ObjXp

YAIL (Yet Another IPC/IO Library)

Hello Yail

```
HelloYail.cpp
-----------------------------------------------------------------------------------------------

#include <Yail.h>

using namespace Yail;

YAIL_BEGIN_CLASS(Hello, EXTENDS(YObject))
  public:
      void init(String msg) { msg_ = msg; }
      void print() {
          std::cout << msg_ << std::endl;
      }

  private:
      String msg_;
YAIL_END_CLASS

int main(int argc, char **argv) {
    SPtr<Hello> myObject = CreateObject<Hello>("Hello Yail!");
    myObject->print();
    return 0;
}
```

```
How to build & run
----------------------------

$ BUILD_EXAMPLES=yes cmake .
$ make all # Build in multi-jobs is supported. Use -j option for that.

$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:`pwd`/lib
$ ./bin/HelloYail
```

Yail.h and YObject

- YObject is the top of all Yail classes. It's defined in Yail.h
- Yail.h also defines common types and macros (See Yail.h for details)

Namespace Yail::

- All Yail classes are defined in "Yail::" namespace
- Full class path is "Yail::SomeYailClass"
- By declaring "using namespace Yail;", you can use the class name directly like "YObject" instead of "Yail::YObject"

Class Definition (1)

- Yail class hierarchy resembles Java.
- A class can inherit from single parent class and multiple interfaces
- http://stackoverflow.com/questions/10839131/implements-vs-extends-when-to-use-whats-the-difference
- In reality, Yail is written in c++, which means the compiler will allow the c++ style multiple inheritance if you insist on defining that way. But do NOT design that way to avoid future headaches.

Class Definition (2)

```
YAIL_BEGIN_CLASS(ClassName,
                 EXTENDS(ParentClass),
                 IMPLEMENTS(SupportedInterface), IMPLEMENTS(SupportedInterface)[, more Interfaces if any ])
          public:
                    void init( init args ...);

                    < public methods and members >
          protected:
                    < protected methods and members >
          private:
                    < private methods and members >
YAIL_END_CLASS

YAIL_BEGIN_INTERFACE(InterfaceName)
          virtual <return_type> methodSignature(<args...>) = 0; // pure virtual function
          // int intData; <- DO NOT define data member in Interface
YAIL_END_INTERFACE
```

Class definition (3) - Extended mode

YAIL_BEGIN_CLASS_EXT() macro is similar to YAIL_BEGIN_CLASS() except that it forces to implement two functions __new__ClassName() and __del__ClassName(), which are called from the constructor and destructor respectively.

```
YAIL_BEGIN_CLASS_EXT(ClassName,
                 EXTENDS(ParentClass),
                 IMPLEMENTS(SupportedInterface), IMPLEMENTS(SupportedInterface)[, more Interfaces if any ])
          void __new__ClassName() { /* This function will be called by constructor */  }
          void __del__ClassName() { /* This function will be called by destructor */ }
          public:
                    void init( init args ...);

                    < public methods and members >
          protected:
                    < protected methods and members >
          private:
                    < private methods and members >
YAIL_END_CLASS
```

Object Creation

- SPtr<FooType> myObj = CreateObject<FooType>(init args...)

  What happens?

  1. Creates a FooType instance
  2. Call FooType::init(init args ...) on the instance
  3. Return smart pointer of the instance(will be explained later)

- init(...) function is NOT a constructor. ParentClass::init(...) is not called automatically. Must be called explicitly if needed.

Smart Pointer (1)

```
C language
----------------------------------------

void foo() {
  void* objData = malloc(1024);
  bar(objData);
  // forgot to free(objData)
}
```

Memory leak every time foo() is called

```
Regular C++ class
------------------------------

void foo() {
  Foo* obj = new Foo();
  obj->bar();
  // forgot to delete obj;
}
```

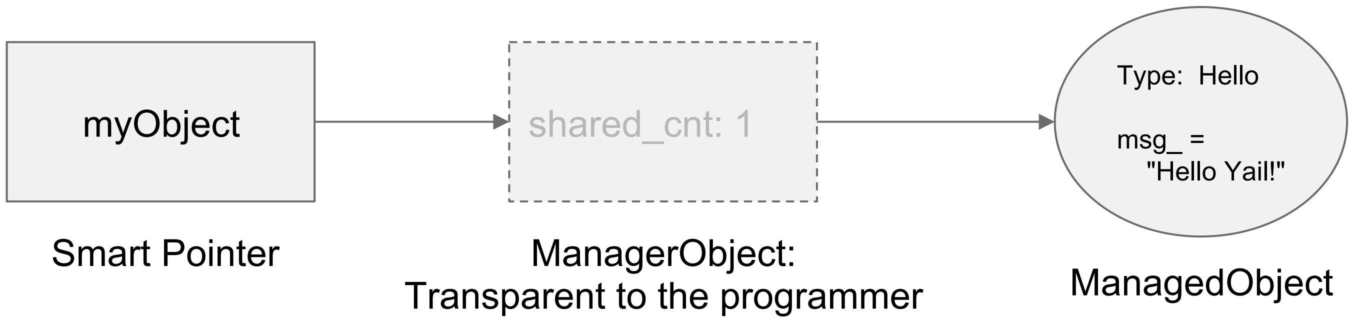
Memory leak every time foo() is called

```
C++ with smart pointer
-------------------------------------------------------

void foo() {
  SPtr<Foo> obj = CreateObject<Foo>();
  obj->bar();
  // nothing to do to delete obj
}
```

No Memory leak.
'obj' will be destroyed
automatically when leaving
the scope of foo() function.

