Life after Pluggo

Getting started programming plugins in C++ using IPlug

Oli Larkin
contact@olilarkin.co.uk

What are the options for building plugins?

High Level Options

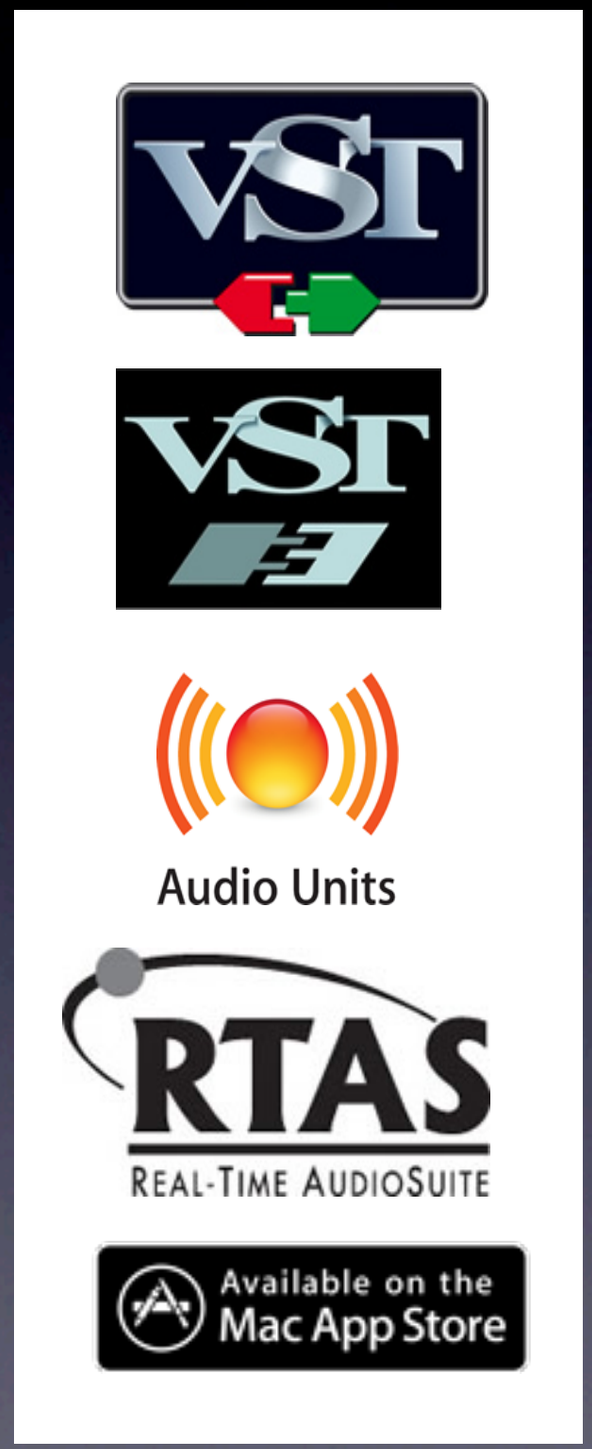
- Max4 / Pluggo (VST/AU/RTAS)
- Synthedit (Windows VST)
- Synthmaker (Windows VST/Standalone)
- Sonicbirth (Mac VST/AU)
- Plugtastic / Jamoma (VST/AU)

Why go Low Level?

- easier to manage projects long term
- control over finer details / polish
- better performance?
- profit?
- fun?
- better job prospects?

Ideally...

- Only code it once
- Multi platform (win/mac/linux)
- Target all common plugin formats
- 32 & 64 bit binaries
- Work consistently & reliably in all common plugin hosts
- Support the majority of API features
- Make standalone app versions



Low Level Options

- Three parts to think about

DSP

GUI & OS

APIs

OPTION #0

- Build everything using different APIs (VST2, AU, RTAS)
- Shared, separated c++ class for the DSP
- VSTGUI or other framework for graphics

OPTION #1

- Build with VST2 SDK, wrap to AudioUnit with VSTAU or Symbiosis
- VSTGUI or other framework for graphics

OPTION #2

- Build using the VST3 SDK, which includes VST2 and AU wrappers
- VSTGUI or other framework for graphics

OPTION #3

- Build with JUCE
  - targets VST2, AU, RTAS, Standalone
  - includes GUI toolkit
- GPL... free to use for open source - £399 for a single closed source product, £799 for multiple

other options

- LibNUI
- Infinity API
- DestroyFX plugin library
- VST.NET
- jVSTwRapper
- ?

other gui options

- Instead of VSTGUI you can use other GUI frameworks e.g.
  - OpenGL, wxWidgets, QT, Win32, Cocoa
- But a lot more work is involved

or use IPlug

- Originally developed by John Schwartz (Schwa) ~2007
- Part of Cockos' WDL C++ code library
- Free for closed source
- Now several modified versions online, including WDL-OL

