Tutorial prático para o desenvolvimento de *externals* em C para o Pure Data

Flávio Luiz Schiavoni - André Bianchi

São Paulo, 29 de maio de 2014

Sumário

Parte I

Primeiros passos

Capítulo 1

Introdução

Pure Data, ou simplesmente Pd, é um ambiente visual de programação musical que permite a criação de aplicações musicais complexas a partir da combinação de componentes visuais mais simples chamados **objetos**. As distribuições oficiais do Pure Data contêm diversos objetos prontos para o uso, mas também permitem a extensão de suas funcionalidades através da criação de novos objetos utilizando C/C++. Desta forma, novas linhas de código escritas pelo usuário são compilados como bibliotecas dinâmicas e podem ser carregadas pelo programa em tempo de execução. Objetos desta forma levam o nome de ***externals***.

Este é um tutorial prático para o desenvolvimento de *externals* em C para o Pure Data. A iniciativa de escrever este documento surgiu no primeiro semestre de 2011, durante a disciplina de Computação Musical ministrada pelo professor Marcelo Gomes de Queiroz no Instituto de Matemática e Estatística da Universidade de São Paulo. A intenção deste tutorial é auxiliar programadores a desenvolver *externals* de maneira bastante simples através de exemplos práticos.

Mais do que ampliar a gama de objetos do Pure Data e criar novos objetos, o objetivo deste trabalho é também fornecer ao pesquisador de computação musical uma ferramenta para implementar e testar algoritmos de processamento de áudio para caráter de estudo. Isto significa que podemos reimplementar várias coisas que já existem no Pure Data simplesmente porque é didático programar e colocar algoritmos para funcionar.

Não é objetivo deste tutorial ensinar processamento de som, ensinar algoritmos, ensinar programar em C/C++ ou ensinar a utilizar o Pure Data. Também não é objetivo questionar o modo como o Pd e seus externals foram feitos.

É importante dizer que nada no mundo se aprende sozinho. Foi graças aos vários *externals* escritos para o Pd, com seu código aberto e documentado que conseguimos reunir o conhecimento que aqui presente. Seria impossível citar todos os autores de *externals* que nos ajudaram sem saber. No entanto, não deixamos de agradecer ao que chamamos de comunidade de software livre, ao autor do Pd (seria Public Domain?), Miller Puckette e ao autor do outro tutorial, IOHannes Zmölnig. Muito obrigado.

Este tutorial está acompanhado de vários exemplos cujos códigos ilustram os nossos passos. A estes tutoriais foi adicionado exemplos de Tk e também um makefile que permite compilá-los em vários sistemas operacionais.

Este tutorial ainda não está pronto e por isto você encontrará caixinhas como esta com notas do que mais temos de fazer.

1.1 Escrevendo *externals*

O código fonte do Pure Data é organizado de acordo com convenções de programação orientada a objetos. Para o desenvolvimento de *externals*, é necessário seguir estas convenções e fornecer ao ambiente uma nova classe com alguns métodos específicos, como veremos mais adiante. Para desenvolver para o Pure Data, é necessário importar o arquivo de cabeçalho `m_pd.h`[1], que contém definições de constantes, tipos e funções.

Uma boa fonte de informação é o tutorial de *externals*[2] escrito pelo IOHannes[3], um dos programadores do Pure Data. Apesar de ter utilizado este documento como ponto de partida, boa parte do que está

[1] `http://pure-data.git.sourceforge.net/git/gitweb.cgi?p=pure-data/pure-data;a=blob_plain;f=src/m_pd.h;hb=HEAD`

[2] `http://iem.at/pd/externals-HOWTO/pd-externals-HOWTO.pdf`

[3] `http://puredata.info/author/zmoelnig`

incluso no presente tutorial foi aprendido a partir da leitura do código-fonte de *externals* contidos no repositório oficial do Pure Data[4].

Navegando pelos códigos-fonte deste repositório você poderá notar que os programadores que escreveram os externals que hoje estão disponível para o Pd seguiram estas convenções e por isto a leitura destes códigos-fonte pode ser didática e simples.

Por esta razão, o primeiro conselho que damos para quem irá escrever *externals* é seguir estas convenções, mesmo que as mesmas não sejam a maneira como você está acostumado a programar deste jeito pois assim seu código também será didático e simples de entender.

Este tutorial não pretende cobrir os algoritmos de processamento de sinais mas explicar como implementar estes algoritmos como objetos do Pd. Para processamento de sinais há uma vasta bibliografia disponível que possui os algoritmos e o ferramental matemático necessário para sua implementação.

Será que podemos citar aqui algum livro ou material para DSP?

1.2 Organização do código-fonte e do objeto compilado

Um novo *external* corresponde a uma nova classe na arquitetura orientada a objetos do Pure Data. Para que o carregamento da biblioteca dinâmica em tempo de execução funcione corretamente, é necessário que o arquivo binário produzido possua o mesmo nome que a classe correspondente ao *external*.

Para criar, por exemplo, um *external* chamado "passa-baixas", podemos escrever seu código-fonte em um arquivo chamado `passa-baixas.c`, e em seguida compilar um objeto de biblioteca compartilhada chamado `passa-baixas.pd_linux`, no caso do sistema GNU/Linux. Outras arquiteturas de sistema utilizam outras extensões para o nome do objeto com a biblioteca compartilhada do *external*, como por exemplo `.dll` (M$ Windows), `.pd_irix5` (SGI Irix) ou `.pd_darwin` (Mac OS X).

Importante: O nome do arquivo com o código-fonte não possui formato obrigatório, mas o nome do objeto compilado com a biblioteca dinâmica deve sempre corresponder ao nome da classe, assim como sua extensão deve sempre corresponder à arquitetura do sistema utilizado.

O mesmo cuidado é recomendado para os métodos que serão definidos internamente no objeto. Os nomes de métodos que serão apresentados neste material seguem o padrão encontrado no repositório do Pd. É fortemente recomendado que o mesmo padrão seja utilizado em seu texto.

Para gerar a estrutura básica de um *external* sugerimos utilizar o gerador de *external* disponível em `www.ime.usp.br/~fls/PDExternal-generator/PDExternal_generator.html`.

1.3 Compilação

Para criar um objeto binário que pode ser carregado no Pure Data em tempo de execução, primeiro compilamos o código fonte, criando assim um ou mais objetos intermediários, e em seguida utilizamos um ligador (*linker*) para criar um objeto de biblioteca compartilhada.

No GNU/Linux, uma forma de realizar o processo `example01.c` → `example01.o` → `example01.pd_linux` é a seguinte:

```
1 EXTNAME=example01
2 cc -DPD -fPIC -Wall -o ${EXTNAME}.o -c ${EXTNAME}.c
3 ld -shared -lc -lm -o ${EXTNAME}.pd_linux ${EXTNAME}.o
4 rm ${EXTNAME}.o
```

Listing 1.1: Compilação de um objeto

A opção de compilação `-fPIC` resulta na criação de código binário que roda independente de sua posição na memória, adequado para geração de bibliotecas compartilhadas. A opção `-shared` passada para o ligador determina a criação de uma biblioteca compartilhada.

Para facilitar a compilação, é interessante utilizar um `makefile`. Os exemplos deste tutorial estão acompanhadas de um `makefile` encontrado na seção de desenvolvedores do Pure Data [5].

Para compilar *externals* no MacOS é necessário instalar o XCode. Tem uma dezena de jeito de compilar pro Windows, usando o Mingw ou o C++ Builder. Aqui[6] temos exemplos e muitas discussões de como compilar externals no Windows.

[4] `http://pure-data.svn.sourceforge.net/viewvc/pure-data/trunk/externals/`
[5] `http://puredata.info/docs/developer/MakefileTemplate`
[6] `http://puredata.hurleur.com/sujet-1029-problem-compiling-external-windows`

Não testamos. Faltou coragem. Será que compensa testarmos isto tudo?

1.3.1 Debugando códigos

Para verificar um erro, inicie o PD com seu patch de teste pelo terminal dentro do ambiente gdb com o comando

gdb –args pd -path caminho-do-external

run

Caso o PD tenha algum problema em sua execução o GDB pode te ajudar a encontrá-lo.

Outros comandos básicos do gdb são **where** (que apresenta o arquivo e a linha onde o erro ocorreu) e **list** (que mostra o código deste trecho). Para navegar entre os arquivos, utilize **up** e **down**. Para maiores informações, procure um tutorial sobre o gdb.

1.3.2 Misturando código C e C++

Existem algumas diferenças entre compiladores C e C++ que tornam a sintaxe das linguagens incompatíveis, gerando resultados diferentes para um mesmo trecho de código. Um exemplo disso que influencia o funcionamento de *externals* no Pure Data é a geração da tabela de símbolos dos objetos binários.

Compiladores C++ realizam um processo chamado *name mangling* (ou "dilaceramento de nomes"), que consiste em alterar o nome de funções, estruturas, classes, etc, incluindo informações sobre o espaço de nomes do objeto em questão. Isto resulta em nome diferentes gravados nas tabelas de símbolos dos objetos binários, o que pode confundir o Pure Data no momento do carregamento de um *external*.

Para garantir que um compilador C++ gere nomes compatíveis com objetos binários C, utilize a expressão `extern "C"` na frente dos nomes das funções que serão chamadas pelo Pure Data:

```
1 extern "C" example01_setup(void);
2 extern "C" example01_new(void);
```

Listing 1.2: Externalização de código C++

1.4 Arquivos de ajuda

É importante distribuir, junto com novos *externals*, um arquivo de ajuda do Pure Data com instruções e exemplos de utilização. Como convenção, o arquivo de ajuda deve ter o mesmo nome que o *external*, acrescido do sufixo `-help.pd`. Por exemplo, para o código fonte `example01.c`, que gera o objeto `example01.pd_linux`, escrevemos também o arquivo `example01-help.pd`.

No próximo capítulo veremos uma forma de associar o arquivo de ajuda com a opção de ajuda que aparece no menu contextual com um clique do botão direito no objeto do external dentro do Pure Data.

1.5 Utilizando *externals*

Para carregar um *external* em um `patch` do Pure Data em tempo de execução, basta criar um objeto (com `CTRL+1` ou acessando o menu `Put` → `Object`) com o caminho (relativo ou absoluto) para o objeto compilado com a biblioteca compartilhada, omitindo a extensão.

É possível adicionar o diretório que contém o arquivo binário do *external* ao caminho de busca do Pure Data, de forma que para acessá-lo de dentro de um *patch* não seja necessário digitar o caminho inteiro até o objeto. Isto pode ser feito através da passagem de um parâmetro na linha de comando do Pure Data com a opção `-path <caminho>`, ou de forma gráfica acessando a opção `File` → `Path...` no menu do Pure Data, como pode ser visto na figura 1.1.

Para carregar uma biblioteca de *externals* (mais de um *external* no mesmo arquivo-fonte), é possível indicar o nome da bibliotecai na linha de comando do Pure Data utilizando a opção `-lib <biblioteca>`, ou também graficamente através do menu `File` → `Startup...`, como pode ser visto na figura 1.2.

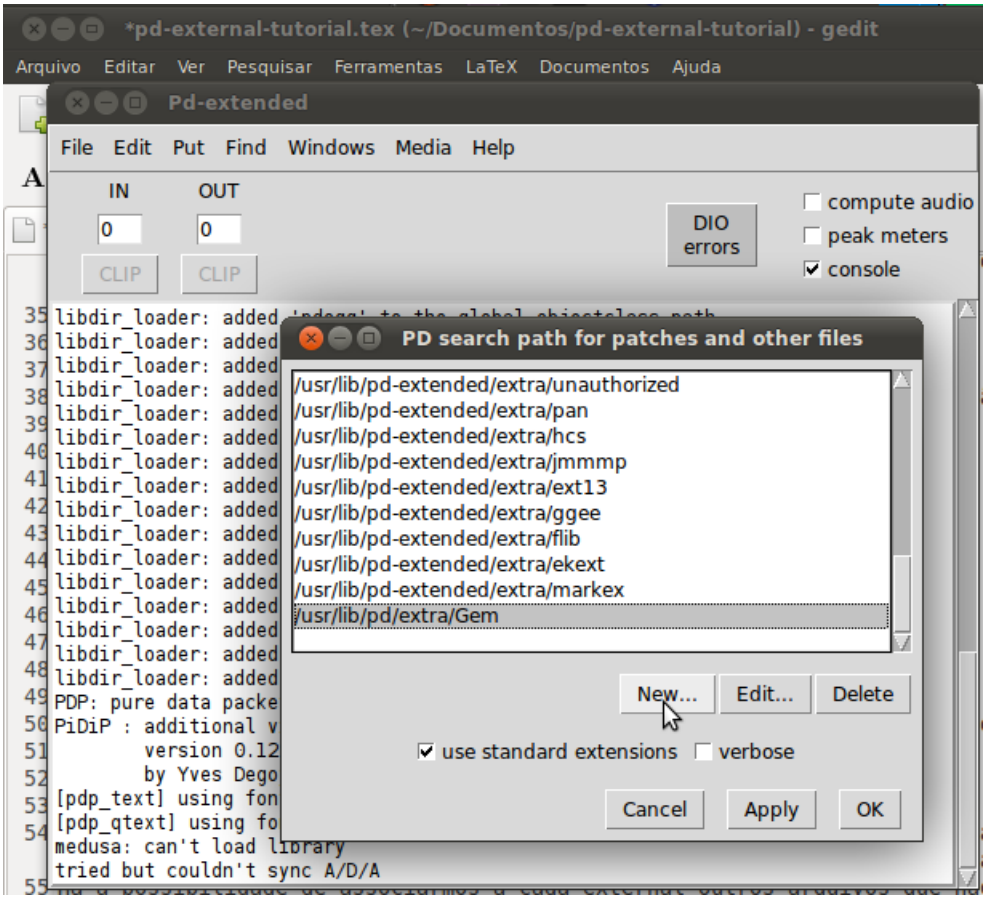

Figura 1.1: Adicionando o diretório de um *external* ao caminho de busca do Pure Data.

1.6 Exemplos

Este tutorial é acompanhado de diversos exemplos de *externals* que ilustram o conteúdo coberto pelo mesmo. Vale lembrar que muitos destes objetos são de utilidade duvidosa e servem apenas como exemplos didáticos.

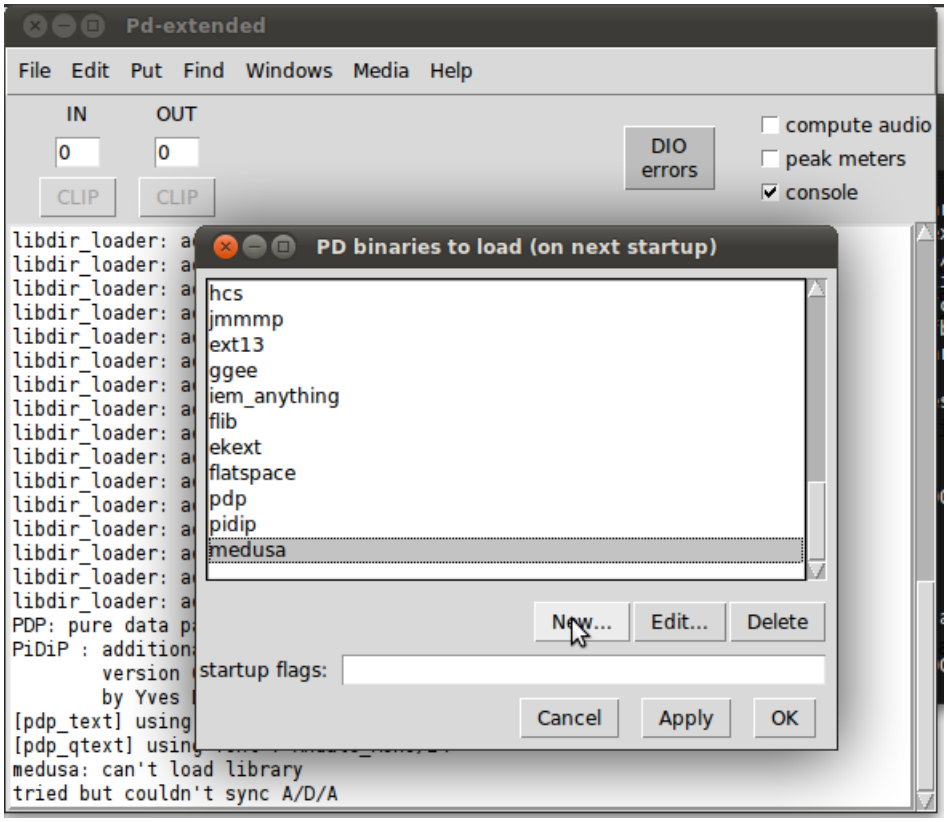

Figura 1.2: Adicionando uma biblioteca ao Pure Data.

Capítulo 2

O básico de um *external*

Escrever um *external* significa seguir as recomendações da API. Peço ao leitor bastante paciência pois este tutorial pretende andar um pouco devagar para mostrar os passos da escrita de um *external*.

2.1 Um *external* Hello World

Como dissemos anteriormente, a arquitetura do Pure Data é organizada de acordo com o paradigma de orientação a objetos: cada objeto gráfico do Pure Data corresponde a uma instância de uma classe. Neste sentido, um *external* está associado a um conjunto de estruturas de dados que representam classes em C. Veja que o conceito de objeto aqui não remete ao conceito de objetos de linguagens de programação como Java ou C++. Para cada classe é necessário haver métodos de instanciação, destruição, processamento de sinais, tratamento de mensagens, etc.

A infraestrutura mínima para o funcionamento de um *external* (de nome, digamos, `<external>`) consiste em uma estrutura de dados para a representação de uma classe, que deve ter nome `t_<external>`, e dois métodos obrigatórios, chamados `<external>_setup()` e `<external>_new()`. Note que a convenção de nomes utilizada no Pure Data é de que toda função deve ser nomeada da forma `<contexto>_<funcao>()`.

A estutura de dados que representa uma classe do Pure Data deve obrigatoriamente possuir o primeiro atributo do tipo `t_object`, no qual é armazenado o objeto criado no momento da instanciação. Outros atributos podem ser adicionados a esta estrutura de maneira que cada instância da mesma classe possua os atributos necessários para seu funcionamento. Uma classe que acessa um arquivo, por exemplo, pode possuir como atributos uma string para guardar o caminho e um inteiro para guardar o descritor do arquivo.

Um exemplo de estrutura de dados para representação de uma classe chamada `helloworld` consiste no seguinte:

```
1  // -------------------------------------------------
2  // Class definition
3  // -------------------------------------------------
4  static t_class *helloworld_class;
5
6  // -------------------------------------------------
7  // Data structure definition
8  // -------------------------------------------------
9  typedef struct _helloworld {
10    t_object x_obj;
11 } t_helloworld;
```

Listing 2.1: Estruturas de dados de um *external*

Tal objeto é passado para as funções que tratam mensagens e por isto tudo o que for necessário para o funcionamento de seu external deve estar contido neste objeto. Aproveitamos para recomendar que isto inclua toda alocação e liberação de memória que possa ser necessária.

