Allegato alla puntata n. 15 Cubasis VST Facile

Vita da plug-in
Con i sei precedenti passi (vedi puntata n.15 sulla rivista) abbiamo dunque dato vita ad un plug-in, davvero ipersemplice certamente, ma pur sempre appartenente alla famiglia dei plug-in ed in quanto tale rappresentativo di alcuni suoi elementi caratteristici. Vediamo ora di osservare appunto alcuni di questi elementi, cercando di stendere una sorta di **identikit** del nostro piccolo plug-in. Per seguire tali osservazioni vi consigliamo di consultare il **codice allegato al presente articolo** che non è altro che un semplice adattamento/personalizzazione di quello originale Steinberg ma che pensiamo possa aiutare un poco nell'identificare i **concetti di fondo** in gioco ed a questo proposito abbiamo cercato di commentare il più possibile le parti di maggiore interesse. Nei paragrafi seguenti riporteremo comunque all'occorrenza anche qualche stralcio laddove si renda necessario evidenziare qualche punto specifico.

Dunque, come segnalato, il plug-in di esempio è basato completamente su **una classe C++ derivata** da quella base fondamentale *AudioEffectX*. Questo, per le proprietà della Programmazione ad Oggetti (OOP), vale sostanzialmente per qualsiasi plug-in che in tal modo può ereditare in un sol colpo tutte le caratteristiche della classe base. Al **costruttore** della classe del nostro plug-in (che ricordiamo essere non altro che la sua funzione membro, per così dire, dedicata alle operazioni di 'inizializzazione') viene passato un parametro *audioMaster* di tipo *audioMasterCallback* e la classe del plug-in è costruita a cura dell'applicazione host la quale passa proprio un oggetto di tipo audioMasterCallback che gestirà l'interazione con il plug-in stesso. Alcuni flag ed identificatori vengono impostati per dichiarare i requisiti di ingressi/uscite.
La classe ridefinisce poi le **due importanti funzioni membro** responsabili dell'effettivo processamento del segnale (ancorché, ovviamente, estremamente semplice nel caso attuale).
Tali funzioni (caratterizzate da tre parametri: un puntatore ai dati, un puntatore all'area di memora dove porre i dati elaborati ed infine la dimensione del blocco dati stesso)
sono quelle ripetutamente invocate dall'applicazione host, ogni volta che è disponibile un nuovo blocco di dati.
Ricordiamo che in *AudioEffect.hpp* troviamo:

```
class AudioEffect
{
public:

...

virtual void process(float **inputs, float **outputs, long sampleFrames) = 0;
virtual void processReplacing(float **inputs, float **outputs, long sampleFrames) {inputs = inputs; outputs = outputs; sampleFrames = sampleFrames;}

...

};
```

Riassumendo e tornando ai concetti di fondo, per il momento si notino i tre punti seguenti:

- La Classe del plug-in viene derivata da quella base AudioEffectX e vengono forniti i due metodi di elaborazione del segnale audio (processing methods).

- La funzione ‘main’ del plug-in contenutanel file *AgainMain.cpp* è utilizzata dall'applicazione host per istanziare il plug-in stesso.

- Viene utilizzata l'interfaccia utente standard del VST per visualizzare i parametri.

Il plug-in dal punto di vista dell'host e la funzione main()
Come accennato è l'applicazione host ad istanziare effettivamente un oggetto della classe del plug-in invocando tramite un puntatore la **funzione main()** di quest'ultimo e passandogli un oggetto audioMaster di tipo audioMasterCallback contenente dati per il controllo dell'interazione fra host e plug-in e passato al costruttore della classe base AudioEffectX.
L'host ottiene un puntatore al plug-in appena istanziato e lo controlla per verificare che risponda ai requisiti di un plug-in.
In *AEffect.h* abbiamo:

```
// semplificando al solo caso Windows
#define VSTCALLBACK __cdecl
...
typedef struct AEffect AEffect;
typedef long (VSTCALLBACK* audioMasterCallback)(AEffect* effect, long opcode, long index, long value, void* ptr, float opt);

...

struct AEffect
{
```

```
...

void (VSTCALLBACK* process)(AEffect* effect, float** inputs, float** outputs, long sampleframes);

...

void (VSTCALLBACK* processReplacing)(AEffect *effect, float** inputs, float** outputs, long sampleframes);

};

...

enum
{
audioMasterAutomate = 0,
audioMasterVersion, // versione vst, attualm. 2 (0 = preced.)
audioMasterCurrentId,
audioMasterIdle,
audioMasterPinConnected
};
```

È nel file *AGainMain.cpp* che troviamo la funzione main() del plug-in:

```
static AudioEffect *effect = 0;

...

AEffect* main(audioMasterCallback audioMaster)
{
// controlla la versione del vst
if (!audioMaster (0, audioMasterVersion, 0, 0, 0, 0))
{
// se ritorna 0 è vecchia
return 0;
}
// istanzia un nuovo oggetto plug-in di Classe AGain
effect = new AGain (audioMaster);
if (!effect)
{
return 0;
}
//... semplificando ...
return effect->getAeffect ();
}
```