Smart Pointer (2) - Basic Idea

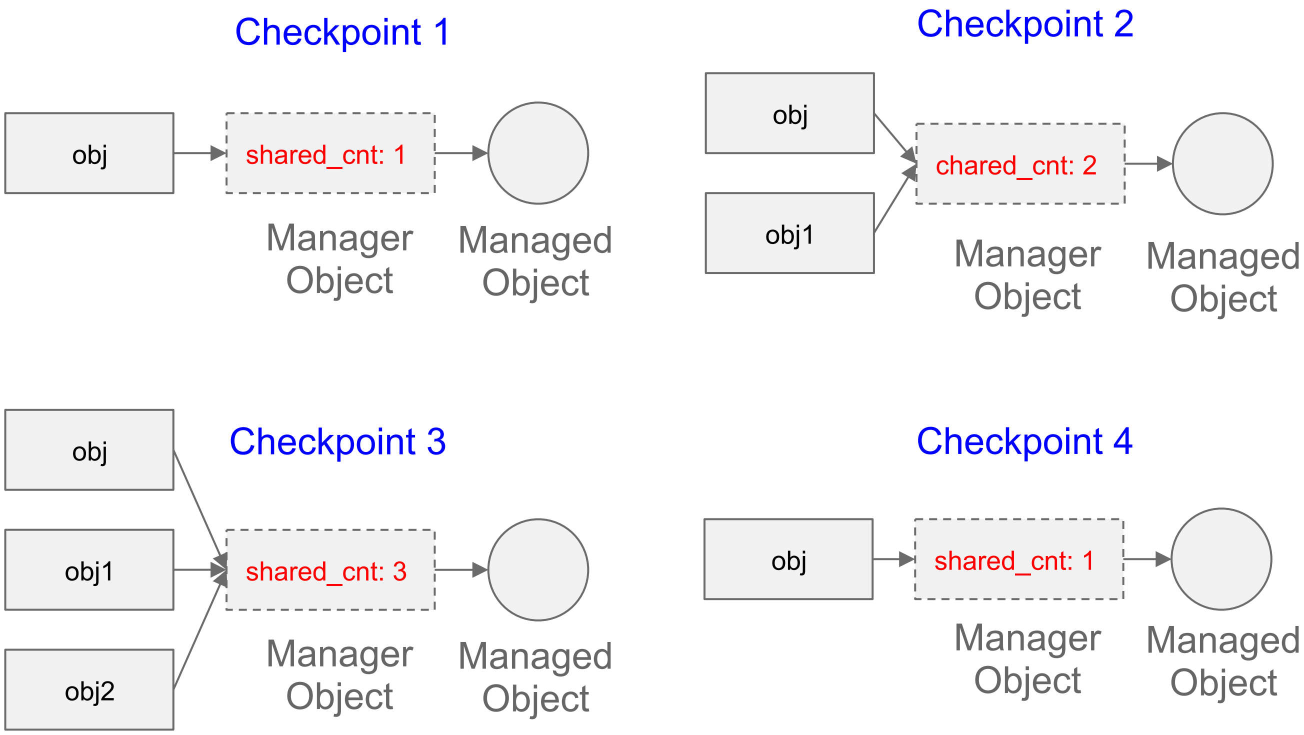
SPtr<Hello> myObject = CreateObject<Hello>("Hello Yail!");



- When an object(managed object) is created, its manager object is also created.
- ManagerObject keeps track of the reference count and will delete the managed object when the reference count becomes zero.

Smart Pointer (3) - Reference Count Changes

```
void foo(SPtr<Hello> obj1) {
  // checkpoint 2
  ......
  SPtr<Hello> obj2 = obj1;
  // checkpoint 3
  ......
}

int main(int argc, char** argv) {
  SPtr<Hello> obj =
             CreateObject<Hello>("Hi!");
  // checkpoint 1
  ......

  foo(obj);
  // checkpoint 4
  ......

  return 0;
}
```