Sempre que um *external* é carregado pelo Pure Data, o método de nome `<external>_setup()` é executado. No exemplo dado acima, o Pure Data irá procurar, no arquivo binário `example1.pd_linux`

que contém a biblioteca compartilhada, o método de nome `example1_setup(void)`. Este método é utilizado para realizar a inicialização da classe, informando ao Pure Data da existência de uma nova classe no sistema e associando a ela os métodos de instanciação e destruição, além de outras informações:

```
1 void helloworld_setup(void) {
2    helloworld_class = class_new(
3       gensym("helloworld"),           // Nome simbolico
4       (t_newmethod) helloworld_new,   // Construtor
5       (t_method) helloworld_destroy, // Destrutor
6       sizeof (t_helloworld),          // Tamanho do objeto
7       CLASS_NOINLET,                  // Flags com o tipo da
            classe
8       0);                             // Argumentos do construtor
9 }
```

Listing 2.2: Método setup

Dentro do método `<external>_setup()` não há limite para o número de classes a definir, de forma que é possível definir apenas uma classe (como no exemplo helloworld.c) ou uma biblioteca inteira com várias classes (como no exemplo 3). A introdução de uma nova classe no sistema é realizada pela função `class_new()`. São parâmetros da função `class_new()`:

- Nome simbólico da classe.
- Método construtor de um objeto.
- Método destrutor de um objeto.
- Tamanho do espaço de dados dos atributos de um objeto.
- Flags que definem a representação gráfica do objeto.
- Tipos dos parâmetros a serem passados para o construtor quando da instanciação de um objeto (veja o próximo capítulo).

Os tipos de Flags aceitas para representar um objeto são:

- CLASS_DEFAULT - Para objetos padrões do PD com 1 inlet
- CLASS_NOINLET - Objetos sem inlet "Mágico"
- CLASS_PD - Para objetos sem representação gráfica, como inlets
- CLASS_GOBJ - Para objetos gráficos como arrays e graphs
- CLASS_PATCHABLE - Para objetos internos do PD como "message" ou "text"

É necessário terminar a lista de tipos de parâmetros com um número inteiro 0, para indicar ao Pure Data que a lista de tipos terminou. Consulte a documentação da função `class_new()` para mais detalhes[1].

O método `<external>_new()`, que foi associado como método de instanciação de objetos na chamada de `class_new()`, realiza a instanciação de objetos propriamente dita. Neste método, além da instanciação de um novo objeto através da função `pd_new()`, é possível definir os valores dos atributos da estrutura de dados da classe e também inicializar quaisquer outros contextos que sejam necessários, como por exemplo abrir arquivos, preencher vetores, alocar memória, etc.

```
1 // ------------------------------------------------
2 // Construtor da classe
3 // ------------------------------------------------
4 void * helloworld_new(void){
5    t_helloworld *x = (t_helloworld *) pd_new(helloworld_class);
```

[1] `http://pdstatic.iem.at/externals-HOWTO/node9.html#SECTION00092100000000000000`

```
6     // Something else?
7     return (void *) x;
8 }
```

Listing 2.3: Construtor de uma classe

Após a criação da estrutura de dados dos métodos da forma mencionada acima, a compilação realizada da forma descrita na seção 1.3, e a criação do objeto no Pure Data como descrito na seção 1.5, o resultado pode ser visto na figura 2.1.

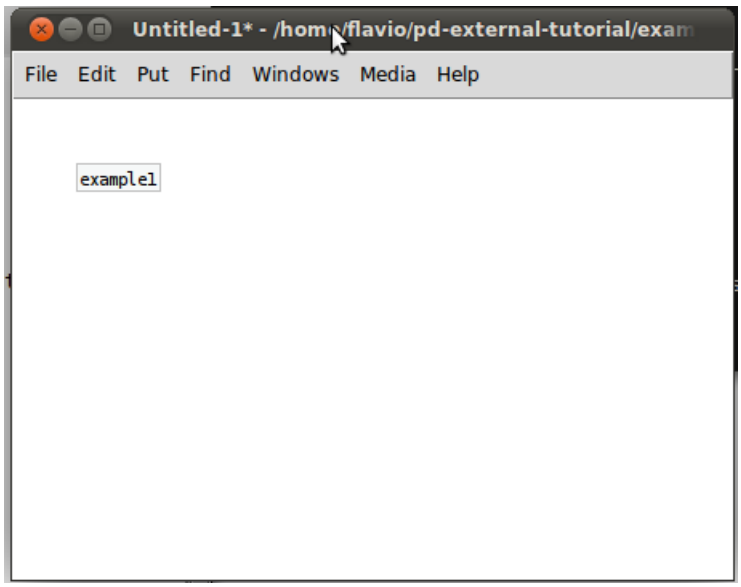

Figura 2.1: Nosso primeiro *external* do PD. Ainda inútil. :-)

2.2 Uma biblioteca simples

Um mesmo método `<external>_setup()` pode definir várias classes diferentes. A isto damos o nome de biblioteca. Neste cenário, o método `<external>_setup()` possui o mesmo nome do arquivo com a biblioteca, mas cada classe podem ter um nome diferente (veja o exemplo 3).

```
1  void example3_setup(void) {
2    post("Initializing my library");
3
4    myobj1_class = class_new(
5      gensym("myobj1"),
6      (t_newmethod) myobj1_new, // Constructor
7      0,
8      sizeof (t_myobj1),
9      CLASS_NOINLET,
10     0);
11   class_sethelpsymbol(myobj1_class, gensym("myobj1-help"));
12
13   myobj2_class = class_new(
14     gensym("myobj2"),
15     (t_newmethod) myobj2_new, // Constructor
16     0,
17     sizeof (t_myobj2),
18     CLASS_NOINLET,
19     0);
20   class_sethelpsymbol(myobj2_class, gensym("myobj2-help"));
```

```
21 }
```

Listing 2.4: Exemplo de arquivo com duas classes

Se o arquivo foi preenchido corretamente, compilado corretamente e adicionado ao caminho do PureData, teremos o resultado visto na figura 2.2.

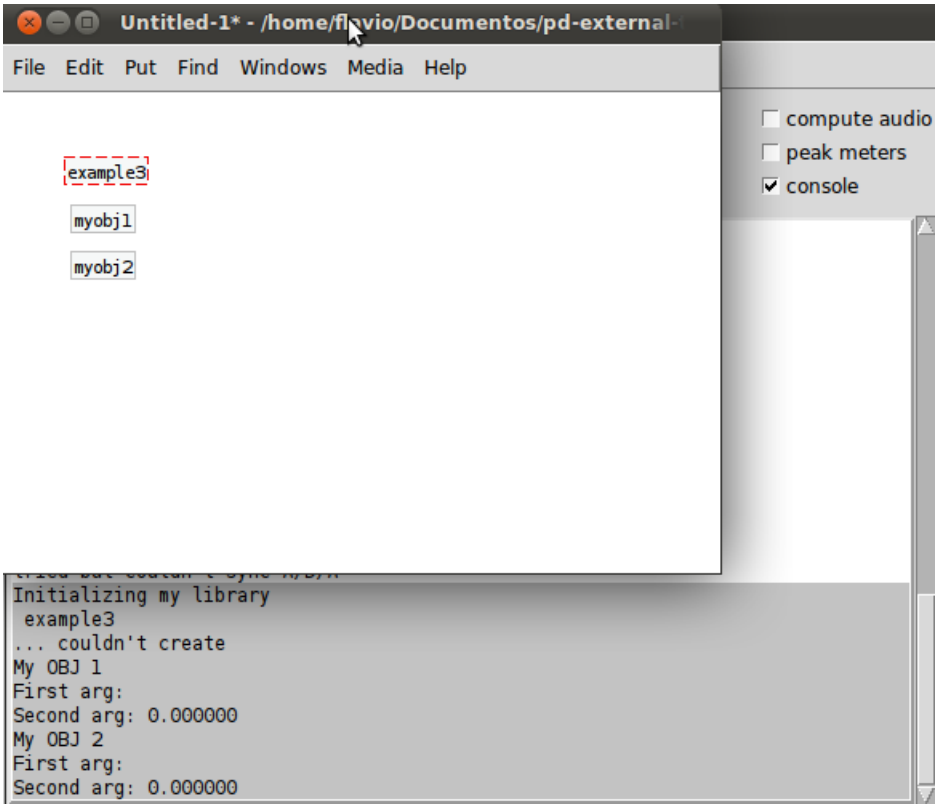

Figura 2.2: Nosso segundo *external* do PD. Ainda inútil. :-)

Apesar de este tipo de biblioteca ser bastante utilizado no código-fonte do PD, é recomendado separar os *external* em arquivos individuais para simplificar a leitura e manutenção do código-fonte. Sendo assim, é recomendado que o *external* "myobj1" esteja em um arquivo chamado "myobj1.c" e que o mesmo seja feito com o *external* "myobj2".

Dentro do Pure Data, um clique com o botão direito em um objeto abre um menu no qual uma das opções é `Ajuda`. Quando esta opção é selecionada, o Pure Data abre um patch associado ao objeto, que deve conter instruções e exemplos de uso. Por padrão, o Pure Data procura um arquivo com o mesmo nome que o external (acrescido da extensão `-help.pd`) no diretório padrão de documentação (`doc/5.reference`). Para associar um arquivo diferente do padrão, basta utilizar a função `class_sethelpsymbol`:

```
1 class_sethelpsymbol(myclass_class, gensym("my_class-help"));
```

Listing 2.5: Definição de arquivo de help

Um objeto pode ainda ter outros nomes (*aliases*). Para definir isto podemos utilizar a função `class_addcreator()`. Veja o exemplo:

```
1 class_addcreator((t_newmethod)medusa_new, gensym("med"), 0);
```

Listing 2.6: Definição de alias para um objeto

Exemplos comuns de aliases são os objetos [send], [receive] e [trigger], que podem ser instanciados pelos aliases [s], [r] e [t] respectivamente.

2.3 Variáveis globais

É possível utilizar variáveis globais para armazenar dados de um *external*. Estas variáveis são visíveis para todas as intâncias de objetos do *external* e todas podem alterar seus valores. Isto pode ser útil ou um desastre (veja o exemplo16). Por exemplo, cada instância do *external* `example16` definido a partir do código a seguir incrementa em uma unidade o valor do contador, como pode ser visto na figura 2.3:

```
1 int count = 0;
2
3 void * example16_new(void) {
4     t_example16 *x = (t_example16 *) pd_new(example16_class);
5     post("Counter value: %d",count);
6     count++;
7     return (void *) x;
8 }
```

Listing 2.7: Exemplo de uma variável global

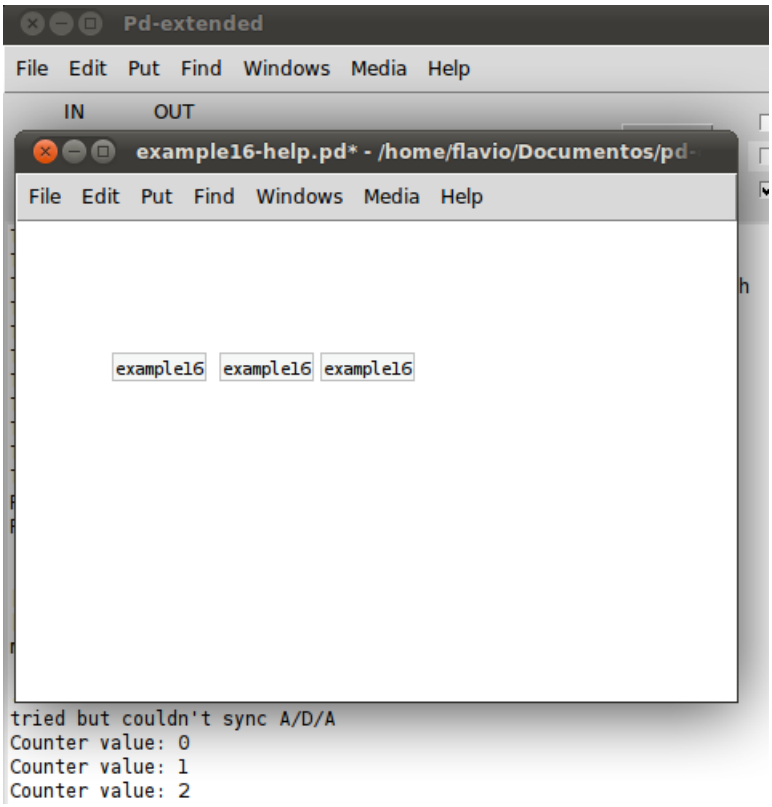

Figura 2.3: Repare na saída da janela principal.

Caso isto não seja desejável, o ideal é incluir as variáveis dentro da estrutura do objeto. Assim, neste exemplo cada instância terá seu próprio contador:

```
1 static t_class *counter_class;
2
3 typedef struct _counter {
4    t_object x_obj;
5    t_float counter;
6    t_outlet * x_outlet_output_float;
7 } t_counter;
8
```

```
9  void * counter_new(void){
10     t_counter *x = (t_counter *) pd_new(counter_class);
11     x->counter = 0;
12     x->x_outlet_output_float = outlet_new(&x->x_obj, gensym("float
           "));
13     return (void *) x;
14 }
```

Listing 2.8: Objeto contador

Vale notar que adicionar a estrutura de dados do *external* os atributos necessários para seu funcionamento é a abordagem padrão para o desenvolvimento. É recomendado que o compartilhamento de dados entre dois objetos ocorra pela troca de mensagem e não por variáveis globais.

Capítulo 3

Os tipos de dados do PD

Uma vez que o Pure Data é utilizado em diversas plataformas, muitos tipos comuns de variáveis, como `int`, são redefinidos. Para escrever um *external* que seja portável para qualquer plataforma, é razoável que você utilize os tipos providos pelo Pure Data. Como dissemos na seção 1.2, para escrever um *external*, é necessário incluir o arquivo `m_pd.h` que possui definições de constantes (versão do Pure Data, sistema operacional, compilador, etc), estruturas, assinaturas de funções e tipos de dados.

Existem muitos tipos predefinidos que devem fazer a vida do programador mais simples. Em geral, os tipos do pd têm nome iniciado por `t_`.

tipo do pd	descrição
`t_atom`	átomo
`t_float`	valor de ponto flutuante
`t_symbol`	símbolo
`t_gpointer`	ponteiro (para objetos gráficos)
`t_int`	valor inteiro
`t_signal`	estrutura de um sinal
`t_sample`	valor de um sinal de áudio (ponto flutuante)
`t_outlet`	*outlet* de um objeto
`t_inlet`	*inlet* de um objeto
`t_object`	objeto gráfico
`t_class`	uma classe do pd
`t_method`	um método de uma classe
`t_newmethod`	ponteiro para um construtor (uma função `_new`)

3.1 Símbolos

Um símbolo corresponde a um valor constante de uma *string*, ou seja, uma sequência de letras que formam uma palavra única.

Cada símbolo é armazenado em uma tabela de busca por razões de performance. A função `gensym(char *)` procura por uma string em uma tabela de busca e retorna o endereço daquele símbolo. Se a string não foi encontrada na tabela, um novo símbolo é adicionado.

Estes símbolos serão usados para várias coisas como para criar e associar mensagens entre objetos, definir ações esperadas para mensagens recebidas por inlets, criar comunicação entre a GUI e o Pd, entre outras.

Para imprimir, por exemplo, o valor de uma String contida em um `t_symbol` é necessário acessar sua propriedade `s_name`.

Exemplo:

```
1    printf("%s\n", my_symbol->s_name);
```

Listing 3.1: Imprimindo o texto de um símbolo

3.2 Mensagens

Dados que não correspondem a áudio são distribuídos via um sistema de mensagens. Cada mensagem é composta de um "seletor"e uma lista de átomos.

3.2.1 Átomos

Um átomo é um tipo de dado do PD que possui um valor e uma identificação. Os tipos de átomo mais utilizados são:

- `A_FLOAT`: um valor numérico (de ponto flutuante).
- `A_SYMBOL`: um valor simbólico (string).
- `A_POINTER`: um ponteiro.

Valores numéricos são sempre considerados valores de ponto flutuante (`t_float`), mesmo que possam ser exibidos como valores inteiros.

Átomos do tipo `A_POINTER` não são muito importantes (para *externals* simples).

O tipo de um átomo `a` é armazenado no elemento da estrutura `a.a_type`.

Outros tipos de átomo definidos no arquivo `m_pd.h` são:

- `A_NULL`,
- `A_FLOAT`,
- `A_SYMBOL`,
- `A_POINTER`,
- `A_SEMI`,
- `A_COMMA`,
- `A_DEFFLOAT`,
- `A_DEFSYM`,
- `A_DOLLAR`,
- `A_DOLLSYM`,
- `A_GIMME`,
- `A_CANT`

Nem todos estes átomos são utilizados no desenvolvimento de *externals* e alguns são representações internas do PD para símbolos reservados como vírgula, cifrão ou nulo. Assim, se um objeto precisa passar para outro objeto um valor nulo, um cifrão ou vírgula, estes tipos devem ser utilizado.

A manipulação de átomos pode ser feita pelas seguintes funções:

`SETFLOAT`

`SETFLOAT(atom, f)`

Esta macro define o tipo do átomo como `A_FLOAT` e armazena no mesmo o valor de f. É necessário passar um ponteiro para o átomo.

`SETSYMBOL`

`SETSYMBOL(atom, s)`

Esta macro define o tipo do átomo como `A_SYMBOL` e armazena no mesmo um ponteiro para o símbolo s. É necessário passar um ponteiro para o átomo.

`SETPOINTER`

`SETPOINTER(atom, pt)`
Esta macro define o tipo do átomo como `A_POINTER` e armazena no mesmo o ponteiro pt. É necessário passar um ponteiro para o átomo.

`atom_getfloat`

`t_float atom_getfloat(t_atom *a)`
Se o átomo for do tipo `A_FLOAT`, retorna o valor do float, caso contrário, retorna 0.0.

`atom_getfloatarg`

`t_float atom_getfloatarg(int which, int argc, t_atom *argv)`
Se o tipo do átomo encontrado na lista de átomos argv, de tamanho argc e na posição which for `A_FLOAT`, retorna o valor deste átomo. Caso contrário, retorna 0.0.

`atom_getint`

`t_int atom_getint(t_atom *a)`
Se o átomo for do tipo `A_INT`, retorna o valor inteiro, caso contrário, retorna 0.

`atom_getintarg`

`t_int atom_getintarg(int which, int argc, t_atom *argv)`
Se o tipo do átomo encontrado na lista de átomos argv, de tamanho argc e na posição which for `A_INT`, retorna o valor deste átomo. Caso contrário, retorna 0.

`atom_getsymbol`

`t_symbol atom_getsymbol(t_atom *a)`
Se o átomo for do tipo `A_SYMBOL`, retorna um ponteiro para este símbolo, caso contrário, retorna "0".

`atom_getsymbolarg`

`t_symbol atom_getsymbolarg(int which, int argc, t_atom *argv)`
Se o tipo do átomo encontrado na lista de átomos argv, de tamanho argc e na posição which for `A_SYMBOL`, retorna um ponteiro para este símbolo. Caso contrário, retorna "0".

`atom_gensym`

`t_symbol *atom_gensym(t_atom *a)`
Se o átomo for do tipo `A_SYMBOL`, retorna um ponteiro para este símbolo. Átomos de outros tipos são convertidos de maneira "razoável" em string, adicionados na tabela de símbolos, e um ponteiro para este símbolo é retornado.

`atom_string`

`void atom_string(t_atom *a, char *buf, unsigned int bufsize)`
Converte um átomo em uma string (char *) previamente alocada e de tamanho bufsize.

3.2.2 Seletores

Um seletor é um símbolo que define o tipo de uma mensagem. Existe cinco seletores pré-definidos:

- `bang`: rotula um gatilho de evento. Uma mensagem de `bang` consiste somente do seletor e não contém uma lista de átomos.
- `float` rotula um valor numérico. A lista de uma mensagem `float` contém um único átomo de tipo `A_FLOAT`.

- `symbol` rotula um valor simbólico. A lista de uma mensagem `symbol` consiste em um único átimo do tipo `A_SYMBOL`.
- `pointer` rotula um valor de ponteiro. A lista de uma mensagem do tipo `pointer` contém um único átimo do tipo `A_POINTER`.
- `list` rotula uma lista de um ou mais átomos de tipos arbitrários.

