HOWTO
write an External
for *Pure Data*

IOhannes m zmölnig

institute of electronic music and acoustics

July 27, 2021

Abstract

Pd is a graphical real-time computer-music system that follows the tradition of IRCAMs *ISPW-max*.

Although plenty of functions are built into Pd, it is sometimes a pain or simply impossible to create a patch with a certain functionality out of the given primitives and combinations of these.

Therefore, Pd can be extended with self made primitives ("objects") that are written in complex programming-languages, like `C/C++`.

This document aims to explain how to write such primitives in `C`, the popular language that was used to realize Pd.

Contents

1 definitions and prerequisites

Pd refers to the graphical real-time computer-music environment *Pure Data* by Miller S. Puckette.

To fully understand this document, it is necessary to be acquainted with Pd and to have a general understanding of programming techniques especially in `C`.

To write externals yourself, a `C`-compiler that supports the `ANSI-C`-Standard, like the *Gnu C-compiler* (gcc) on linux-systems or *Visual-C++* on windos-plattforms, will be necessary.

1.1 classes, instances, objects

Pd is written in the programming-language `C`. Due to its graphical nature, Pd is a *object-oriented* system. Unfortunately `C` does not support very well the use of classes. Thus the resulting source-code is not as elegant as `C++`-code would be, for instance.

In this document, the expression *class* refers to the realisation of a concept combining data and manipulators on this data.

Concrete *instances of a class* are called *objects*.

1.2 internals, externals und libraries

To avoid confusion of ideas, the expressions *internal*, *external* and *library* should be explained here.

Internal An *internal* is a class that is built into Pd. Plenty of primitives, such as "+", "pack" or "sig~" are *internals*.

External An *external* is a class that is not built into Pd but is loaded at runtime. Once loaded into Pd's memory, *externals* cannot be distinguished from *internals* any more.

Library A *library* is a collection of *externals* that are compiled into a single binary-file.

Library-files have to follow a system dependent naming convention:

library	linux	irix	Win32
`my_lib`	`my_lib.pd_linux`	`my_lib.pd_irix`	`my_lib.dll`

The simplest form of a *library* includes exactly one *external* bearing the same name as the *library*.

Unlike *externals*, *libraries* can be imported by Pd with special operations. After a *library* has been imported, all included *externals* have been loaded into memory and are available as objects.

Pd supports two modes to import *libraries*:

- via the command line-option "`-lib my_lib`"
- by creating an object "`my_lib`"

The first method loads a *library* when Pd is started. This method is preferably used for *libraries* that contain several *externals*.

The other method should be used for *libraries* that contain exactly one *external* bearing the same name. Pd checks first, whether a class named "my_lib" is already loaded. If this is not the case[1], all paths are searched for a file called "`my_lib.pd_linux`"[2]. If such file is found, all included *externals* are loaded into memory by calling a routine `my_lib_setup()`. After loading, a class "my_lib" is (again) looked for as a (newly loaded) *external*. If so, an instance of this class is created, else the instantiation fails and an error is printed. Anyhow, all *external*-classes declared in the *library* are loaded by now.

2 my first external: `helloworld`

Usually the first attempt learning a programming-language is a "hello world"-application.

In our case, an object class should be created, that prints the line "hello world!!" to the standard error every time it is triggered with a "bang"-message.

2.1 the interface to Pd

To write a Pd-external a well-defined interface is needed. This is provided in the header-file "m_pd.h".

```
#include "m_pd.h"
```

[1] If a class "my_lib" is already existent, an object "my_lib" will be instantiated and the procedure is done. Thus, no *library* has been loaded. Therefore no *library* that is named like an already used class-name like, say, "abs", can be loaded.

[2] or another system-dependent filename-extensions (s.a.)

2.2 a class and its data space

First a new class has to be prepared and the data space for this class has to be defined.

```
static t_class *helloworld_class;

typedef struct _helloworld {
  t_object  x_obj;
} t_helloworld;
```

`hello_worldclass` is going to be a pointer to the new class.

The structure `t_helloworld` (of the type `_helloworld`) is the data space of the class.

An absolutely necessary element of the data space is a variable of the type `t_object`, which is used to store internal object-properties like the graphical presentation of the object or data about inlets and outlets.

`t_object` has to be the first entry in the structure !

Because a simple "hello world"-application needs no variables, the structure is empty apart from the `t_object`.

2.3 method space

Apart from the data space, a class needs a set of manipulators (methods) to manipulate the data with.

If a message is sent to an instance of our class, a method is called. These methods are the interfaces to the message system of Pd. On principal they have no return argument and are therefore of the type `void`.

```
void helloworld_bang(t_helloworld *x)
{
  post("Hello world !!");
}
```

This method has an argument of the type `t_helloworld`, which would enable us to manipulate the data space.

Since we only want to output "Hello world!" (and, by the way, our data space is quite sparse), we renounce a manipulation.

The command `post(char *c,...)` sends a string to the standard error. A carriage return is added automatically. Apart from this, the `post`-command works like the `C`-command `printf()`.

2.4 generation of a new class

To generate a new class, information of the data space and the method space of this class, have to be passed to Pd when a library is loaded.

On loading a new library "my_lib", Pd tries to call a function "my_lib_setup()". This function (or functions called by it) declares the new classes and their properties. It is only called once, when the library is loaded. If the function-call fails (e.g., because no function of the specified name is present) no external of the library will be loaded.

```
void helloworld_setup(void)
{
  helloworld_class = class_new(gensym("helloworld"),
        (t_newmethod)helloworld_new,
        0, sizeof(t_helloworld),
        CLASS_DEFAULT, 0);

  class_addbang(helloworld_class, helloworld_bang);
}
```

class_new The function `class_new` creates a new class and returns a pointer to this prototype.

The first argument is the symbolic name of the class.

The next two arguments define the constructor and destructor of the class.

