Introduction to the C Object System

COS version 0.7

Laurent Deniau

CERN – Accelerator Technologies Department

laurent.deniau@cern.ch

September 11, 2008

Content

Outline

Context – Scientific Software Development @ CERN

Context of software development

- inexperienced resources *(technical and PhD students)*
- non-developer resources *(applied physics students)*
- discontinuous resources *(6 – 36 months)*
- long term development *(>10 years)*
- large scale software *(>100K loc)*
- complex software *(scientific computing)*

☞ This context is not specific to CERN non-IT departments

Motivation I – Simplify Design

Important concepts to simplify software design

- **simplicity** *(easy to understand)*
- flexibility *(easy to change)*
- extensibility *(easy to extend)*
- reusability *(easy to reuse)*
- efficiency *(easy is enough)*
- portability *(easy to port)*

Motivation II – Simplify Programming

Important concepts to simplify software programming

- uniform object model: everything is an object *(simplicity, reusability)*
- functional polymorphism: duck typing *(simplicity, reusability)*
- functional collaboration: multi-methods *(simplicity, extensibility)*
- functional composition: delegation *(flexibility, extensibility)*
- strong encapsulation: ADT *(flexibility, extensibility)*
- open class model: extensible class *(extensibility)*
- design on need: no anticipation *(simplicity, extensibility)*

Motivation III – Simplify Learning

Important points for easy learning

- widely known programming language *(syntax, grammar)*
- "simple" language with "simple" concepts *(background)*
- efficient language with wide spectrum of use *(easy is enough)*
- high portability and availability
- observable behavior easy to understand
- non-ambiguous concepts, free of pitfall

Proposal I – Library Centric Design

Why a library?

- Thousands programming languages already exist
  Would you adopt yet another programming language?
- Comparing to other programming languages implementation
  - ☺ it is easier to develop, improve, port and support
  - ☺ it gets all the benefits and improvements of the underlying language
  - ☹ syntax extensions are restricted by the expressiveness of the language
  - ☹ compile-time errors are compiler-dependent
- Better to lift a well supported mainstream programming language
  - if the underlying language has a wide spectrum of use
  - if the underlying language is expressive enough
- Recycling is fashion

Proposal II – The C Programming Language

Why C?

- C is "simple", efficient and portable
- C is (still) widely used and available
- C is normalized (ISO/IEC 9899:1999)
- C is low-level but still expressive enough (wide spectrum)
- C is the reference language for the foreign function interfaces (FFI)

Proposal III – The C Object System

What is COS?

- The C Object System is *only* a C library
- COS is strongly inspired from OBJECTIVE-C and CLOS
- COS lifts C to the level of other OO programming languages
  - it has its own syntax and grammar
  - it uses the preprocessor to parse and translate COS syntax
- COS implements high-level concepts like
  - uniform object model (everything is an object)
  - polymorphism and strong encapsulation
  - generic functions and multi-methods
  - fast generic delegation
  - exceptions and contracts
  - introspection
  - ownership
  - closure

"Hello World!" Example

Cos

```
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
}
```

☞ Remember, Cos is a C library! Why not use the classic version?

Cos *with streams*

```
#include <cos/Object.h>
#include <cos/File.h>
#include <cos/String.h>
#include <cos/gen/container.h>

int main(void) {
  gput(aFile(stdout), aStr("Hello World!\n"));
}
```

Building Programs

The compile – collect – link – run cycle

```
ld:~/hello$ c99 -W -Wall -pedantic -O3 -c *.c
ld:~/hello$ cossym *.o
ld:~/hello$ c99 *.o _cossym.c -o hello -lCosBase
ld:~/hello$ ./hello
```

- COS must collect its symbols before the final linking step (cossym)
  - methods can be defined anywhere (open class model)
  - cossym is a tiny shell script (≈100 lines)
- COS is a C library that your program must be linked with
- COS provides makefiles to build projects (program, library, plug-in)
- COS can be used with C89 compilers
  - requires an external C99 preprocessor like ucpp

Outline

Components Overview

Cos components

FUNCTION/BEHAVIOR
defgeneric
function declaration

TYPE/STATES
defclass
structure definition

1

1..5

SPECIALIZATION
defmethod
function definition

Basic Types

from cos/Object.h

```
typedef _Bool                 BOOL;     // boolean NO or YES           (as in Objective-C)
typedef const char*           STR;      // string literal              (as in Objective-C)
typedef struct OBJ*           OBJ;      // ADT, never defined        (as id in Objective-C)
typedef const struct Generic* SEL;      // generic as method selector  (as in Objective-C)
```

. . . assuming LP64 data model

```
typedef signed   int     I32;     // at least 32 bits wide
typedef unsigned int     U32;
typedef signed   long    I64;     // at least 64 bits wide
typedef unsigned long    U64;
typedef double           F64;
typedef _Complex double  C64;
```

☞ Basic types are all uppercase (naming convention)