Uma vez que os símbolos para estes seletores são utilizados com frequência, seu endereço na tabela de símbolos pode ser utilizado diretamente, sem a necessidade da utilização de `gensym`:

seletor	rotina de busca	endereço de busca
`bang`	`gensym("bang")`	`&s_bang`
`float`	`gensym("float")`	`&s_float`
`symbol`	`gensym("symbol")`	`&s_symbol`
`pointer`	`gensym("pointer")`	`&s_pointer`
`list`	`gensym("list")`	`&s_list`
`--` (signal)	`gensym("signal")`	`&s_signal`

Outros seletores também podem ser utilizados. A classe receptora tem que prover um médodo para um seletor específico ou para `anything`, que corresponde a qualquer seletor arbitrário.

Mensagens que não possuem seletor explícito e começam com um valor numérico são reconhecidas automaticamente como mensagens `float` (se consistirem de apenas um átomo) ou como mensagens `list` (se forem compostas de diversos átomos).

Por exemplo, as mensagens `12.429` e `float 12.429` são idênticas. Da mesma forma, as mensagens `list 1 para voce` é idêntica a `1 para voce`.

Cabe escrever sobre binbuf?

Capítulo 4

Construtor e destrutor

Um objeto em PD pode precisar de determinados parâmetros para ser iniciado. Por exemplo, para criar um oscilador é necessário passar para este objeto a frequência que o mesmo irá funcionar. Assim, o objeto recebe um parâmetro como, por exemplo, [osc 440]. Para receber parâmetros na criação de um objeto é necessário informar o PD do mesmo. Para isto, é necessário tanto informar os parâmetros na definição da classe quanto definir corretamente a assinatura da função do construtor. Estes parâmetros são ilustrados abaixo.

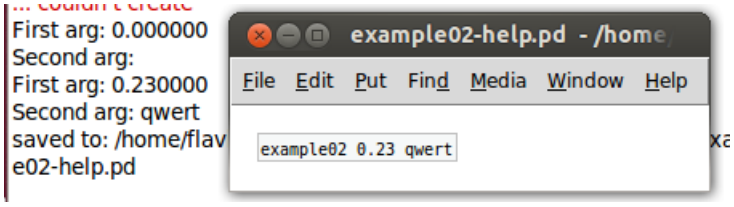

Figura 4.1: External recebendo parâmetros. Note a tela de saída no fundo da imagem.

4.1 Tipos de parâmetros

Os tipos de parâmetros aceitos para o construtor com checagem de tipo são:

- `A_DEFSYMBOL` para Strings
- `A_DEFFLOAT` para números

Por questões de implementação, o PD pode verificar apenas 5 parâmetros com tipo. Para passar mais parâmetros ao construtor é necessário utilizar um outro tipo de parâmetro:

- `A_GIMME` para qualquer tipo de parâmetro

Caso este tipo seja utilizado, o construtor deverá receber uma lista de átomos de tamanhos e tipos arbitrários e não faz a verificação de tipos.

A assinatura do construtor para receber ester parâmetros são:

```
1 void *myclassnew(void) // Construtor sem parametros
2 void *myclassnew(t_symbol *arg)  // Para parametro tipo
      A_DEFSYMBOL
3 void *myclassnew(tfloatarg arg) // Para parametro tipo A_DEFFLOAT
4 void *myclass_new(t_symbol *s , int argc , t_atom * argv) // Para
      A_GIMME
```

Listing 4.1: Assinatura do construtor

4.2 Construtor

Parâmetros de inicialização no construtor podem permitir que inicializemos o external com determinados valores. Isto é feito definindo os parâmetros no métodos `class_new()` quanto na definição da função construtora. (Veja o exemplo02).

```
1  // Constructos of the class
2  void * example02_new(t_floatarg arg1, t_symbol * arg2) {
3      t_example02 *x = (t_example02 *) pd_new(example02_class);
4      post("First arg: %f", arg1);
5      post("Second arg: %s", arg2->s_name);
6      return (void *) x;
7  }
8
9  void example2_setup(void) {
10     example2_class = class_new(gensym("example2"),
11             (t_newmethod) example2_new, // Constructor
12             0,
13             sizeof (t_example2),
14             CLASS_NOINLET,
15             A_DEFFLOAT, // First Constructor parameter
16             A_DEFSYMBOL, // Second Constructor parameter
17             0);
18 }
```

Listing 4.2: Passagem de parâmetro para o construtor

Notem que os parâmetros são definidos com um tipo e são recebidos com outro. Como explicado na seção 3.2, todos os dados que não correspondem a sinais de áudio são transmitidos como mensagens, compostas de átomos. Para ver os tipos de átomo que podem ser utilizados na passagem de parâmetros, veja a seção 3.2.1.

Para aceitar qualquer tipo de átomo na passagem de um parâmetro específico, utilize o tipo de átomo `A_GIMME` (veja o exemplo09).

```
1  // Constructor of the class
2  void * example9_new(t_symbol *s, int argc, t_atom * argv) {
3     t_example9 *x = (t_example9 *) pd_new(example9_class);
4     post("%d parameters received",argc);
5     return (void *) x;
6  }
7
8  void example9_setup(void) {
9     example9_class = class_new(gensym("example9"),
10      (t_newmethod) example9_new, // Constructor
11      (t_method) example9_destroy, // Destructor
12      sizeof (t_example9),
13      CLASS_NOINLET,
14      A_GIMME, // Allows various parameters
15      0); // LAST argument is ALWAYS zero
16 }
```

Listing 4.3: Objeto que recebe qualquer tipo de parâmetro

Quando utilizamos o tipo de átomo `A_GIMME` o método construtor funciona como uma função `main()` em C: ela recebe os parâmetros `argc`, que indica o número de átomos na lista, e `*argv`, que aponta para a lista de átomos de fato. Veja o exemplo na figura 4.2.

Neste caso, diferentemente da função main na linguagem C, o primeiro parâmetro não é o nome do external. O nome do external é o primeiro parâmetro recebido pela função, em nosso exemplo, "t_symbol *s".

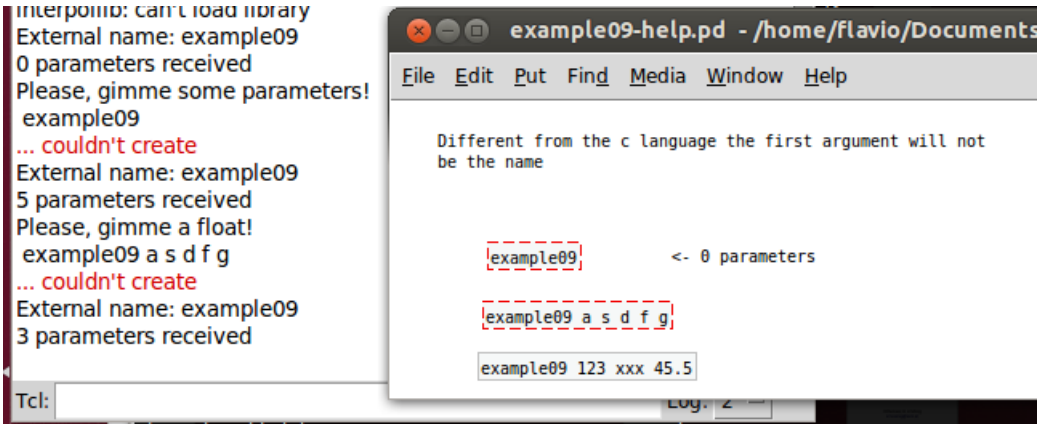

Figura 4.2: Diferente da linguagem C, o primeiro parâmetro não é o nome do external.

4.3 Validando parâmetros no construtor

Note que o Pure Data não obriga que o usuário passe parâmetros para o objeto. Todo construtor, independentemente de como ele está definido, aceita sua instanciação vazia. Cabe ao programador verificar se os parâmetros recebidos são em quantidade, tipo e valor esperado e, caso não seja, abortar a construção do objeto e não retornar sua instância.

```
1  // Constructor of the class
2  void * example9_new(t_symbol *s, int argc, t_atom * argv) {
3     t_example9 *x = (t_example9 *) pd_new(example9_class);
4     post("External name: %s", s->s_name);
5     post("%d parameters received",argc);
6     if(argc < 1){
7        post("Please, gimme some parameters!");
8        return NULL;
9     }
10    if(argv[0]->a_type != A_FLOAT{
11       post("Gimme a float!");
12       return NULL;
13    }
14    return (void *) x;
15 }
```

Listing 4.4: Validando parâmetros na construção de um objeto

Como podemos verificar no exemplo código acima, caso não seja passado um parâmetro do tipo float para o example09 o mesmo não irá retornar uma instância do objeto e apresentará a mensagem de que parâmetros devem ser passado ao objeto.

4.4 Outras tarefas para o construtor

Como deve-se saber, não é aconselhável alocar memória durante o bloco de processamento de sinais quando trabalhamos com processamento em tempo real. Por esta razão, é aconselhável alocar a memória de variáveis que iremos utilizar em nosso *external* no construtor.

Um exemplo de dados que deve ser alocado e instanciado no construtor seria uma tabela seno para criar um oscilador por consulta a tabela.

```
1  void *getbytes(size_t nbytes);
```

Listing 4.5: Função para a alocação de memória

A alocação de memória deve ser feita preferencialmente pela função `getbytes`. Esta função utiliza internamente a função padrão `malloc` porém é portável para os sistemas operacionais onde o PD funciona.

Outra tarefa que pode ser realizada pelo construtor é a criação de iolets passivos. Tal funcionalidade será coberta no próximo capítulo deste documento.

4.5 Destrutor

O destrutor de uma classe permite liberar alguma memória eventualmente alocada pelo construtor ou por outras funções do *external* (veja o exemplo 07).

```
1  // Destroy the object
2  void example9_destroy(t_example9 *x) {
3    post("You say good bye and I say hello");
4  }
5
6  void example9_setup(void) {
7     example9_class = class_new(gensym("example9"),
8       (t_newmethod) example9_new, // Constructor
9       (t_method) example9_destroy, // Destructor
10      sizeof (t_example9),
11      CLASS_NOINLET,
12      A_GIMME, // Allows various parameters
13      0); // LAST argument is ALWAYS zero
14 }
```

Listing 4.6: Exemplo de destrutor

De maneira análoga a alocação de memória, o PD também disponibiliza uma função portável para a liberação de memória. A liberação da memória pode ser feita utilizando a função `freebytes()` definida na API do Pure Data. Tal função deve chamar internamente a função padrão `free` sendo, porém, portável entre diferentes sistemas operacionais.

```
1  void freebytes(void *x, size_t nbytes)
```

Capítulo 5

Inlets e outlets

Os objetos que criamos até agora são inúteis. Não servem para nada, pois não se comunicam com outros objetos nem modificam sinais de áudio. Para dar utilidade a um *external*, é necessário que ele comunique com outros objetos do Pure Data. Isto é feito por meio de *inlets* e *outlets*, portas de entrada e saída (respectivamente) de sinais de áudio e/ou mensagens.

Neste capítulo vamos tratar exclusivamente de inlets e outlets de mensagens. Entre os inlets de mensagem há os tipos passivos e ativos, que os usuários costumam chamar de inlets frios e quentes. Inlets **passivos** são inlets cujo valor recebido é associada diretamente a um atributo do objeto. São chamados de passivos pois a alteração do seu valor não resulta na chamada de um método e a atribuição do valor recebido ao atributo do objeto é feita automaticamente. Inlets **ativos**, por outro lado, são associados a funções e permitem a execução de uma função arbitrária quando um valor é recebido no inlet.

As mensagens passadas para o objeto em seus inlets ocorre por passagem de valor para o caso de inteiros e por passagem de parâmetros para os demais tipos. Por isto, é necessário cuidado ao manipular tais mensagens pois a alteração do valor de um ponteiro implica na alteração do mesmo em todos os objetos que o recebe. Veja o exemplo `inverter.c`. Neste exemplo o valor de um Symbol é alterado e resulta no mesmo invertido. Caso você inverta este valor, salve o patch, feche-o e abra-o novamente, encontrará o valor deste símbolo invertido, conforme apresentado nas Figuras 5.1 e 5.2.

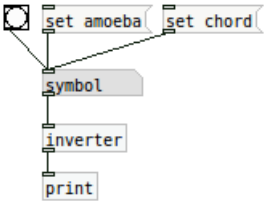

Figura 5.1: Alteração de uma mensagem recebida.

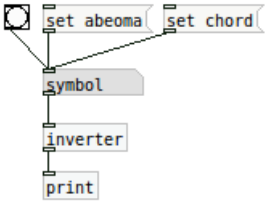

Figura 5.2: Alteração de uma mensagem recebida, ao reabrir.

5.1 Inlets ativos

Um inlet ativo sempre recebe mensagens no primeiro inlet do objeto e por isto o mesmo deve ser utilizado com a classe do tipo `CLASS_DEFAULT`. Estes inlets são sempre associados a uma função. A criação de um inlet ativo define o tipo do átomo que o inlet receberá (veja o exemplo 05 e o resultado na figura 5.3).

```
1  // all inlet-methods receive the object as their first argument.
2  void example5_bang(t_example5 *x) {
3    post("BANGED!");
4  }
5
6  void example5_anything(t_example5 *x, t_symbol *s, int argc,
       t_atom *argv){
7    post("ANYTHING!");
8  }
9
10 void example5_setup(void) {
11   example5_class = class_new(gensym("example5"),
12     (t_newmethod) example5_new, // Constructor
13     0,
14     sizeof (t_example5),
15     CLASS_DEFAULT,
16     0); // LAST argument is ALWAYS zero
17   class_addbang(example5_class, example5_bang);
18   class_addanything(example5_class, example5_anything);
19 }
```

Listing 5.1: Exemplo de objeto com inlet ativo

Neste exemplo, definimos duas funções associadas ao primeiro inlet, uma para receber uma mensagem `bang` e outra para receber qualquer tipo de dado. O resultado desta implementação pode ser vista na Figura 5.3.

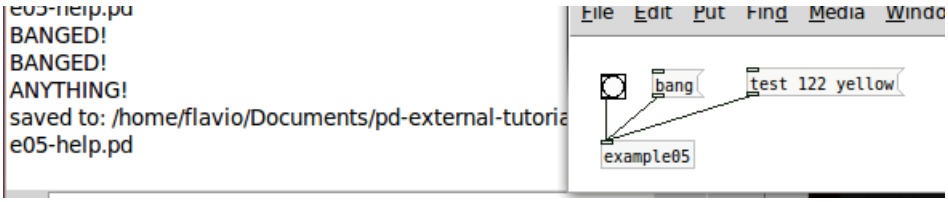

Figura 5.3: Inlets ativos.

Abaixo, a tabela com os métodos que criam inlets ativos, e assinaturas possíveis para funções associadas a cada tipo:

método do pd para criar inlet ativo	assinatura para o método associado ao inlet
`class_addbang(t_class *c, t_method fn);`	`my_b(t_myt *x);`
`class_addfloat(t_class *c, t_method fn);`	`my_f(t_myt *x, t_floatarg f);`
`class_addsymbol(t_class *c, t_method fn);`	`my_s(t_myt *x,t_symbol *s);`
`class_addpointer(t_class *c, t_method fn);`	`my_p(t_myt *x, t_gpointer *pt);`
`class_addlist(t_class *c, t_method fn);`	`my_l(t_myt *x, t_symbol *s, int argc, t_atom *argv);`
`class_addanything(t_class *c, t_method fn);`	`my_a(t_mydata *x, t_symbol *s, int argc, t_atom *argv);`

5.2 Mensagens para o primeiro inlet

Da mesma maneira que é possível mapear os tipos de dados recebidos no primeiro inlet para funções, também é possível definir tipos de mensagens separadamente. Isto é feito através da função `add_method()` (veja o exemplo 08). Esta função permite que a mensagem possua um identificador com seu tipo e outros dados que acompanham esta mensagem.

```
1  // Constructor of the class
2  void * example8_new(void) {
3    t_example8 *x = (t_example8 *) pd_new(example8_class);
4    return (void *) x;
5  }
6
7  void example8_start(t_example8 *x){
8    post("START / BANG");
9  }
10
11 void example8_open(t_example8 *x, t_symbol *s){
12   post("open %s",s->s_name);
13 }
14
15
16 void example8_alfa(t_example8 *x, t_floatarg f){
17   post("ALFA VALUE %f",f);
18 }
19
20 void example8_setup(void) {
21   example8_class = class_new(gensym("example8"),
22     (t_newmethod) example8_new, // Constructor
23     (t_method) example8_destroy, // Destructor
24     sizeof (t_example8),
25     CLASS_DEFAULT,
26     0); // LAST argument is ALWAYS zero
27   // All these messages will be received by the first left inlet
28   class_addmethod(example8_class, (t_method) example8_start,
29     gensym("start"), 0); // two messages, the same function
30   class_addmethod(example8_class, (t_method) example8_start,
31     gensym("bang"),  0); // may be "start" or "bang" messages
32   class_addmethod(example8_class, (t_method) example8_open,
33     gensym("open"),  A_DEFSYMBOL,0);
34   class_addmethod(example8_class, (t_method) example8_alfa,
35     gensym("alfa"), A_DEFFLOAT,0);
36 }
```

Listing 5.2: Passagem de mensagens para o primeiro inlet

A listagem acima mostra que as mensagens `bang` e `start` são associadas ao mesmo método. Além disto, a mensagem `open` recebe um texto como parâmetro e a mensagem `alfa` recebe um float como parâmetro. Como no construtor, a função `class_addmethod` pode receber uma lista de e receber um valor zero como último argumento.

Desta forma não precisamos tratar a mensagem que o inlet recebe mas definí-las de antemão e criar funções que mapeiem a mensagem recebida. Veja a Figura 5.4.

5.3 Inlets passivos

Um inlet passivo é a forma de o objeto receber uma mensagem que não está associado a uma função mas a um atributo do objeto. Assim, ao criarmos um inlet que recebe um valor float, o mesmo deverá alterar um atributo float do objeto sem que o mesmo possua uma função associada para verificar esta alteração. Por não possuir uma função associada, tal inlet é comumente chamado de "inlet frio". Abaixo vemos um exemplo de objeto com um inlet passivo (veja a figura 5.5):

```
1  static t_class *example4_class;
2
3  typedef struct _example4 {
4    t_object x_obj;
```

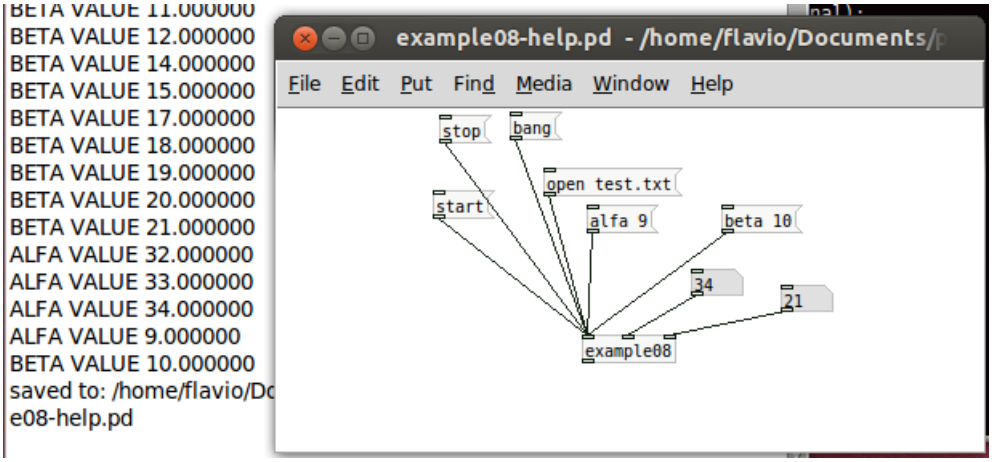

Figura 5.4: Mais inlets.

```
5     t_float my_float;
6   } t_example4;
7
8   // Constructor of the class
9   void * example4_new(t_symbol * arg1, t_floatarg arg2) {
10    t_example4 *x = (t_example4 *) pd_new(example4_class);
11    floatinlet_new(&x->x_obj, &x->my_float);
12    return (void *) x;
13  }
```

Listing 5.3: Exemplo de inlet passivo

Neste exemplo, o atributo `my_float` do objeto é associado a um inlet do tipo float. Isto significa que, caso tenhamos uma mensagem float ligada a este objeto, o valor desta mensagem ficará armazenada no atributo my_float.