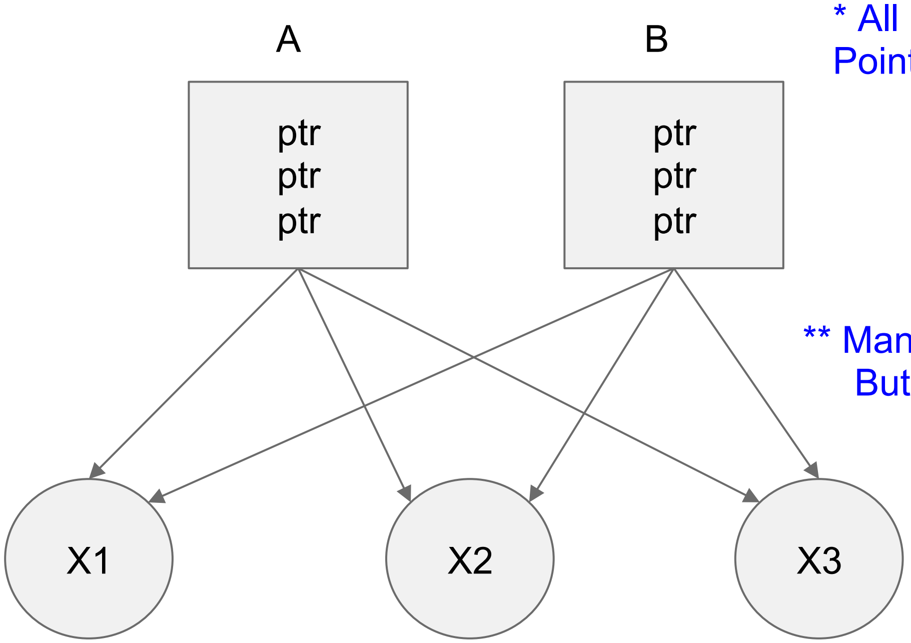
Checkpoint 1

Checkpoint 2

Checkpoint 3

Checkpoint 4

Shared Pointer (SPtr) (1)



* All Smart pointers mentioned so far are Shared Pointers(SPtr)

** ManagerObjects are not shown in this picture for simplicity.
But Xn is still managed via its ManagerObject.

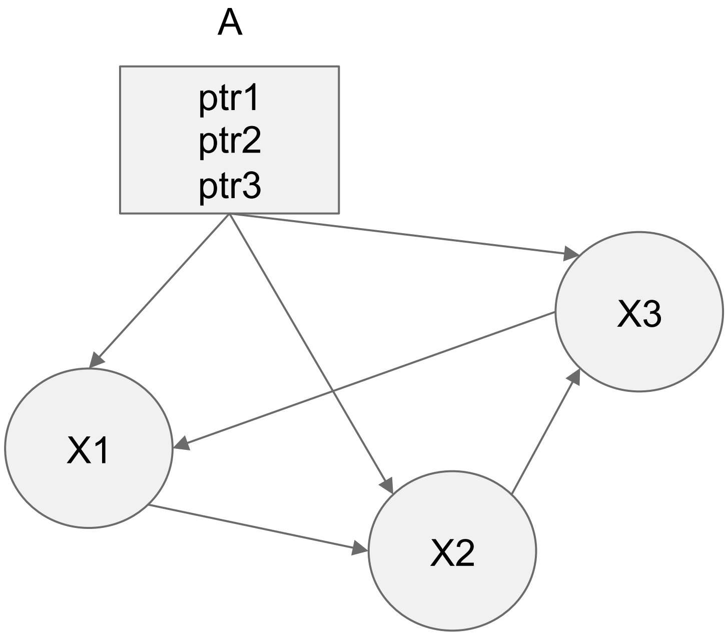
Xn will not be deleted as long as there is any reference to it.

Shared Pointer (SPtr) (2)

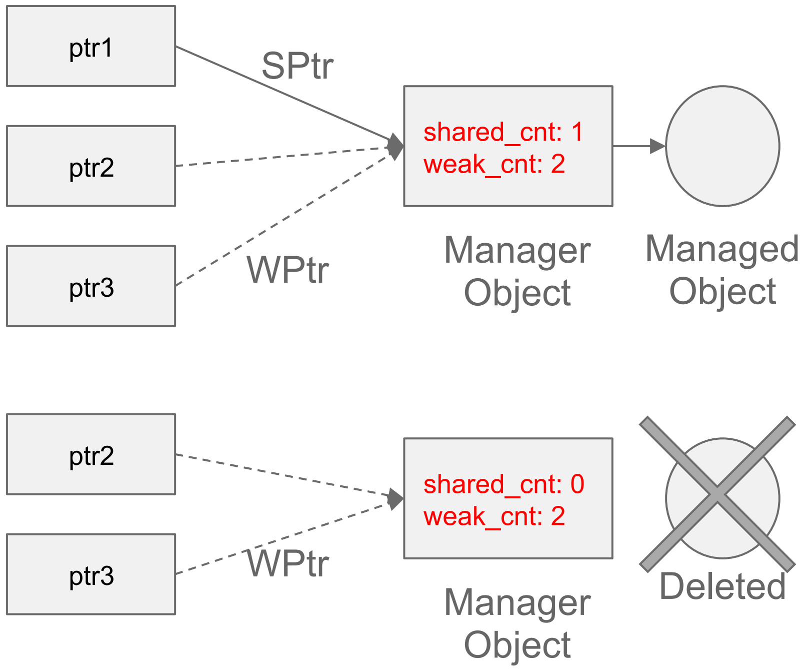
Problem Case:

- Let's say "A" is an object container that owns X1, X2 and X3.
- X1, X2, X3 happen to be referencing each other and making a loop.
- X1, X2, X3 will not be deleted even after "A" gives up all its pointers.