Token recognition

✍ `OBJ`, `void` *and* `va_list` *must not be* typedefed *in generic definition*

Class Interface

from Vector.h

Cos

```
defclass(Vector, Object)
  U32     size;
  double *value;
endclass
```

from Vector.h

Objective-C

```
@interface Vector : Object {
  U32     size;
  double *value;
}
  // methods declarations
@end
```

from Vector.cl

Clos

```
(defclass Vector (Object)
  ((size) (value))
)
```

C structure

```
struct Vector {
  struct Object Object;
  U32     size;
  double *value;
};
```

☞ Class names start by an uppercase (naming convention)

Class Implementation

from Vector.c

Cos

```
#include <cos/Object.h>
#include "Vector.h"

makclass(Vector, Object);
// methods definitions
```

from Vector.m

Objective-C

```
#include <objc/Object.h>
#include "Vector.h"

@implementation Vector : Object
  // methods definitions
@end
```

from Vector.cl

Clos

```
// methods definitions
```

- `makclass` creates instances of `Class`

Compile-time check

- `defclass` *and* `makclass` *must be identical*

Generic

from cos/gen/init.h

```
defgeneric(OBJ, ginitWithDbl, _1, (double)val); // val has a closed type (monomorphic)
```

from cos/gen/container.h

```
defgeneric(OBJ, ggetAt, _1, at);          // rank 2: _1, at      have open types (polymorphic)
defgeneric(OBJ, gputAt, _1, at, obj);     // rank 3: _1, at, obj have open types (polymorphic)
```

C declarations

```
OBJ ginitWithDbl (OBJ _1, double val);
OBJ ggetAt       (OBJ _1, OBJ _2);
OBJ gputAt       (OBJ _1, OBJ _2, OBJ _3);
```

- Generic names start by a lowercase 'g' or 'v' (naming convention)
- Constructors start by 'ginit' or 'vinit' (naming convention)
- Tags of open types are discarded

Compile-time check

- *Name of closed types arguments (monomorphic) are defined forever*

Method

from Vector.c

Cos

```
#include <cos/Object.h>
#include <cos/gen/init.h>
#include "Vector.h"

makclass(Vector, Object);

// defgeneric(OBJ, ginitWithDbl, _1, (double)val);
defmethod(OBJ, ginitWithDbl, Vector, (double)val)
  U32 n = self->size;
  double *value = self->value;

  while (n--) *value++ = val;

  retmethod(_1);      // return the vector
endmethod
```

from Vector.m

Objective-C

```
#include <objc/Object.h>

#include "Vector.h"

@implementation Vector : Object


- (id) initWithDbl: (double)val {
  U32 n = self->size;
  double *value = self->value;

  while (n--) *value++ = val;

  return(self);      // return the vector
}
@end
```

☞ `defmethod` creates instances of `Method`

Compile-time check

☞ `defgeneric` *and* `defmethod` *must be identical* *(tags of open types are discarded)*

Multi-Method

from Stack.c

Cos (only)

```
#include <cos/Object.h>
#include <cos/gen/container.h>
#include "Stack.h"

/* definitions provided by headers above

   defgeneric(OBJ, gpush, to, what);

   defclass(Stack, Collection)
     OBJ *pos;
     OBJ *top;
     OBJ  stk[];
   endclass
*/

makclass(Stack, Collection);

defmethod(OBJ, gpush, Stack, Object)      // rank 2
  test_assert( self->pos < self->top, "stack is full" );

  *self->pos++ = gretain(_2);             // _2 refers to instance of Object (root class)

  retmethod(_1);                          // return the stack
endmethod
```

Message *(method invocation)*

Cos

```
#include <cos/Object.h>
#include <cos/gen/container.h>

void dump_objects(OBJ stream, OBJ obj[])
{
  while (*obj)
    gput(stream, gtoString(*obj++));
}
```

Objective-C

```
#include <objc/Object.h>
#include <stdio.h>

void dump_objects(FILE* stream, id obj[])
{
  while (*obj)
    [[*obj++ toString] printToFile: stream];
}
```

Cos multimethod

```
void dump_objects(OBJ stream, OBJ obj[])
{
  while (*obj) gput(stream, *obj++);
}
```

Compile-time check

☞ *Messages signatures are statically checked against generics messages are C functions that must conform to their prototype*

Components Relationship

Summary

FUNCTION/BEHAVIOR
defgeneric
function declaration

TYPE/STATES
defclass
structure definition

1

1..5

SPECIALIZATION
defmethod
function definition

- Generics must remain simple, clear and generic (reusability)
- Messages and methods must conform to their generics (static typing)
- Methods are bound to generics, not classes (open class model)

Outline

Multi-Method

from Vector.c

```
#include <cos/Object.h>
#include <cos/Number.h>         // for Int and Float
#include <cos/gen/object.h>     // for defgeneric(OBJ, gputAt, to, at, what);
#include "Vector.h"

makclass(Vector, Object);

defmethod(OBJ, gputAt, Vector, Int, Float)    // rank 3
  U32 i = self2->value;

  test_assert( i < self->size, "index out of range" );

  self->value[i] = self3->value;

  retmethod(_1);      // return the vector
endmethod
```

C definitions

```
struct Vector* const restrict self1 = (struct Vector*)_1;
struct Float * const restrict self2 = (struct Float *)_2;
struct Int   * const restrict self3 = (struct Int   *)_3;
```

(self ≡ self1)