Um inlet passivo é associado a um tipo do Pure Data, e requer que o atributo associado seja do mesmo tipo do valor recebido através do inlet. Para cada tipo do Pure Data, utiliza-se uma função diferente para criar inlets que recebam aquele tipo (veja o exemplo evenodd.c).

As funções para criar os inlets passivos são:

- `floatinlet_new(t_object *owner, t_float *fp)`
- `symbolinlet_new(t_object *owner, t_symbol **sp)`
- `pointerinlet_new(t_object *owner, t_gpointer *gp)`

Para adicionar inlets passivos de um tipo genérico, veja a Subseção Proxy de inlets adiante.

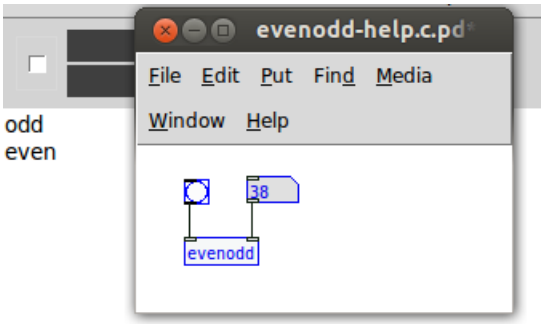

Figura 5.5: Inlet passivo (do arquivo exemplo evenodd.c)

5.4 Um inlet ativo extra

A função `inlet_new()` pode adicionar novos inlets a um objeto sem que estes inlets sejam passivos. Tal função depende de haver uma mensagem associada ao primeiro inlet e utiliza a função presente nesta associação para receber os dados deste inlet. Por esta razão, apesar de o mesmo parecer um inlet passivo, este novo inlet não é associado a um atributo mas a uma função associada a símbolos de mensagens:

```
1 t_inlet *inlet_new(t_object *owner, t_pd *dest,
2       t_symbol *s1, t_symbol *s2);
```

Listing 5.4: Criando inlets ativos extras

Neste caso, quando um átomo do tipo `s1` for recebido neste inlet, o mesmo será passado para a função associada a mensagem `s2`. Veja o exemplo abaixo.

```
1
2 // Constructor of the class
3 void * example08_new(void) {
4    t_example08 *x = (t_example08 *) pd_new(example08_class);
5    // creates inlets for distinct messages
6    inlet_new(&x->x_obj, &x->x_obj.ob_pd, gensym("float"), gensym(
        "alfa"));
7    inlet_new(&x->x_obj, &x->x_obj.ob_pd, gensym("float"), gensym(
        "beta"));
8    (...)
9 }
10
11 void example08_setup(void) {
12   example08_class = class_new(gensym("example08"),
13     (t_newmethod) example08_new, // Constructor
14    ...
15   // associate messages with inlets 2 and 3
16   class_addmethod(example08_class, (t_method) example08_alfa,
        gensym("alfa"), A_DEFFLOAT,0);
17   class_addmethod(example08_class, (t_method) example08_beta,
        gensym("beta"), A_DEFFLOAT,0);
18 }
```

Neste exemplo, criamos um método para aceitar as mensagens `alfa` e `beta`. Se estas mensagens forem recebidas pelo inlet quente, suas funções serão chamadas para tratar a mensagem. Além disto, dois inlets passivos foram criados para receber dados do tipo float e tais inlets foram associados às mensagens `alfa` e `beta`. Por esta razão este inlet não é tão passivo assim. O método para processar a mensagem alfa será chamado tanto se esta mensagem for enviada quanto se um valor float for passaro para o segundo inlet, como mostrado na figura 5.6.

5.5 Proxy de inlets

Com os inlets apresentados até agora é impossível criar um objeto que possua um inlet passivo que aceite qualquer tipo de mensagem pois o PD não possui um método para isto. Tal implementação só é possível utilizando um proxy de inlet. Esta técnica pode ser vista no exemplo "yourclass.c"[1].

A ideia básica deste *external* é definir não um mas dois objetos sendo primeiro utilizado para suas funcionalidades e o outro utilizado apenas para ser o proxy.

```
1 /*
2  * declare the proxy object class
3  */
```

[1]Exemplo adaptado do site:`http://puredata.info/Members/mjmogo/proxy-example-for-pd.zip/at_download/file`. Este exemplo possui modificações feitas pelos autores deste tutorial.

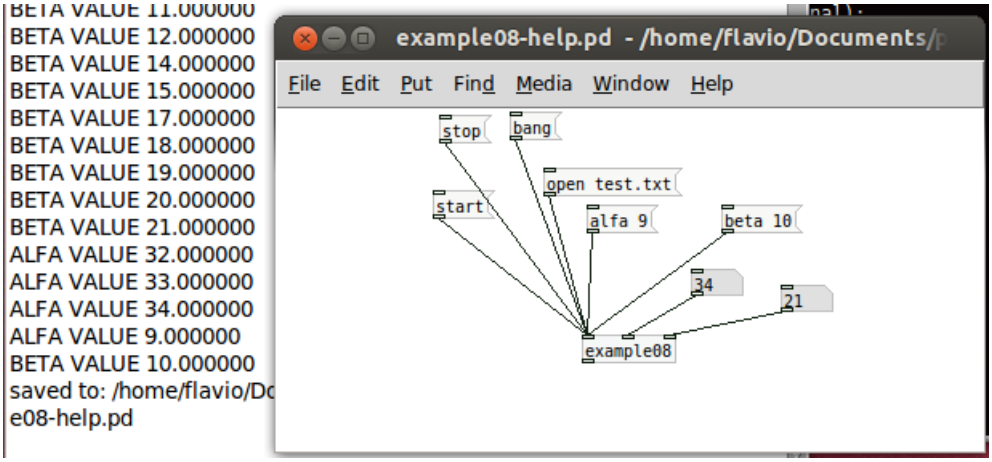

Figura 5.6: Inlets criado pela função inlet_new

```
4  t_class *proxy_class = NULL;
5
6  /*
7   * declare your class
8   */
9  t_class *yourclass_class = NULL;
10
11 typedef struct _proxy {
12   t_pd l_pd;
13   // if you want to maintain a pointer to your main class,
14   // if not, besure to change the 'init' function
15   void *yourclass;
16 } t_proxy;
17
18 typedef struct _yourclass {
19   t_object x_obj;
20   t_proxy pxy;
21 } t_yourclass;
```

Listing 5.5: Estruturas de dados para um proxy

A classe proxy não tem um atributo do tipo `t_object` mas um um objeto do tipo `t_pd`. A sua classe terá um atributo do tipo sua classe proxy.

O próximo passo é o método setup de sua classe chamar o método setup da classe proxy, que contará com um inlet do tipo anything.

```
1  static void proxy_setup(void) {
2    post("proxy_setup");
3    proxy_class =
4      (t_class *)class_new(gensym("proxy"),
5                 (t_newmethod)proxy_new,
6                 (t_method)proxy_free,
7                 sizeof(t_proxy),
8                 0,
9                 A_GIMME,
10                0);
11   class_addanything(proxy_class, (t_method)proxy_anything);
12 }
13
14 void yourclass_setup(void) {
15   post("yourclass_setup");
16   yourclass_class =
```

```
17      (t_class *)class_new(gensym("yourclass"),
18                  (t_newmethod)yourclass_new,
19                  (t_method)yourclass_free,
20                  sizeof(t_yourclass),
21                  0,
22                  A_GIMME,
23                  0);
24
25    // be sure to call the proxy class setup before we finish
26    proxy_setup();
27  }
```

Listing 5.6: configuração de um inlet proxy

Na criação da sua classe, o atributo `yourclass` e o atributo `l_pd` da classe proxy recebem atribuições. Um inlet é criado e associado a classe proxy e seu método `proxy_anything`.

```
1  static void *yourclass_new(t_symbol *s, int argc, t_atom *argv) {
2     t_yourclass *x = (t_yourclass *)pd_new(yourclass_class);
3     if (x) {
4        // first make the sql_buffer
5        x->pxy.l_pd = proxy_class;
6        x->pxy.yourclass = (void *) x;
7
8        // this connects up the proxy inlet
9        inlet_new(&x->x_obj, &x->pxy.l_pd, 0, 0);
10       post("yourclass_new");
11    }
12    return x;
13 }
```

Listing 5.7: Criação da classe com um inlet proxy

Agora basta definir o método anything para a classe proxy e um método anything para a sua clase.

```
1  void yourclass_anything(t_yourclass *x, t_symbol *s, int argc,
       t_atom *argv){
2     int i;
3     char buf[SYMBOL_LENGTH];
4     post("yourclass_anything: %s", s -> s_name);
5     for(i = 0; i < argc; i++) {
6        atom_string(&argv[i], buf, SYMBOL_LENGTH);
7        post("argv[%d]: %s", i, buf);
8     }
9
10 }
11
12 static void proxy_anything(t_proxy *x, t_symbol *s, int argc,
       t_atom *argv){
13    post("Proxy Anything");
14    yourclass_anything((t_yourclass *) x->yourclass, s, argc, argv
          );
15 }
```

Listing 5.8: Passagem de dados da classe proxy para a classe principal

Vale notar que nem sempre é necessário que a classe proxy passe os dados para a classe que possui o inlet, podendo o tratamento da mensagem ser feito na função do próprio inlet.

O resultado desta implementação pode ser verificado na Figura5.7.

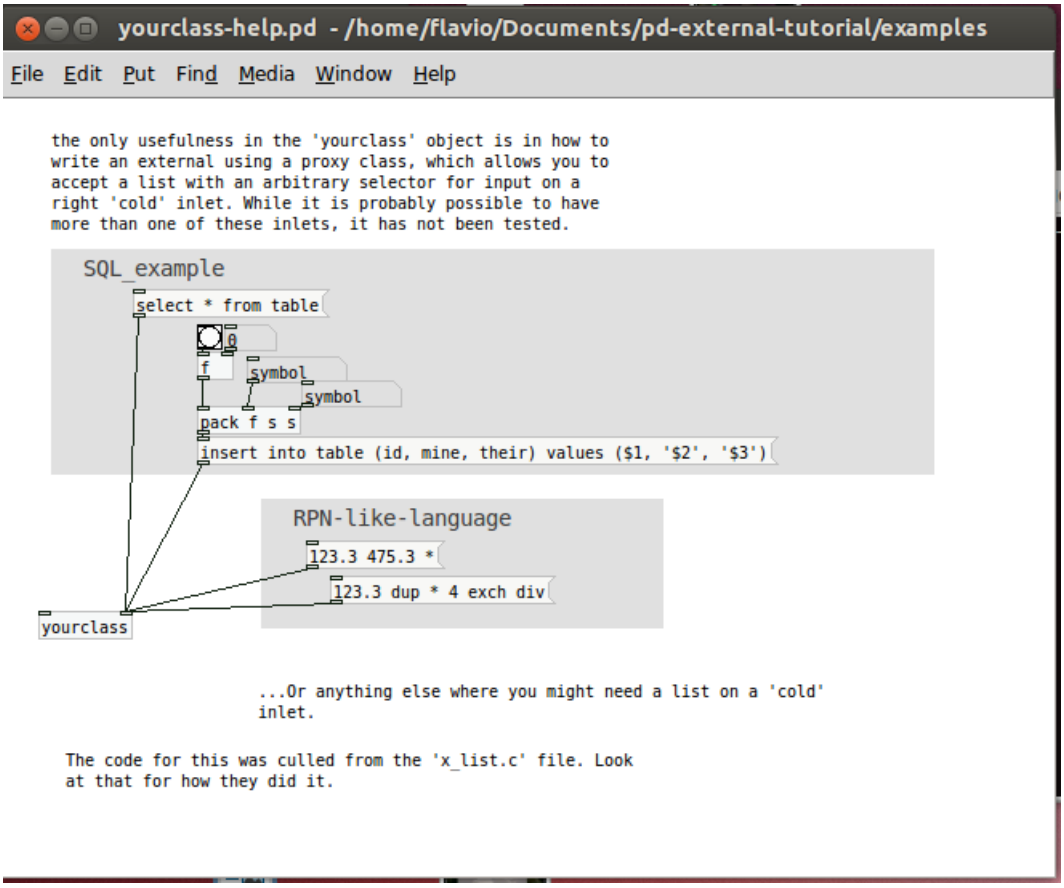

Figura 5.7: Proxy

5.6 Oulets

Depois de termos tratado as formas de entrada de dados através de inlets do Pure Data, chegou a hora de falarmos das saídas. A saída de dados dos objetos do Pure Data é feita por meio de outlets (veja o exemplo 06).

```
1  typedef struct _example6 {
2    t_object x_obj;
3    t_outlet *my_outlet; // Defines an outlet
4  } t_example6;
5
6  // The BANG method, first inlet
7  void example6_bang(t_example6 *x) {
8    post("BANGED!");
9    outlet_bang(x->my_outlet); // Bang my outlet
10 }
11
12 // Constructor of the class
13 void * example6_new(t_symbol * arg1, t_floatarg arg2) {
14   t_example6 *x = (t_example6 *) pd_new(example6_class);
15   x->my_outlet = outlet_new(&x->x_obj, gensym("bang"));
16   return (void *) x;
17 }
18
19 void example6_setup(void) {
20     example6_class = class_new(gensym("example6"),
21       (t_newmethod) example6_new, // Constructor
22       0,
```

```
23          sizeof (t_example6),
24          CLASS_DEFAULT,
25          A_DEFFLOAT, // First Constructor parameter
26          A_DEFSYMBOL, // Second Consctructo parameter
27          0); // LAST argument is ALWAYS zero
28      class_addbang(example6_class, example6_bang);
29  }
```

Listing 5.9: Exemplo de outlet

Um outlet deve ser definido na estrutura do objeto e instanciado pela função `outlet_new()`, definindo também o tipo do átomo associado. No caso deste exemplo, o outlet é do tipo `bang` e dispara um bang toda vez que recebe um bang (veja a figura 5.8).

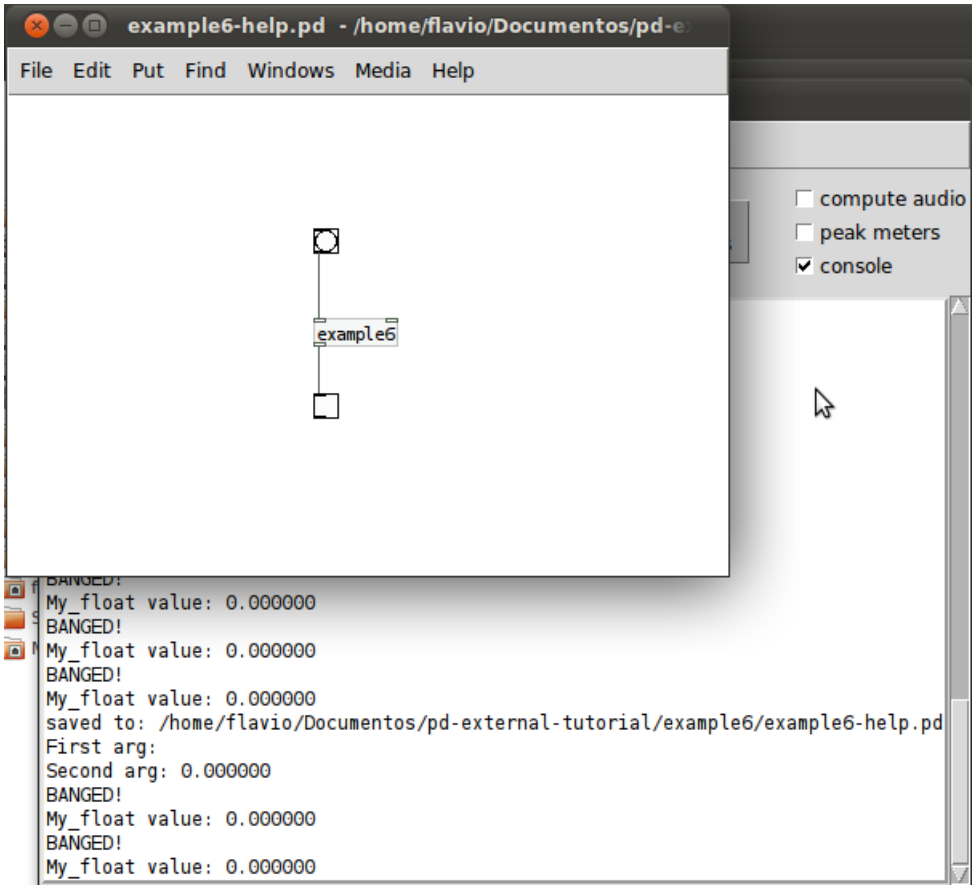

Figura 5.8: Um external bem útil que recebe um bang e envia um bang.

Há funções definidas para enviar vários tipos diferentes para um outlet. São elas:

- void outlet_bang(t_outlet *x);
- void outlet_pointer(t_outlet *x, t_gpointer *gp);
- void outlet_float(t_outlet *x, t_float f);
- void outlet_symbol(t_outlet *x, t_symbol *s);
- void outlet_list(t_outlet *x, t_symbol *s, int argc, t_atom *argv);
- void outlet_anything(t_outlet *x, t_symbol *s, int argc, t_atom *argv);

Vale lembrar que é de bom tom liberar a memória alocada para o outlet no destrutor da classe. Esta desalocação pode ser feita pela função abaixo:

```
1    outlet_free(x->x_outlet_inverted_symbol);
```

Listing 5.10: Desalocando a memória

5.7 IOlets dinâmicos

Alguns objetos, como o trigger, cria outlets dinamicamente conforme a quantidade de parâmetros recebidos. Tal abordagem pode ser utilizada tanto para inlets passivos quanto para outlets pois a criação destes ocorre no método construtor e não no método setup da classe. No exemplo "multiplus.c", a quantidade de inlets e outlets depende de um parâmetro passado para o construtor. Neste caso, guardamos na estrutura do objeto uma lista de outlets e um atributo com a quantidade de outlets.

```
1  typedef struct _multiplus {
2     t_object x_obj;
3     t_float count;
4     t_float * values;
5     t_outlet ** my_outlets; // Defines a outlet
6  } t_multiplus;
```

Listing 5.11: Estrutura de um objeto com outlets dinâmicos

No construtor, dependendo da passagem de um parâmetro que nos diz quantos outlets desejamos possuir, criamos e alocamos dinamicamente os outlets.

```
1  void * multiplus_new(t_floatarg count_arg){
2     t_multiplus *x = (t_multiplus *) pd_new(multiplus_class);
3     x->count = count_arg;
4     x->my_outlets = getbytes(x->count * sizeof(t_outlet*));
5     x->values = getbytes(x->count * sizeof(t_float));
6     int i = 0;
7     for(i = 0 ; i < (int) x->count; i++){
8        x->my_outlets[i] = outlet_new(&x->x_obj, gensym("bang"));
9        floatinlet_new (&x->x_obj, &x->values[i]);
10    }
11    return (void *) x;
```

Listing 5.12: Criação dinâmica de inlets e outlets

Não esquecer de desalocar os outlets no destrutor dos objetos.

```
1  void outlet_dinamico_destroy(t_outlet_dinamico *x) {
2     int i = 0;
3     for(i = 0 ; i < (int) x->outlet_count; i++){
4       outlet_free(x->my_outlets[i]);
5     }
6  }
```

Listing 5.13: Destrutor com outlets dinâmicos

Neste exemplo hipotético e talvez nada útil, ao receber uma mensagem de bang o objeto irá retornar os valores recebidos em seus inlets + 1.

```
1  void multiplus_bang(t_multiplus *x){
2     int i = 0;
3     for(; i < (int) x->count ; i++){
4        outlet_float(x->my_outlets[i], x->values[i] + 1);
5     }
6  }
```

Listing 5.14: Destrutor com outlets dinâmicos

O resultado desta implementação pode ser visto na Figura5.9.