Whenever a class object is created in a Pd-patch, the class-constructor `(t_newmethod)helloworld_new` instantiates the object and initialises the data space.

Whenever an object is destroyed (either by closing the containing patch or by deleting the object from the patch) the destructor frees the dynamically reserved memory. The allocated memory for the static data space is automatically reserved and freed.

Therefore we do not have to provide a destructor in this example, the argument is set to "0".

To enable Pd to reserve and free enough memory for the static data space, the size of the data structure has to be passed as the fourth argument.

The fifth argument has influence on the graphical representation of the class objects. The default-value is `CLASS_DEFAULT` or simply "0".

The remaining arguments define the arguments of an object and its type.

Up to six numeric and symbolic object-arguments can be defined via `A_DEFFLOAT` and `A_DEFSYMBOL`. If more arguments are to be passed to the

object or if the order of atom types should by more flexible, `A_GIMME` can be used for passing an arbitrary list of atoms.

The list of object-arguments is terminated by "0". In this example we have no object-arguments at all for the class.

class_addbang We still have to add a method space to the class.

`class_addbang` adds a method for a "bang"-message to the class that is defined in the first argument. The added method is defined in the second argument.

2.5 constructor: instantiation of an object

Each time, an object is created in a Pd-patch, the constructor that is defined with the `class_new`-command, generates a new instance of the class.

The constructor has to be of type `void *`.

```
void *helloworld_new(void)
{
  t_helloworld *x = (t_helloworld *)pd_new(helloworld_class);

  return (void *)x;
}
```

The arguments of the constructor-method depend on the object-arguments defined with `class_new`.

`class_new`-argument	constructor-argument
`A_DEFFLOAT`	`t_floatarg f`
`A_DEFSYMBOL`	`t_symbol *s`
`A_GIMME`	`t_symbol *s, int argc, t_atom *argv`

Because there are no object-arguments for our "hello world"-class, the constructor has anon too.

The function `pd_new` reserves memory for the data space, initialises the variables that are internal to the object and returns a pointer to the data space.

The type-cast to the data space is necessary.

Normally, the constructor would initialise the object-variables. However, since we have none, this is not necessary.

The constructor has to return a pointer to the instantiated data space.

2.6 the code: helloworld

```
#include "m_pd.h"

static t_class *helloworld_class;

typedef struct _helloworld {
  t_object  x_obj;
} t_helloworld;

void helloworld_bang(t_helloworld *x)
{
  post("Hello world !!");
}

void *helloworld_new(void)
{
  t_helloworld *x = (t_helloworld *)pd_new(helloworld_class);

  return (void *)x;
}

void helloworld_setup(void) {
  helloworld_class = class_new(gensym("helloworld"),
        (t_newmethod)helloworld_new,
        0, sizeof(t_helloworld),
        CLASS_DEFAULT, 0);
  class_addbang(helloworld_class, helloworld_bang);
}
```

3 a simple external: counter

Now we want to realize a simple counter as an external. A "bang"-trigger outputs the counter-value on the outlet and afterwards increases the counter-value by 1.

This class is similar to the previous one, but the data space is extended by a variable "counter" and the result is written as a message to an outlet instead of a string to the standard error.

3.1 object-variables

Of course, a counter needs a state-variable to store the actual counter-value.

State-variables that belong to class instances belong to the data space.

```
typedef struct _counter {
  t_object  x_obj;
  int i_count;
} t_counter;
```

The integer variable `i_count` stores the counter-value.

3.2 object-arguments

It is quite useful for a counter, if a initial value can be defined by the user. Therefore this initial value should be passed to the object at creation-time.

```
void counter_setup(void) {
  counter_class = class_new(gensym("counter"),
        (t_newmethod)counter_new,
        0, sizeof(t_counter),
        CLASS_DEFAULT,
        A_DEFFLOAT, 0);

  class_addbang(counter_class, counter_bang);
}
```

So we have an additional argument in the function `class_new`: `A_DEFFLOAT` tells Pd, that the object needs one argument of the type `t_floatarg`. If no argument is passed, this will default to "0".

3.3 constructor

The constructor has some new tasks. On the one hand, a variable value has to be initialised, on the other hand, an outlet for the object has to be created.

```
void *counter_new(t_floatarg f)
{
  t_counter *x = (t_counter *)pd_new(counter_class);

  x->i_count=f;
  outlet_new(&x->x_obj, &s_float);
```

```
  return (void *)x;
}
```

The constructor-method has one argument of type `t_floatarg` as declared in the setup-routine by `class_new`. This argument is used to initialise the counter.

A new outlet is created with the function `outlet_new`. The first argument is a pointer to the interna of the object the new outlet is created for.

The second argument is a symbolic description of the outlet-type. Since out counter should output numeric values it is of type "float".

`outlet_new` returns a pointer to the new outlet and saves this very pointer in the `t_object`-variable `x_obj.ob_outlet`. If only one outlet is used, the pointer need not additionally be stored in the data space. If more than one outlets are used, the pointers have to be stored in the data space, because the `t_object`-variable can only hold one outlet pointer.

3.4 the counter method

When triggered, the counter value should be sent to the outlet and afterwards be incremented by 1.

```
void counter_bang(t_counter *x)
{
  t_float f=x->i_count;
  x->i_count++;
  outlet_float(x->x_obj.ob_outlet, f);
}
```

The function `outlet_float` sends a floating-point-value (second argument) to the outlet that is specified by the first argument.

We first store the counter in a floating point-buffer. Afterwards the counter is incremented and not before that the buffer variable is sent to the outlet.

What appears to be unnecessary on the first glance, makes sense after further inspection: The buffer variable has been realized as `t_float`, since `outlet_float` expects a floating point-value and a typecast is inevitable.

If the counter value was sent to the outlet before being incremented, this could result in an unwanted (though well defined) behaviour: If the counter-outlet directly triggered its own inlet, the counter-method would be called although the counter value was not yet incremented. Normally this is not what we want.