More Multi-Method

from Vector.c

```
defmethod(OBJ, gputAt, Vector, Slice, Float)
  test_assert( Slice_first(self2) < self->size
            && Slice_last (self2) < self->size, "slice out of range" );

  for (U32 i = self2->start; i != Slice_end(self2) ; i += self2->stride)
    self->value[i] = self3->value;

  retmethod(_1);     // return the vector
endmethod
```

from Vector.c

```
defmethod(OBJ, gputAt, Vector, Slice, Vector)
  test_assert( Slice_first(self2) < self->size
            && Slice_last (self2) < self->size, "slice out of range" );
  test_assert( self2->size <= self3->size, "incompatible vectors sizes" );

  F64 *value = self3->value;

  for (U32 i = self2->start; i != Slice_end(self2) ; i += self2->stride)
    self->value[i] = *value++;

  retmethod(_1);     // return the vector
endmethod
```

Asymmetric Multi-Method

from DenseMatrix.c

```
#include "DenseMatrix.h"

defmethod(OBJ, gaddTo, DenseMatrix, DenseMatrix) // in place addition _1 += _2;
  test_assert( self->nrow == self2->nrow  &&  self->ncol == self2->ncol );

  for (U32 row = 0; row < self->nrow; row++)
  for (U32 col = 0; col < self->ncol; col++)
    self->value[row][col] += self2->value[row][col];

  retmethod(_1);      // return the dense matrix
endmethod
```

from DiagonalMatrix.c

```
#include "DiagonalMatrix.h"

defmethod(OBJ, gaddTo, DenseMatrix, DiagonalMatrix) // in place addition _1 += _2;
  test_assert( self->nrow == self->ncol  &&  self2->nrow == self->ncol );

  for (U32 i = 0; i < self->nrow; i++)
    self->value[i][i] += self2->value[i];

  retmethod(_1);      // return the dense matrix
endmethod
```

Next-Method and Specialization

```
defclass(A)   int val; endclass
defclass(B,A)          endclass

defmethod(int, gjustDoIt, A, A)
  retmethod( self1->val - self2->val );
endmethod

defmethod(int, gjustDoIt, A, B)
  next_method(self1, self2);        // call (A,A) specialization
endmethod                           // transparently return the result (type int)

defmethod(int, gjustDoIt, B, A)
  next_method(self1, self2);        // call (A,A) specialization
  RETVAL = -RETVAL;                 // intercept and change the returned value
endmethod
```

- `next_method` transfers the returned value from the callee to the caller
- `next_method` can be tested for existence (`next_method_p != 0`)
- Returned value can be intercepted with `RETVAL`

Compile-time check

- `next_method` *must conform to its* `defmethod` *(monomorphic)*

Next-Method and Around-Method

```
defmethod(int, gjustDoIt, A, B)
  next_method(self1, self2);        // call (A,A) specialization as before
endmethod                           // transparently return the result (type int)

defmethod(int, (gjustDoIt), A, B)
  next_method(self1, self2);        // call (A,B) specialization
endmethod                           // transparently return the result (type int)

defmethod(int, gjustDoIt, B, B)
  next_method(self1, self2);        // call (A,B) around specialization
endmethod
```

- Around methods are more specialized than their primary methods
- Around methods are useful for *decorating* already defined behaviors
- Around methods are useful for Key Value Observing
  - Around methods allow to add message-specific notification
- Primary methods can have many around methods
  - Around methods will be chained in an unspecified order
  - Around methods can be tagged after their name like in `(gjustDoIt)1` or `(gjustDoIt)around`

Inheritance and Specialization

```
defclass(A)   .. endclass // least derived
defclass(B,A) .. endclass
defclass(C,B) .. endclass // most  derived

defmethod(void, gjustDoIt, A, A) .. endmethod // least specialized
defmethod(void, gjustDoIt, A, B) .. endmethod // next_method -> (A,A)
defmethod(void, gjustDoIt, B, A) .. endmethod // next_method -> (A,A)
defmethod(void, gjustDoIt, A, C) .. endmethod // next_method -> (A,B)
defmethod(void, gjustDoIt, B, B) .. endmethod // next_method -> (B,A)
defmethod(void, gjustDoIt, C, A) .. endmethod // next_method -> (B,A)
defmethod(void, gjustDoIt, B, C) .. endmethod // next_method -> (B,B)
defmethod(void, gjustDoIt, C, B) .. endmethod // next_method -> (C,A)
defmethod(void, gjustDoIt, C, C) .. endmethod // next_method -> (C,B)
```

Specialization rules and properties

- Dispatcher calls the most specialized method fitting arguments types
- Specialization use natural left-to-right precedence (C,A) > (B,B) > (A,C)
- Specialization is non-ambiguous, monotonic and totally ordered
- `next_method` goes up (back) along applicable specializations

Multi-Method Summary

Summary

- `_1`, `_2`, ..., `_5` are of type `OBJ`
- `self1`, `self2`, ..., `self5` are of the types of the specializers `struct` *Class1*`*`, `struct` *Class2*`*`, ..., `struct` *Class5*`*`
- `self1`, `self2`, ..., `self5` are bound to `_1`, `_2`, ..., `_5`
- `retmethod()` replaces `return()` *(compile-time error)*
- `next_method()` replaces `super` invocation of other OOP languages
- `RETVAL` intercepts returned value