PITCH
TONE
TUNING
VIBRATO
SPEED
DEPTH
VOLUME
ADJUST
SWELL
KEY CONTACTS
ATTACK
RELEASE
MIDI
CHANNEL
M.T. BOOSTER
KNEE LEVER
PEDAL
KIRNU
GLOBAL SETTINGS
Rate
Type
Direction
Hold
Velocity Mode
Scale
Key Mode
STEP
Sort order
Custom
Gate
Swing
Note
Side
SPLIT
SCALE SETTINGS
Octaves
Transpose
FREE VERSION
PATTERN
RESET
On new note
Note list
On beat
Beat
Steps
LENGTH
GATE
ACCENT
TRANSP
DIV
SHIFT
DATA
CC1
CC2
Arto Vaarala KIRNU
мод
мешалка
фильтр
вибратор
частота
тремоло
вибрато
STILLWELL
THRESHOLD
ATTACK
RELEASE
RATIO
IMPETUS
The Rocket
COMPRESSOR
PARALLEL COMPRESSION
DETECTOR HPF
VU MODE
COMPENSATION
INPUT
OUTPUT / GR
OUT
ALL
schOPE

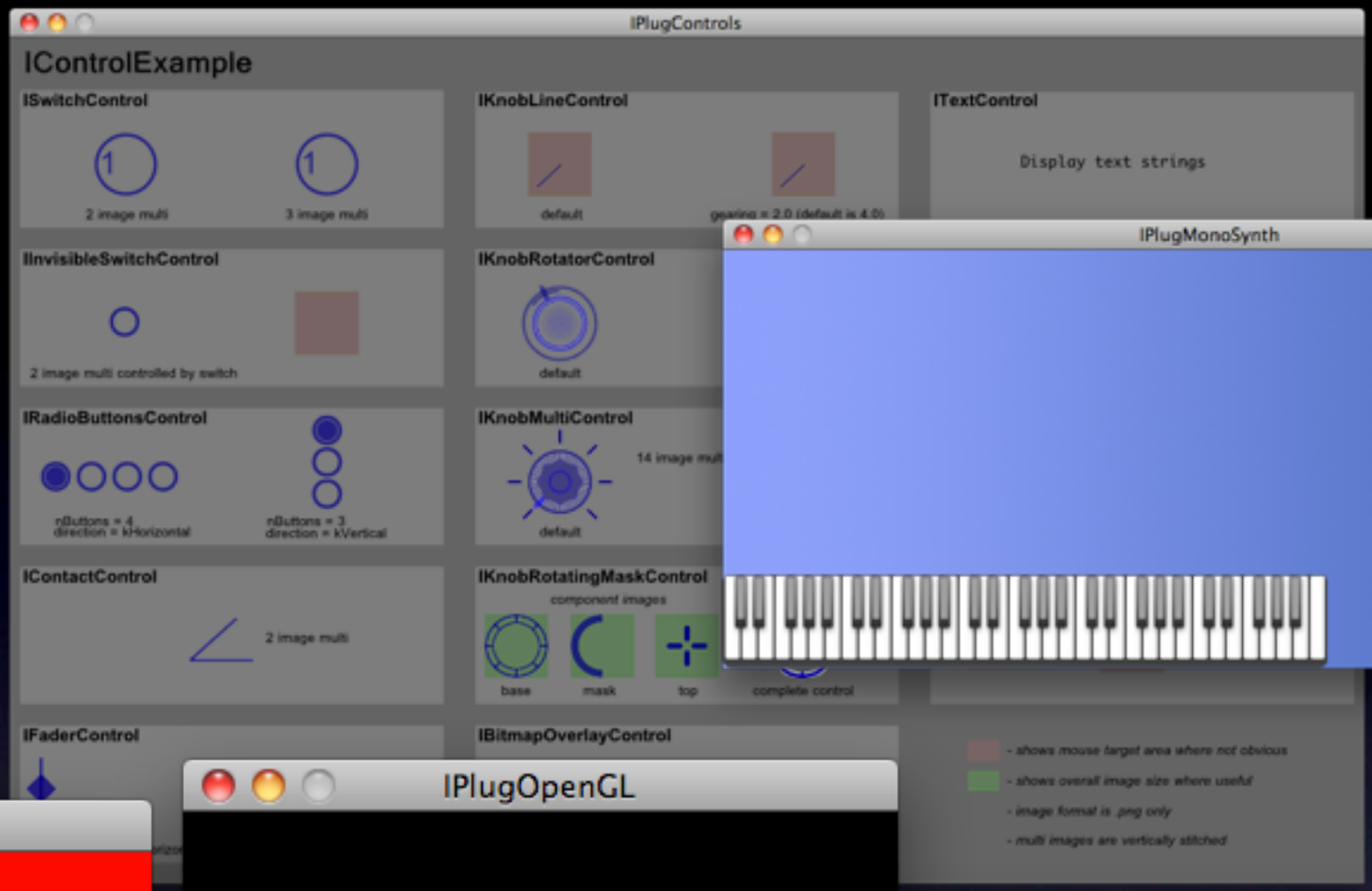
Features

- Targets multiple plugin APIs from the same code
- Includes a GUI framework with lots of common controls (dials, sliders, buttons, switches)
- Drawing and image compositing is done by Cockos' LICE library
- VERY simple to implement a plugin, with much less code than other options