The same (correct) result could of course be obtained with a single line, but this would obscure the *reentrant*-problem.

3.5 the code: `counter`

```
#include "m_pd.h"

static t_class *counter_class;

typedef struct _counter {
  t_object  x_obj;
  int i_count;
} t_counter;

void counter_bang(t_counter *x)
{
  t_float f=x->i_count;
  x->i_count++;
  outlet_float(x->x_obj.ob_outlet, f);
}

void *counter_new(t_floatarg f)
{
  t_counter *x = (t_counter *)pd_new(counter_class);

  x->i_count=f;
  outlet_new(&x->x_obj, &s_float);

  return (void *)x;
}

void counter_setup(void) {
  counter_class = class_new(gensym("counter"),
        (t_newmethod)counter_new,
        0, sizeof(t_counter),
        CLASS_DEFAULT,
        A_DEFFLOAT, 0);

  class_addbang(counter_class, counter_bang);
}
```

4 a complex external: `counter`

The simple counter of the previous chapter can easily be extended to more complexity. It might be quite useful to be able to reset the counter to an initial value, to set upper and lower boundaries and to control the step-width. Each overrun should send a "bang"-Message to a second outlet and reset the counter to the initial value.

4.1 extended data space

```
typedef struct _counter {
  t_object  x_obj;
  int i_count;
  t_float step;
  int i_down, i_up;
  t_outlet *f_out, *b_out;
} t_counter;
```

The data space has been extended to hold variables for step width and upper and lower boundaries. Furthermore pointers for two outlets have been added.

4.2 extension of the class

The new class objects should have methods for different messages, like "set" and "reset". Therefore the method space has to be extended too.

```
  counter_class = class_new(gensym("counter"),
        (t_newmethod)counter_new,
        0, sizeof(t_counter),
        CLASS_DEFAULT,
        A_GIMME, 0);
```

The class generator `class_new` has been extended by the argument `A_GIMME`. This enables a dynamic number of arguments to be passed at the instantiation of the object.

```
  class_addmethod(counter_class,
        (t_method)counter_reset,
        gensym("reset"), 0);
```

`class_addmethod` adds a method for an arbitrary selector to an class.

The first argument is the class the method (second argument) will be added to.

The third argument is the symbolic selector that should be associated with the method.

The remaining "0"-terminated arguments describe the list of atoms that follows the selector.

```
class_addmethod(counter_class,
      (t_method)counter_set, gensym("set"),
      A_DEFFLOAT, 0);
class_addmethod(counter_class,
      (t_method)counter_bound, gensym("bound"),
      A_DEFFLOAT, A_DEFFLOAT, 0);
```

A method for "set" followed by a numerical value is added, as well as a method for the selector "bound" followed by two numerical values.

```
class_sethelpsymbol(counter_class, gensym("help-counter"));
```

If a Pd-object is right-clicked, a help-patch describing the object-class can be opened. By default, this patch is located in the directory "*doc/5.reference/*" and is named like the symbolic class name.

An alternative help-patch can be defined with the `class_sethelpsymbol`-command.

4.3 construction of in- and outlets

When creating the object, several arguments should be passed by the user.

```
void *counter_new(t_symbol *s, int argc, t_atom *argv)
```

Because of the declaration of arguments in the `class_new`-function with `A_GIMME`, the constructor has following arguments:

`t_symbol *s`	the symbolic name, that was used for object creation
`int argc`	the number of arguments passed to the object
`t_atom *argv`	a pointer to a list of `argc` atoms

```
t_float f1=0, f2=0;

x->step=1;
```

```
switch(argc){
default:
case 3:
  x->step=atom_getfloat(argv+2);
case 2:
  f2=atom_getfloat(argv+1);
case 1:
  f1=atom_getfloat(argv);
  break;
case 0:
  break;
}
if (argc<2)f2=f1;
x->i_down = (f1<f2)?f1:f2;
x->i_up   = (f1>f2)?f1:f2;

x->i_count=x->i_down;
```

If three arguments are passed, these should be the *lower boundary*, the *upper boundary* and the *step width*.

If only two arguments are passed, the step-width defaults to "1". If only one argument is passed, this should be the *initial value* of the counter with step-width of "1".

```
inlet_new(&x->x_obj, &x->x_obj.ob_pd,
      gensym("list"), gensym("bound"));
```

The function `inlet_new` creates a new "active" inlet. "Active" means, that a class-method is called each time a message is sent to an "active" inlet.

Due to the software-architecture, the first inlet is always "active".

The first two arguments of the `inlet_new`-function are pointers to the interna of the object and to the graphical presentation of the object.

The symbolic selector that is specified by the third argument is to be substituted by another symbolic selector (fourth argument) for this inlet.

Because of this substitution of selectors, a message on a certain right inlet can be treated as a message with a certain selector on the leftmost inlet.

This means:

- The substituting selector has to be declared by `class_addmethod` in the setup-routine.
- It is possible to simulate a certain right inlet, by sending a message with this inlet's selector to the leftmost inlet.

- It is not possible to add methods for more than one selector to a right inlet. Particularly it is not possible to add a universal method for arbitrary selectors to a right inlet.

```
floatinlet_new(&x->x_obj, &x->step);
```

`floatinlet_new` generates a new "passive" inlet for numerical values. "Passive" inlets allow parts of the data space-memory to be written directly from outside. Therefore it is not possible to check for illegal inputs.

The first argument is a pointer to the internal infrastructure of the object. The second argument is the address in the data space-memory, where other objects can write too.

"Passive" inlets can be created for pointers, symbolic or numerical (floating point[3]) values.

```
x->f_out = outlet_new(&x->x_obj, &s_float);
x->b_out = outlet_new(&x->x_obj, &s_bang);
```

The pointers returned by `outlet_new` have to be saved in the classdata space to be used later by the outlet-routines.

The order of the generation of inlets and outlets is important, since it corresponds to the order of inlets and outlets in the graphical representation of the object.