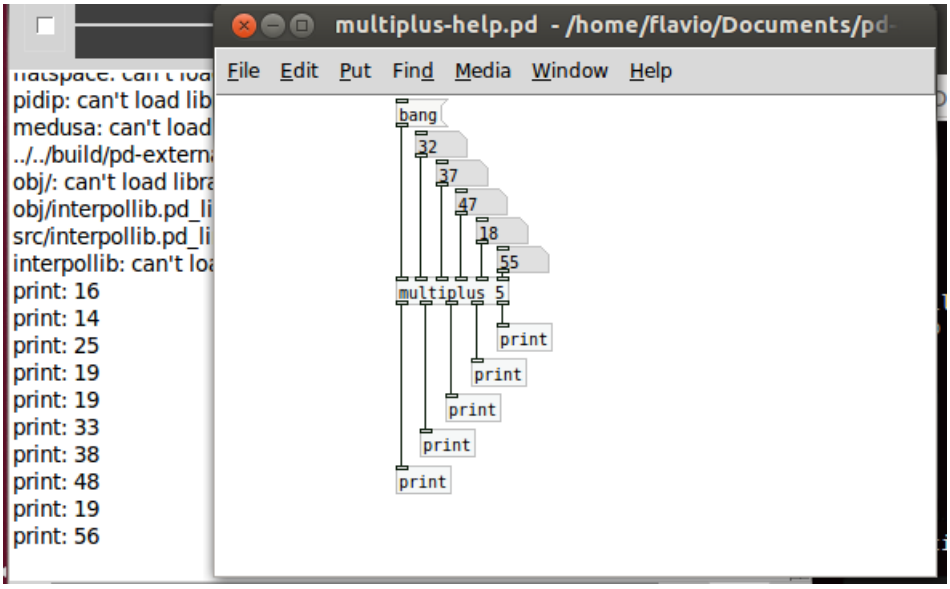

Figura 5.9: Inlets e outlets criados dinamicamente.

Capítulo 6

Processamento de Sinais Digitais

Enfim chegamos no processamento de áudio propriamente dito: *Digital Signal Processing* ou processamento de sinal digital. O Pure Data possui inlets e outlets específicos para o processamento de sinal. É fácil reconhecer: eles são pintados de cinza escuro. Além disto, classes que operam com processamento de sinal e que possuem iolets DSP costumam ser nomeadas `class ~`.

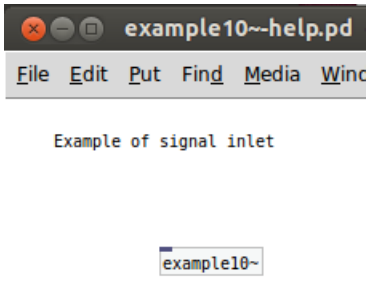

Figura 6.1: Primeiro Inlet DSP

6.1 O método DSP

Para que um objeto faça processamento de áudio, o primeiro passo é adicionar ao mesmo um método DSP. O método DSP é definido como os outros métodos tendo seu símbolo associado a mensagem `dsp` e nenhum parâmetro.

```
1  void dsponbang_tilde_setup(void) {
2     dspbang_class = class_new(gensym("dsponbang~"),
3        (t_newmethod) dsponbang_new, // Constructor
4        (t_method) dsponbang_destroy, // Destructor
5        sizeof (t_dsponbang),
6        CLASS_NOINLET,
7        0);//Must always ends with a zero
8
9     class_addmethod(dsponbang_class, (t_method) dsponbang_dsp,
         gensym("dsp"), 0);
10 }
```

Listing 6.1: Adicionando um método para DSP

Este método irá chamar a função associada ao mesmo toda vez que o DSP do Pure Data for ligado. Este método recebe como parâmetro um bloco de sinais do PD. No exemplo dspbang.c, toda vez que o DSP for ligado, o objeto irá emitir um bang.

```
1  static void dsponbang_dsp(t_dsponbang *x, t_signal **sp){
2     (void) sp;
```

```
3     outlet_bang(x->x_outlet_output_bang);
4 }
```

Listing 6.2: O método DSP

6.2 A função Perform

Normalmente, o método DSP é utilizado para adicionar o objeto ao ciclo DSP do PD. Isto é feito pelo método de processamento de sinais definido pela função `dsp_add()`.

```
1 static void dspbang_dsp(t_dspbang *x, t_signal **sp){
2     dsp_add(dspbang_perform, 1, x);
3 }
```

Listing 6.3: O método DSP add

A função `dsp_add` recebe como parâmetros uma função que será chamada a cada bloco de DSP do PD, o número de parâmetros que será passado para esta função e estes parâmetros. Neste exemplo, apenas um parâmetro é passado, o próprio objeto.

```
1 static t_int * dspbang_perform(t_int *w){
2     t_dspbang *x = (t_dspbang *)(w[1]);
3     outlet_bang(x->x_outlet_output_bang);
4     return (w + 2); // proximo bloco
5 }
```

Listing 6.4: A função perform

A função perform, definida como função DSP, receberá como parâmetro um ponteiro. O primeiro parâmetro deste ponteiro é o endereço da própria função. Os demais parâmetros são os parâmetros definidos na definição da função, neste caso, o próprio objeto. Esta função deverá retornar o próximo bloco de processamento, o que significa o vetor de entrada acrescido do tamanho de seus argumentos mais um.

Todo método de processamento de sinais será executado em todo ciclo DSP enquanto o processamento de sinais estiver ligado para o Pure Data ou para aquela janela específica, através do objeto `switch` . Por isto, cuidado com alocações de memória, inicialização de variáveis, etc.

Podemos passar para o método perform quaisquer parâmetros em qualquer ordem. Só é importante e óbvio que devemos lembrar quais parâmetros foram passados e em qual ordem.

6.3 Primeiro inlet para DSP

Uma vez que já vimos como adicionar o método DSP e como utilizar adicionar nosso objeto na cadeia DSP do PD, podemos adicionar inlets e outlets para conexões de áudio. Caso o objeto possua apenas um inlet DSP, é possível utilizar o primeiro inlet para receber tanto mensagens quanto sinal de áudio.

Para este caso específico, é necessário possuir na estrutura de dados um atributo do tipo `t_float` para armazenar o valor de entrada do inlet.

```
1 typedef struct _example10 {
2     t_object x_obj;
3     t_float x_f; /* inlet value when set by message */
4 } t_example10;
```

Listing 6.5: Estrutura de dados para utilizar o primeiro inlet para áudio

Se o *external* necessita de apenas um inlet DSP, a macro `CLASS_MAINSIGNALIN()` define um inlet DSP no primeiro inlet da esquerda. Para esta macro funcionar, é necessário que a classe seja do tipo `CLASS_DEFAULT`, e que um método seja associado à mensagem "dsp", de forma que será executado quando o DSP for iniciado. A forma de declaração de outros inlets DSP será vista logo adiante.

```
void example10_tilde_setup(void) {
   example10_class = class_new(
      (...)
   );
   // this declares the leftmost, "main" inlet
   // as taking signals.
   CLASS_MAINSIGNALIN(example10_class, t_example10, x_f);
   class_addmethod(example10_class, (t_method) example10_dsp,
      gensym("dsp"), 0);
   class_addbang(example10_class, example10_bang_method);
}
```

Listing 6.6: Definição da macro que permite ao primeiro inlet receber sinal de áudio

Note que este inlet também poderá receber mensagens ou outros tipos de dados.

O próximo passo é definir o método DSP que associamos na função `_setup()`:

```
static void example10_dsp(t_example10 *x, t_signal **sp){
  // adds a method for dsp
  dsp_add(example10_perform, 3, sp[0]->s_vec, sp[0]->s_n, x);
}
```

Listing 6.7: Definição da função DSP

Neste exemplo, o método `example10_perform()` é associado ao processamento de áudio do Pure Data e será chamada em cada execução do ciclo DSP com 3 argumentos: o endereço para o sinal de entrada (`sp[0]->s_vec`), a quantidade de amostras no bloco (`sp[0]->s_n`), e o endereço da estrutura que contém sua instância (`x`).

O próximo passo é criar o método perform propriamente dito.

```
static t_int * example10_perform(t_int *w){
  t_float     *in = (t_float *) (w[1]);
  int          n  = (int) (w[2]);
  t_example10 *x  = (t_example10 *) (w[3]);
  // do something ...
  return (w + 4); // next block's address
}
```

Listing 6.8: Definição da função DSP

O método perform receberá como parâmetro um vetor com os valores definidos no método `dsp_add()`. A posição 0 deste vetor sempre conterá o endereço para a própria função `_perform()`. Nas próximas posições devemos rever o que definimos na função DSP acima.

Na posição 1, o sinal de entrada; na posição 2 o tamanho do vetor de entrada e na posição 3 a estrutura de dados correspondente ao objeto. Note que podemos enviar estes dados em outra ordem ou ainda enviar outros dados para a função `perform()`. Este método deve retornar a próxima posição do vetor, ou seja, o endereço dado na chamada somado com a quantidade de atributos do método mais um.

6.4 Vários inlets DSP

É possível definir vários inlets de DSP para um *external* (veja o exemplo 11). A criação de inlets adicionais não é feita no método `_setup()` mas sim no construtor do objeto. Quanto ao primeiro inlet, só é necessário definí-lo explicitamnete no construtor se a macro `CLASS_MAINSIGNALIN` não for utilizada.

```
// Constructor of the class
void * example11_new(t_symbol *s, int argc, t_atom * argv) {
    t_example11 *x = (t_example11 *) pd_new(example11_class);
    inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
       // second signal inlet
    inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
       // third signal inlet
```

```
6       inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
            // fourth signal inlet
7       return (void *) x;
8   }
```

Listing 6.9: Criação de vários inlets DSP

Neste caso, a utilização do método `class_addmethod` é exatamente igual à anterior, a menos de uma mudança na quantidade de parâmetros por causa da quantidade de inlets:

```
1   static void example11_dsp(t_example11 *x, t_signal **sp){
2     dsp_add(example11_perform, 6, sp[0]->s_vec, sp[1]->s_vec, sp
          [2]->s_vec, sp[3]->s_vec, sp[0]->s_n, x);
3   }
```

Listing 6.10: Definição do método DSP

Note que precisamos agora alterar a quantidade de parâmetros passadas ao método `_perform()`, que ficará assim:

```
1   static t_int * example11_perform(t_int *w){
2     t_float *in1 = (t_float *)(w[1]);
3     t_float *in2 = (t_float *)(w[2]);
4     t_float *in3 = (t_float *)(w[3]);
5     t_float *in4 = (t_float *)(w[4]);
6     int n = (int)(w[5]);
7     t_example11 *x = (t_example11 *)(w[6]);
8     return (w + 7); // proximo bloco
9   }
```

Listing 6.11: Definição da função perform

Neste ponto, este external deve ter uma aparência como na figura 6.2.

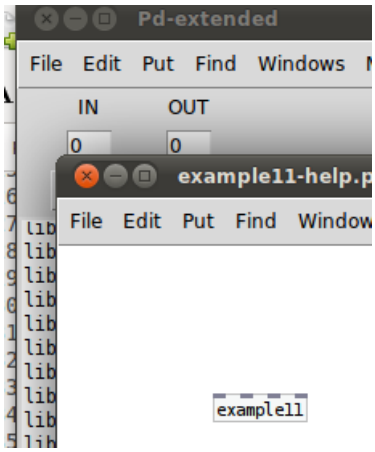

Figura 6.2: Vários inlets DSP.

6.5 Primeiro outlet DSP

A criação dos outlets é feita no construtor do external (veja o exemplo 12) e é necessário adicionar os outlets à estrutura da classe para que os mesmos possam ser desalocados quando o objeto for destruído.

```
1   void * example12_new(void){
2      t_example12 *x = (t_example12 *) pd_new(example12_class);
3
4      x->x_outlet_dsp_0 = outlet_new(&x->x_obj, &s_signal);
```

```
5     x->x_outlet_dsp_1 = outlet_new(&x->x_obj, &s_signal);
6     x->x_outlet_dsp_2 = outlet_new(&x->x_obj, &s_signal);
7     x->x_outlet_dsp_3 = outlet_new(&x->x_obj, &s_signal);
8
9     return (void *) x;
10 }
```

Listing 6.12: Criação de outlets DSP

A definição do método `_perform()` será idêntica ao do exemplo anterior, quando criamos quatro inlets:

```
1 static void example12_dsp(t_example12 *x, t_signal **sp){
2     dsp_add(example12_perform, 6, x, sp[0]->s_n, sp[0]->s_vec, sp
          [1]->s_vec, sp[2]->s_vec, sp[3]->s_vec);
3 }
```

Listing 6.13: Método DSP para outlets

O método perform também será quase idêntico ao do exemplo anterior, porém recebendo quatro outlets:

```
1 static t_int * example12_perform(t_int *w){
2     t_example12 *x = (t_example12 *)(w[1]);
3     int n = (int)(w[2]);
4     t_float *out1 = (t_float *)(w[3]);
5     t_float *out2 = (t_float *)(w[4]);
6     t_float *out3 = (t_float *)(w[5]);
7     t_float *out4 = (t_float *)(w[6]);
8     return (w + 7); // proximo bloco
9 }
```

Listing 6.14: Método Perform para outlets

O resultado pode ser visto na figura 6.3.

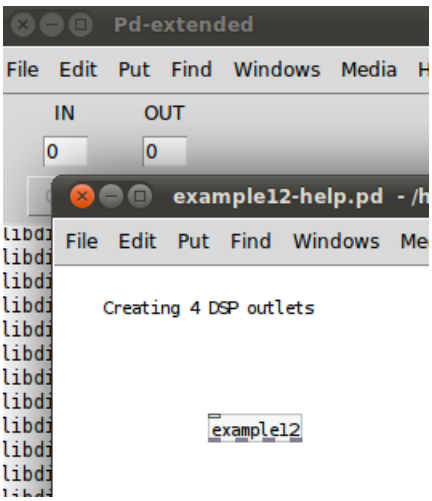

Figura 6.3: Primeiro Outlet DSP.

6.6 Inlets e outlets DSP

Nosso próximo exemplo (veja o exemplo 13) mistura no mesmo objeto inlets e outlets DSP, o que é bastante comum. Neste ponto, deve estar mais ou menos claro como é feita a construção de um objeto assim. Neste exemplo, não utilizaremos o primeiro inlet mágico.

```
1  void * example13_new(void){
2     t_example13 *x = (t_example13 *) pd_new(example13_class);
3
4     x->x_inlet_dsp_0 = inlet_new(&x->x_obj, &x->x_obj.ob_pd, &
          s_signal, &s_signal);
5     x->x_inlet_dsp_1 = inlet_new(&x->x_obj, &x->x_obj.ob_pd, &
          s_signal, &s_signal);
6     x->x_inlet_dsp_2 = inlet_new(&x->x_obj, &x->x_obj.ob_pd, &
          s_signal, &s_signal);
7     x->x_inlet_dsp_3 = inlet_new(&x->x_obj, &x->x_obj.ob_pd, &
          s_signal, &s_signal);
8
9     x->x_outlet_dsp_0 = outlet_new(&x->x_obj, &s_signal);
10    x->x_outlet_dsp_1 = outlet_new(&x->x_obj, &s_signal);
11    x->x_outlet_dsp_2 = outlet_new(&x->x_obj, &s_signal);
12    x->x_outlet_dsp_3 = outlet_new(&x->x_obj, &s_signal);
13
14    return (void *) x;
15 }
```

Listing 6.15: Criação de vários inlets e outlets DSP

No método seguinte associamos o método `_perform()` à cadeia DSP do Pure Data:

```
1  static void example13_dsp(t_example13 *x, t_signal **sp){
2     dsp_add(example13_perform, 10, x, sp[0]->s_n, sp[0]->s_vec, sp
          [1]->s_vec, sp[2]->s_vec, sp[3]->s_vec, sp[4]->s_vec, sp
          [5]->s_vec, sp[6]->s_vec, sp[7]->s_vec);
3  }
```

Listing 6.16: Método DSP para vários inlets e outlets DSP

No método `_perform()` recebemos como argumento um bloco de memória que contém primeiro os buffers de entrada e em seguida os buffers de saída:

```
1  static t_int * example13_perform(t_int *w){
2     t_example13 *x = (t_example13 *)(w[1]);
3     int n = (int)(w[2]);
4     t_float *in1 = (t_float *)(w[3]);
5     t_float *in2 = (t_float *)(w[4]);
6     t_float *in3 = (t_float *)(w[5]);
7     t_float *in4 = (t_float *)(w[6]);
8     t_float *out1 = (t_float *)(w[7]);
9     t_float *out2 = (t_float *)(w[8]);
10    t_float *out3 = (t_float *)(w[9]);
11    t_float *out4 = (t_float *)(w[10]);
12    return (w + 11); // proximo bloco
13 }
```

Listing 6.17: Método Perform para vários inlets e outlets DSP

Note que não precisamos do inlet "mágico" pois este objeto não irá receber outras mensagens que não sinais de áudio. Caso queira receber outras mensagens, basta utilizar a macro já apresentada lembrando que, neste caso, não será necessário adicionar o primeiro inlet.

O resultado pode ser visto na figura 6.4.

6.7 Inlets e outlets DSP criados dinamicamente

Como visto anteriormente, o método DSP passa para o método perform como parâmetro a quantidade de inlets e outlets. Por esta razão, a criação dinâmica de inlets e outlets de áudio não é tão simples quanto a criação dinâmica de inlets e outlets para mensagens.

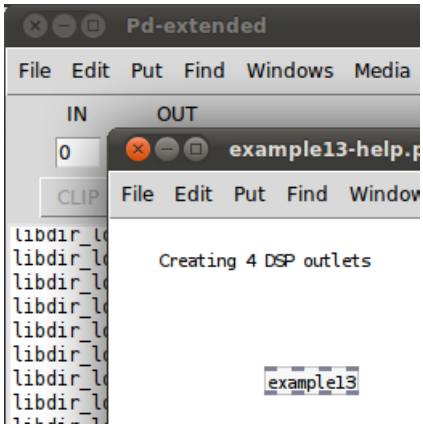

Figura 6.4: Vários inlets e outlets DSP.