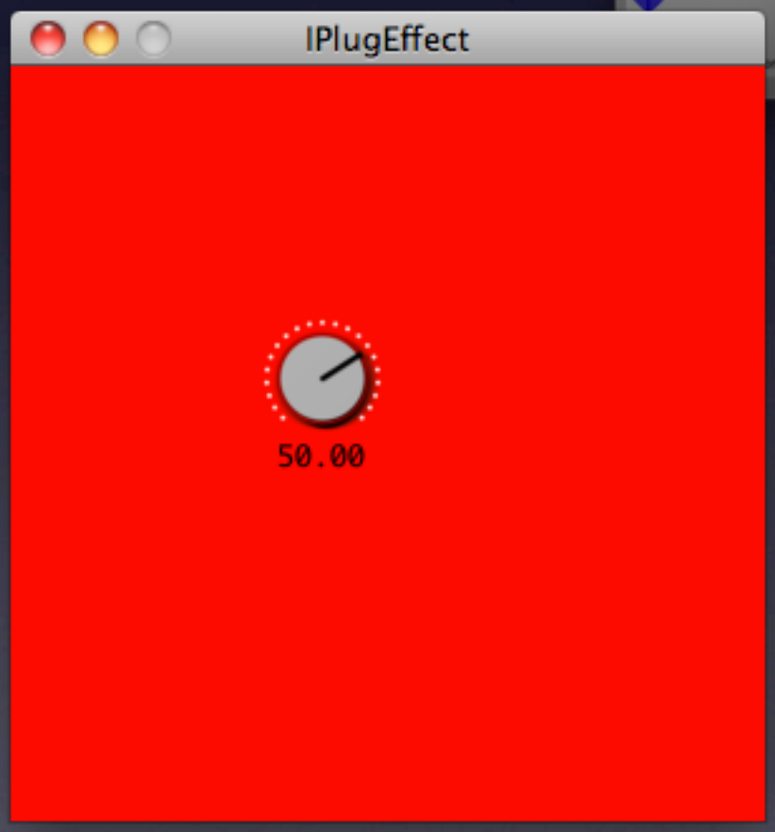
WDL-OL IPlug

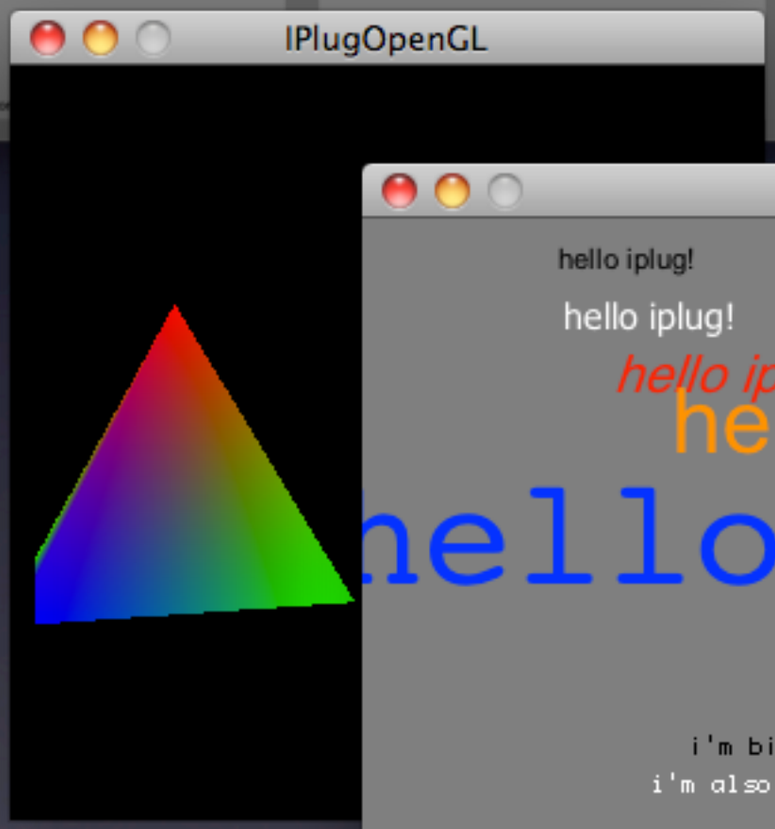
- Targets VST2, VST3, AU, RTAS, Standalone apps and also IOS apps
- New IDE projects and structure
- Works with 32/64 bit “out of the box”
- Many enhancements
- More examples
- Helper scripts
- Build-chain including installer scripts


Carrier
9:58 PM
GainL
-12.0dB
GainR
-12.0dB
Mode
a
Play Midi Note


IPlugControls
IControlExample
ISwitchControl
2 image multi
3 image multi
IKnobLineControl
default
ITextControl
Display text strings
IInvisibleSwitchControl
2 image multi controlled by switch
IKnobRotatorControl
default
IRadioButtonsControl
nButtons = 4
direction = kHorizontal
nButtons = 3
direction = kVertical
IKnobMultiControl
default
IContactControl
2 image multi
IKnobRotatingMaskControl
component images
base
mask
top
complete control
IFaderControl
IBitmapOverlayControl
- shows overall image size where useful
- image format is .png only
- multi images are vertically stitched