4.4 extended method space

The method for the "bang"-message has to full fill the more complex tasks.

```
void counter_bang(t_counter *x)
{
  t_float f=x->i_count;
  int step = x->step;
  x->i_count+=step;
  if (x->i_down-x->i_up) {
    if ((step>0) && (x->i_count > x->i_up)) {
      x->i_count = x->i_down;
      outlet_bang(x->b_out);
    } else if (x->i_count < x->i_down) {
      x->i_count = x->i_up;
      outlet_bang(x->b_out);
```

[3]That's why the `step`-width of the classdata space is realized as `t_float`.

```
    }
  }
  outlet_float(x->f_out, f);
}
```

Each outlet is identified by the `outlet_...`-functions via the pointer to this outlets.

The remaining methods still have to be implemented:

```
void counter_reset(t_counter *x)
{
  x->i_count = x->i_down;
}

void counter_set(t_counter *x, t_floatarg f)
{
  x->i_count = f;
}

void counter_bound(t_counter *x, t_floatarg f1, t_floatarg f2)
{
  x->i_down = (f1<f2)?f1:f2;
  x->i_up   = (f1>f2)?f1:f2;
}
```

4.5 the code: `counter`

```
#include "m_pd.h"

static t_class *counter_class;

typedef struct _counter {
  t_object  x_obj;
  int i_count;
  t_float step;
  int i_down, i_up;
  t_outlet *f_out, *b_out;
} t_counter;

void counter_bang(t_counter *x)
{
```

```
  t_float f=x->i_count;
  int step = x->step;
  x->i_count+=step;

  if (x->i_down-x->i_up) {
    if ((step>0) && (x->i_count > x->i_up)) {
      x->i_count = x->i_down;
      outlet_bang(x->b_out);
    } else if (x->i_count < x->i_down) {
      x->i_count = x->i_up;
      outlet_bang(x->b_out);
    }
  }

  outlet_float(x->f_out, f);
}

void counter_reset(t_counter *x)
{
  x->i_count = x->i_down;
}

void counter_set(t_counter *x, t_floatarg f)
{
  x->i_count = f;
}

void counter_bound(t_counter *x, t_floatarg f1, t_floatarg f2)
{
  x->i_down = (f1<f2)?f1:f2;
  x->i_up   = (f1>f2)?f1:f2;
}

void *counter_new(t_symbol *s, int argc, t_atom *argv)
{
  t_counter *x = (t_counter *)pd_new(counter_class);
  t_float f1=0, f2=0;

  x->step=1;
  switch(argc){
  default:
```

```
  case 3:
    x->step=atom_getfloat(argv+2);
  case 2:
    f2=atom_getfloat(argv+1);
  case 1:
    f1=atom_getfloat(argv);
    break;
  case 0:
    break;
  }
  if (argc<2)f2=f1;

  x->i_down = (f1<f2)?f1:f2;
  x->i_up   = (f1>f2)?f1:f2;

  x->i_count=x->i_down;

  inlet_new(&x->x_obj, &x->x_obj.ob_pd,
        gensym("list"), gensym("bound"));
  floatinlet_new(&x->x_obj, &x->step);

  x->f_out = outlet_new(&x->x_obj, &s_float);
  x->b_out = outlet_new(&x->x_obj, &s_bang);

  return (void *)x;
}

void counter_setup(void) {
  counter_class = class_new(gensym("counter"),
        (t_newmethod)counter_new,
        0, sizeof(t_counter),
        CLASS_DEFAULT,
        A_GIMME, 0);

  class_addbang  (counter_class, counter_bang);
  class_addmethod(counter_class,
        (t_method)counter_reset, gensym("reset"), 0);
  class_addmethod(counter_class,
        (t_method)counter_set, gensym("set"),
        A_DEFFLOAT, 0);
  class_addmethod(counter_class,
```

```
        (t_method)counter_bound, gensym("bound"),
        A_DEFFLOAT, A_DEFFLOAT, 0);

  class_sethelpsymbol(counter_class, gensym("help-counter"));
}
```

5 a signal-external: pan~

Signal classes are normal Pd-classes, that offer additional methods for signals.

All methods and concepts that can be realized with normal object classes can therefore be realized with signal classes too.

Per agreement, the symbolic names of signal classes end with a tilde ~.

The class "pan~" shall demonstrate, how signal classes are written.

A signal on the left inlet is mixed with a signal on the second inlet. The mixing-factor between 0 and 1 is defined via a `t_float`-message on a third inlet.

5.1 variables of a signal class

Since a signal-class is only an extended normal class, there are no principal differences between the data spaces.

```
typedef struct _pan_tilde {
  t_object x_obj;

  t_sample f_pan;
  t_float  f;

  t_inlet *x_in2;
  t_inlet *x_in3;

  t_outlet*x_out;

} t_pan_tilde;
```

Only one variable `f_pan` for the *mixing-factor* of the panning-function is needed.

The other variable `f` is needed whenever a signal-inlet is needed too. If no signal but only a float-message is present at a signal-inlet, this variable is used to automatically convert the float to signal.

Finally, we have the members `x_in2`, `x_in3` and `x_out`, which are needed to store handles to the various extra inlets (resp. outlets) of the object.

5.2 signal-classes

```
void pan_tilde_setup(void) {
  pan_tilde_class = class_new(gensym("pan~"),
        (t_newmethod)pan_tilde_new,
        (t_method)pan_tilde_free,
        sizeof(t_pan_tilde),
        CLASS_DEFAULT,
        A_DEFFLOAT, 0);

  class_addmethod(pan_tilde_class,
        (t_method)pan_tilde_dsp, gensym("dsp"), 0);
  CLASS_MAINSIGNALIN(pan_tilde_class, t_pan_tilde, f);
}
```

Something has changed with the `class_new` function: the third argument specifies a "free-method" (aka *destructor*), which is called whenever an instance of the object is to be deleted (just like the "new-method" is called whenever an instance is to be created). In the prior examples this was set to `0` (meaning: we don't care), but in this example we have to clean up some ressources when we don't need them any more.