Software design

- ☞ *Object attributes are visible only inside its class methods*
- ☞ *Multi-Methods enhance collaboration and encapsulation*
- ☞ *Multi-Methods reduce drastically the need for getters and setters*
- ☞ *Multi-Methods are bound to generics: open class model*

Outline

Handling Unrecognized Messages *(Object)*

from Object.c

```
useclass(ExBadMessage);

defmethod(void, gunrecognizedMessage1, Object) // rank 1
  THROW( gnewWithStr(ExBadMessage, _sel->name) );
endmethod

defmethod(void, gunrecognizedMessage2, Object, Object) // rank 2
  THROW( gnewWithStr(ExBadMessage, _sel->name) );
endmethod

defmethod(void, gunrecognizedMessage3, Object, Object, Object) // rank 3
  THROW( gnewWithStr(ExBadMessage, _sel->name) );
endmethod

// etc... for rank 4, and 5
```

- ☞ Each `gunderstandMessageN` correspond to each generic ranks (1..5)
- ☞ Unknown messages are substituted by the `unrecognizedMessageN` but `_sel`, `_arg` and `_ret` remain unchanged

COS run-time

☞ *Unrecognized messages throw an* `ExBadMessage` *exception by default*

Forwarding Unrecognized Messages *(Proxy)*

from Proxy.c

```
#include <cos/Proxy.h>  // for defclass(Proxy, Object) OBJ obj; endclass

defmethod(void, gunrecognizedMessage1, Proxy) // rank 1
  forward_message(self1->obj);
endmethod

defmethod(void, gunrecognizedMessage2, Proxy, Object) // rank 2
  forward_message(self1->obj, _2);
endmethod

defmethod(void, gunrecognizedMessage2, Object, Proxy) // rank 2
  forward_message(_1, self2->obj);
endmethod

defmethod(void, gunrecognizedMessage2, Proxy, Proxy) // rank 2
  forward_message(self1->obj, self2->obj);
endmethod

// etc... for rank 3, 4, and 5
```

Cos run-time

☞ `forward_message` *forwards original* `_sel`, `_arg` *and* `_ret` *to different receivers*

Checking Unrecognized Messages

from example.c

```
#include <cos/gen/object.h>  // for defgeneric(OBJ, gunderstandMessage1, _1, (SEL)msg);
#include <cos/gen/stream.h>  // for defgeneric(OBJ, gprint, _1);

static void
print_objects(OBJ obj[])
{
  usegeneric( (gprint)gprint_s ); // use generic object of gprint but rename it gprint_s

  for (U32 i = 0; obj[i] != 0; i++)
    if (gunderstandMessage1(obj[i], (SEL)gprint_s) == True)
      gprint(obj[i]); // ok, message is understood by obj[i]
}
```

Applications of Unrecognized Messages I

```
OBJ obj = aFloat(3.14);
gprint(obj);          // print the double
gput(obj,obj);        // error, throw ExBadMessage
```

transparent element

```
OBJ obj = gnewWith(Proxy, aFloat(3.14));
gprint (obj);         // transparent -> forward print message
gaddTo (obj,obj);     // transparent -> forward addTo message obj += obj;
gdelete(obj);         // transparent -> forward delete message, proxy and obj are destroyed
```

- ☞ `_sel`, `_ret` and `_arg` are hidden arguments of all multimethods
- ☞ `_sel`, `_ret` and `_arg` can be used within multimethods to respectively
  - check current (original) message selector
  - read or modify current (original) returned value
  - read or modify current (original) monomorphic arguments

Applications of Unrecognized Messages II *(Tracer)*

from Tracer.c

```
defclass(Tracer, Proxy) endclass
makclass(Tracer, Proxy);

defmethod(void, gunrecognizedMessage1, Tracer) // rank 1
  trace_msg1(_sel,self1->Proxy.obj); // if enabled, display message sent
  next_method(self1);
endmethod

defmethod(void, gunrecognizedMessage2, Tracer, Object) // rank 2
  trace_msg2(_sel,self1->Proxy.obj,_2); // if enabled, display message sent
  next_method(self1,self2);
endmethod

defmethod(void, gunrecognizedMessage2, Object, Tracer) // rank 2
  trace_msg2(_sel,_1,self2->Proxy.obj); // if enabled, display message sent
  next_method(self1,self2);
endmethod

// etc... for rank 3, 4, and 5
```

trace behavior

```
OBJ obj = gnewWith(Tracer, aFloat(3.14));
gprint (obj);        // trace then forward print message
gaddTo (obj,obj);    // trace then forward addTo message obj += obj;
gdelete(obj);        // tracer and obj are destroyed
```