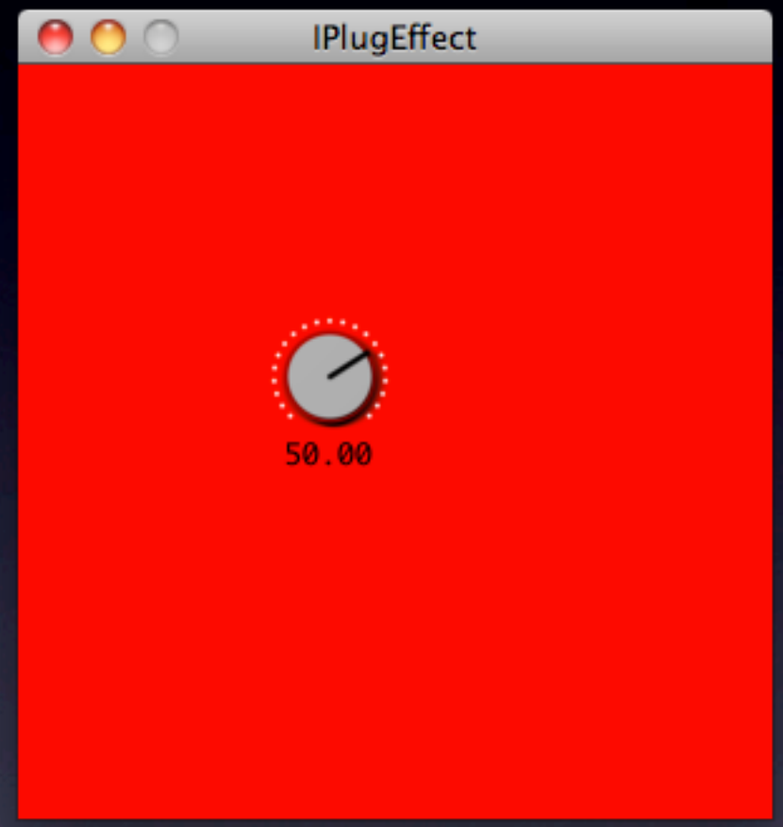
IPlugMonoSynth
IPlug Example


IPlugEffect
50.00


IPlugOpenGL


IPlugText
hello iplug!
hello iplug!
hello iplug!
hello iplug!
hello iplug!
i'm bitmap monospace text
i'm also bitmap monospace text

```cpp

#include "MyPlugin.h" // Plugin header, class interface
#include "IPlug_include_in_plug_src.h"
#include "resource.h" // Define resources and channel count etc
#include "IControl.h" // library of IPlug gui controls

MyPlugin::MyPlugin(IPlugInstanceInfo instanceInfo)
: IPLUG_CTOR(kNumParams, kNumPrograms, instanceInfo), mGain(1.)
{
  // add a parameter with type, name, default, min, max, step, unit label
  GetParam(kGain)->InitDouble("Gain", 50., 0., 100.0, 0.01, "%");

  // create graphics context
  IGraphics* pGraphics = MakeGraphics(this, kWidth, kHeight);

  // add a colored background
  pGraphics->AttachPanelBackground(&COLOR_RED);

  // add a PNG image-based knob control, link to parameter and position on gui
  IBitmap knob = pGraphics->LoadIBitmap(KNOB_ID, KNOB_FN, kKnobFrames);
  pGraphics->AttachControl(new IKnobMultiControl(this, kGainX, kGainY, kGain, &knob));

  // attach graphics context
  AttachGraphics(pGraphics);

  // create factory presets
  MakeDefaultPreset((char *) "-", kNumPrograms);
}

// process a block of audio
void MyPlugin::ProcessDoubleReplacing(double** inputs, double** outputs, int nFrames)
{
  double* in1 = inputs[0];
  double* out1 = outputs[0];

  // process each sample
  for (int s = 0; s < nFrames; s++)
  {
    out1[s] = in1[s] * mGain;
  }
}

// handle parameter changes
void MyPlugin::OnParamChange(int paramIdx)
{
  IMutexLock lock(this);

  switch (paramIdx)
  {
    case kGain:
      mGain = GetParam(kGain)->Value() / 100.;
      break;
    default:
      break;
  }
}

```



WDL-OL IPlug project folder structure

- On Mac open PluginName.xcodeproj
- On Windows open PluginName.sln

Parts of an IPlug-Plugin

- 3 main source code files you need to edit...
  - resource.h
  - MyPlugin.h
  - MyPlugin.cpp

resource.h

- specifies various properties of the plugin
  - plugin name, manufacturer name
  - valid channel i/o / side-chain
  - plugin unique ID, manufacturer unique ID
  - version number
  - plugin type: instrument/effect/midi-controlled effect
  - plugin category (VST3/RTAS)
  - plugin does chunks or not
- specifies resource IDs and paths for each unique PNG image you need to use in a control

plugin interface (.h)

- *interface* of your plugin class
  - declare *member variables* here
  - specify which *methods* from IPlugBase you want to *override*
  - at least...
    ProcessDoubleReplacing() and
    OnParamChange()
  - probably also...
    Reset() and
    ProcessMidiMsg(IMidiMsg* pMsg)

implementation (.cpp)

- *implementation* of your plugin class
  - **plugin constructor**
    - initialise *member variables*
    - add plugin parameters
    - create gui and attach controls
    - create factory presets
  - **plugin destructor**
    - free any memory that your plugin allocated
- ...

