

# A Smart Pointer Capable of Object Level Thread Synchronization and Reference Counting Garbage Collection

---

This article was contributed by **Stefan Chekanov**.

- [What is new](#)
- [Introduction](#)
- [Examples of how to use SmartPtr](#)
- [How does SmartPtr work?](#)

## What is new

- SmartPtr objects can accept different pointer types - [details](#)
- Passing a SmartPtr object as function parameter or function result is as efficient as passing an integer value or regular pointer - [details](#).

## Introduction

The SmartPtr class is a generic wrapper that encapsulates pointers. It keeps track of the total number of reference counts and performs garbage collection when the pointed object is no more referenced by any "pointer". In addition SmartPtr can do Object Level Thread Synchronization for pointed objects. This means that calling the members of the pointed object is enclosed in Windows critical section and only one thread at a time can use the object.

SmartPtr stores reference counter and critical section in separate memory block - not in the pointed object. This causes that you don't have to inherit your classes from special base one to be able to use SmartPtr as a pointer. And one other think is that you could use SmartPtr to point objects of scalar types - int, short, long, double ....

SmartPtr is implemented as a template class. Because it has three template parameters it is annoying to specify all of them each time. This is the reason I have defined three additional template classes inherited from SmartPtr with appropriate default template parameters to get three diferent behaviours:

- **Reference Counting Garbage Collection** - SmartPtr will free dynamically allocated memory automatically.
- **Synchronized Access without Reference Counting Garbage Collection** - SmartPtr will perform only Thread synchronization. You should free dynamically allocated memory in your own.
- **Synchronized Access with Reference Counting Garbage Collection** - SmartPtr will perform thread synchronization and will free dynamically allocated memory for you.

These are the classes:

Class Name	Description
<b>SmartPtrBase</b>	The SmartPtrBase class is a non template base class for the SmartPtr class. It should never be used directly. SmartPtr uses it to perform appropriate type casting
<b>SmartPtr</b>	The SmartPtr class is a template base class for the next classes
<b>RefCountPtr</b>	Smart pointers that perform only Reference Counting Garbage Collection
<b>SyncPtr</b>	Smart pointers that perform only Synchronized Access without Reference Counting Garbage Collection
<b>SyncRefCountPtr</b>	Smart pointers that perform both Synchronized Access and Reference Counting Garbage Collection

All of these classes are equal in their implementation. The different behaviour is achieved via different default template parameters. You could use any of these four classes to get any of the three behaviours but then you should specify all parameters.

The following classes are used inside the SmartPtr implementation and SHOULD NEVER be used directly.

- CRefCountRep
- CSyncAccessRep
- CSyncRefCountRep
- CSyncAccess

## Examples of how to use SmartPtr

You should include the file `SmartPtr.h` in your project.

I usually include this line in my `StdAfx.h` file.

```
#include "SmartPtr.h"
```

Let's have a class `CSomething`

### 1. Reference Counting Garbage Collection on `CSomething` class

```
class CSomething {
    CSomething();
    ~CSomething();
    .....
    void do();
};

typedef RefCountPtr<CSomething> LPSOMETHING;

void TestFunc() {
    LPSOMETHING p1 = new CSomething;
    LPSOMETHING p2 = p1;

    if( p1 == NULL ) {
        ....
    }

    p2->do();
    p1 = NULL;

}

// Here the object pointed by p2 WILL BE destroyed automatically
///////////////////////////////////////////////////////////////////
```

### 2. Object Level Thread Synchronization for objects of `CSomething`

```
class CSomething {
    CSomething();
    ~CSomething();
    .....
    void do();
};

typedef SyncPtr<CSomething> LPSOMETHING;

void TestFunc() {
    LPSOMETHING p1 = new CSomething;
```

```

LPSOMETHING      p2 = p1;

if( p1.IsNull() ) {
    ....
}
StartThread( p1 );

p2->do();          //      Synchronized with the other thread
p1 = NULL;

}      //      Here the object pointed by p2 will NOT be destroyed automatically

void      ThreadFunc( LPSOMETHING p ) {
    p->do();          //      Synchronized with the other thread
}//      Here the object pointed by p will NOT be destroyed automatically
////////////////////////////////////

```

In this example you will get memory leaks, but the two threads will be synchronized when trying to call object's members. It is your care to free dynamically allocated memory.