Si noti che l'oggetto audioMaster di tipo audioMasterCallback viene invocato con il valore enumerativo audioMasterVersion per assicurarsi che il plug-in venga aperto nell'ambito di un'applicazione host VST effettivamente valida.
Dopodiché vien chiamato il costruttore del plug-in e se tutto funziona bene (in caso contrario, come in presenza della tipica e peraltro assai temuta mancanza di memoria disponibile, ritorna un valore nullo all'applicazione VST host) viene invocata una delle funzioni membro dell'oggetto audioeffect, precisamente la getAeffect (), il cui risultato sarà ancora una volta passato all'applicazione host. Si ricordi che in *AudioEffect.hpp* abbiamo:

```
class AudioEffect
{
...
public:
  AEffect *getAeffect() {return &cEffect;}
...
protected:
  AEffect cEffect;
};
```

Come i più attenti lettori avranno certamente notato quello che è effettivamente restituito all'applicazione host è una struttura C anche se lavoriamo in un contesto C++.
Allorché l'host istanzia un plug-in, dopo la chiamata alla funzione main() di quest'ultimo, provvede anche ad informarlo su parametri fondamentali di sistema quali ad esempio la frequenza di campionamento e la dimensione dei blocchi di campioni.

Dati membro del plug-in
In *AGain.hpp* (che, ricordiamo, è il file di intestazione per la classe del plug-in, dunque contenente la sua dichiarazione, le funzioni ed i dati membro) troviamo due parametri di tipo 'protected' (protetto) in quanto non potranno essere manipolati direttamente come quelli public ma esclusivamente per tramite delle funzioni membro adibite al loro trattamento. In questo modo si assicura una **protezione ai dati** ed il **rispetto dei concetti di base della OOP**.
Riportiamo un estratto di *AGain.hpp* per aiutare nella 'localizzazione' di tali due dati membro (in pratica il guadagno ed il nome programma).

```
class AGain : public AudioEffectX
{
public:
  AGain(audioMasterCallback audioMaster);

...

protected:
  // dato membro contenente il 'Guadagno' del plug-in
  float fGain;
  // dato membro contenente il nome del programma
  char programName[32];
};
```

Il costruttore del plug-in
Il **costruttore** del plug-in si trova nel file *AGain.cpp*:

```
AGain::AGain(audioMasterCallback audioMaster)
        : AudioEffectX(audioMaster, 1, 1)    // 1 programma, 1 parametro
{
fGain = 0.5;
setNumInputs(2);
setNumOutputs(2);
setUniqueID('CMGN');
canMono();
canProcessReplacing();
setProgramName("CMGain Default Prog.");
}
```

Si noti che il costruttore presenta come parametro la callback verso l'host (audioMaster) che viene a sua volta passata alla classe base AudioEffectX (principio di ereditarietà in azione!), il cui costruttore presenta due parametri aggiuntivi relativi rispettivamente al numero di programmi ed al numero di parametri utente per il plug-in.
Ricordiamo infatti che in *audioeffectx.h* troviamo:

```
class AudioEffectX : public AudioEffect
{
public:
 AudioEffectX (audioMasterCallback audioMaster, long numPrograms, long numParams);
...

};
```

Nel nostro caso abbiamo **un solo parametro**, il guadagno (gain), corrispondente ad un dato membro della classe e che scegliamo di initializzare a 0.5 (per porre lo slider del cursore associato al centro). Tutti i parametri VST sono dichiarati come *float* (virgola mobile) con intervalli compresi fra 0.0 e 1.0.
Vengono poi specificati il **numero di ingressi** ed **il numero di uscite** che il plug-in è in grado di suppotare/elaborare (2 per il caso stereo).
Segue la specificazione dell'**identificativo univoco del plug-in**, usato dall'applicazione host per distinguere ed identificare appunto il plug-in.
La chiamata a **canMono()** comunica all'applicazione host la caratteristica mono del plug-in (un ingresso/due uscite) di modo che quando l'host si troverà nella situazione di presentare ad esempio mandate effetti mono ed un bus stereo per i ritorni di questi ultimi, controllando tale parametro potrà immediatamente aggiungere il plug-in all'elenco di quelli utilizzabili in tale maniera. Se è selezionato un plug-in stereo l'host può decidere se inviare lo stesso segnale ad entrambi gli ingressi.
Tornando al costruttore possiamo poi notare il supporto del processamento di segnale con rimpiazzo (**replacing processing**) tramite la chiamata a canProcessReplacing() .
Infine, troviamo l'unico nome programma che impostiamo con la funzione strcpy(programName, "CMGain Default Prog."). Tale nome programma è quello visualizzato nel rack effetti di *Cubasis* VST e potrà essere modificato dall'utente.