ProcessDoubleReplacing()

- Called by the plugin host every time it needs to process a block of samples
- Passes *pointers* to buffers of double precision samples for each input and output channel
- IPlug guarantees valid buffers, just full of zeros if the host hasn't connected them

OnParamChange()

- Called for each parameter change
  - whenever a parameter is automated
  - when you edit a parameter via the GUI
  - when you recall a preset

Reset()

- Called when the processing is started, or restarted, e.g. on change of samplerate
- use **GetSampleRate()** to check the new sample rate and update your DSP accordingly

ProcessMidiMsg()

- This is called whenever a MIDI event is received
- MIDI events are sample-accurate and are timestamped with the *sample offset* into the current block at which the event should occur

Chunks?

- If a plugin needs to store arbitrary data in its state or presets, e.g. a string of text representing the location of an audio file on the hard disk, you need to support chunks.
- Otherwise each parameter value is stored as a single floating point number

Defining Parameters

- Each parameter must have an index (starting from 0). The EParams enum is used to enumerate the parameters and give them individual IDs ... e.g. *kGain*
- In the plugin constructor add parameters with **GetParam(idx)->init...** like this...

```
// add a parameter with type, name, default, min, max, step, unit label
GetParam(kGain)->InitDouble("Gain", 50., 0., 100.0, 0.01, "%");
```

4 different kinds of IPlug parameter

Double, Enum, Int, Bool

```
// add a floating point, continuous parameter,
//                          name,   default,  min,  max,    step, unit label
GetParam(kGain)->InitDouble("Gain", 50.,      0.,   100.0,  0.01, "%");

// add an enumerated list, discrete parameter,
//                         name,   default,  numitems
GetParam(kMode)->InitEnum("Mode", 0,        4);

GetParam(kMode)->SetDisplayText(0, "a"); // set display text for first item to "a"
GetParam(kMode)->SetDisplayText(1, "b"); // ...
GetParam(kMode)->SetDisplayText(2, "c"); // ...
GetParam(kMode)->SetDisplayText(3, "d"); // ...

// add a boolean, discreet parameter,
//                           name,     default,
GetParam(kEnable)->InitBool("Enable", 0);

// add an integer, discreet parameter,
//                         name,      default,  min,  max,  unit label
GetParam(kGain)->InitInt("Count",   50,       0,    100,  "%");
```

Handling Parameter Changes

```
    void ParameterTypes::OnParamChange(int paramIdx)
{
  IMutexLock lock(this);

  switch (paramIdx)
  {
    case kGain:
      DBGMSG( "value of gain : %f\n", GetParam(kGain)->Value() );
      break;
    case kMode:
      DBGMSG( "value of mode : %i\n", GetParam(kMode)->Int() );
      break;
    case kEnable:
      DBGMSG( "value of enable : %i\n", GetParam(kEnable)->Bool() );
      break;
    case kCount:
      DBGMSG( "value of count : %i\n", GetParam(kCount)->Int() );
      break;
    default:
      break;
  }
}
```

Processing Audio

- you can call **IsInChannelConnected(ch)** and **IsOutChannelConnected(ch)** to find out what channel config is used
- this is where your CPU will get taxed

```
void MyPlugin::ProcessDoubleReplacing(double** inputs, double** outputs, int nFrames)
{
  // block rate ... many calculations can be done here to save cpu cycles
  double* in1 = inputs[0];
  double* in2 = inputs[1];
  double* out1 = outputs[0];
  double* out2 = outputs[1];

  // process each sample in the buffers
  for (int sampleIdx = 0; sampleIdx < nFrames; sampleIdx++)
  {
    // sample rate ... code in here should be as fast as possible!
    out1[sampleIdx] = in1[sampleIdx] * mGain;
    out2[sampleIdx] = in2[sampleIdx] * mGain;
  }
}
```

Handling MIDI

- You can use an IMidiQueue (helper class by Theo Niessink) to queue all the time-stamped messages that arrive via **ProcessMidiMsg()** and handle them sample-accurately when you process the corresponding sample in **ProcessDoubleReplacing()**

Attaching a GUI

- Add PNG resource info to resource.h & *myplugin.rc* on windows
- Add PNG resources to your xcode targets
- Create graphics context and attach controls, linked to parameter IDs in plugin constructor

```
// create graphics context
IGraphics* pGraphics = MakeGraphics(this, kWidth, kHeight);

// add a coloured background
pGraphics->AttachPanelBackground(&COLOR_RED);

// add a PNG image-based knob control, link to parameter and position on gui
IBitmap knob = pGraphics->LoadIBitmap(KNOB_ID, KNOB_FN, kKnobFrames);
pGraphics->AttachControl(new IKnobMultiControl(this, kGainX, kGainY, kGain, &knob));

// attach graphics context
AttachGraphics(pGraphics);
```