More interestingly, a method for signal-processing has to be provided by each signal class.

Whenever Pd's audio engine is started, a message with the selector "dsp" is sent to each object. Each class that has a method for the "dsp"-message is recognised as signal class.

Signal classes that want to provide signal-inlets have to declare this via the `CLASS_MAINSIGNALIN`-macro. This enables signals at the first (default) inlet. If more than one signal-inlet is needed, they have to be created explicitly in the constructor-method.

Inlets that are declared as signal-inlets cannot provide methods for `t_float`-messages any longer.

The first argument of the macro is a pointer to the signal class. The second argument is the type of the class's data space.

The last argument is a dummy-variable out of the data space that is needed to replace non-existing signal at the signal-inlet(s) with `t_float`-messages.

5.3 construction of signal-inlets and -outlets

```
void *pan_tilde_new(t_floatarg f)
{
  t_pan_tilde *x = (t_pan_tilde *)pd_new(pan_tilde_class);

  x->f_pan = f;

  x->x_in2 = inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
  x->x_in3 = floatinlet_new (&x->x_obj, &x->f_pan);

  x->x_out = outlet_new(&x->x_obj, &s_signal);

  return (void *)x;
}
```

Additional signal-inlets are added like other inlets with the routine `inlet_new`. The last two arguments are references to the symbolic selector "signal" in the lookup-table.

Signal-outlets are also created like normal (message-)outlets, by setting the outlet-selector to "signal".

The newly created inlets/outlets are "user-allocated" data. Pd will keep track of all the ressources it automatically creates (like the default inlet), and will eventually free these ressources once they are no longer needed. However, if we request an "extra" ressource (like the additional inlets/outlets in this example; or - more commonly - memory that is allocated via `malloc` or similar), we have to make sure ourselves, that these ressources are freed when no longer needed. If we fail to do so, we will invariably create a dreaded *memory leak*.

Therefore, we store the "handles" to the newly created inlets/outlets as returned by the `..._new` routines for later use.

5.4 DSP-methods

Whenever Pd's audio engine is turned on, all signal-objects declare their perform-routines that are to be added to the DSP-tree.

The "dsp"-method has two arguments, the pointer to the class-data space, and a pointer to an array of signals.

The signals are arranged in the array in such way, that they are ordered in a clockwise way in the graphical representation of the object.[4]

[4]If both left and right in- and out-signals exist, this means: First is the leftmost in-

```
void pan_tilde_dsp(t_pan_tilde *x, t_signal **sp)
{
  dsp_add(pan_tilde_perform, 5, x,
          sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec, sp[0]->s_n);
}
```

`dsp_add` adds a *perform*-routine (as declared in the first argument) to the DSP-tree.

The second argument is the number of the following pointers to diverse variables. Which pointers to which variables are passed is not limited.

Here, sp[0] is the first in-signal, sp[1] represents the second in-signal and sp[3] points to the out-signal.

The structure `t_signal` contains a pointer to the its signal-vector `().s_vec` (an array of samples of type `t_sample`), and the length of this signal-vector `().s_n`.

Since all signal vectors of a patch (not including it's sub-patches) are of the same length, it is sufficient to get the length of one of these vectors.

5.5 perform-routine

The perform-routine is the DSP-heart of each signal class.

A pointer to an integer-array is passed to it. This array contains the pointers, that were passed via `dsp_add`, which must be casted back to their real type.

The perform-routine has to return a pointer to integer, that points to the address behind the stored pointers of the routine. This means, that the return argument equals the argument of the perform-routine plus the number of pointer variables (as declared as the second argument of `dsp_add`) plus one.

```
t_int *pan_tilde_perform(t_int *w)
{
  t_pan_tilde *x = (t_pan_tilde *)(w[1]);
  t_sample  *in1 =    (t_sample *)(w[2]);
  t_sample  *in2 =    (t_sample *)(w[3]);
  t_sample  *out =    (t_sample *)(w[4]);
  int          n =           (int)(w[5]);

  t_sample f_pan = (x->f_pan<0)?0.0:(x->f_pan>1)?1.0:x->f_pan;
```

signal followed by the right in-signals; after the right out-signals, finally there comes the leftmost out-signal.

```
  while (n--) *out++ = (*in1++)*(1-f_pan)+(*in2++)*f_pan;

  return (w+6);
}
```

Each sample of the signal vectors is read and manipulated in the `while`-loop.

Optimisation of the DSP-tree tries to avoid unnecessary copy-operations. Therefore it is possible, that in- and out-signal are located at the same address in the memory. In this case, the programmer has to be careful not to write into the out-signal before having read the in-signal to avoid overwriting data that is not yet saved.

5.6 destructor

```
void pan_tilde_free(t_pan_tilde *x)
{
  inlet_free(x->x_in2);
  inlet_free(x->x_in3);
  outlet_free(x->x_out);
}
```

The object has some dynamically allocated ressources, namely the additional inlets and outlets we created in the constructor.

Since Pd doesn't track dynamically allocated ressources for us, we have to free them manually in the "free-method" (aka: destructor). We do so by calling `inlet_free` (resp. `outlet_free`) on the handles to our additional iolets.

Note that we do not need to free the default first outlet. As it is created automatically by Pd, it is also freed automatically.