==> Memory Leak



Weak Pointer (WPtr) (1)



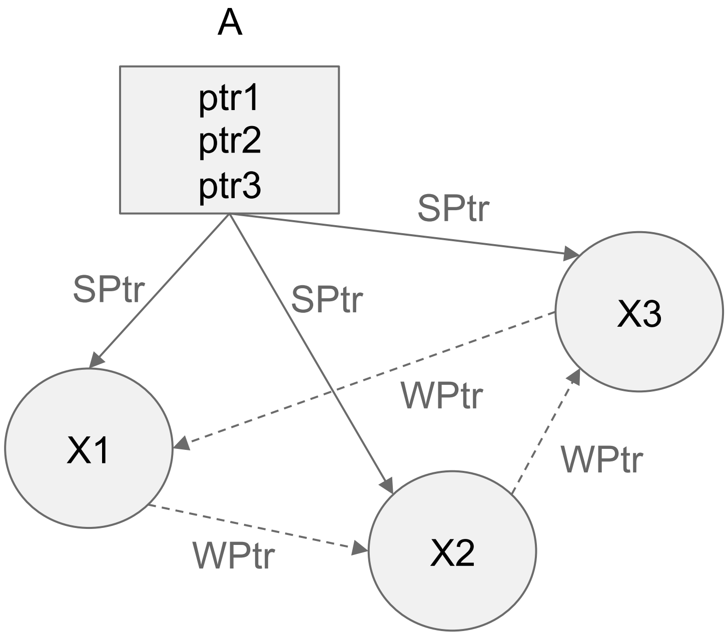
- ManagerObject keeps two counters separately
  - shared_count
  - weak_count
- Initial creation is always with SPtr
- ManagedObject will be deleted when shared_count becomes zero. But ManagerObject is still remaining as long as weak_count is non-zero.
- Once deleted, ManagedObject cannot be restored from WPtr.
- ManagerObject will be deleted when weak_count becomes zero.

Weak Pointer (WPtr) (2)

Going back to the previous problem case:

- Let's say "A" is an object container that owns X1, X2 and X3.
- X1, X2, X3 happen to be referencing each other and making a loop.
- X1, X2, X3 will be deleted as expected when "A" gives up the shared pointers.

==> No Memory Leak



Using WPtr (1)

```
void foo(WPtr<Hello> wPtr) {
    wPtr->print(); // Compile time Error. WPtr cannot directly access the class members.

    SPtr<Hello> sPtr = wPtr.lock(); // Convert WPtr to SPtr
    sPtr->print(); // Compilation Ok, But may cause a crash!!!
}
```

Using WPtr (2)

```
void foo(WPtr<Hello> wPtr) {
    SPtr<Hello> sPtr = wPtr.lock(); // Convert WPtr to SPtr

    if(sPtr) {// sPtr is not null and lock() succeeded, which means
                 // at least one other SPtr was holding the object before my lock() trial above,
                 // and now my sPtr is holding the object again.
        sPtr->print(); // Ok
    } else {
                // Handle the case the object was deleted already. (shared_count == 0)
                cout << "Warning: target object deleted" << endl;
    }
}
```

Using WPtr (3) - Pointer Assignment

```
SPtr<Hello> tmp; // Automatically initialized with nullPtr

SPtr<Hello> sPtr1 = CreateObject<Hello>("Hi!");

SPtr<Hello> sPtr2 = sPtr1; // Assign SPtr to SPtr: Ok

WPtr<Hello> wPtr1 = sPtr2; // Assign SPtr to WPtr: Ok

WPtr<Hello> wPtr2 = wPtr1; // Assign WPtr to WPtr: Ok

SPtr<Hello> sPtr2 = wPtr2; // Assign WPtr to SPtr: Compile Error

SPtr<Hello> sPtr2 = wPtr2.lock(); // Convert WPtr to SPtr: Ok
```

USE_WPTR macro (1)

To make the WPtr scope clear and to help enforce null pointer handling

```
#define USE_WPTR(sPtrType, sPtr, wPtr,                                          \
                 doItBefore, doItIfSPtrIsNotNull, doItIfSPtrIsNull, doItAfter)  \
 {                                                                              \
   SPtr<sPtrType> sPtr = (wPtr).lock();                                         \
   { doItBefore }                                                               \
   if(sPtr) { doItIfSPtrIsNotNull }                                             \
   else { doItIfSPtrIsNull }                                                    \
   { doItAfter }                                                                \
 }

void foo(WPtr<Hello> wPtr) {
   USE_WPTR(Hello, sPtr, wPtr,
         {},
         { sPtr->print(); },
         { cout << "Warning: target object deleted" << endl; },
         {}
   );
   // sPtr is not visible here. Out of scope
}
```