Ainda assim, é possível definir através de parâmetros para o construtor a quantidade de inlets e/ou outlets DSP que um external deve possuir. Isso significa que o número de inlets e outlets é definido dinamicamente, em tempo de execução, através de um argumento para o construtor, da mesma maneira que criamos inlets e outlets de mensagens dinamicamente no capítulo anterior,

Neste caso, além de armazenarmos a quantidade de canais, armazenaremos na estrutura da classe um vetor para o sinal de entrada e outro vetor para o sinal de saída.

```
1 typedef struct _multigain {
2    t_object x_obj;
3    t_int count;
4    t_float gain;
5    t_inlet * x_inlet_gain_float;
6    t_sample ** invec;
7    t_sample ** outvec;
8 } t_multigain;
```

Listing 6.18: Estrutura da classe para inlets e outlets DSP dinâmicos

O construtor cria a quantidade de inlets e outlets passada como argumento na criação do objeto. Aqui, poderíamos utilizar `getbytes()` para alocar o vetor com os dados de portável.

```
1  void * multigain_new(t_floatarg count_arg){
2     t_multigain *x = (t_multigain *) pd_new(multigain_class);
3     x->count = (int)count_arg;
4     short i;
5     for (i = 0; i < x->count; i++) {
6        inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal)
             ; // signal inlets
7        outlet_new(&x->x_obj, &s_signal);
8     }
9     x->outvec = getbytes(sizeof(t_sample) * x->count);
10    x->invec = getbytes(sizeof(t_sample) * x->count);
11    x->x_inlet_gain_float = floatinlet_new(&x->x_obj, &x->gain);
12    return (void *) x;
13 }
```

Listing 6.19: Estrutura do construtor para iolets DSP dinâmicos

O método DSP irá alocar no próprio objeto os vetores de entrada e saída de DSP e passar apenas o objeto para o método `_perform()`.

```
1 static void multigain_dsp(t_multigain *x, t_signal **sp){
2    if(x->count < 1) return;
3    int i = 0;
```

```
4     for(; i < x->count; i++){
5        x->invec[i] = getbytes(sys_getblksize() * sizeof(t_sample))
             ;
6        x->invec[i] = sp[i]->s_vec;
7     }
8     for(i = 0; i < x->count ; i++){
9        x->outvec[i] = getbytes(sys_getblksize() * sizeof(t_sample)
             );
10       x->outvec[i] = sp[x->count + i]->s_vec;
11    }
12    dsp_add(multigain_perform, 2, x, sp[0]->s_n);
13 }
```

Listing 6.20: Método DSP para iolets DSP dinâmicos

A função perform irá, neste exemplo, alterar o ganho dos sinais de entrada, copiando-os para os sinais de saída.

```
1  static t_int * multigain_perform(t_int *w){
2     t_multigain *x = (t_multigain *)(w[1]);
3     int n = (int)(w[2]), i = 0, j = 0;
4     float gain = x->gain;
5     for(; i < x->count ; i++)
6        for(j = 0 ; j < n ; j++)
7           x->outvec[i][j] = x->invec[i][j] * gain;
8     return (w + 3); // proximo bloco
9  }
```

Listing 6.21: Método Perform para iolets DSP dinâmicos

6.8 Alocação de memória para DSP

Na introdução da seção de iolets, apresentamos um exemplo (inverter.c) que modifica um valor recebido. Como estes parâmetros são passados por referência e não por valor, a modificação do valor do mesmo irá ser refletida em todos os objetos que recebem esta referência. No caso de sinais de áudio, não é diferente.

Ao concatenarmos uma série de objetos de áudio, o PD não copia o bloco de áudio de um para o outro mas utiliza passagem por referência. Assim, se um objeto altera o bloco recebido, o mesmo será alterado em toda cadeia de objetos concatenados com o mesmo.

Imaginemos um *external* que calcula a mediana de um sinal de áudio [1]. A maneira mais simples de calcular a mediana é ordenar as amostras e pegar o valor do meio. Caso façamos a ordenação no vetor de entrada iremos alterar a ordem das amostras recebidas e todos os objetos conectados a este sinal receberão o mesmo com as amostras ordenadas. Por esta razão, é importante verificar se podemos ou não alterar o valor de um vetor de amostras recebidos.

6.9 Outras funcionalidades para DSP

Vários processamentos em sinais dependem da taxa de amostragem, tamanho de bloco e quantidade de canais de entrada e saída. Para acessar estas informações no PD, utilizamos duas funções da API:

- `int sys_getblksize(void);` Retorna o tamanho do bloco de processamento do Pure Data.
- `t_float sys_getsr(void);` Retorna qual a amostragem (Sample Rate) atual do Pure Data.
- `int sys_get_inchannels(void);` Retorna a quantidade de canais de entrada do Pure Data.
- `int sys_get_outchannels(void);` Retorna a quantidade de canais de saída do Pure Data.

Estas e outras funções podem ser encontradas no último capítulo deste tutorial.

[1] Tal exemplo é real e o *external* pode ser encontrado aqui: `http://sourceforge.net/projects/median/`.

Parte II

Avançando...

Capítulo 7

Multithreading

Como vimos no capítulo anterior, o bloco de processamento do Pd possui um limite de tempo para a execução. É possível utilizar threads para separar processos que consumam mais tempo do que o o período do bloco DSP, como por exemplo no vaso de processos do tipo produtor/consumidor.

Explicar que o PD não suporta multithread

A programação multithread não é exatamente comum no Pure Data mas pode ser útil para várias coisas como escrita em arquivo, envio de dados para a rede ou atualização da interface gráfica (como veremos no próximo capítulo).

Apesar de existirem várias bibliotecas para programação paralela, como por exemplo a simples utilização do comando fork do GNU/Linux, é desejável que os externals do Pure Data sejam compatíveis com diversos sistemas operacionais. Nos repositórios do Miller Puckette, autor do Pure Data, encontramos patches nos quais ele utiliza threads POSIX implementadas pela biblioteca `pthread` [1].

Note que esta solução, que em muito se aproxima da última forma de criar inlets e outlets DSP implica em não trabalharmos mais em tempo real. Implementações deste tipo não podem ser pensadas para processamentos aonde a entrada de áudio será processada e devolvida na saída de áudio no mesmo bloco de processamento do Pd.

7.1 Criando threads

Para utilizar a biblioteca de threads do POSIX é necessário incluir o arquivo de cabeçalho correspondente. Em seguida, para armazenar as threads que criamos, utilizamos uma variáveis que armazenam threads (veja o exemplo 20).

```
1 #include <pthread.h>
2 ...
3 typedef struct _example20 {
4     t_object x_obj;
5     pthread_t example20_thread;
6 } t_example20;
```

O próximo passo é criar uma função associada a esta thread e a enfim lançar a thread. O lançamento da thread pode ser feito na função DSP. Isto implica criar e iniciar uma thread nova em cada ciclo DSP do Pd.

```
1 void * example20_thread_function(void * arg) {
2   t_example20 *x = (t_example20 *) arg;
3   while(1){
4     // DO SOMETHING
5     printf("Threading running!\n");
6     sleep(1);
7   }
8   return 0;
9 }
```

[1] Para maiores informações, visite: https://computing.llnl.gov/tutorials/pthreads/

```
10
11 static void example20_dsp(t_example20 *x, t_signal **sp){
12   pthread_create(&x->example20_thread, NULL,
         example20_thread_function, x);
13   dsp_add(example20_perform, 1 , x);
14 }
```

A função de criação da thread recebe o endereço da variável aonde a thread será armazenada, os atributos da thread sendo criada[2], a função de inicialização associada a esta thread e os argumentos passados para esta função.

Caso seja passado mais de um argumento, é recomendado que se crie uma estrutura de dados e que esta seja passada como argumento para a thread.

7.2 Gerenciamento de threads

Há diversas funções para o gerenciamento de threads, definidas no arquivo de cabeçalho `pthread.h`. Entre elas:

- `pthread_detach(threadid)`: Indica para a implementação que o armazenamento da thread pode ser recuperado quando a mesma se encontra terminada
- `pthread_join(threadid,status)`: Indica para o trecho de código que chamou a thread que o mesmo deve esperar que a mesma tenha terminado sua execução.
- `pthread_exit(void *value_ptr)`: Encerra a execução de uma thread e libera sua alocação de memória.

Em princípio, threads POSIX não possuem funções para interromper e continuar a execução. Apesar disto, é possível implementar estes comandos por meio de mutexes, como veremos a seguir.

7.3 Controle de concorrência

Uma das dificuldades de utilização de threads é o controle da concorrência por recursos entre threads. Em situações de race condition, é necessário que controlemos o acesso de threads concorrentes a trechos de código que acessam dados comuns. Isto é feito por meio de mutex (mutual exclusion), sistemas de controle atômicos que garantem que apenas uma thread será executada sobre um trecho de código por vez.

Um mutex é definido da seguinte forma:

```
1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
3 int play = 0;
```

O controle ao trecho de código pode ser feito da seguinte forma:

```
1 for(;;) { /* Playback loop */
2   pthread_mutex_lock(&lock);
3   while(!play) { /* We're paused */
4     pthread_cond_wait(&cond, &lock); /* Wait for play signal */
5   }
6   pthread_mutex_unlock(&lock);
7   /* Continue playback */
8 }
```

[2]No caso, passando `NULL` serão utilizados os atributos padrão. Para uma lista completa dos atributos, visite: http://sourceware.org/pthreads-win32/manual/pthread_attr_init.html

Creio que seria mais útil um exemplo funcional...

7.4 Controle via Pure Data

Além de podermos trabalhar threads para nosso external, na biblioteca do Pd há funções para que possamos criar mutex no Pd. São elas:

integrando multith-read com o pd

- `void sys_lock(void);`
- `void sys_unlock(void);`
- `int sys_trylock(void);`

Capítulo 8

Send e Receive

A comunicação do PD com *externals* nem sempre é feita por conexões explícitas. Outra maneira de permitir a comunicação entre objetos é por meio de send e receive. Esta opção está presente em objetos gráficos como o toggle e bang, por exemplo. Para utilizar tal tipo de mensagem é necessário definir o nome da mensagem que o objeto pretende receber.

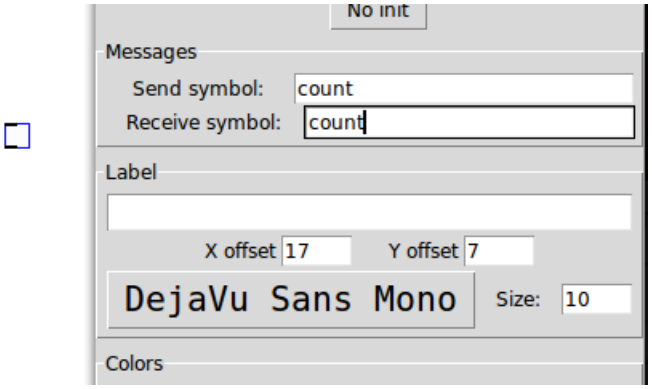

Figura 8.1: Exemplo de configuração de um toggle para envio e recebimento de mensagens

Neste capítulo vamos verificar como enviar e receber mensagens. Para maiores informações quanto a este tipo de mensagens, consulte o código dos objetos send e receive no repositório do PD[1].

8.1 Enviando mensagens

O envio de mensagem pode ser feito por meio de algumas funções:

- `pd_bang(t_pd *x)`
- `pd_float(t_pd *x, t_float f)`
- `pd_symbol(t_pd *x, t_symbol *s)`
- `pd_pointer(t_pd *x, t_gpointer *gp)`
- `pd_list(t_pd *x, t_symbol *s, int argc, t_atom *argv)`
- `typedmess(t_pd *x, t_symbol *s, int argc, t_atom *argv)`

A última função, que não define tipo para a mensagem, é uma espécie de "anything". O primeiro parâmetro para todas as funções pode ser conseguido em um símbolo em seu atributo `s_thing`.

No exemplo mailman.c, o objeto envia mensagens para um conjunto de diferentes seletores recebidos como parâmetro. As mensagens são do tipo bang e enviadas quando o objeto recebe um bang.

[1] `https://github.com/libpd/libpd/blob/master/pure-data/src/x_connective.c`

```
void mailman_bang(t_mailman *x){
   int i = 0;
   for(; i < x->argc ; i++){
      if (x->messages[i]->s_thing)
         pd_bang(x->messages[i]->s_thing);
   }
}
```

Listing 8.1: Envio de mensagens bang por send

O resultado pode ser visto na Figura **??**.

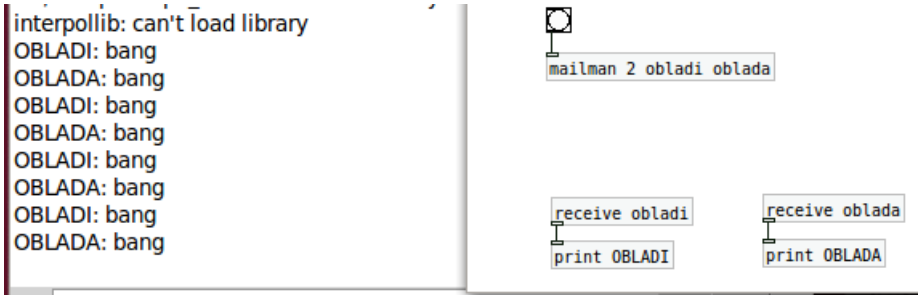

Figura 8.2: Envio de mensagens por send

8.2 Receive

O recebimento de mensagens junto ao PD ocorre por meio de uma função `bind` com o o símbolo esperado.

- `pd_bind(t_pd *x, t_symbol *s)`
- `pd_unbind(t_pd *x, t_symbol *s)`

Nestas funções, o primeiro parâmetro é seu objeto e o segundo parâmetro é a mensagem que vc espera receber.

```
void * postbox_new(t_symbol *s, int argc, t_atom *argv){
   t_postbox *x = (t_postbox *) pd_new(postbox_class);
   int i = 0;
   int counter = 0;
   for(; i < argc ; i++){
      if((argv + i)->a_type == A_SYMBOL)
         counter++;
   }
   x->messages = getbytes(counter * sizeof(t_symbol *));
   counter = 0;
   for(i = 0; i < argc ; i++){
      if((argv + i)->a_type == A_SYMBOL){
         x->messages[counter] = atom_getsymbol(argv + i);
         pd_bind(&x->x_obj.ob_pd, x->messages[counter]);
         counter++;
      }
   }
//    pd_bind(&x->x_obj.ob_pd, gensym("#key"));
//    pd_bind(&x->x_obj.ob_pd, gensym("#keyname"));
//    pd_bind(&x->x_obj.ob_pd, gensym("#keyup"));
   x->argc = counter;
   x->x_outlet = outlet_new(&x->x_obj, gensym("bang"));
   return (void *) x;
```

```
24 }
```

Listing 8.2: Exemplo de objeto que recebe várias mensagens

É importante que, no destrutor do objeto, a ligação seja desfeita para evitar que o PD aborte ao tentar enviar uma mensagem para um objeto que não existe mais. Isto é feito pela função `unbind`, apresentada abaixo.

```
1  void postbox_destroy(t_postbox *x) {
2     outlet_free(x->x_outlet);
3     int i = 0;
4     for(; i < x->argc ; i++){
5        pd_unbind(&x->x_obj.ob_pd, x->messages[i]);
6     }
7  //    pd_unbind(&x->x_obj.ob_pd, gensym("#key"));
8  //    pd_unbind(&x->x_obj.ob_pd, gensym("#keyname"));
9  //    pd_unbind(&x->x_obj.ob_pd, gensym("#keyup"));
10 }
```

Listing 8.3: Desvinculando o objeto com a mensagem no destrutor

Uma vez associado o recebimento de um determinado símbolo, é necessário definir qual método será chamado para cada tipo de mensagem recebida. A definição destes métodos é similar a definição dos inlets ativos. É ideal que um objeto que possua um receive possua métodos para receber todos os tipos de mensagens do PD, mesmo que tais métodos não sejam implementados.

```
1  void postbox_list_method(t_postbox *x, t_symbol *s, int argc,
      t_atom *argv){
2     post("list %s", atom_getsymbolarg(1, argc, argv)->s_name);
3  }
4
5  void postbox_setup(void) {
6     postbox_class = class_new(gensym("postbox"),
7        (t_newmethod) postbox_new, // Constructor
8        (t_method) postbox_destroy, // Destructor
9        sizeof (t_postbox),
10       CLASS_NOINLET,
11       A_GIMME,
12       0);//Must always ends with a zero
13
14    class_addbang(postbox_class, postbox_bang_method);
15    class_addfloat(postbox_class, postbox_float_method);
16    class_addlist(postbox_class, postbox_list_method);
17 }
```

Listing 8.4: Associando métodos para receber mensagens

Note que este objeto (postobox.c) não possui inlets e por isto tais métodos só serão usados para mensagens. Caso o mesmo possua inlets, o tratamento de mensagens recebidas pelo inlet ou pelo receive será exatamente o mesmo, o que é muito bacana.

8.3 Indo além disto

Entender como receber mensagens enviadas pelo PD ajudará a entender como trocar mensagens entre um objeto PD em C e sua interface em tcl/tk. Além disto, é possível receber e enviar mensagens padrões do PD, como movimentos de teclado, entradas MIDI e assim por diante.

Com isto, é possível desenvolver um *external* que envia eventos de teclado ou que recebe eventos de teclado diretamente.

Alguns exemplos de mensagens internas do PD que são trocadas por send / receive são:

- Eventos de teclado[2]

[2]Retirados de: `https://github.com/libpd/libpd/blob/master/pure-data/src/x_gui.c`

  - `#key`
  - `#keyup`
  - `#keyname`
- Eventos MIDI[3]
  - `#midiin`
  - `#sysexin`
  - `#notein`
  - `#ctlin`
  - `#pgmin`
  - `#bendin`
  - `#touchin`
  - `#polytouchin`
  - `#midiclkin`
  - `#midirealtimein`
  - `#midiclkin`
  - `#midiclkin`

Acho que seria bacana alguns exemplos disto tudo...

[3] Retirados de: `https://github.com/libpd/libpd/blob/master/pure-data/src/x_midi.c`

Capítulo 9

Clock

Clock
- Canvas
- DSP
- Comunicação por send / receive

Parte III

GUI

Capítulo 10

Manipulando GUI

O PD foi desenvolvido utilizando um modelo que separa a apresentação das funcionalidades. Isto rola com 2 linguagens de programação: C para a engine e tcl/tk para as GUI. GUI e engine se comunicam por um socket que permite, inclusive, que a engine do PD seja executada em uma máquina e a GUI em outra.

10.1 Iniciando no Tcl/Tk

O Tcl (Tool Command Language) é uma linguagem de programação dinâmica bastante poderosa e simples de ser utilizada. Tk é um conjunto de ferramentas para construção de GUI de aplicações desktop e é a GUI padrão não apenas do TCL mas de várias outras linguagens e pode ser executada nativamente em vários sistemas operacionais modernos como Windows, Mac OS X, Linux, entre outros [1].

O Tk possui vários objetos prontos para GUI como botões, labels, janelas, checkbox, entre outros. Os objetos criados devem ser armazenados em uma variável. Toda variável em Tk possui um nome que inicia com ponto (.). Após criar o objeto e definir seus atributos, basta que o mesmo seja empacotado (pack).

```
1 label .hello -text "Hello World"
2 pack .hello
```

Uma vez que um programa tk esteja pronto, basta salvá-lo com a extensão .tcl e utilizar um interpretador para executá-lo. Um exemplo deste interpretador no Linux é o wish. Assim, salvando o exemplo anterior com o nome de helloWorld.tcl e executando

```
1 wish helloWorld.tcl
```

Teremos o resultado:

Figura 10.1: Hello World no Tcl/Tk

[1] Visite: http://www.tcl.tk/ para maiores informações

No diretório tk deste tutorial temos exemplos mais interessantes de GUI com Tk mas a prática desta linguagem vai além do escopo deste tutorial. Um tutorial mais completo de Tcl/Tk pode ser encontrado em [2] e uma lista dos objetos e parâmetros pode ser encontrada em [3].

[2] http://www.bin-co.com/tcl/tutorial/
[3] http://www.tkdocs.com/widgets/index.html

Capítulo 11

Externals com GUI - Usando o Tcl/Tk

O Pd permite que externals possuam interfaces gráficas mais rebuscadas que as simples caixinhas que o mesmo desenha. Isto pode ser feito adicionando código Tcl/tk ao external. Isto porque a GUI do próprio Pd é feita nesta linguagem.