5.7 the code: pan~

```
#include "m_pd.h"

static t_class *pan_tilde_class;

typedef struct _pan_tilde {
  t_object  x_obj;
  t_sample f_pan;
```

```
  t_sample f;

  t_inlet *x_in2;
  t_inlet *x_in3;
  t_outlet*x_out;
} t_pan_tilde;

t_int *pan_tilde_perform(t_int *w)
{
  t_pan_tilde *x = (t_pan_tilde *)(w[1]);
  t_sample  *in1 =    (t_sample *)(w[2]);
  t_sample  *in2 =    (t_sample *)(w[3]);
  t_sample  *out =    (t_sample *)(w[4]);
  int          n =           (int)(w[5]);
  t_sample f_pan = (x->f_pan<0)?0.0:(x->f_pan>1)?1.0:x->f_pan;

  while (n--) *out++ = (*in1++)*(1-f_pan)+(*in2++)*f_pan;

  return (w+6);
}

void pan_tilde_dsp(t_pan_tilde *x, t_signal **sp)
{
  dsp_add(pan_tilde_perform, 5, x,
          sp[0]->s_vec, sp[1]->s_vec, sp[2]->s_vec, sp[0]->s_n);
}

void pan_tilde_free(t_pan_tilde *x)
{
  inlet_free(x->x_in2);
  inlet_free(x->x_in3);
  outlet_free(x->x_out);
}

void *pan_tilde_new(t_floatarg f)
{
  t_pan_tilde *x = (t_pan_tilde *)pd_new(pan_tilde_class);

  x->f_pan = f;

  x->x_in2=inlet_new(&x->x_obj, &x->x_obj.ob_pd, &s_signal, &s_signal);
```

```
  x->x_in3=floatinlet_new (&x->x_obj, &x->f_pan);
  x->x_out=outlet_new(&x->x_obj, &s_signal);

  return (void *)x;
}

void pan_tilde_setup(void) {
  pan_tilde_class = class_new(gensym("pan~"),
        (t_newmethod)pan_tilde_new,
        0, sizeof(t_pan_tilde),
        CLASS_DEFAULT,
        A_DEFFLOAT, 0);

  class_addmethod(pan_tilde_class,
        (t_method)pan_tilde_dsp, gensym("dsp"), 0);
  CLASS_MAINSIGNALIN(pan_tilde_class, t_pan_tilde, f);
}
```

A Pd's message-system

Non-audio-data are distributed via a message-system. Each message consists of a "selector" and a list of atoms.

A.1 atoms

There are three kinds of atoms:

- *A_FLOAT*: a numerical value (floating point)
- *A_SYMBOL*: a symbolic value (string)
- *A_POINTER*: a pointer

Numerical values are always floating point-values (`t_float`), even if they could be displayed as integer values.

Each symbol is stored in a lookup-table for reasons of performance. The command `gensym` looks up a string in the lookup-table and returns the address of the symbol. If the string is not yet to be found in the table, a new symbol is added.

Atoms of type *A_POINTER* are not very important (for simple externals).

The type of an atom `a` is stored in the structure-element `a.a_type`.

A.2 selectors

The selector is a symbol that defines the type of a message. There are five predefined selectors:

- "`bang`" labels a trigger event. A "bang"-message consists only of the selector and contains no lists of atoms.
- "`float`" labels a numerical value. The list of a "float"-Message contains one single atom of type *A_FLOAT*
- "`symbol`" labels a symbolic value. The list of a "symbol"-Message contains one single atom of type *A_SYMBOL*
- "`pointer`" labels a pointer value. The list of a "pointer"-Message contains one single atom of type *A_POINTER*
- "`list`" labels a list of one or more atoms of arbitrary type.

Since the symbols for these selectors are used quite often, their address in the lookup-table can be queried directly, without having to use `gensym`:

selector	lookup-routine	lookup-address
`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_symbol`

Other selectors can be used as well. The receiving class has to provide a method for a specifique selector or for "anything", which is any arbitrary selector.

Messages that have no explicit selector and start with a numerical value, are recognised automatically either as "float"-message (only one atom) or as "list"-message (several atoms).

For example, messages "`12.429`" and "`float 12.429`" are identical. Likewise, the messages "`list 1 for you`" is identical to "`1 for you`".

B Pd-types

Since Pd is used on several platforms, many ordinary types of variables, like `int`, are re-defined. To write portable code, it is reasonable to use types provided by Pd.

Apart from this there are many predefined types, that should make the life of the programmer simpler.

Generally, Pd-types start with `t_`.

Pd-type	description
`t_atom`	atom
`t_float`	floating point value
`t_symbol`	symbol
`t_gpointer`	pointer (to graphical objects)
`t_int`	pointer-sized integer value (only used for perform routines)
`t_signal`	structure of a signal
`t_sample`	audio signal-value (floating point)
`t_outlet`	outlet of an object
`t_inlet`	inlet of an object
`t_object`	object-interna
`t_class`	a Pd-class
`t_method`	class-method
`t_newmethod`	pointer to a constructor (new-routine)

C important functions in "m_pd.h"

C.1 functions: atoms

C.1.1 SETFLOAT

```
SETFLOAT(atom, f)
```

This macro sets the type of `atom` to `A_FLOAT` and stores the numerical value `f` in this atom.

C.1.2 SETSYMBOL

```
SETSYMBOL(atom, s)
```

This macro sets the type of `atom` to `A_SYMBOL` and stores the symbolic pointer `s` in this atom.

C.1.3 SETPOINTER

```
SETPOINTER(atom, pt)
```

This macro sets the type of `atom` to `A_POINTER` and stores the pointer `pt` in this atom.

C.1.4 atom_getfloat

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

If the type of the atom `a` is `A_FLOAT`, the numerical value of this atom else "0.0" is returned.

C.1.5 atom_getfloatarg

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

If the type of the atom – that is found at in the atom-list `argv` with the length `argc` at the place `which` – is `A_FLOAT`, the numerical value of this atom else "0.0" is returned.

C.1.6 atom_getint

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

If the type of the atom `a` is `A_FLOAT`, its numerical value is returned as integer else "0" is returned.

C.1.7 atom_getsymbol

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

If the type of the atom `a` is `A_SYMBOL`, a pointer to this symbol is returned, else a null-pointer "0" is returned.

C.1.8 atom_gensym

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

If the type of the atom `a` is `A_SYMBOL`, a pointer to this symbol is returned.