USE_WPTR macro (2) - STL Container Example

```
// Let's assume there is a list of WPtr<Data>
List<WPtr<Data>> wDataList;

void produceData(SPtr<Data> sData) {
  WPtr<Data> wData = sData;
  wDataList.push_back(wData);
}
```

See the following link for more about std::list.
http://www.cplusplus.com/reference/list/list/

```
SPtr<Data> consumeData() {
  while(!wDataList.empty()) {
    USE_WPTR(Data, sData, wDataList.front(),
      { wDataList.pop_front(); },
      { return sData; },
      {},
      {}
    );
    return nullPtr(Data);
  }
}
```

=

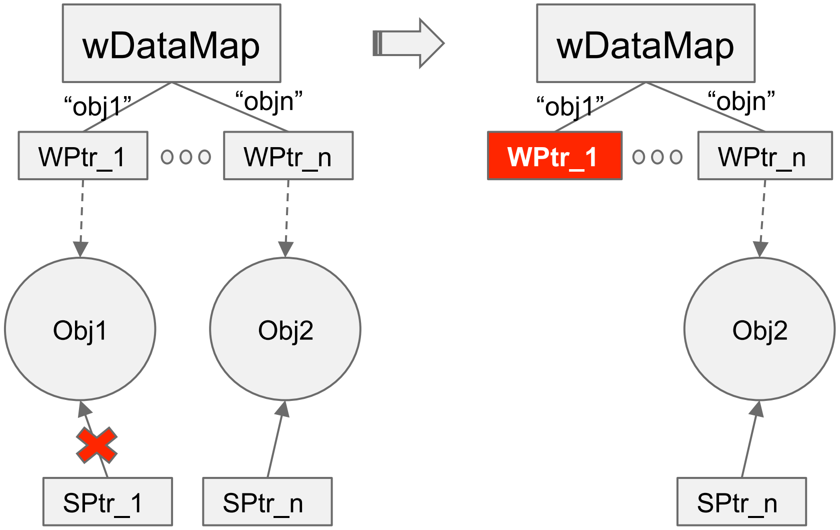
```
SPtr<Data> getData() {
  while(!wDataList.empty()) {
    { SPtr<Data> sData = (wDataList.front()).lock();
      { { wDataList.pop_front(); } }
      if(sData) { { return sData; } }
      else { { } }
      { { } }
    };
    return nullPtr(Data);
  }
}
```

USE_WPTR macro (3) - STL Container Potential Memory Leak

```
// Let's assume there is a String to WPtr<Data> map
Map<String, WPtr<Data>> wDataMap;

void insertData(String key, SPtr<Data> sData) {
  WPtr<Data> wData = sData;
  wDataMap.insert(Pair<String, WPtr<Data>>(key,wData));
}

SPtr<Data> findData(String key) {
  Map<String, WPtr<Data>>::iterator iter = wDataMap.find(key);
  if(iter != wDataMap.end()) {
    USE_WPTR(Data, sData, iter->second,
      {},
      { return sData; }
      { wDataMap.erase(key); }
      {}
    );
  }
  return nullPtr(Data);
}
```

Memory leak if this statement is not called here or anywhere else.

See the following link for more about std::map.
http://www.cplusplus.com/reference/map/map/
and
http://stackoverflow.com/questions/36132936/what-happens-to-an-expired-weak-ptr-in-a-map



When SPtr_1 is removed, Obj1 will be destroyed.
But WPtr_1 still remains until wDataMap.erase("obj1") is called.

USE_WPTR macro (4) - Avoid Nesting

```
USE_WPTR(Hello, hsPtr, hwPtr1,
  {},
  { USE_WPTR(World, wsPtr, wwPtr,
        {},
        { foo(hsPtr, wsPtr); },
        { /* handle null wsPtr */ },
        {}
      );
  },
  { /* handle null hsPtr */ },
  {}
);
```

