RFC Sample

Leonard Moşescu
(lemo@metasoft.ro)

1 General guidelines

LFC development is intended to be an open process which should benefit from the best ideas and experience from all of the team members. Request For Comments (RFC) are descriptions of new ideas and proposals and they are intended to:

1. request feedback on the proposed solution
2. keep the rest of the team up-to-date with the new stuff in LFC
3. force the developer to put order in his/her ideas
4. serve as base for the implementation/doc/tests

Each new concrete idea/proposal that will affect LFC should be described in a RFC. There can be multiple incremental versions of an RFC before a final solution is proposed for approval. An RFC could describe not only technical solutions, but ideas for project organization, LFC marketing, ...

An RFC should contain:

1. descriptive name of the RFC
2. clear description of the problem it tries to solve
3. a complete set of requirements
4. an usage example of the solution (code if the solution is part of LFC)
5. the solution (without implementation details, except for the complexity class of the algorithm used when is the case, side effects of some functions, ...)
6. known limitations of the solution

Notes:

- not all RFCs will have the exact same structure (for example an RFC that describes non technical solution might not contain code samples, ...)
- try not to construct the description of the problem and the set of requirements as to fit the proposed solution. Those two parts are very important, and it's normally that in some cases the solution will not fit all the requirements
- the last, but certainly not the least, RFCs are a very important piece of the design puzzle and should be intensively used

2 An RFC Sample

Name **LFC2 Smart Pointers (rev2)**

Description
Smart pointers are objects that (try to) behave like build-in C++ pointers (aka dumb pointers) adding extra functionality.

As you know, pointers are extremely important in C++, but working with pointers and dynamically allocated objects has some serious drawbacks:

- using uninitialised pointers
- dangling pointers
- memory leaks

I propose here a design of a reference-counting smart pointer variant that try to address the problems above, offering:

- auto init to NULL
- auto free memory when unreferenced
- detect NULL pointers indirection

Requirements

- easy to use
- non intrusive
- small speed, mem overhead
- good integration with LFC:
  - callbacks
  - persistent streams
  - ...
- thread safe (?)
- look and feel like build-in pointers as much as possible
  - `->` and `*` operators
  - equivalent conversions with build-in pointers
  - point to const/nonconst objects
  - ...

Known limitations

- doesn't support operator `->*`
- can't be transparently mixed with build-in pointers
- some conversions ambiguities (for ex. the direct base class is not proffered to some indirect base class in overload resolution)
- it's not thread safe (yet?)
- support for 'volatile' modifier(?)
- support for object arrays(?)
- cyclic references(!)

Some usage examples

```
// smart pointer declarations
Ptr<Base> sp1; // null initialized
Ptr<Deriv> sp2(new Deriv); // point to new Base object
Ptr<Base> sp3(sp2); // init form another smart ptr (note Deriv -> Base conversion)
//Ptr<Base> sp4 = new Base; // fail:  Ptr<T>(T*) is explicit
Ptr<Deriv> sp4 = lfcNull; // init to NULL (useful for func args)
Ptr<const Base> sp5 = sp3; // Ptr to const object (note non-const -> const conversion)
Ptr<double> sp6; // Ptr to a build-in type

// basic usage
sp2->f();
sp2->x = 10;
Deriv obj = *sp2;
//sp1->f(); // will throw an exception (null dereferencing)
//*sp1; // idem

// copying and conversions
// (similar to build-in pointers, if T1* -> T2* then Ptr<T1> -> Ptr<T2>)
sp1 = sp2; // ok, Ptr<Deriv> -> Ptr<Base>
//sp2 = sp3; // fail, Ptr<Base> -> Ptr<Deriv> is illegal
sp2 = sp2.dynamicCast<Deriv>(); // ok, dynamic cast (may return a null ptr)

// comparations (2 smart ptr are equal if they point to the same object)
sp1 == sp2; // false
sp1 == sp3; // true
sp1 == lfcNull; // true
lfcNull == sp1; // true
```

```
sp1.isNull(); // true

// special usage
Base *p = sp2.realPointer(); // use with caution!

// now, all unreferenced objects
// will be automaticaly deleted
```

My proposed solution (`Ptr<T>`*)*

```
//!  Ptr<T> represent a smart pointer to a T object
template<class T>
class Ptr :  private PtrBase
{
public:
     explicit Ptr(T *p = NULL) throw();
     Ptr(const Ptr &ptr) throw();
     Ptr(Null) throw();
     ~Ptr() throw();
     template<class X>
     Ptr(const Ptr<X> &ptr) throw();

public:
     const Ptr &operator=(const Ptr &ptr) throw();
     template<class X>
     const Ptr &operator=(const Ptr<X> &ptr) throw();
     const Ptr &operator=(T *p) throw();
     template<class X>
     const Ptr<X> dynamicCast() const;

     bool isNull() const throw();
     T *operator->() const;
     T &operator*() const;
     T *realPointer() const;
};

template<class T1, class T2>
inline bool operator==(const Ptr<T1> &p1, const Ptr<T2> &p2) throw();
template<class T>
inline bool operator==(const Ptr<T> &ptr, Null) throw();
template<class T>
inline bool operator==(Null, const Ptr<T> &ptr) throw();
```

I would appreciate any comments (especially regarding the thread-safe issue).
Lemo.