Atoms of a different type, are "reasonably" converted into a string. This string is – on demand – inserted into the symbol-table. A pointer to this symbol is returned.

C.1.9 atom_string

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

Converts an atom `a` into a `C`-string `buf`. The memory to this char-Buffer has to be reserved manually and its length has to be declared in `bufsize`.

C.1.10 gensym

```
t_symbol *gensym(char *s);
```

Checks, whether the C-string `*s` has already been inserted into the symbol-table. If no entry exists, it is created. A pointer to the symbol is returned.

C.2 functions: classes

C.2.1 class_new

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

Generates a class with the symbolic name `name`. `newmethod` is the constructor that creates an instance of the class and returns a pointer to this instance.

If memory is reserved dynamically, this memory has to be freed by the destructor-method `freemethod` (without any return argument), when the object is destroyed.

`size` is the static size of the class-data space, that is returned by `sizeof(t_mydata)`.

`flags` define the presentation of the graphical object. A (more or less arbitrary) combination of following objects is possible:

flag	description
`CLASS_DEFAULT`	a normal object with one inlet
`CLASS_PD`	*object (without graphical presentation)*
`CLASS_GOBJ`	*pure graphical object (like arrays, graphs,...)*
`CLASS_PATCHABLE`	*a normal object (with one inlet)*
`CLASS_NOINLET`	the default inlet is suppressed

Flags the description of which is printed in *italic* are of small importance for writing externals.

The remaining arguments `arg1,...` define the types of object-arguments passed at the creation of a class-object. A maximum of six type checked arguments can be passed to an object. The list of argument-types are terminated by "0".

Possible types of arguments are:

`A_DEFFLOAT`	a numerical value
`A_DEFSYMBOL`	a symbolical value
`A_GIMME`	a list of atoms of arbitrary length and types

If more than six arguments are to be passed, `A_GIMME` has to be used and a manual type-check has to be made.

C.2.2 class_addmethod

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

Adds a method `fn` for a selector `sel` to a class `c`.

The remaining arguments `arg1,...` define the types of the list of atoms that follow the selector. A maximum of six type-checked arguments can be passed. If more than six arguments are to be passed, `A_GIMME` has to be used and a manual type-check has to be made.

The list of arguments is terminated by "0".

Possible types of arguments are:

`A_DEFFLOAT`	a numerical value
`A_DEFSYMBOL`	a symbolical value
`A_POINTER`	a pointer
`A_GIMME`	a list of atoms of arbitrary length and types

C.2.3 class_addbang

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

Adds a method `fn` for "bang"-messages to the class `c`.

The argument of the "bang"-method is a pointer to the class-data space:

```
void my_bang_method(t_mydata *x);
```

C.2.4 class_addfloat

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

Adds a method `fn` for "float"-messages to the class `c`.

The arguments of the "float"-method is a pointer to the class-data space and a floating point-argument:

```
void my_float_method(t_mydata *x, t_floatarg f);
```

C.2.5 class_addsymbol

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

Adds a method `fn` for "symbol"-messages to the class `c`.

The arguments of the "symbol"-method is a pointer to the class-data space and a pointer to the passed symbol:

```
void my_symbol_method(t_mydata *x, t_symbol *s);
```

C.2.6 class_addpointer

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

Adds a method `fn` for "pointer"-messages to the class `c`.

The arguments of the "pointer"-method is a pointer to the class-data space and a pointer to a pointer:

```
void my_pointer_method(t_mydata *x, t_gpointer *pt);
```

C.2.7 class_addlist

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

Adds a method `fn` for "list"-messages to the class `c`.

The arguments of the "list"-method are – apart from a pointer to the class-data space – a pointer to the selector-symbol (always `&s_list`), the number of atoms and a pointer to the list of atoms:

```
void my_list_method(t_mydata *x,
   t_symbol *s, int argc, t_atom *argv);
```

C.2.8 class_addanything

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

Adds a method `fn` for an arbitrary message to the class `c`.

The arguments of the anything-method are – apart from a pointer to the class-data space – a pointer to the selector-symbol, the number of atoms and a pointer to the list of atoms:

```
void my_any_method(t_mydata *x,
   t_symbol *s, int argc, t_atom *argv);
```

C.2.9 class_addcreator

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

Adds a creator-symbol `s`, alternative to the symbolic class name, to the constructor `newmethod`. Thus, objects can be created either by their "real" class name or an alias-name (p.e. an abbreviation, like the internal "float" resp. "f").

The "0"-terminated list of types corresponds to that of `class_new`.

C.2.10 class_sethelpsymbol

```
void class_sethelpsymbol(t_class *c, t_symbol *s);
```

If a Pd-object is right-clicked, a help-patch for the corresponding object class can be opened. By default this is a patch with the symbolic class name in the directory "*doc/5.reference/*".

The name of the help-patch for the class that is pointed to by `c` is changed to the symbol `s`.

Therefore, several similar classes can share a single help-patch.

Path-information is relative to the default help path *doc/5.reference/*.

C.2.11 pd_new

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

Generates a new instance of the class `cls` and returns a pointer to this instance.

C.3 functions: inlets and outlets

All routines for inlets and outlets need a reference to the object-interna of the class-instance. When instantiating a new object, the necessary data space-variable of the `t_object`-type is initialised. This variable has to be passed as the `owner`-object to the various inlet- and outlet-routines.

C.3.1 inlet_new

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

Generates an additional "active" inlet for the object that is pointed at by `owner`. Generally, `dest` points at "`owner.ob_pd`".

The selector `s1` at the new inlet is substituted by the selector `s2`.

If a message with selector `s1` appears at the new inlet, the class-method for the selector `s2` is called.

This means

- The substituting selector has to be declared by `class_addmethod` in the setup-routine.
- It is possible to simulate a certain right inlet, by sending a message with this inlet's selector to the leftmost inlet.

