HXCPP

HaXe girt by C++

Girt

Australians all let us rejoice
For we are young and free
We've golden soil and wealth for toil,
Our home is girt by sea

From *Australian National Anthem*

Things to Cover

- The overall architecture
- Some gory implementation details
- Future developments

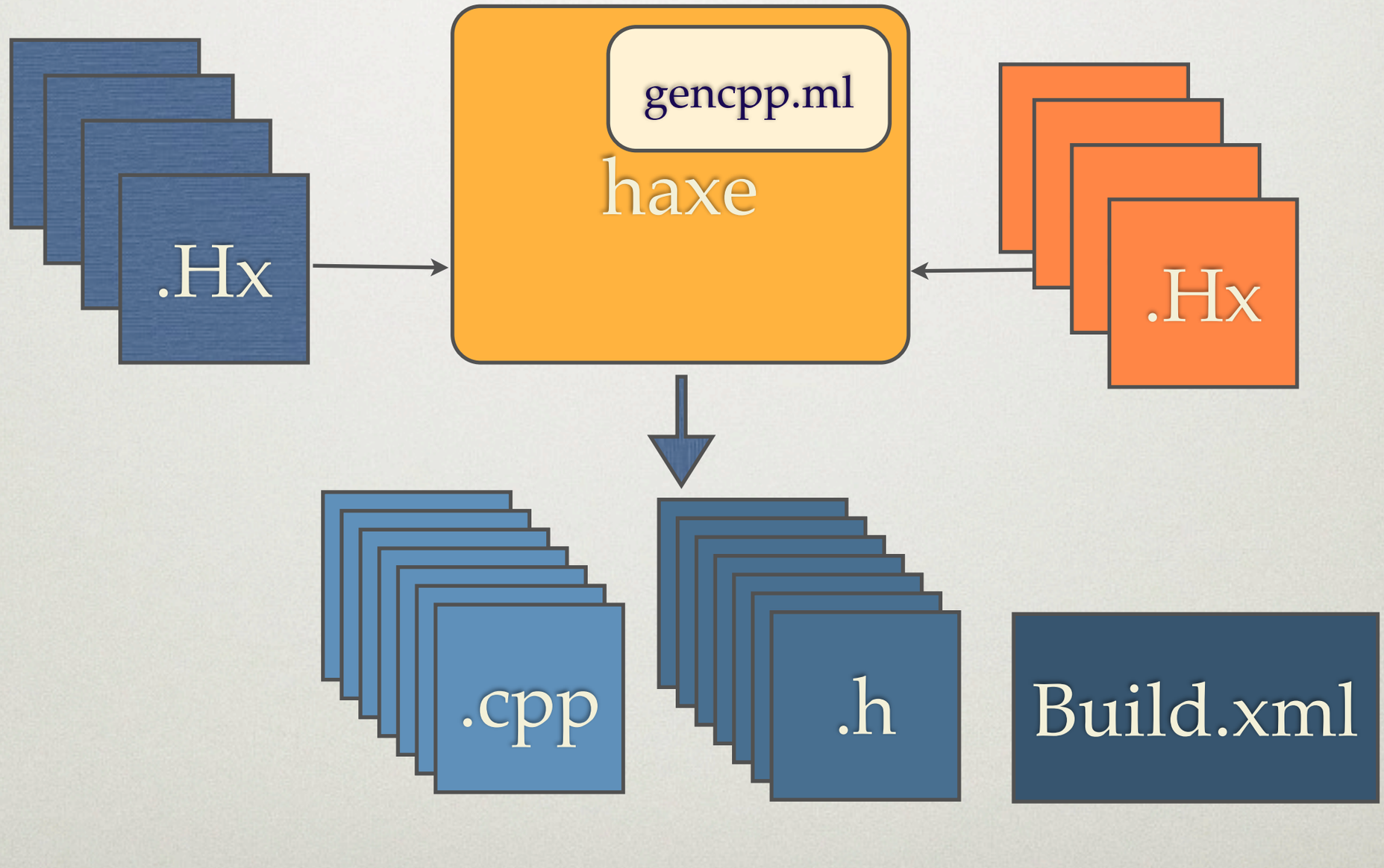
Why HXCPP?