11.1 Escrevendo externals com GUI

A primeira necessidade para implementar um external com GUI é a inclusão da biblioteca g_canvas.h. Esta biblioteca encontra-se disponível em [1] e nela estarão as funcionalidades necessárias para que o Pd desenhe nossas GUI. Veja que não queremos alterar a GUI do Pd mas apenas fazer uma GUI para o external. Isto significa que teremos o cuidade de pedir ao Pd que desenhe nossa GUI.

O segundo passo é definirmos uma variável para armazenar o comportamento do nosso objeto (Veja o exemplo 14).

```
1 t_widgetbehavior widgetbehavior; // This represents the external
    GUI
```

Nesta variável teremos as funções definidas para o que acontece com nosso objeto gráfico. Isto deve ser feito no método setup do objeto.

```
1  void example14_setup(void) {
2      example14_class = class_new(gensym("example14"),
3              (t_newmethod) example14_new, // Constructor
4              (t_method) example14_destroy, // Destructor
5              sizeof (t_example14),
6        CLASS_DEFAULT,
7        A_GIMME, // Allows various parameters
8              0); // LAST argument is ALWAYS zero
9
10     // The external GUI rectangle definition
11     widgetbehavior.w_getrectfn = my_getrect;
12     //How to make ir visible / invisible
13     widgetbehavior.w_visfn= my_vis;
14     //what to do whe moved
15     widgetbehavior.w_displacefn= my_displace;
16     // What to do when selected
17     widgetbehavior.w_selectfn= my_select;
18     // What to do when active
19     widgetbehavior.w_activatefn = my_activate;
```

[1] http://www.koders.com/c/fid97160440CD236854358462C336536646E0933C46.aspx

```
20      // What to do when deleted
21      widgetbehavior.w_deletefn = my_delete;
22      // What to do when clicked
23      widgetbehavior.w_clickfn = my_click;
24
25      // What about object properties?
26      class_setpropertiesfn(example14_class, my_properties);
27      // How to save its properties with the patch?
28      class_setsavefn(example14_class, my_save);
29
30      //Associate the widgetbehavior with the class
31      class_setwidget(example14_class, &widgetbehavior);
32
33  }
```

Além de definir as funções callback da GUI é necessário associar o widgetbehavior a classe. Uma vez que isto for feito, o Pd não mais irá renderizar a famosa caixinha. A partir disto a responsabilidade de renderizar a GUI passa a ser do programador.

Vamos ver as funções criadas para desenhar uma GUI.

```
1  // THE BOUNDING RECTANGLE
2  static void my_getrect(t_gobj *z, t_glist *glist, int *xp1, int *
       yp1, int *xp2, int *yp2){
3    // This function is always called.
4    // Better do not put a post here...
5    // post("GETRECT");
6    t_example14 *x = (t_example14 *)z;
7    *xp1 = x->x_obj.te_xpix;
8    *yp1 = x->x_obj.te_ypix;
9    *xp2 = x->x_obj.te_xpix + 30;
10   *yp2 = x->x_obj.te_ypix + 50;
11 }
```

Esta função recebe ponteiros para inteiros que deverão ser apontados para os valores que queremos definir como o retângulo do nosso objeto. Veja que isto não é necessariamente o tamanho do retângulo do objeto gráfico mas aonde o mesmo poderá ser clicado na tela. Um exemplo disto é o comentário do Pd. Apesar do mesmo as vezes ficar maior, sua área clicável é sempre um quadradinho na esquerda. Isto significa que nem sempre a representação gráfica da área do objeto é a mesma da sua área de desenho.

```
1  // MAKE VISIBLE OR INVISIBLE
2  static void my_vis(t_gobj *z, t_glist *glist, int vis){
3    t_example14 *x = (t_example14 *)z;
4
5    // takes the Canvas to draw a GUI
6    t_canvas * canvas = glist_getcanvas(glist);
7
8    if(vis){ // VISIBLE
9    post("VISIBLE");
10
11         sys_vgui(".x%lx.c create rectangle %d %d %d %d -tags %xrr
               -fill #FF0000\n",
12     glist_getcanvas(glist),
13     x->x_obj.te_xpix,
14     x->x_obj.te_ypix,
15     x->x_obj.te_xpix + 70,
16     x->x_obj.te_ypix + 50,
17     x
```

```
18      );
19          sys_vgui(".x%x.c create text %d %d -text {example14} -
               anchor w  -tags %xlb\n",
20                   canvas,
21      x->x_obj.te_xpix + 2,
22      x->x_obj.te_ypix + 12,
23      x);
24    }else{ // INVISIBLE
25      post("INVISIBLE");
26      sys_vgui(".x%x.c delete %xrr\n",canvas, x);
27      sys_vgui(".x%x.c delete %xlb\n",canvas, x);
28    }
29  //  canvas_fixlinesfor(glist, (t_text *)x);
30  }
```

Deixar o objeto visível ou invisível significa pedir a GUI do Pd que desenhe ou apague um desenho. Neste caso nosso desenho é um retângulo vermelho. Usamos o nome da própria instância como nome do objeto Tk para evitar que sobrescrevamos o nome de algum outro componente gráfico do Pd. Também desenhamos um texto com o nome do objeto pois o Pd não fará isto. Caso nosso objeto se torne invisível, temos de remover ambos do canvas.

```
1   // WHAT TO DO IF SELECTED?
2   static void my_select(t_gobj *z, t_glist *glist, int state){
3     t_example14 *x = (t_example14 *)z;
4     if (state) {
5     post("SELECTED");
6           sys_vgui(".x%x.c create rectangle %d %d %d %d -tags %xSEL
                -outline blue\n",
7       glist_getcanvas(glist),
8       x->x_obj.te_xpix,
9       x->x_obj.te_ypix,
10      x->x_obj.te_xpix + 70,
11      x->x_obj.te_ypix + 50,
12      x
13      );
14    }else {
15    post("DESELECTED");
16    sys_vgui(".x%x.c delete %xSEL\n",glist_getcanvas(glist), x);
17    }
18  }
```

O que acontece quando um objeto do Pd é selecionado? Ele ganha um contorno azul. O que acontece quando nosso objeto é selecionado? Nada. Por isto desenhamos aqui um contorno azul. Quando ele é deselecionado, removemos o contorno azul.

```
1   // DISPLACE IT
2   void my_displace(t_gobj *z, t_glist *glist,int dx, int dy){
3     post("MOVED");
4     t_canvas * canvas = glist_getcanvas(glist);
5     t_example14 *x = (t_example14 *)z;
6     x->x_obj.te_xpix += dx; // x movement
7     x->x_obj.te_ypix += dy; // y movement
8
9           sys_vgui(".x%lx.c coords %xSEL %d %d %d %d \n", //MOVE O
               SELECIONADO
10      canvas,
11      x,
```

```
12      x->x_obj.te_xpix,
13      x->x_obj.te_ypix,
14      x->x_obj.te_xpix + 70,
15      x->x_obj.te_ypix + 50
16      );
17          sys_vgui(".x%x.c coords %xrr %d %d %d %d\n",canvas,x,x->
                x_obj.te_xpix,x->x_obj.te_ypix,x->x_obj.te_xpix + 70,x
                ->x_obj.te_ypix + 50);
18          sys_vgui(".x%x.c coords %xlb %d %d \n",canvas,x,x->x_obj.
                te_xpix + 2, x->x_obj.te_ypix + 12);
19
20    canvas_fixlinesfor(glist, (t_text *)x);
21  }
```

O que fazer quando movemos um objeto. Aqui será necessário mover todos os objetos pois não temos um container que possui todos os objetos. Inclusive é necessário mover o contorno azul que desenhamos quando o mesmo se tornou selecionado.

```
1  // What to do if activated?
2  static void my_activate(t_gobj *x, struct _glist *glist, int
       state){
3    post("Activated");
4  }
```

O método será executado quando um objeto se tornar ativo.

```
1  static int my_click(t_gobj *z, struct _glist *glist, int xpix,
       int ypix, int shift, int alt, int dbl, int doit){
2    post("Clicked xpix:%d ypix:%d shift:%d alt:%d dbl:%d doit:%d",
         xpix,ypix, shift, alt, dbl, doit);
3    return 0;
4  }
```

O método click é retornado com qualquer evento do mouse. Caso ele seja clicado, o mesmo receberá o valor 1 no parâmetro doit. Vale lembrar que este método só será executado quando o Pd não estiver no modo de edição.

```
1  static void my_delete(t_gobj *z, t_glist *glist){
2    t_text *x = (t_text *)z;
3    canvas_deletelinesfor(glist_getcanvas(glist), x);
4    post("Object deleted!");
5  }
```

Este método será o destrutor da GUI.

```
1  void my_save(t_gobj *c, t_binbuf *b){
2    post("SAVE");
3  }
```

Quando salvamos o *patch* pode ser necessário armazenar parâmetros para recuperar nosso external como GUI na mesma situação. Este método recebe um buffer aonde poderá armazenar dados que serão devolvidos quando o mesmo for recriado. Note que isto não está associado com a GUI mas com a classe.

Nunca implementei Salvar. Parece legal...

```
1 void my_properties(t_gobj *c, t_glist *list){
2   post("PROPERTIES");
3 }
```

Caso o usuário pressione o botão contrário ele pode alterar as propriedades do objeto. Este método permite definir a lista de propriedades que o objeto aceita. Note que isto não está associado com a GUI mas com a classe.

Ainda não implementei propriedades.

Caso nosso objeto tenha sido criado corretamente, ele ficará como a seguir

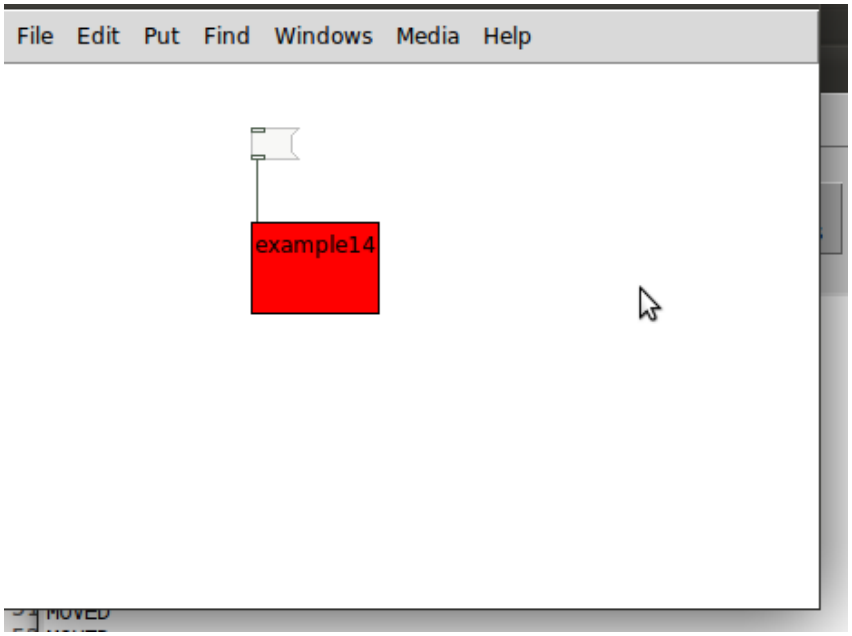

Figura 11.1: Adicionando GUI tk.

Note que este objeto foi definido em sua criação como um CLASS_DEFAULT, o que implica que o mesmo possui um inlet. Este inlet existe e está funcionando e aceitando conexões de outros objetos, mensagens e números. Só não aceita conexões DSP pois as mesmas não foram definidas para este objeto. Mesmo assim o Pure Data não irá desenhá-lo pois precisamos definir tudo na GUI. Caso queira um retângulo em cima para mostrar ao usuário que temos um inlet, temos que desenhá-lo, movê-lo quando necessário e também apagá-lo quando o objeto se tornar invisível.

11.2 Adicionando componentes gráficos

O Objeto Tk que o Pd nos disponibiliza é um canvas. Um canvas é, em princípio, uma tela de pintura. Em um canvas podemos adicionar linhas, ovals, pontos, textos, retângulos e janelas. É por se tratar de um canvas e não de uma janela que não temos o conceito de um container para os objetos. Desta maneira, para adicionarmos um componente como um botão ou um slider, é necessário adicionarmos uma janela ao canvas para que a mesma abrigue este componente gráfico. Há vários componentes gráficos que podem ser adicionados em um external. Vejamos um exemplo (exemplo 15).

```
1 // MAKE VISIBLE OR INVISIBLE
2 static void my_vis(t_gobj *z, t_glist *glist, int vis){
3   t_example15 *x = (t_example15 *)z;
4   t_canvas * canvas = glist_getcanvas(glist);
5
6   if(vis){ // VISIBLE
7   // Define the tk/tcl commands / functions
```

```
8   sys_vgui("proc do_something {} {\n set name [.x%x.c.s%xtx get]\
       n puts \"OIA: $name\" \n}\n",canvas,x);
9   sys_vgui("proc do_otherthing {val} {\n set name [.x%x.c.s%xtx
       get]\n puts \"OIA: $name\" \n}\n",canvas,x);
10      // The text field
11  sys_vgui("entry .x%x.c.s%xtx -width 12 -bg yellow \n", canvas,x
       );
12  // The button
13  sys_vgui("button .x%x.c.s%xbb -text {click} -command
       do_something\n", canvas,x);
14  // The radio button
15  sys_vgui("radiobutton .x%x.c.s%xrb -value 1 -command
       do_something\n", canvas,x);
16  // The h slider
17  sys_vgui("scale .x%x.c.s%xsb -orient horizontal -command
       do_otherthing \n", canvas,x);
18  // A checkbutton
19  sys_vgui("checkbutton .x%x.c.s%xcb -foreground blue -background
        yellow -command do_something\n", canvas,x);
20  // The red rectangle
21        sys_vgui(".x%lx.c create rectangle %d %d %d %d -tags %xrr
              -fill #FF0000\n",
22    glist_getcanvas(glist),
23    x->x_obj.te_xpix,
24    x->x_obj.te_ypix,
25    x->x_obj.te_xpix + 170,
26    x->x_obj.te_ypix + 150,
27    x
28    );
29  // A window to the button (bb)
30  sys_vgui(".x%x.c create window %d %d -anchor nw -window .x%x.c.
       s%xbb -tags %xbb\n",
31    canvas,
32    x->x_obj.te_xpix + 70,
33    x->x_obj.te_ypix + 120,
34    canvas,
35    x,
36    x);
37  // A window to the radiobutton
38  sys_vgui(".x%x.c create window %d %d -anchor nw -window .x%x.c.
       s%xrb -tags %xrb\n",
39    canvas,
40    x->x_obj.te_xpix + 70,
41    x->x_obj.te_ypix,
42    canvas,
43    x,
44    x);
45  // A window to the combo box
46  sys_vgui(".x%x.c create window %d %d -anchor nw -window .x%x.c.
       s%xcb -tags %xcb\n",
47    canvas,
48    x->x_obj.te_xpix  + 100,
49    x->x_obj.te_ypix,
50    canvas,
51    x,
52    x);
53  // A window to the slider button
54  sys_vgui(".x%x.c create window %d %d -anchor nw -window .x%x.c.
```

```
      s%xsb -tags %xsb\n",
55      canvas,
56      x->x_obj.te_xpix,
57      x->x_obj.te_ypix + 80,
58      canvas,
59      x,
60      x);
61    // A window to the text
62    sys_vgui(".x%x.c create window %d %d -anchor nw -window .x%x.c.
          s%xtx -tags %xtx\n",
63      canvas,
64      x->x_obj.te_xpix,
65      x->x_obj.te_ypix + 60,
66      canvas,
67      x,
68      x);
69    // a label field
70          sys_vgui(".x%x.c create text %d %d -text {example15~} -
              anchor w  -tags %xlb\n",
71                    canvas,
72      x->x_obj.te_xpix,
73      x->x_obj.te_ypix + 40,
74      x);
75
76    }else{ // INVISIBLE
77      sys_vgui(".x%x.c delete %xbb\n",canvas, x);
78      sys_vgui(".x%x.c delete %xcb\n",canvas, x);
79      sys_vgui(".x%x.c delete %xrb\n",canvas, x);
80      sys_vgui(".x%x.c delete %xsb\n",canvas, x);
81      sys_vgui(".x%x.c delete %xrr\n",canvas, x);
82      sys_vgui(".x%x.c delete %xlb\n",canvas, x);
83      sys_vgui(".x%x.c delete %xtx\n",canvas, x);
84    }
85  }
```

Criamos os componentes gráficos e pedimos ao canvas para criar uma janela para que a mesma abrigue o componente. Assim, na hora de remover podemos remover apenas a janela. O mesmo ocorre na hora de mover. Note que além de adicionarmos componentes gráficos associamos eles a um comando. Este será nosso próximo tópico.

```
1   void my_displace(t_gobj *z, t_glist *glist,int dx, int dy){
2     t_canvas * canvas = glist_getcanvas(glist);
3     t_example15 *x = (t_example15 *)z;
4     x->x_obj.te_xpix += dx;
5     x->x_obj.te_ypix += dy;
6
7           sys_vgui(".x%lx.c coords %xSEL %d %d %d %d \n", //MOVE O
                SELECIONADO
8       glist_getcanvas(glist),
9       x,
10      x->x_obj.te_xpix,
11      x->x_obj.te_ypix,
12      x->x_obj.te_xpix + 170,
13      x->x_obj.te_ypix + 150
14      );
15          sys_vgui(".x%x.c coords %xrr %d %d %d %d\n",canvas,x,x->
              x_obj.te_xpix,x->x_obj.te_ypix,x->x_obj.te_xpix + 170,
              x->x_obj.te_ypix + 150);
```

```
16          sys_vgui(".x%x.c coords %xbb %d %d \n",canvas,x,x->x_obj.
               te_xpix,x->x_obj.te_ypix);
17          sys_vgui(".x%x.c coords %xcb %d %d \n",canvas,x,x->x_obj.
               te_xpix + 30,x->x_obj.te_ypix);
18          sys_vgui(".x%x.c coords %xsb %d %d \n",canvas,x,x->x_obj.
               te_xpix,x->x_obj.te_ypix + 80);
19          sys_vgui(".x%x.c coords %xtx %d %d \n",canvas,x,x->x_obj.
               te_xpix,x->x_obj.te_ypix + 120);
20          sys_vgui(".x%x.c coords %xrb %d %d \n",canvas,x,x->x_obj.
               te_xpix + 60,x->x_obj.te_ypix);
21          sys_vgui(".x%x.c coords %xlb %d %d \n",canvas,x,x->x_obj.
               te_xpix,x->x_obj.te_ypix + 40);
22
23    canvas_fixlinesfor(glist, (t_text *)x);
24  }
```

11.3 Adicionando comandos

Os comandos dos nossos componentes gráficos podem ser recebidos e interagirem com o external assim como o external consegue alterar os valores da GUI dependendo do que recebe em seus inlets. Veja o exemplo 21.