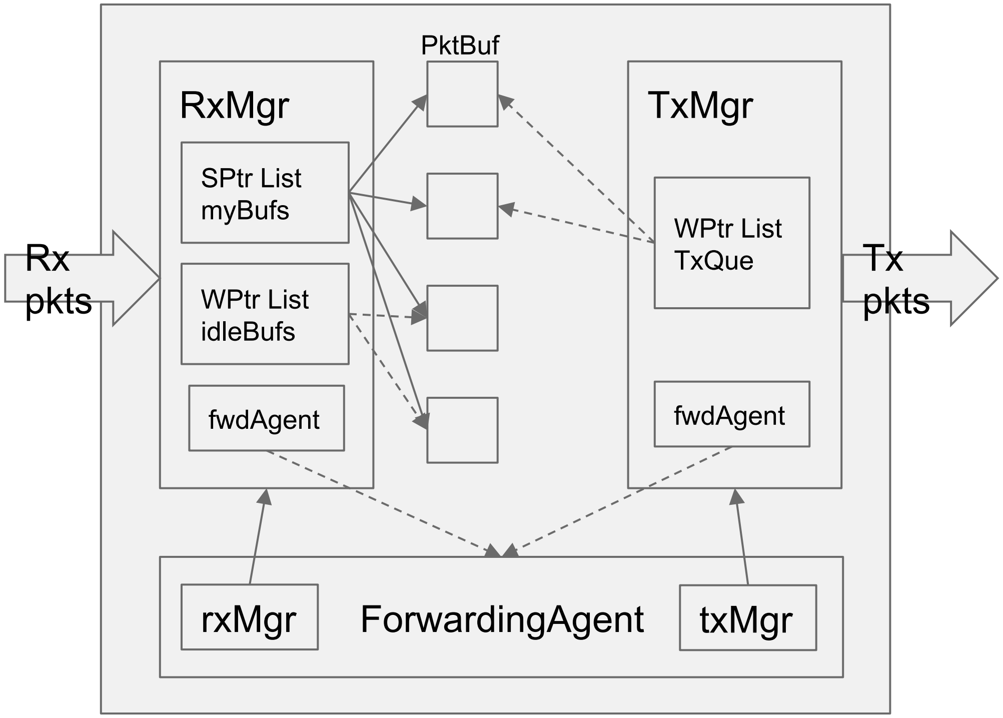
Avoid nesting

```
SPtr<Hello> hsPtr;
SPtr<World> wsPtr;

USE_WPTR(Hello, _hsPtr, hwPtr1,
  {},
  { hsPtr = _hsPtr; },
  { /* handle null _hsPtr */ },
  {}
);

USE_WPTR(World, _wsPtr, wwPtr,
  {},
  { wsPtr = _wsPtr; },
  { /* handle null _wsPtr */ },
  {}
);

if(hsPtr && wsPtr) {
   foo(hsPtr, wsPtr);
}
```

Design Guide - SPtr vs WPtr

- If an object is never going to be removed, it doesn't matter how many SPtrs share it.
- If object is created/removed dynamically, it is usually easy to handle single SPtr and multiple WPtrs. But not necessarily, it depends on you judgement.
- In general, define ObjectOwner as SPtr and ObjectUser as WPtr
- Pointers that may form a loop should have WPtr(s) on at least one of the links.
- More to come ...

Design Example - Packet Forwarder



- ForwardingAgent owns the SPtrs for RxMgr and TxMgr
- RxMgr and TxMgr have the reverse WPtr pointer to their owner ForwardingAgent.
- RxMgr allocates packet buffers and owns the SPtrs in myBufs list. It also maintains the current idle buffers in idleBufs WPtr list.
- On rx packet arrival, RxMgr removes an idle buffer from idleBufs list, fills the rx data, then passes to TxQue of the TxMgr.
- TxMgr transmits the queued packets and returns the completed buffers to RxMgr.
- RxMgr recycles the buffer to the idleBufs list.

- If Rx interface is shut down suddenly, RxMgr may remove the buffers already queued in TxQue.
- TxMgr should handle the null pointer case when processing the TxQue.

Explicit Pointer Release - reset()

- Reference count usually decreases automatically when exiting a function scope. (Implicit Release)
- In some cases, you may want to release the pointer explicitly.
- reset() is applicable for both SPtr and WPtr

```
SPtr<Hello> obj1 = CreateObject<Hello>("Hi!");  // Create an object
assert(obj1.use_count() == 1)

SPtr<Hello> obj2 = obj1;
assert(obj1.use_count() == 2);
assert(obj2.use_count() == 2);

obj1.reset(); // Release obj1 explicitly
assert(obj1.use_count() == 0); // obj1 is nullptr
assert(obj2.use_count() == 1); // but obj2 is still pointing the object. The managed object is still there.

obj2.reset(); // Release obj2 explicitly. At this point the managed object will be deleted.
assert(obj1.use_count() == 0);
assert(obj2.use_count() == 0);
```

Dot(.) vs Arrow(->) Operator

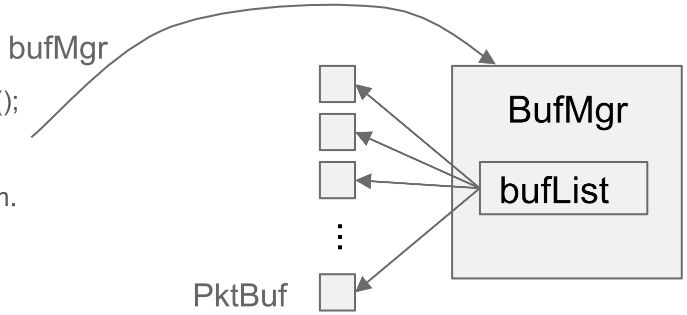
- Dot(.) operator is for ManagerObject
- Arrow(->) operator is for ManagedObject
- Examples
  - SPtr<Hello> myObj = CreateObject<Hello>("Hi!");
  - myObj.print(); // Error. print() is defined in managed object class. Should be myObj->print()
  - myObj.use_count(); // Ok
  - myObj.reset(); // Ok
- Avoid using the same function name in your Class as the ones in ManagerObject
  - e.g) If you have reset() function defined in your class and then you use like myObj.reset() instead of myObject->reset() by mistake, that will result in unexpected behavior which might be hard to debug.
- See the following link for details of ManagerObject methods http://www.boost.org/doc/libs/1_63_0/libs/smart_ptr/shared_ptr.htm