Applications of Unrecognized Messages III *(Locker)*

from Locker.c

```
defclass(Locker, Proxy) pthread_mutex_t mutex; endclass
makclass(Locker, Proxy);

defmethod(void, gunrecognizedMessage1, Locker) // rank 1
  lock1(self1);    // lock the locker
  next_method(self1);
  unlock1(self1); // unlock the locker
endmethod

defmethod(void, gunrecognizedMessage2, Locker, Locker) // rank 2
  sorted_lock2(self1,self2);    // lock the lockers by ascending memory addresses
  next_method(self1,self2);
  sorted_unlock2(self1,self2); // unlock the lockers by descending memory addresses
endmethod

// etc... for rank 2, 3, 4, and 5
```

lock shared objects

```
OBJ obj = gnewWith(Locker, aFloat(3.14));
gprint (obj);        // lock then forward print message
gaddTo (obj,obj);   // lock then forward addTo message obj += obj;
gdelete(obj);        // locker and obj are destroyed
```

Applications of Unrecognized Messages IV *(Units)*

from Units.c

```
defclass(Units, Proxy)
  U64 unit;
endclass

makclass(Units, Proxy);

defmethod(void, gaddTo, Units, Units)
  test_assert( self1->unit == self2->unit );
  next_method(self1,self2);
endmethod

defmethod(OBJ, gprint, Units)
  next_method(self);
  printf("%s", Units_str(self));
endmethod
```

units behavior

```
OBJ obj = gnewWith2(Units, aFloat(3.14), aLong(SI_meter * SI_meter)); // square meters
gprint (obj);       // print object first then  print units
gaddTo (obj,obj);   // check unit then forward addTo message obj += obj;
gdelete(obj);       // unit and obj are destroyed
```

Outline

Reference Counting – Object Creation and Destruction

common life-cycle

```
useclass(MyClass);

OBJ obj = gnew(MyClass);   // object creation
// ...
gdelete(obj);              // object destruction?
```

detailed life-cycle

```
useclass(MyClass);

OBJ obj = ginit(galloc(MyClass));   // object creation, alloc set reference counting to 1
// ...
if (gretainCount(obj) == 1)         // check for count (gdelete/grelease do much more!)
  gdealloc(gdeinit(obj));           // object destruction!
```

☞ `gnew`*XXX*`()` *(when it exists) sends the messages* `ginit`*XXX*`(galloc())`

Reference Counting – Ownership

collections own their elements (default in COS)

```
defmethod(OBJ, gpush, Stack, Object)
  test_assert( self->pos < self->top );
  *self->pos++ = gretain(_2);
  retmethod(_1);
endmethod

defmethod(OBJ, gtop, Stack)
  test_assert( self->pos > self->stk );
  retmethod( *(self->pos-1) );
endmethod

defmethod(void, gpop, Stack)
  test_assert( self->pos > self->stk );
  grelease( * --self->pos );
endmethod
```

collections do not own their elements

```
defmethod(OBJ, gpush, Stack, Object)
  test_assert( self->pos < self->top );
  *self->pos++ = _2;
  retmethod(_1);
endmethod

defmethod(OBJ, gtop, Stack)
  test_assert( self->pos > self->stk );
  retmethod( *(self->pos-1) );
endmethod

defmethod(void, gpop, Stack)
  test_assert( self->pos > self->stk );
  --self->pos;
endmethod
```

☞ Reference counting is simple, but requires clear conventions

collections share ownership responsibility

```
defmethod(void, gpop, Stack)
  test_assert( self->pos > self->stk );
  grelease( * --self->pos );
endmethod
```

collections share ownership responsibility (cont.)

```
defmethod(OBJ, gpush, Stack, Object)
  test_assert( self->pos < self->top );
  *self->pos++ = _2;
  retmethod(_1);
endmethod
```

Reference Counting – Late Destruction

hidden object factory problem

```
OBJ do_something() {
  OBJ obj = gnew(MyClass);
  // ...
  return obj;
}
```

```
int main(void) {
  OBJ obj = do_something();
  // ...
} // memory leak
```

hidden object factory reloaded

```
OBJ do_something() {
  OBJ obj = gnew(MyClass);
  // ...
  return gautoDelete(obj);
}
```

```
int main(void) {
  OBJ obj = do_something();
  // ...
} // automatically release default AutoRelease pool
```

```
int main(void) {
  OBJ pool = gnew(AutoRelease);
  OBJ obj = do_something();
  // ...
  gdelete(pool); // destroying pool grelease objects
}
```

- ☞ `gnew`*`XXX`*`()` and `alloc`*`XXX`*`()` are object factories (naming convention)
- ☞ Autorelease pools work (almost) as in OBJECTIVE-C

Reference Counting – Automatic and Static Objects

automatic objects

```
OBJ do_something() {


  OBJ obj = aFloat(3.14);
  gprint(obj);

  OBJ obj2 = gretain(obj);
  grelease(obj2);

  // return a copy of obj
  return gautoDelete(obj);
} // obj is automatically discarded
```

static objects

```
OBJ do_something() {
  useclass(String);

  OBJ obj = String;
  gprint(obj);

  OBJ obj2 = gretain(obj);
  grelease(obj2);


  return gautoDelete(obj);
}
```

- ☞ Static objects are insensitive to ownership
- ☞ Automatic objects are cloned as soon as dynamic scope is requested

