como en el siguiente ejemplo:

```
Vector v;
VectorLoop { v = Vector(1.0, 2.0, 3.0); }
```

lo cual es equivalente a

```
Vector v;
v[0] = 1.0;
v[1] = 2.0;
v[2] = 3.0;
```

5. Uso de `@define`

La instrucción `@define` sirve para definir constantes y pequeñas funciones, haciendo el papel de las macros `#define` del preprocesador de C. El siguiente ejemplo muestra ambos usos de `@define`.

```
@define KBOLTZMANN = 8.617E-05
@define Suma(x, y) = (x+y)

@slot Prueba(reader)
{
 double x = Suma(5.0, 3.0*KBOLTZMANN);
}
```

6. Uso de `@macro`

La instrucción `@macro` define una macro, es decir, un bloque de código con nombre, en el cual cero o más parámetros son expandidos al momento de ser llamado. Las macros no pueden retornar resultados como si fueran funciones, aunque por supuesto siempre es posible retornar valores a través de parámetros de salida.

```
@macro Igualar(v1, v2)
{
 VectorLoop { v1 = v2; }
}

@slot Prueba2(reader)
{
 Vector a, b;
 Igualar (b, Vector(1,2,3));
 Igualar (a, b);
}
```

7. Simulando captura de valores con `@define` y `@macro`

Es posible simular *clausuras* usando `@define` y `@macro`. Ya que ambas son expandidas textualmente, es posible hacer referencia en ellas a variables "colgantes" (*unbounded*) que no han sido definidas hasta el momento de la llamada. A continuación el típico ejemplo de clausura en el cual se `@define` una función que suma $a = 10$ a su argumento:

```
@define SumaDiez(x) = (x+a)
int a = 10;
int y = SumaDiez(5);
assert (y == 15);
```

```
a = 20;
y = SumaDiez(5);
assert (y == 25);
```

Note que `SumaDiez` no es una clausura genuina, uno, porque no es un objeto sino una macro expandida en tiempo de compilación (similar a un template de C++), y dos, porque el valor a no queda en realidad capturado dentro de la función, sino que se toma el valor actual de a del ámbito superior. Al cambiar el valor de a a 20, `SumaDiez(x)` cambia su resultado.

8. Referencia de PlugCode

8.1. Tipos de datos

Nativos (heredados de C):

- `int`
- `long`
- `float`
- `double`
- `bool`
- `char`

Emulados:

- `Vector`
- `Matrix`
- `AtomPair`
- `NeighborList`
- `AtomSelection`

"Constructores"[2] de los tipos emulados

- `Vector Vector(double x, double y, double z)`
- `Matrix Matrix(int columns, int rows)`

- `NeighborList NeighborList(AtomPair * pairs, long n)`
- `AtomSelection AtomSelection(long * indices, long n)`

8.2. Funciones de manejo de memoria

- `T * Allocate(T, long N)`
- `T * Allocate(T, long N, T * p)`
- `void Deallocate(T * p)`

8.3. Funciones que involucran objetos `Vector`

- `const char * VectorFormat(const char * format, Vector v)`
- `double Module(Vector v)`
- `double SquareModule(Vector v)`
- `double Dot(Vector v)`
- `Vector Cross(Vector a, Vector b)`
- `Vector Unitary(Vector v)`

8.4. Funciones de generación de números aleatorios

- `double Random()`
- `Vector RandomVector()`

8.5. Funciones relacionadas con la celda de simulación

- `double Distance(Vector dr, Vector pi, Vector pj)`
- `void Fractional(Vector cart, Vector frac)`
- `void Cartesian(Vector frac, Vector cart)`
- `void CenterOfMass(Vector cm)`

[2]En realidad como C no tiene el concepto de constructor, éstos son más bien "funciones de fábrica" (*factory functions*).

- `int InsideNode(Vector v)`
- `void AddAtom(const char * names, ...)`
- `void * GetArray(const char * name)`
- `void GetArrays(const char * names, ...)`
- `void SetArray(const char * name, long n, void * p)`
- `void SetArrays(const char * names, long n, ...)`
- `void * GetTotalArray(const char * name, long a, long b, int sorted)`
- `void GetTotalArrays(const char * names, long a, long b, int sorted, ...)`
- `int HasTag(const Tag tag, int key)`
- `void SetTag(Tag * tag, int key)`
- `void UnsetTag(Tag * tag, int key)`

8.6. Depuración y tests unitarios

- `void LogMessage(const char * format, ...)`

8.7. Variables especiales disponibles en un "slot"

long size : Tamaño local del conjunto de átomos.

long totalsize : Tamaño total del conjunto de átomos.

long extsize : Tamaño extendido del conjunto de átomos.

int masternode : 1 si el nodo es el nodo maestro, de lo contrario 0.

NeighborList neighborlist : La lista de vecinos precalculada.