- The original motivation was speed.
- The neko vm design is very nice, but just too slow for numeric algorithms.
- Currently, for gaming style algorithms:
  - 10x neko
  - 2-3x flash/v8
  - 0.5x hand-written native

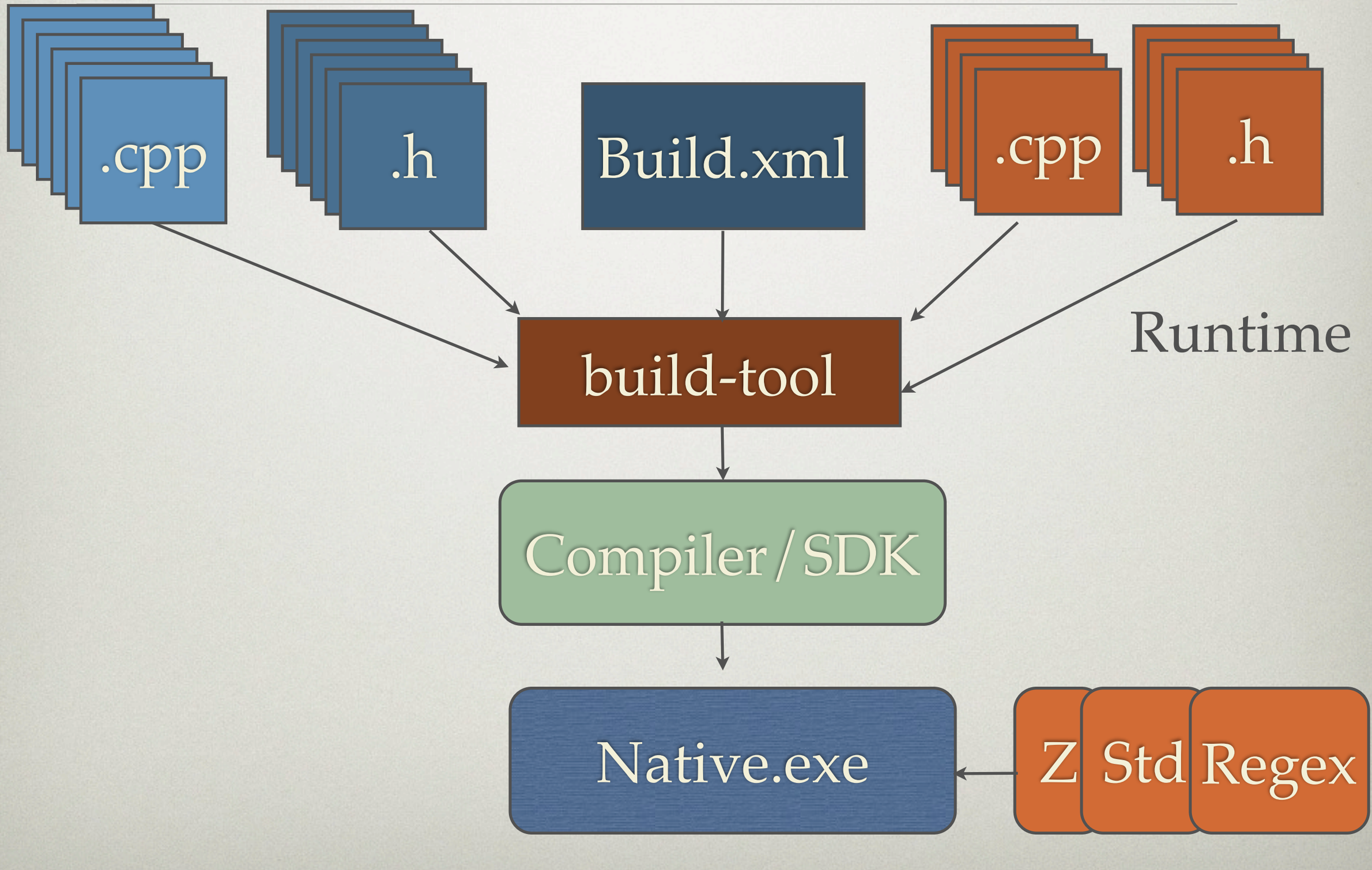
HXCPP IN HAXE

User Code

Library Code



HXCPP IN HAXELIB



Gencpp.ml

- The haXe compiler is written in ocaml, a functional language
- Each "backend" is implemented in a single file
- Gencpp is about 3000 lines of code
- I gained most of my ocaml knowledge from the existing haXe compiler code base
- I probably speak ocaml with a French accent
- Has evolved well beyond initial design