IControls

- See IControl.h for the standard IControls
- Many take an IRECT struct to specify their location on the GUI, others take just x, y coordinates

Creating Factory Presets

- At the end of the plugin constructor, there are several ways to create presets...

```
MakeDefaultPreset() // fills kNumPrograms with “blank” presets based on the default parameter values
MakePreset() // create a preset with raw float values
MakePresetFromNamedParams() // create a preset with param index, value pairs

MakePresetFromChunk() // fill a preset with binary data (if you are using chunks)
MakePresetFromBlob() // fill a preset with bin64 encoded binary data (if you are using chunks)
```

Handling Tempo Info

- At block rate (i.e. outside the sample loop in ProcessDoubleReplacing())
  - **GetTempo()**
  - **GetSamplePos()**
  - **GetSamplesPerBeat()**
  - **GetTime(ITimeInfo* pTimeInfo)**

C++ intro/recap

Source code

- C++ *source code* files have the extensions .cpp and .h
- You can use the *preprocessor* #include statement to import other source code files into your source code in order to make that functionality available to your program

Compiling

- The *compiler* converts each C++ *source code* file in your project into low level instructions (object files)
- The *linker* links the object files, system *libraries* and any third party *libraries* specified
- The output is usually either an *executable*, a *static library* or a *dynamic library*

Static vs Dynamic Libraries

- *Static libraries* are linked at *compile time*
- *Dynamic libraries* are linked at *run time*
- *Plugins* are usually *Dynamic libraries*

Debug/Release

- A *debug* build is used during *development* to test the program functionality
- A *release* build is used for *deployment* when the program is finished
- *Release* builds are optimised

C++ Language

- semicolons delimitate statements;
- certain *keywords* are reserved
- memory management is the main cause of bugs
- C++ is *strongly typed*

Variables

- allocate part of the computers memory to store something
- have a *type*, which determines the number of bytes used and a *name*
- integer variables can be *signed* or *unsigned*
- must not be used without being *initialised* first

type	**size**	**range**
char	8 bits	0-255
short int	16 bits	0-65536
long int	32 bits	0-(2^32)
float	32 bits	-
double	64 bits	-
custom	?	-

```
int a = 100;
double b = 0.5;
```

Functions

- take *arguments*
- have a return *type*
- if they don't return anything, the return type is *void*

```
int add(int a, int b)
{
  return a + b;
}

void myFunction()
{
  // do something
}
```

Conditional operations

- Used to execute different code depending on a condition
- *if statement*
- *switch statement*

```
if(a == 100)
{
  printf("a equals 100");
}
else
{
  printf("a doesn't equal 100");
}
```

```
switch (a)
{
  case 100:
    printf("a equals 100");
    break;
  default:
    printf("a doesn't equal 100");
    break;
}
```

Loops

- execute code multiple times

- *for loop*
- *while loop*
- *do while loop*

```
int nFrames = 128;

for (int s = 0; s < nFrames; s++)
{
  printf("s = %i\n", s);
}
```

```
while (nFrames--)
{
  printf("s = %i\n", s);
}
```

Stack vs Heap

- Memory can either be allocated on the *Stack* or the *Heap*
- *Heap* memory is allocated *dynamically*
- the *Heap* is used for storing larger data, the *Stack* is quicker and usually used for *local* variables
- memory *allocated* on the *Heap* must be *deallocated*!

```
int myStackVar;
int *myHeapVar = new int();
delete myHeapVar;
```

Arrays

- Static declaration, on the stack
  - int myArray[512];
- Dynamic declaration on the heap
  - int *myArray = new int[512];
  - must remember to free the memory else we have a *memory leak*!
    - delete [] myArray;

- In both cases the array values need to be initialised before they are used!

Arrays 2

- setting values:
  - myArray[0] = 100;
- getting values
  - int valueAtIndexZero = myArray[0];

Pointers

- Each variable is located somewhere in the computer's memory indexable in bytes
- That location is referred to as the *address*
- An *address* is a 32 or 64 bit integer
- A *pointer* is a kind of variable that holds an *address* and specifies the type (and therefore the size) of the data stored at that *address*

Arrays 3

- Arrays are often accessed using pointer syntax, where the name of the array is a pointer to the first element of the array in memory
- set values at array index via *pointer arithmetic*
  - *(myArray + i) = 100;
- to get values we can *dereference* a pointer
  - int valueAtIndexZero = *myArray;

Structs & Classes

- Group variables and functions into *objects*
  - *object's* variables are called *member variables*
  - *object's* functions are called *methods*
- In a *Class* the keywords *public*, *private* and *protected* limit access to *members* of an object
- A *Struct* is essentially the same as a *Class* except all the members are *public*