Il distruttore del plug-in

Nel caso specifico il distruttore del plug-in in realtà non fa...un bel niente; tuttavia vale la pena di citarlo in questa sede per ricordare come ciascun oggetto di una data classe una volta 'costruito' tramite il proprio costruttore prevede anche un apposito 'distruttore'.

```
AGain::~AGain()
{
// in questo caso niente di particolare da compiere
}
```

Parametri e programmi
Il nostro 'micro' plug-in presenta, come s'è evidenziato sopra, oltre ad un solo programma, **un solo parametro** pertanto è possibile tralasciare nelle funzioni membro l'index che entra in gioco appunto nel caso di più parametri.
Con le funzioni setParameter/getParameter rispettivamente impostiamo e recuperiamo dunque il valore corrente del dato membro fGain:

```
void AGain::setParameter(long index, float value)
{
fGain = value;
}

float AGain::getParameter(long index)
{
return fGain;
}
```

In questo semplice esempio ed in generale in tutti i casi in cui si intenda utilizzare l'interfaccia standard offerta dall'applicazione host anche per i plug-in è necessario poi ridefinire alcune funzioni affinché possano venir visualizzati valori pertinenti all'utente.
In sostanza è l'apllicazione VST host che fornisce le modalità di visualizzazione dei parametri ed un plug-in deve solo comunicare le stringhe rappresentative dei propri valori, etichette e nomi programma. Ricordiamo che in *AudioEffect.hpp* troviamo:

```
class AudioEffect
{
public:

...

virtual long getProgram() {return curProgram;}
virtual void setProgram(long program) {curProgram = program;}
virtual void setProgramName(char *name) {*name = 0;}
virtual void getProgramName(char *name) {*name = 0;}
virtual void getParameterLabel(long index, char *label) {index = index; *label = 0;}
virtual void getParameterDisplay(long index, char *text) {index = index; *text = 0;}
virtual void getParameterName(long index, char *text) {index = index; *text = 0;}

...

protected:
        // members
        float sampleRate;
        AEffEditor *editor;
        audioMasterCallback audioMaster;
        long numPrograms;
        long numParams;
        long curProgram;
        long blockSize;
        AEffect cEffect;

};
```

Il parametro *index* è sempre ignorato nel nostro caso dato che, come s'è detto, abbiamo un solo parametro per il plug-in.
Come ridefinizioni avremo ad esempio:

```
void AGain::getParameterName(long index, char* label)
{
 strcpy(label, "  Guadagno  ");
}
void AGain::getParameterDisplay(long index, char* text)
```

```
{
 dB2string(fGain, text);
}
void AGain::getParameterLabel(long index, char* label)
{
 strcpy(label, "   [dB]   ");
}
```

Da tener presente che le lunghezze previste per default dall'interfcaccia VST sono tipicamente limitate a 24 caratteri e per evitare problemi è bene mantenersi entro tale valore massimo.
La classe base AudioEffectX offre anche alcune funzioni membro di utilità da adoperare in sede di conversioni di parametri in valori e stringhe:

```
virtual void dB2string(float value, char *text);
virtual void Hz2string(float samples, char *text);
virtual void ms2string(float samples, char *text);
virtual void float2string(float value, char *string);
virtual void long2string(long value, char *text);
```

Plug-in-conclusioni
Bene, le fasi necessarie per creare un plug-in funzionante sono quelle incontrate nel corso del presente appuntamento. Al di là della semplicità e ovviamente del carattere assolutamente *entry-level* del plug-in esaminato, che ci ha peraltro permesso sia di continuare la nostra **esplorazione del VST SDK** (attualmente fra gli strumenti più appetibili ed interessanti per lo sviluppo di plug-in audio/musicali, destinato com'è ad una piattaforma, quale *Cubase/is VST*, di così enorme diffusione ed utilizzo a tutti i livelli) sia di delineare **i capisaldi** dell' l'architettura posta alla base di un qualsiasi plug-in.
C'è da aggiungere a questo punto che, almeno in linea generale, chiunque disponga di, o metta a punto, un adeguato algoritmo di elaborazione del segnale, non dovrà far altro che 'infilarlo' nelle funzioni di processamento del segnale e partire con lo sviluppo, utilizzando il nostro 'piccolo amico' magari come semplice test di partenza.
Si noti che con il mini plug-in esaminato sopra abbiamo visto in sostanza le principali problematiche comuni a qualsiasi plug-in quali quelle relative a:

- inizializzazione ed istanziazione
- gestione di base dei parametri
- interfaccia utente standard

Per coloro che vogliano aggiungere più parametri e, della serie... *l'appetito vien mangiando*, dotare magari il proprio plug-in di una *pirotenica* interfaccia utente più sofisticata, consigliamo a questo punto di procedere con l'analisi degli esempi *ADelay* (plug-in che implementa un ritardo) e *ADelayEditGUI* (Delay con Editor custom) presenti nella solita cartella source del VST SDK medesimo. Ai novelli progettisti di Sintetizzatori Virtuali suggeriamo infine di partire dall'esempio *Vstxsynth*.

Ok, vi giungano i miei consueti ma sinceri saluti a tutti ed un arrivederci alla prossima puntata.