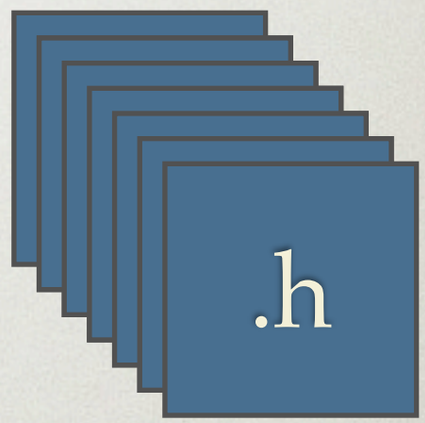
gencpp.ml

The .hx Library Files

- More than a passing resemblance to the neko library files
- Laziness is a virtue
- Use “compiler magic” to delegate the work to the runtime files

HXCPP Output

- "Readable" C++ (not c++11)
  - local functions are a bit hairy
- C++ native virtual functions
- C++ native RTTI
- C++ native exceptions
- Templates do "casting" work
- Macros allow a lot of development to be done in C++, not ocaml -> more accessible



The HXCPP Runtime Files

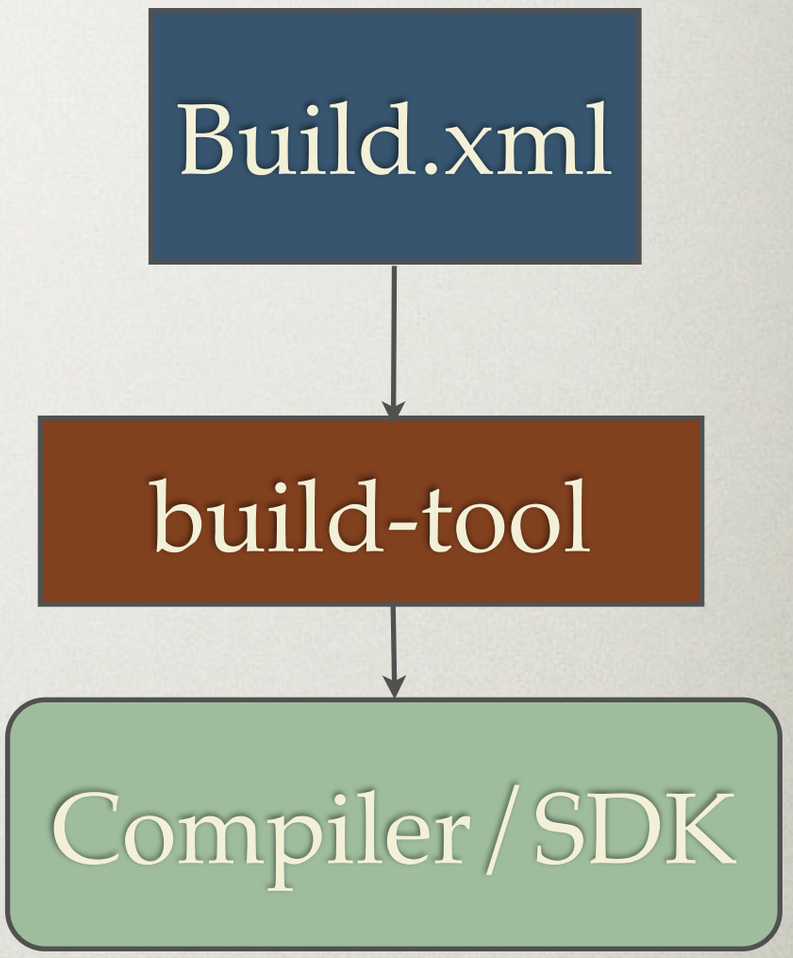
- Are provided as *source*, not binary
  - Good: debugging, tinkering, identical compiler, no binary distribution
  - Bad: extra compile time (maybe solved by compiler cache?)
- Includes: Initialisation, Arrays, Dates, Reflection, Strings, Garbage Collection, CFFI, Threading, Math, Debug etc, etc.

HXCPP RUNTIME DLLS

- Binary distribution (but source included)
- Also have more than a passing resemblance to neko ndlls
- Could not use the neko binaries, but can get a lot of help from source code
- Laziness is still a virtue

Build System

- Neko based replacement for “make” / “ant”, without cross platform pain
- Very easy to extend - got nice user contributions
- android, blackberry, cygwin, gcc, ios (armv6/armv7/i386), linux, mac, mingw, msvc, webos



Implementation

The less nerdy amongst you may want to power down to save some batteries...