Reference Counting – AutoRelease Pools

survive to autorelease pool

```
OBJ foo(void) {
  return gautoDelete( aInt(10) ); // implicit object factory
}

OBJ bar(void) {
  useclass(AutoRelease);

  OBJ pool = gnew(AutoRelease);    // top pool
  OBJ obj  = gretain( foo() );

  gdelete(pool);                   // destroy top pool, obj will survive
  return gautoRelease(obj);        // balance the retain
}
```

- `gautoDelete` balances `gnew`*XXX* or automatic constructors
- `gautoDelete` is equivalent to a delayed `gdelete`
- `gautoRelease` balances `gretain`
- `gautoRelease` is equivalent to a delayed `grelease`

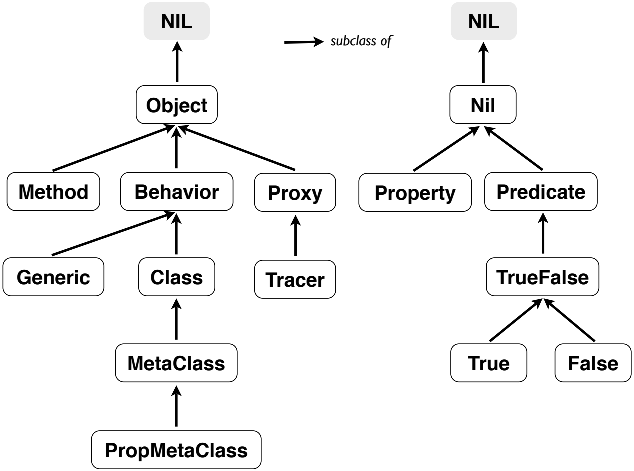
Objects Life-Cycle Summary

Summary

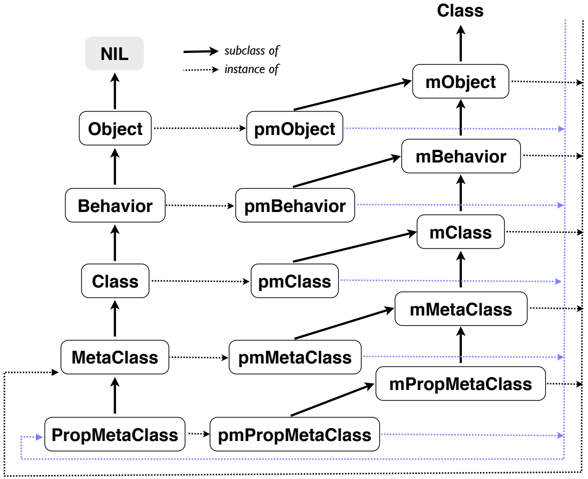
- Ownership requires balanced reference counting
- Reference counting is simple but requires convention
- Reference counting must be balanced locally (high cohesion)
- Static objects are insensitive to ownership (longest lifespan)
- Automatic objects are cloned on first retain (dynamic lifespan)
- Autorelease pools behave like (manual) garbage collectors
  - Autorelease pools are chained (last created = active pool)
  - `gautoRelease` save objects into the active pool for future release
  - Autorelease pools send `grelease` message to their objects
  - Autorelease pools solve the problem of implicit object factory
- By default (can be overridden):
  - `gdelete` ≡ `grelease`
  - `gautoDelete` ≡ `gautoRelease`

Outline

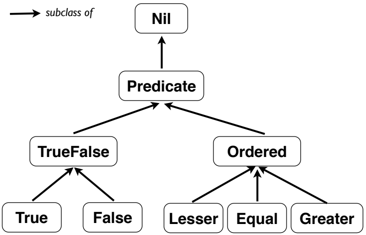
Core Classes Hierarchy



Meta Classes Hierarchy



Predicate Classes Hierarchy



from cos/Object.h

```
useclass(Nil, True, False);
```

from cos/TrueFalse.h

```
defclass(TrueFalse, Nil      ) endclass
defclass(True      , TrueFalse) endclass
defclass(False     , TrueFalse) endclass
```

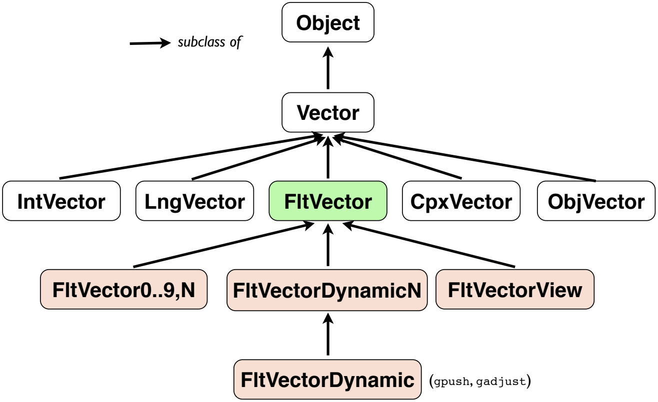
from cos/gen/logic.h

```
defgeneric(OBJ, gand, _1, _2);
```

from TrueFalse.c

```
/*
 AND | F | T | ?
 ----+---+---+---
   F | F | F | F
   T | F | T | ?
   ? | F | ? | ?
*/

defmethod(OBJ, gand, mFalse, mTrueFalse)
  retmethod(False);
endmethod

defmethod(OBJ, gand, mTrueFalse, mFalse)
  retmethod(False);
endmethod

defmethod(OBJ, gand, mTrueFalse, mTrueFalse)
  useclass (TrueFalse);
  retmethod(TrueFalse);
endmethod

defmethod(OBJ, gand, mTrue, mTrue)
  retmethod(True);
endmethod
```