Scope

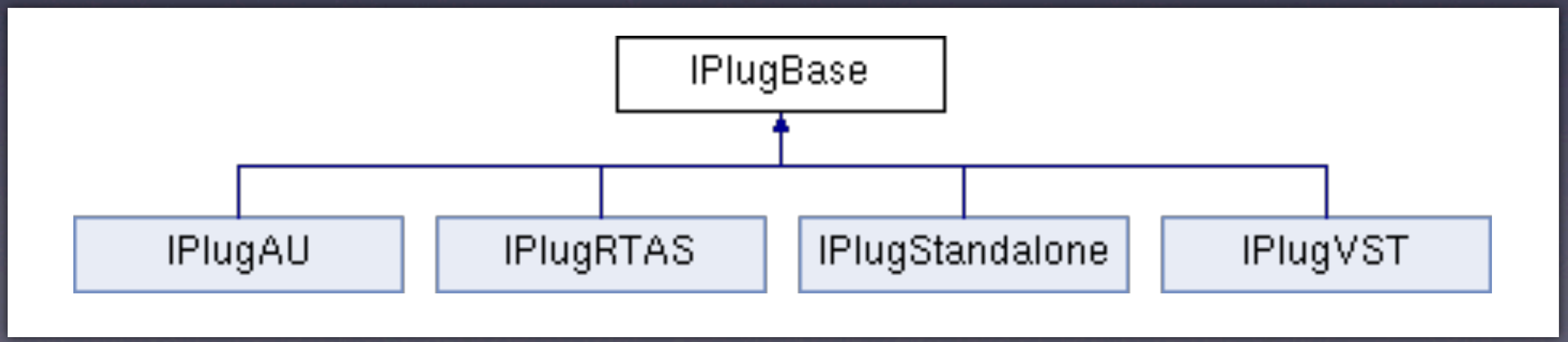
- a variable has *scope* which means the part/s of the code/program in which it is valid
- variables declared at *global scope* are *in scope* at any point in the program
- Class *member variables* are *in scope* inside that Class' methods
- variables declared in a function are *in scope* until that function ends *(local scope)*

Inheritance

- C++ is an object-oriented language
- Classes can *inherit* from *base classes*, obtaining the functionality of the base class, adding new functionality and customising certain functionality

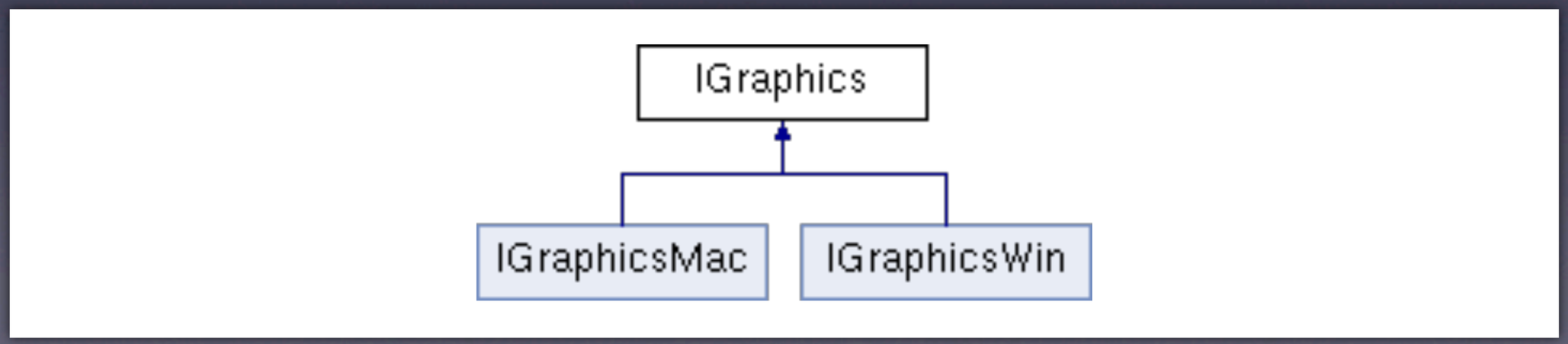
IPlug Class inheritance

- When you create your plugin's class its base class is *IPlug*
- *IPlug* is actually either *IPlugAU*, *IPlugVST* etc, depending on the *preprocessor* macro definitions, so inherits from one of those classes
- The base class of *IPlugAU* etc is IPlugBase