Strong Typing

- HXCPP gets its speed from strong typing
- I don't mean hitting the keyboard hard
- When the compiler knows a type, it can use registers and perform optmiztion
- It would be easy just to make everything "Dynamic", but performance would suffer
- Laziness is not a virtue
- Knowing what HXCPP does with certain types is a key to writing high performance code

Typing Concepts

- **Dynamic** - when i refer to variables as Dynamic, I mean members are identified by name only, rather than ordinal structure position or vtable position. All values can be represented by Dynamic.
- **Boxing** - when a non-class variable (Bool, Int, Float, String) is treated as Dynamic, a “boxing” class must be allocated to provide the Dynamic access. This allocation can be expensive.

Where is Dynamic Used?

- SomeClass<T> - T variables are stored and arguments are passed as Dynamic (what else could they be?)
  - Can use haxe.rtti.Generic to create multiple types
  - Create Bool, Int, Float, String (just Float) variations?
- Functions stored in variables (closures)
  - Have strongly typed functor classes (blowout ?), or just Float variations?
- function foo(bar:{run:Void->Void})
  - Fake interfaces?
- var bar = { x : 20 }
  - Fake classes?

Arrays

- Arrays are strongly typed, and have natural C++ implementation
- Are fast - except they have a range check on every access
- Can even avoid this with “unsafe” access
- Array<Dynamic> is actually Dynamic, so it can cope with both Array<Bool> and Array<SomeClass>

Interfaces

- Originally, I used C++ virtual & multiple inheritance. This was going to cause problems with GC due to vtable offset thunks
- Changed to use delegation (interface object has a pointer to original object) - everything was much nicer
- Variables in interfaces are treated with get/set functions. I considered using references, but it was technically challenging

Long Outstanding Issue

- C++ allows some flexibility in the order of some calculations
- Nice to optimizing compilers
- Nasty for deterministic results
- Because of the way arguments are pushed on the stack, they most efficient order for calculation is right-to-left :(
- Working around this has daunted me:

  x = (i++) ? b++ : f(c++ || i++, i++)

CFFI

- C Foreign Functional Interface (actually, C++ FFI)
- Dynamic libraries (static for iPhone) can be built externally and then used. The HXCPP build tool is good for this.
- CFFI allows external code to interact with HXCPP objects by treating them as "handles"
    - This is VM implementation independent to a large extent (can use the same ndll for neko,v8,cpp ...)
- Boxing issues (looking at Float variations)
- Some care must be taken with garbage collection and blocking operations - make porting neko ndlls harder

Metadata Magic

- You can inject code into your classes to call "normal" C++ code.
- Because HXCPP uses natural C++, passing and converting types is quite easy
- May be easier in some cases than CFFI
- Or, maybe get you out of trouble, or get maximum speed (eg, raw function pointers)
- Newer development, but HXCPP is starting to use it for the runtime calls

Garbage Collection(GC)

- HXCPP has built-in GC, with minimal assumptions:
  1. a standard stack layout,
  2. registers can be forced on the stack with a sufficiently complex function call.
- So far, so good!!
- The collection is “mostly precise”, where the haxe objects are marked precisely, but the stack is also scanned conservatively
- This allows objects to be found in CFFI routines, and optimised code.

...Garbage Collection

- All allocating threads need to cooperate with GC
- The compiler creates routines to do the mark phase - there may be some optimisations that can be done here
- The code is pretty simple - and compiles easily on all targets
- There is plenty of scope for research + optimization

Future

I never think of the future - it comes soon enough. ~Albert Einstein

Future

- Work is ongoing to support haXe features as they are developed
- Add new compilers as required
- Improve performance by making fewer things "Dynamic" (probably in order of importance)
  - function variables -> full or partial typing
  - numeric versions of template classes
  - anonymous structs -> fake classes
  - type restrictions -> fake interfaces

...Future

- Improve Garbage Collection speed
  - multi-thread the mark/collection
  - defragmentation/moving
  - revisit marking strategy
- Would be a good project for someone interested

...Future

- Improve instrumentation in debug mode
- Integrated socket based debugger
- Profiling / code coverage metrics
- Tracing objects for GC
- Graphical views of memory usage etc. etc.

// Comments?