  Using an empty symbol (`gensym("")`) as selector makes it impossible to address a right inlet via the leftmost one.
- It is not possible to add methods for more than one selector to a right inlet. Particularly it is not possible to add a universal method for arbitrary selectors to a right inlet.

C.3.2 floatinlet_new

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

Generates a new "passive" inlet for the object that is pointed at by `owner`. This inlet enables numerical values to be written directly into the memory `fp`, without calling a dedicated method.

C.3.3 symbolinlet_new

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

Generates a new "passive" inlet for the object that is pointed at by `owner`. This inlet enables symbolic values to be written directly into the memory `*sp`, without calling a dedicated method.

C.3.4 pointerinlet_new

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

Generates a new "passive" inlet for the object that is pointed at by `owner`. This inlet enables pointer to be written directly into the memory `gp`, without calling a dedicated method.

C.3.5 outlet_new

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

Generates a new outlet for the object that is pointed at by `owner`. The Symbol `s` indicates the type of the outlet.

symbol	symbol-address	outlet-type
"bang"	`&s_bang`	message (bang)
"float"	`&s_float`	message (float)
"symbol"	`&s_symbol`	message (symbol)
"pointer"	`&s_gpointer`	message (pointer)
"list"	`&s_list`	message (list)
—	0	message
"signal"	`&s_signal`	signal

There are no real differences between outlets of the various message-types. At any rate, it makes code more easily readable, if the use of outlet is shown at creation-time. For outlets for any messages a null-pointer is used. Signal-outlet must be declared with `&s_signal`.

Variables if the type `t_object` provide pointer to one outlet. Whenever a new outlet is generated, its address is stored in the object variable `(*owner).ob_outlet`.

If more than one message-outlet is needed, the outlet-pointers that are returned by `outlet_new` have to be stored manually in the data space to address the given outlets.

C.3.6 outlet_bang

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

Outputs a "bang"-message at the outlet specified by `x`.

C.3.7 outlet_float

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

Outputs a "float"-message with the numeric value `f` at the outlet specified by `x`.

C.3.8 outlet_symbol

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

Outputs a "symbol"-message with the symbolic value `s` at the outlet specified by `x`.

C.3.9 outlet_pointer

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

Outputs a "pointer"-message with the pointer `gp` at the outlet specified by `x`.

C.3.10 outlet_list

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

Outputs a "list"-message at the outlet specified by `x`. The list contains `argc` atoms. `argv` points to the first element of the atom-list.

Independent of the symbol `s`, the selector "list" will precede the list.

To make the code more readable, `s` should point to the symbol list (either via `gensym("list")` or via `&s_list`)

C.3.11 outlet_anything

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

Outputs a message at the outlet specified by `x`.

The message-selector is specified with `s`. It is followed by `argc` atoms. `argv` points to the first element of the atom-list.

C.4 functions: DSP

If a class should provide methods for digital signal-processing, a method for the selector "dsp" (followed by no atoms) has to be added to this class

Whenever Pd's audio engine is started, all objects that provide a "dsp"-method are identified as instances of signal classes.

DSP-method

```
void my_dsp_method(t_mydata *x, t_signal **sp)
```

In the "dsp"-method a class method for signal-processing is added to the DSP-tree by the function `dsp_add`.

Apart from the data space `x` of the object, an array of signals is passed. The signals in the array are arranged in such a way, that they can be read in the graphical representation of the object clockwisely.

In case there are both two in- and out-signals, this means:

pointer	to signal
sp[0]	left in-signal
sp[1]	right in-signal
sp[2]	right out-signal
sp[3]	left out-signal

The signal structure contains apart from other things:

structure-element	description
`s_n`	length of the signal vector
`s_vec`	pointer to the signal vector

The signal vector is an array of samples of type `t_sample`.

perform-routine

```
t_int *my_perform_routine(t_int *w)
```

A pointer `w` to an array (of integer) is passed to the perform-routine that is inserted into the DSP-tree by `class_add`.

In this array the pointers that are passed via `dsp_add` are stored. These pointers have to be casted back to their original type.

The first pointer is stored at `w[1]` !!!

The perform-routine has to return a pointer to integer, that points directly behind the memory, where the object's pointers are stored. This means, that the return-argument equals the routine's argument `w` plus the number of used pointers (as defined in the second argument of `dsp_add`) plus one.

C.4.1 CLASS_MAINSIGNALIN

```
CLASS_MAINSIGNALIN(<class_name>, <class_data>, <f>);
```

The macro `CLASS_MAINSIGNALIN` declares, that the class will use signal-inlets.

The first macro-argument is a pointer to the signal-class. The second argument is the type of the class-data space. The third argument is a (dummy-)floating point-variable of the data space, that is needed to automatically convert "float"-messages into signals if no signal is present at the signal-inlet.

No "float"-methods can be used for signal-inlets, that are created this way.

C.4.2 dsp_add

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

Adds the perform-routine `f` to the DSP-tree. The perform-routine is called at each DSP-cycle.

The second argument`n` defines the number of following pointer-arguments

Which pointers to which data are passes is not limited. Generally, pointers to the data space of the object and to the signal-vectors are reasonable. The length of the signal-vectors should also be passed to manipulate signals effectively.

C.4.3 sys_getsr

```
float sys_getsr(void);
```

Returns the sampler ate of the system.

C.5 functions: memory

C.5.1 getbytes

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

Reserves `nbytes` bytes and returns a pointer to the allocated memory.

C.5.2 copybytes

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

Copies `nbytes` bytes from `*src` into a newly allocated memory. The address of this memory is returned.

C.5.3 freebytes

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

Frees `nbytes` bytes at address `*x`.

C.6 functions: output

C.6.1 post

```
void post(char *fmt, ...);
```

Writes a `C`-string to the standard error (shell).

C.6.2 error

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

Writes a `C`-string as an error-message to the standard error (shell).

The object that has output the error-message is marked and can be identified via the Pd-menu *Find->Find last error*.