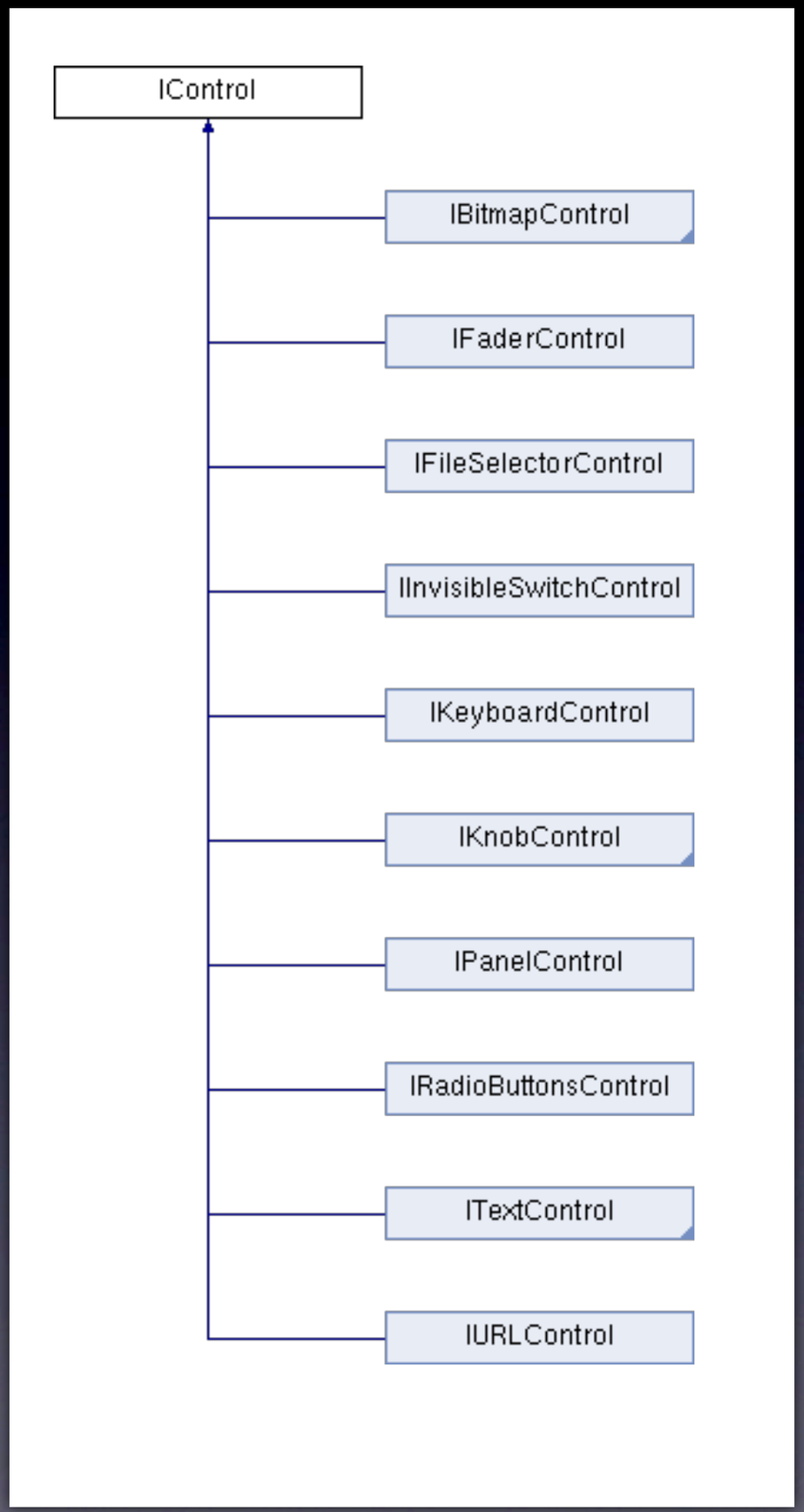
- Likewise when you make an *IGraphics* in the Plugin Constructor its actually either an *IGraphicsMac* or an *IGraphicsWin* depending on the preprocessor macros

the switching happens in:
WDL/IPlug/IPlug_include_in_plug_hdr.h
WDL/IPlug/IPlug_include_in_plug_src.h



- And when you make an *IKnobControl* etc it inherits from *IControl*

- So you can make custom controls by inheriting from these classes and overriding methods (e.g draw())



DSP Tips

- At some point you'll probably want a DSP library so you can build up algorithms quickly with "ugens" like you might do in Max
- If you're not an expert C++ programmer already, don't start off trying to make a DSP library, because as your skills improve you'll keep wanting to rewrite it!

Key issues

- processing precision (float or double samples?)
- readability vs efficiency
- single sample or block-based process() calls?
- sample accurate synchronisation (e.g. how do you trigger?)
- how much should a unit generator do?
- do you separate state and process?
- how do you modulate algorithm parameters?
- denormals

DSP Libraries

there are several open source DSP libraries already available

- Probably OK For closed source (BSD or MIT style license)
  - ICST
    http://www.icst.net/research/downloads/dsp-library/
  - libpd (not yet suitable for plugins)
    https://github.com/libpd
  - Jamoma DSP
    http://redmine.jamoma.org/projects/dsp/
  - Synthesis Tool Kit (STK)
    https://ccrma.stanford.edu/software/stk/
  - Gamma
    http://mat.ucsb.edu/gamma/
  - Maximilian
    http://maximilian.strangeloop.co.uk/
- Not OK for closed source (GPL-style)
  - sndobj
    http://sndobj.sourceforge.net/
  - ugen++
    http://code.google.com/p/ugen/

IPlug Links

- Cockos WDL Page
  http://www.cockos.com/wdl
- Cockos WDL Forum
  http://forum.cockos.com/forumdisplay.php?f=32
- WDL-OL @ github
  https://github.com/olilarkin/wdl-ol