Get Shared Pointer of 'this' Object

- Use getThisPtr<TypeName>()

```
YAIL_BEGIN_CLASS(PktBuf, EXTENDS(YObject))
  public:
      void init(SPtr<BufMgr> bufMgr) {
          SPtr<PktBuf> thisPtr = getThisPtr<PktBuf>();
          registerBuffer(thisPtr);
      }
      ......

  private:
      unsigned char data[2048];
YAIL_END_CLASS

int main(int argc, char** argv) {
    SPtr<BufMgr> bufMgr = CreateObject<BufMgr>();
    for(int i = 0; i < 100; i++) {
        CrateObject<PktBuf>(bufMgr);
        // main() creates buffers and forget about them.
        // BufMgr should take care of them.
    }
    ......
}
```

```
YAIL_BEGIN_CLASS(BufMgr, EXTENDS(YObject))
  public:
      void init() { }
      void registerBuffer(SPtr<PktBuf> buf) {
          ......
      }

  private:
      List<SPtr<PktBuf>> bufList;
YAIL_END_CLASS
```

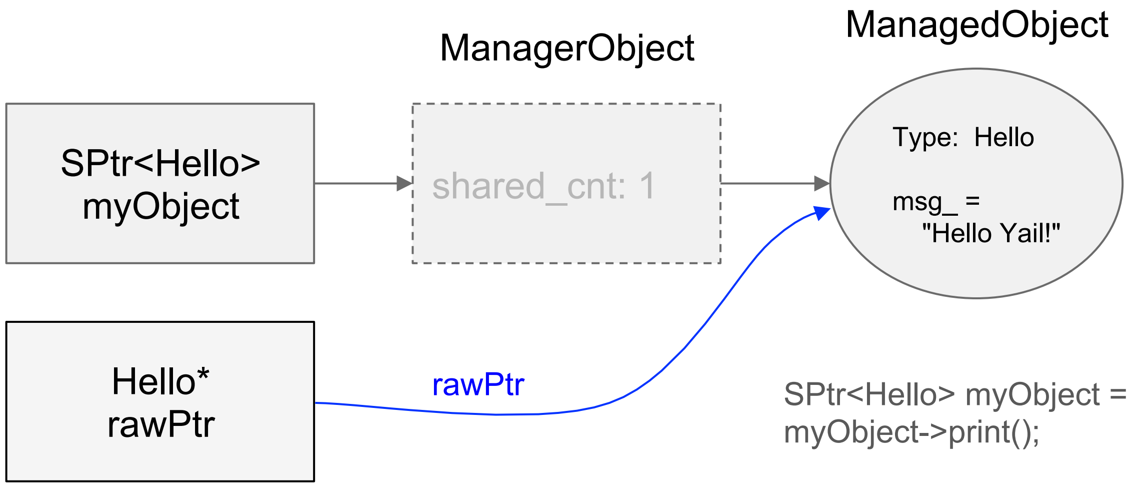


Pointer Type Casting

```
YAIL_BEGIN_CLASS(Parent, EXTENDS(YObject))
public:
   void init() {}
YAIL_END_CLASS

YAIL_BEGIN_CLASS(Child, EXTENDS(Parent))
 public:
   void init() {}
YAIL_END_CLASS

YAIL_BEGIN_CLASS(Other, EXTENDS(YObject))
 public:
   void init() {}
YAIL_END_CLASS

SPtr<Parent> parent = CreateObject<Parent>();
SPtr<Child> child = CreateObject<Child>();
SPtr<Other>  other = CreateObject<Other>();
SPtr<Parent> tmpParent;
SPtr<Child> tmpChild;
SPtr<Other>  tmpOther;
```

```
tmpParent = child; assert(tmpParent);
//tmpParent = other;  // compile error
//tmpChild = parent;  // compile error
//tmpChild = other;  // compile error
//tmpOther = parent;  // compile error
//tmpOther = child;  // compile error

tmpParent = StaticPointerCast<Parent>(child); assert(tmpParent); // upcasting ok
//tmpParent = StaticPointerCast<Parent>(other);   // compile error
tmpChild  = StaticPointerCast<Child>(parent); assert(tmpChild); // downcasting ok
//tmpChild  = StaticPointerCast<Child>(other);  // compile error
//tmpOther  = StaticPointerCast<Other>(parent);  // compile error
//tmpOther  = StaticPointerCast<Other>(child);  // compile error

// DynamicPointerCast is successful at compile time
// But it returns null at run time if type doesn't match.
tmpParent = DynamicPointerCast<Parent>(child); assert(tmpParent); // upcasting ok
tmpChild  = DynamicPointerCast<Child>(tmpParent); assert(tmpChild); // downcasting ok
tmpChild  = DynamicPointerCast<Child>(parent); assert(!tmpChild); // downcasting not allowed
tmpParent = DynamicPointerCast<Parent>(other); assert(!tmpParent);
tmpChild  = DynamicPointerCast<Child>(other);  assert(!tmpChild);
tmpOther  = DynamicPointerCast<Other>(parent); assert(!tmpOther);
tmpOther  = DynamicPointerCast<Other>(child);  assert(!tmpOther);
```