Class Cluster



FltVector is the only class visible and used

Classes Hierarchy Summary

Summary

- Class `Object` is the root class of common COS classes
- Class `Nil` is the root class of predicate and property classes
- Class `Proxy` is the "root class" of proxy classes (transparent)
- `defclass` defines the class, meta-class and property meta-class

- Class `MyClass`
  - instances are of type `struct MyClass*` (or `OBJ`)
- Meta Class `mMyClass` is an instance of `MetaClass`
  - instances are of type `struct Class*` (or `OBJ`)
- Property Meta Class `pmMyClass` is an instance of `PropMetaClass`
  - instances are of type `struct Class*` (or `OBJ`)

Outline

Exceptions *(Non-Local Errors)*

Try section

- `TRY` section defines code that may throw an exception
- `CATCH` section defines code to handle exception thrown in `TRY`
- `CATCH_ANY` is same as `CATCH` but handle *any* exception thrown in `TRY`
- `FINALLY` section defines code that should always be executed

Throw

- `THROW(`*ex*`)` throws the exception *ex*
- `RETHROW` throws the current exception within a `CATCH` or `FINALLY` section
- Both accept *extra optional* arguments for debugging
  - *function-name* (`STR`) location of the test *(generated if not provided)*
  - *file-name* (`STR`) location of the test *(generated if not provided)*
  - *line-number* (`int`) location of the test *(generated if not provided)*

Exceptions *(cont.)*

from Stack.c

```
defmethod(OBJ, gput, Stack, Object)
  if (self->pos == self->top) {
    useclass(ExOverflow);     // local declaration
    THROW( gnewWithStr(ExOverflow, "Stack is full") );
  }
  *self->pos++ = gretain(_2);
  retmethod(_1);
endmethod
```

from main.c

```
int main(void) {
  useclass(Stack, Float, ExOverflow);
  OBJ stk = gnewWithSize(Stack, 1000), val = Nil;
  TRY
    gpush(stk, val = gnew(Float));
  CATCH(ExOverflow, ex)
    gdelete(val);
  FINALLY
    gdelete(stk);
  ENDTRY
}
```

☞ COS exceptions work as in other high-level programming languages

Contracts

Contract sections

- `PRE` section defines pre-conditions executed before `BODY`
- `POST` section defines post-conditions executed after `BODY`
- `BODY` section defines the core of the method
- `PRE` and `POST` sections are optional
- Invariants are automatically checked after post-conditions (if any)
- Throwing an exception breaks the contract

Tests

- `test_assert(`*cond*`)` throws an `ExBadAssert` if *cond* fails (equals zero)
- `test_invariant(_1)` sends the `ginvariant` message to `_1` (explicit)
- Both accept *extra optional* arguments for debugging
  - *tag-name* (`STR`) informative
  - *function-name* (`STR`) location of the test *(generated if not provided)*
  - *file-name* (`STR`) location of the test *(generated if not provided)*
  - *line-number* (`int`) location of the test *(generated if not provided)*

Contracts *(cont.)*

from Stack.c

```
defmethod(OBJ, gput, Stack, Object)
  OBJ* old_pos;

  PRE // executed before BODY
    test_assert( self->pos < self->top );
    old_pos = self->pos;

  POST // executed after BODY
    test_assert( self->pos == old_pos+1 );
    test_assert( self->pos <= self->top );  // could be an invariant
    test_assert( *old_pos == _2 );

  BODY
    *self->pos++ = gretain(_2);
    retmethod(_1);
endmethod
```

tuning contracts level

`COS_CONTRACT`	contract level *(macro)* from `NO` to `COS_CONTRACT_ALL`
`COS_CONTRACT_PRE`	enables `PRE` sections *(default)*
`COS_CONTRACT_POST`	enables `PRE` and `POST` sections
`COS_CONTRACT_ALL`	enables `PRE` and `POST` sections plus invariants

Outline

COS Performances

gcc 4.2 on Linux Intel T9300

```
** C Object System Speed Testsuite (Linux 64 bits) **
 + method (0 argument )                          (0.55 s) -  218182 Kitr/s
 + method (1 argument )                          (0.57 s) -  210526 Kitr/s
 + method (2 arguments)                          (0.65 s) -  184615 Kitr/s
 + method (3 arguments)                          (0.70 s) -  171429 Kitr/s
 + method (4 arguments)                          (0.78 s) -  153846 Kitr/s
 + method (5 arguments)                          (0.90 s) -  133333 Kitr/s
 + multimethod (rank 2)                          (0.80 s) -  150000 Kitr/s
 + multimethod (rank 3)                          (0.99 s) -  121212 Kitr/s
 + multimethod (rank 4)                          (1.34 s) -   89552 Kitr/s
 + multimethod (rank 5)                          (1.55 s) -   77419 Kitr/s
```