Para este exemplo, definimos na criação da classe no método setup() os métodos de retorno de nossa GUI.

```
1  // depois associaremos estes "tipos" de mensagem aos inlets 2 e 3
2      class_addmethod(example21_class, (t_method)example21_alfa,
           gensym("alfa"), A_DEFFLOAT,0);
3      class_addmethod(example21_class, (t_method)example21_beta,
           gensym("beta"), A_DEFFLOAT,0);
4  // metodo do botao ok
5      class_addmethod(example21_class, (t_method)example21_btok,
           gensym("btok"), A_DEFSYMBOL,0);
```

No método vis da GUI, criamos comandos.

```
1        sys_vgui("proc slide_alfa {val} {\n pd [concat example21%x
            alfa $val \\;]\n}\n",x);
2        sys_vgui("proc slide_beta {val} {\n pd [concat example21%x
            beta $val \\;]\n}\n",x);
3        sys_vgui("proc botao_ok {} {\n set name [.x%x.c.s%xtx get]\
            n pd [concat example21%x btok $name \\;]\n}\n",x->canvas
            ,x,x);
4        sys_vgui("proc botao_file_chooser {} {\n\
5                  set filename [tk_getOpenFile]\n\
6                .x%x.c.s%xtx delete 0 end \n\
7                .x%x.c.s%xtx insert end $filename \n}\n",x->canvas,
                    x,x->canvas,x);
8
9    (...)
10   sys_vgui("entry .x%x.c.s%xtx -width 25 -bg white -textvariable
        \"teste\" \n", x->canvas,x);
11   sys_vgui("button .x%x.c.s%xbfc -text {...} -command
        botao_file_chooser\n", x->canvas,x);
12   sys_vgui("scale .x%x.c.s%xsb1 -length 250 -resolution 0.01 -
        from 0.5 -to 2 -orient horizontal -command slide_alfa \n", x
        ->canvas,x);
```

```
13    sys_vgui("scale .x%x.c.s%xsb2 -length 250 -resolution 0.01 -
          from 0.5 -to 2 -orient horizontal -command slide_beta \n", x
          ->canvas,x);
14    sys_vgui("button .x%x.c.s%xbt -text {start} -command botao_ok\n
          ", x->canvas,x);
15    (...)
```

Os comandos alfa, beta e btok serão executados pelo objeto pd. Desta forma pedimos ao pd que, ao chamarmos este método o mesmo chame a função associada a este symbolo anteriormente. Os objetos criados usarão estes comandos para suas ações. Note que no caso do filechooser, o comando irá associar o caminho do arquivo escolhido com nosso campo text e tudo isto será feito diretamente no Tk.

Então basta criarmos as funções associadas:

```
1  static void example21_btok(t_example21* x, t_symbol * file_name){
2      x->file_name = file_name->s_name;
3      example21_bang(x);
4  }
5
6  // Metodo para definir o nome do arquivo com retorno para a GUI
7  void example21_set_file_name(t_example21 *x, char * file_name){
8      sys_vgui(".x%x.c.s%xtx delete 0 end \n", x->canvas, x);
9      sys_vgui(".x%x.c.s%xtx insert end %s\n", x->canvas, x ,
           file_name);
10     x->file_name = file_name;
11     // DO SOMETHING
12 }
13
14 void example21_alfa(t_example21 *x, t_floatarg f){
15     if( f >= 2) f = 2;
16     if( f <= 0.5) f = 0.5;
17     sys_vgui(".x%x.c.s%xsb1 set %f\n",x->canvas,x,f);
18     x->alfa = f;
19 }
20
21 void example21_beta(t_example21 *x, t_floatarg f){
22     if( f >= 2) f = 2;
23     if( f <= 0.5) f = 0.5;
24     sys_vgui(".x%x.c.s%xsb2 set %f\n",x->canvas,x,f);
25     x->beta = f;
26 }
```

O resultado é visto a seguir.

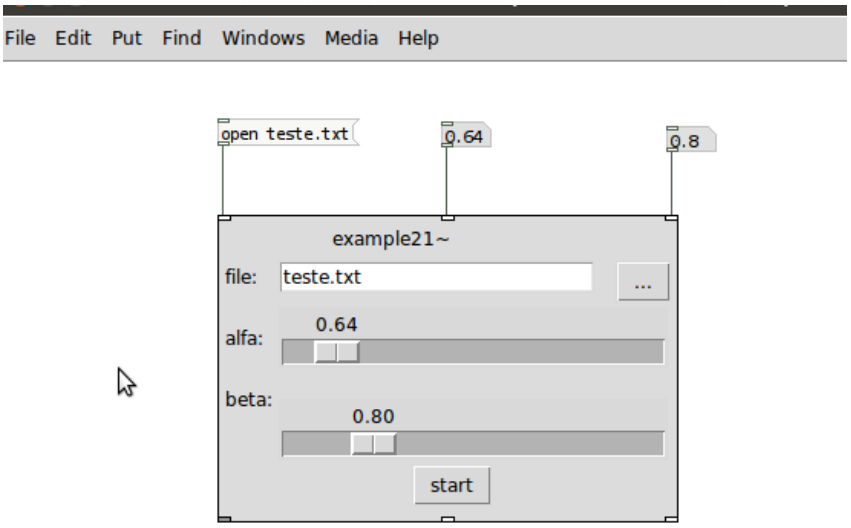

Figura 11.2: External com GUI e comandos

Capítulo 12

Miscelâneas

Aqui temos uma lista de funções definidas pela biblioteca que permite criarmos externals (m_pd.h). Não se trata de todas as funções mas de algumas que podem ser útil termos descrito como material de referência.

Ideal mesmo era rechear esta documentação com exemplos.

12.1 Gerenciamento de memória

```
void *getbytes(size_t nbytes);
```

```
void *getzbytes(size_t nbytes);
```

```
void *copybytes(void *src, size_t nbytes);
```

```
void freebytes(void *x, size_t nbytes);
```

Desaloca um ponteiro da memória.

```
void *resizebytes(void *x, size_t oldsize, size_t newsize);
```

12.2 Atoms

```
t_float atom_getfloat(t_atom *a);
```

```
t_int atom_getint(t_atom *a);
```

```
t_symbol *atom_getsymbol(t_atom *a);
```

```
t_symbol *atom_gensym(t_atom *a);
```

```
t_float atom_getfloatarg(int which, int argc, t_atom *argv);
```

```
t_int atom_getintarg(int which, int argc, t_atom *argv);
```

```
t_symbol *atom_getsymbolarg(int which, int argc, t_atom *argv);
```

```
void atom_string(t_atom *a, char *buf, unsigned int bufsize);
```

12.3 Binbufs

```
t_binbuf *binbuf_new(void);
```

```
void binbuf_free(t_binbuf *x);
```

```
t_binbuf *binbuf_duplicate(t_binbuf *y);
```

```
void binbuf_text(t_binbuf *x, char *text, size_t size);
```

```
void binbuf_gettext(t_binbuf *x, char **bufp, int *lengthp);
```

```
void binbuf_clear(t_binbuf *x);
```

```
void binbuf_add(t_binbuf *x, int argc, t_atom *argv);
```

```
void binbuf_addv(t_binbuf *x, char *fmt, ...);
```

```
void binbuf_addbinbuf(t_binbuf *x, t_binbuf *y);
```

```
void binbuf_addsemi(t_binbuf *x);
```

```
void binbuf_restore(t_binbuf *x, int argc, t_atom *argv);
```

```
void binbuf_print(t_binbuf *x);
```

```
int binbuf_getnatom(t_binbuf *x);
```

```
t_atom *binbuf_getvec(t_binbuf *x);
```

```
void binbuf_eval(t_binbuf *x, t_pd *target, int argc, t_atom *argv);
```

```
int binbuf_read(t_binbuf *b, char *filename, char *dirname, int crflag);
```

```
int binbuf_read_via_canvas(t_binbuf *b, char *filename, t_canvas *canvas, int crflag);
```

```
int binbuf_read_via_path(t_binbuf *b, char *filename, char *dirname, int crflag);
```

```
int binbuf_write(t_binbuf *x, char *filename, char *dir, int crflag);
```

```
void binbuf_evalfile(t_symbol *name, t_symbol *dir);
```

```
t_symbol *binbuf_realizedollsym(t_symbol *s, int ac, t_atom *av, int tonew);
```

12.4 Clocks

```
t_clock *clock_new(void *owner, t_method fn);
```

```
void clock_set(t_clock *x, double systime);
```

```
void clock_delay(t_clock *x, double delaytime);
```

```
void clock_unset(t_clock *x);
```

```
double clock_getlogicaltime(void);
```

```
double clock_getsystime(void);
```
OBSOLETE; use clock_getlogicaltime()

```
double clock_gettimesince(double prevsystime);
```
Elapsed time in milliseconds since the given system time

```
double clock_getsystimeafter(double delaytime);
```

```
void clock_free(t_clock *x);
```

12.5 Pure data

```
t_pd *pd_new(t_class *cls);
```

```
void pd_free(t_pd *x);
```

```
void pd_bind(t_pd *x, t_symbol *s);
```

```
void pd_unbind(t_pd *x, t_symbol *s);
```

```
t_pd *pd_findbyclass(t_symbol *s, t_class *c);
```

Verifica se ha um objeto desta classe instanciado no patch atual.

```
void pd_pushsym(t_pd *x);
```

```
void pd_popsym(t_pd *x);
```

```
t_symbol *pd_getfilename(void);
```

```
t_symbol *pd_getdirname(void);
```

```
void pd_bang(t_pd *x);
```

```
void pd_pointer(t_pd *x, t_gpointer *gp);
```

```
void pd_float(t_pd *x, t_float f);
```

```
void pd_symbol(t_pd *x, t_symbol *s);
```

```
void pd_list(t_pd *x, t_symbol *s, int argc, t_atom *argv);
```

```
void pd_anything(t_pd *x, t_symbol *s, int argc, t_atom *argv);
```

12.6 Pointers

```
void gpointer_init(t_gpointer *gp);
```

```
void gpointer_copy(const t_gpointer *gpfrom, t_gpointer *gpto);
```

```
void gpointer_unset(t_gpointer *gp);
```

```
int gpointer_check(const t_gpointer *gp, int headok);
```

12.7 Inlets and outlets

```
t_inlet *inlet_new(t_object *owner, t_pd *dest, t_symbol *s1, t_symbol *s2);
```

```
t_inlet *pointerinlet_new(t_object *owner, t_gpointer *gp);
```

```
t_inlet *floatinlet_new(t_object *owner, t_float *fp);
```

```
t_inlet *symbolinlet_new(t_object *owner, t_symbol **sp);
```

```
t_inlet *signalinlet_new(t_object *owner, t_float f);
```

```
void inlet_free(t_inlet *x);
```

```
t_outlet *outlet_new(t_object *owner, t_symbol *s);
```

```
void outlet_bang(t_outlet *x);
```

```
void outlet_pointer(t_outlet *x, t_gpointer *gp);
```

```
void outlet_float(t_outlet *x, t_float f);
```

```
void outlet_symbol(t_outlet *x, t_symbol *s);
```

```
void outlet_list(t_outlet *x, t_symbol *s, int argc, t_atom *argv);
```

```
void outlet_anything(t_outlet *x, t_symbol *s, int argc, t_atom *argv);
```

```
t_symbol *outlet_getsymbol(t_outlet *x);
```

```
void outlet_free(t_outlet *x);
```

```
t_object *pd_checkobject(t_pd *x);
```

12.8 Canvases

```
void glob_setfilename(void *dummy, t_symbol *name, t_symbol *dir);
```

```
void canvas_setargs(int argc, t_atom *argv);
```

```
void canvas_getargs(int *argcp, t_atom **argvp);
```

```
t_symbol *canvas_getcurrentdir(void);
```

```
t_glist *canvas_getcurrent(void);
```

```
void canvas_makefilename(t_glist *c, char *file, char *result,int resultsize);
```

```
t_symbol *canvas_getdir(t_glist *x);
```

```
char sys_font[];
    /* default typeface set in s_main.c */
```

```
char sys_fontweight[];
    /* default font weight set in s_main.c */
```

```
int sys_fontwidth(int fontsize);
```

```
int sys_fontheight(int fontsize);
```

```
void canvas_dataproperties(t_glist *x, t_scalar *sc, t_binbuf *b);
```

```
int canvas_open(t_canvas *x, const char *name, const char *ext, char *dirresult, char **nameresult,
unsigned int size, int bin);
```

12.9 Classes

```
t_class *class_new(t_symbol *name, t_newmethod newmethod, t_method freemethod, size_t size,
int flags, t_atomtype arg1, ...);
```

```
void class_addcreator(t_newmethod newmethod, t_symbol *s, t_atomtype type1, ...);
```

```
void class_addmethod(t_class *c, t_method fn, t_symbol *sel, t_atomtype arg1, ...);
```

```
void class_addbang(t_class *c, t_method fn);
```

```
void class_addpointer(t_class *c, t_method fn);
```

```
void class_doaddfloat(t_class *c, t_method fn);
```

```
void class_addsymbol(t_class *c, t_method fn);
```

```
void class_addlist(t_class *c, t_method fn);
```

```
void class_addanything(t_class *c, t_method fn);
```

```
void class_sethelpsymbol(t_class *c, t_symbol *s);
```
Define o arquivo de help para a classe.

```
void class_setwidget(t_class *c, t_widgetbehavior *w);
```
Define qual será o comportamento da GUI deste objeto. Quando esta função é chamada o Pd não se responsabiliza mais quanto ao desenho do external.

```
void class_setparentwidget(t_class *c, t_parentwidgetbehavior *w);
```

```
t_parentwidgetbehavior *class_parentwidget(t_class *c);
```

```
char *class_getname(t_class *c);
```

```
char *class_gethelpname(t_class *c);
```

```
void class_setdrawcommand(t_class *c);
```

```
int class_isdrawcommand(t_class *c);
```

```
void class_domainsignalin(t_class *c, int onset);
```

```
void class_set_extern_dir(t_symbol *s);
```

```
void class_setsavefn(t_class *c, t_savefn f);
```

```
t_savefn class_getsavefn(t_class *c);
```

```
void class_setpropertiesfn(t_class *c, t_propertiesfn f);
```

```
t_propertiesfn class_getpropertiesfn(t_class *c);
```

12.10 Printing

```
void post(const char *fmt, ...);
```
Envia um mensagem para a saída padrão do PD. Funciona de maneira similar ao printf e aceita argumentos como %d ou %f. Ao contrário do printf, quebra linha no final.

```
void startpost(const char *fmt, ...);
```

```
void poststring(const char *s);
```

```
void postfloat(t_floatarg f);
```

```
void postatom(int argc, t_atom *argv);
```

```
void endpost(void);
```

```
void error(const char *fmt, ...);
```

```
void verbose(int level, const char *fmt, ...);
```

```
void bug(const char *fmt, ...);
```

```
void pd_error(void *object, const char *fmt, ...);
```

```
void sys_logerror(const char *object, const char *s);
```

```
void sys_unixerror(const char *object);
```

```
void sys_ouch(void);
```

12.11 System interface routines

```
int sys_isreadablefile(const char *name);
```

```
int sys_isabsolutepath(const char *dir);
```

```
void sys_bashfilename(const char *from, char *to);
```

```
void sys_unbashfilename(const char *from, char *to);
```

```
int open_via_path(const char *name, const char *ext, const char *dir, char *dirresult, char
**nameresult, unsigned int size, int bin);
```

```
int sched_geteventno(void);
```

```
double sys_getrealtime(void);
```

```
int (*sys_idlehook)(void);
```
Hook to add idle time computation

12.12 Threading

```
void sys_lock(void);
```

```
void sys_unlock(void);
```

```
int sys_trylock(void);
```

12.13 Signals

```
t_int *plus_perform(t_int *args);
```

```
t_int *zero_perform(t_int *args);
```

```
t_int *copy_perform(t_int *args);
```

```
void dsp_add_plus(t_sample *in1, t_sample *in2, t_sample *out, int n);
```

```
void dsp_add_copy(t_sample *in, t_sample *out, int n);
```

```
void dsp_add_scalarcopy(t_float *in, t_sample *out, int n);
```

```
void dsp_add_zero(t_sample *out, int n);
```

```
int sys_getblksize(void);
```
Retorna o tamanho do bloco de processamento do Pure Data.

```
t_float sys_getsr(void);
```
Retorna qual a amostragem (Sample Rate) atual do Pure Data.

```
int sys_get_inchannels(void);
```
Retorna a quantidade de canais de entrada do Pure Data.

```
int sys_get_outchannels(void);
```
Retorna a quantidade de canais de saída do Pure Data.

```
void dsp_add(t_perfroutine f, int n, ...);
```

```
void dsp_addv(t_perfroutine f, int n, t_int *vec);
```

```
void pd_fft(t_float *buf, int npoints, int inverse);
```

```
int ilog2(int n);
```

```
void mayer_fht(t_sample *fz, int n);
```

```
void mayer_fft(int n, t_sample *real, t_sample *imag);
```

```
void mayer_ifft(int n, t_sample *real, t_sample *imag);
```

```
void mayer_realfft(int n, t_sample *real);
```

```
void mayer_realifft(int n, t_sample *real);
```

```
float *cos_table;
```

```
int canvas_suspend_dsp(void);
```

```
void canvas_resume_dsp(int oldstate);
```

```
void canvas_update_dsp(void);
```

```
int canvas_dspstate;
```

```
typedef struct _resample {
int method; // up/downsampling method ID
t_int downsample; // downsampling factor
t_int upsample; // upsampling factor
t_sample *s_vec; // here we hold the resampled data
int s_n;
t_sample *coeffs; // coefficients for filtering...
int coefsize;
t_sample *buffer; // buffer for filtering
int bufsize;
} t_resample;
```

```
void resample_init(t_resample *x);
```

```
void resample_free(t_resample *x);
```

```
void resample_dsp(t_resample *x, t_sample *in, int insize, t_sample *out, int outsize, int
method);
```

```
void resamplefrom_dsp(t_resample *x, t_sample *in, int insize, int outsize, int method);
```

```
void resampleto_dsp(t_resample *x, t_sample *out, int insize, int outsize, int method);
```

12.14 Utility functions for signals

```
t_float mtof(t_float);
```
Converte valores MIDI em frequencia.

```
t_float ftom(t_float);
```
Converte frequencias em valores MIDI.

```
t_float rmstodb(t_float);
```
Converte amplitudes RMS em decibéis.

```
t_float powtodb(t_float);
```
Converte potência em decibél.

```
t_float dbtorms(t_float);
```
Converte decibél para RMS.

```
t_float dbtopow(t_float);
```
Converte decibél para potência.

```
t_float q8_sqrt(t_float);
```

```
t_float q8_rsqrt(t_float);
```

12.15 Data

```
t_class *garray_class;
```

```
int garray_getfloatarray(t_garray *x, int *size, t_float **vec);
```

```
int garray_getfloatwords(t_garray *x, int *size, t_word **vec);
```

```
t_float garray_get(t_garray *x, t_symbol *s, t_int indx);
```

```
void garray_redraw(t_garray *x);
```

```
int garray_npoints(t_garray *x);
```

```
char *garray_vec(t_garray *x);
```

```
void garray_resize(t_garray *x, t_floatarg f);
```

```
void garray_usedindsp(t_garray *x);
```

```
void garray_setsaveit(t_garray *x, int saveit);
```

```
t_class *scalar_class;
```

```
t_float *value_get(t_symbol *s);
```

```
void value_release(t_symbol *s);
```

```
int value_getfloat(t_symbol *s, t_float *f);
```

```
int value_setfloat(t_symbol *s, t_float f);
```

12.16 GUI interface - functions to send strings to TK

```
void sys_vgui(char *fmt, ...);
```
Envia comandos Tk para o canvas com argumentos.

```
void sys_gui(char *s);
```
Envia comandos Tk para o canvas.

```
void sys_pretendguibytes(int n);
```

```
void sys_queuegui(void *client, t_glist *glist, t_guicallbackfn f);
```

```
void sys_unqueuegui(void *client);
```

```
void gfxstub_new(t_pd *owner, void *key, const char *cmd);
```

```
void gfxstub_deleteforkey(void *key);
```

```
t_class *glob_pdobject;
```
object to send ”pd”messages

Parte IV

Usando o PD para construir aplicações

Capítulo 13

De *externals* para código C

Como transformar seu patch em código C?
Vamos ver!