Getting Raw Pointer



```
SPtr<Hello> myObject = CreateObject<Hello>("Hello Yail!");
myObject->print();

Hello* rawPtr = myObject.get();
rawPtr->print(); // Ok

myObject.reset(); // ManagedObject will be destroyed here
assert(rawPtr != NULL); // But rawPtr is not aware of that.
......
rawPtr->print(); // Crash or abnormal behavior
```

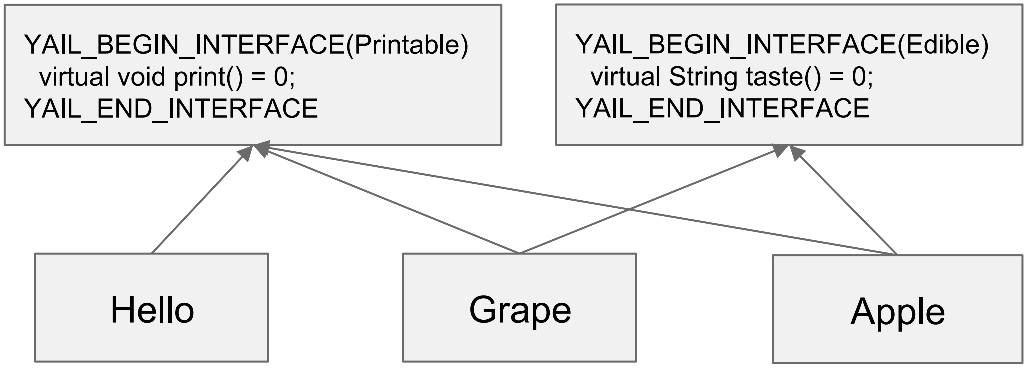
Smart Pointers in Multi-threaded Environment

- Smart pointers should work in multi-threaded environment because the automatic reference counting actions are occurring in object constructor/ destructor which is thread-agnostic.
- But care must be taken and proper mutual exclusion is required to avoid context switching in the middle of ManagerObject operations.

Interface (1)

```
YAIL_BEGIN_INTERFACE(InterfaceName)
                virtual <return_type> methodSignature(<args...>) = 0; // pure virtual function
                // int intData; <- DO NOT define data member in Interface
YAIL_END_INTERFACE
```

Interface (2)



```
YAIL_BEGIN_INTERFACE(Printable)
  virtual void print() = 0;
YAIL_END_INTERFACE
```

```
YAIL_BEGIN_INTERFACE(Edible)
  virtual String taste() = 0;
YAIL_END_INTERFACE
```

Hello

Grape

Apple

```
YAIL_BEGIN_CLASS(Hello,
             EXTENDS(YObject),
             IMPLEMENTS(Printable))
  public:
    void init(String msg) { msg_ = msg; }
    void print() override { // must implement print()
      cout << msg_ << endl;
    }

  private:
    String msg_;
YAIL_END_CLASS
```

```
YAIL_BEGIN_CLASS(Grape,
             EXTENDS(YObject),
             IMPLEMENTS(Printable),
             IMPLEMENTS(Edible)) {
  public:
    void init() {}

    // must implement print() and taste()
    void print() override {
      cout << "A grape is a fruit, botanically a
berry" << endl;
    }
    String taste() override {
      return "sour";
    }
YAIL_END_CLASS
```

```
YAIL_BEGIN_CLASS(Apple,
             EXTENDS(YObject),
             IMPLEMENTS(Printable),
             IMPLEMENTS(Edible))
  public:
    void init() {}

    // must implement print() and taste()
    void print() override {
      cout << "American multinational
technology company headquartered in
Cupertino" << endl;
    }
    String taste() override {
      return "delicious";
    }
YAIL_END_CLASS
```

Interface (3)

```
void showMe(SPtr<Printable> obj) {
    obj->print();
}

String  tasteOf(SPtr<Edible> obj) {
    return obj->taste();
}
```

```
SPtr<Hello> hello = CreateObject<Hello>("Hello World!");
SPtr<World> grape = CreateObject<Grape>();
SPtr<Apple> apple = CreateObject<Apple>();

showMe(hello); // hello is a Printable
showMe(grape); // grape is a Printable
showMe(apple); // apple is a Printable

// grape is an Edible
cout << "Grape is " << tasteOf(grape) << endl;
// apple is an Edible
cout << "Apple is " << tasteOf(apple) << endl;
```