Cos vs C++

g++ 4.2 on Linux Intel T9300

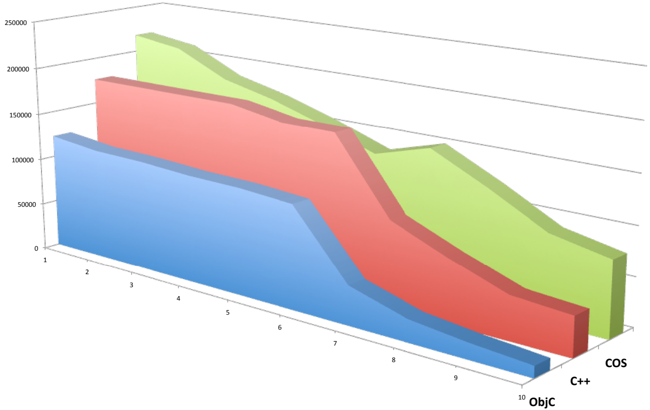
```
** C Object System Speed Testsuite for C++ (Linux 64 bits) **
 + virtual function (0 argument )              (0.68 s) -  176471 Kitr/s   +24%
 + virtual function (1 argument )              (0.68 s) -  176471 Kitr/s   +19%
 + virtual function (2 arguments)              (0.68 s) -  176471 Kitr/s    +5%
 + virtual function (3 arguments)              (0.68 s) -  176471 Kitr/s    -3%
 + virtual function (4 arguments)              (0.72 s) -  166667 Kitr/s    -8%
 + virtual function (5 arguments)              (0.72 s) -  166667 Kitr/s   -20%
 + virtual function & visitor pattern (rank 2) (1.33 s) -   90226 Kitr/s   +66%
 + virtual function & visitor pattern (rank 3) (1.82 s) -   65934 Kitr/s   +84%
 + virtual function & visitor pattern (rank 4) (2.69 s) -   44610 Kitr/s  +101%
 + virtual function & visitor pattern (rank 5) (2.98 s) -   40268 Kitr/s   +92%
```

Cos vs Objective-C

gcc 4.2 on Linux Intel T9300

```
** C Object System Speed Testsuite for Objective-C (Linux 64 bits) **
 + method (0 argument )                            (0.98 s) -  122449 Kitr/s   +78%
 + method (1 argument )                            (1.03 s) -  116505 Kitr/s   +81%
 + method (2 arguments)                            (1.04 s) -  115385 Kitr/s   +60%
 + method (3 arguments)                            (1.07 s) -  112150 Kitr/s   +53%
 + method (4 arguments)                            (1.08 s) -  111111 Kitr/s   +38%
 + method (5 arguments)                            (1.12 s) -  107143 Kitr/s   +24%
 + method & visitor pattern (rank 2)               (3.00 s) -   40000 Kitr/s  +275%
 + method & visitor pattern (rank 3)               (5.23 s) -   22945 Kitr/s  +428%
 + method & visitor pattern (rank 4)               (7.40 s) -   16216 Kitr/s  +452%
 + method & visitor pattern (rank 5)              (10.28 s) -   11673 Kitr/s  +563%
```

Performance Summary



Outline

Conclusion

Pros

- COS is as simple as OBJECTIVE-C but has more features
- COS is nearly as flexible and extensible as PERL, PYTHON, RUBY,..
- COS is nearly as efficient as ADA, C++, EIFFEL,..
- COS is as portable as C ☺☺☺
- COS is easy to learn, use and support
- COS enforces good design *(multi-methods & encapsulation)*
  - bad design is rapidly boring or inefficient *(getters & setters)*

Cons

- COS "standard" library is still under development
- COS has less syntactic sugar than *true* programming languages
- COS has less room for optimization than *true* program. languages

Outline

Alias *(method alias)*

from Stack-ext.c

```
#include <cos/Object.h>
#include <cos/gen/container.h>

/* generics provided by container.h (reminder)

   defgeneric(OBJ, gput , to, what);
   defgeneric(OBJ, gpush, to, what);

   defined in Stack.c (reminder)

   defmethod(OBJ, gput, Stack, Object)
     ...
   endmethod
*/

defalias(OBJ, (gput)gpush, Stack, Object);  // only defgenerics of gput and gpush are required
```

Compile-time checks

- `defmethod` *and* `defalias` *must be identical except for their names*
- *Both generics must be identical* (tags of open types are discarded)

Generic Implementation (optional with COS makefiles)

```
#include <cos/Object.h>
#include <cos/gen/init.h>
#include <cos/gen/container.h>

/* generics provided by headers above (reminder)

   defgeneric(OBJ, ginitWithDbl, _1, (double)val);
   defgeneric(OBJ, gputAt      , _1, at, obj);
   defgeneric(OBJ, ggetAt      , _1, at);
*/

makgeneric(OBJ, ginitWithDbl, _1, (double)val);
makgeneric(OBJ, gputAt      , _1, at, obj);
makgeneric(OBJ, ggetAt      , _1, at);
```

☞ `makgeneric` create instances of `Generic`

Compile-time check

☞ `defgeneric` *and* `makgeneric` *must be identical* *(tags of open types are discarded)*