### 3. Object Level Thread Synchronization and Reference Counting Garbage Collection for objects of CSomething

```

class      CSomething {
    CSomething();
    ~CSomething();
    ....
    void      do();
};

typedef SyncRefCountPtr<CSomething>      LPSOMETHING;

void      TestFunc() {
    LPSOMETHING      p1 = new CSomething;
    LPSOMETHING      p2 = p1;

    if( p1.IsNull() ) {
        ....
    }
    StartThread( p1 );

    p2->do();          //      Synchronized with the other thread
    p1 = NULL;

}//      Here the object pointed by p2 WILL BE destroyed automatically
//      if p in ThreadFunc has already released the object

void      ThreadFunc( LPSOMETHING p ) {
    p->do();          //      Synchronized with the other thread
}//      Here the object pointed by p WILL BE destroyed automatically
//      if p2 in TestFunc has already released the object
////////////////////////////////////

```

In this example you will not get memory leaks and the two threads will be synchronized when trying to call object's members. You don't have to free dynamically allocated memory. SmartPtr will do it for you.

### How does SmartPtr work?

The definition of SmartPtr is:

```

template<class T, class REP=RefCountRep<T>, class ACCESS = T*>
class SmartPtr : public SmartPtrBase {

```

```
.....  
};
```

Where *T* is the type of the pointed objects, *REP* is the representation class used to handle the pointers and *ACCESS* is the class used to get thread safe access to the underlying real pointer.

There is nothing interesting about Reference Counting Garbage Collection. It is a standard implementation. The only interesting thing is that the reference counter and the real pointer are stored in the representation object instead of the pointed object.

The more interesting thing is how Object Level Thread Synchronization works. Let's look at the definition of SyncPtr class:

```
template<class T, class REP=CSyncAccessRep<T>, class ACCESS=CSyncAccess<T> >  
class SyncPtr {  
.....  
    ACCESS operator -> ();  
};
```

Look at the reference operator. It returns object of type ACCESS, which by default is of type CSyncAccess<T>. The trick is that if the reference operator returns something different than a real pointer, the compiler calls the operator ->() of the returned object. If it returns again something different than a real pointer compiler calls the returned object operator ->(). And this continues till some reference operator returns a real pointer like T\*. Well, to get object level synchronization I use this behaviour of the reference operator. I have defined a class CSyncAccess<T>:

```
template <class T>  
class CSyncAccess {  
.....  
    virtual ~CSyncAccess();  
    T*      operator -> ();  
};  
////////////////////////////////////
```

The representation class used in this scenario has a member of type CRITICAL\_SECTION.

```
template <class T>  
class CSyncAccessRep {  
.....  
    CRITICAL_SECTION      m_CriticalSection;  
};  
////////////////////////////////////
```

And now let's look at the example code:

```
typedef SyncPtr<CSomething>      LPSOMETHING;  
LPSOMETHING      p = new CSomething;  
  
p->do();
```

What's going when `p->do()` is called?

1. The compiler calls `SyncPtr::operator->()` which returns object of type `CSyncAccess`. This object is temporary and is stored on top of the stack. In its constructor the critical section object is initialized.
2. The compiler calls `CSyncAccess::operator->()` which owns the critical section and thus protects other threads to own it and then returns the real pointer to the pointed object
3. The compiler calls the method `do()` of the pointed object
4. The compiler destroys `CSyncAccess` object which is on the stack. This calls its destructor where the critical section is released and other threads are free to own it.

The good thing about SmartPtr is that pointed objects do not have to be inherited from any special base class as in many other reference counting implementations. The reference counter and the real object pointer are stored in dynamically allocated objects of types CRefCountRep, CSyncAccessRep and CSyncRefCountRep (called representation objects) depending on what kind of SmartPtr we have. These objects are allocated and freed by SmartPtr. This approach allows us to have SmartPtr for objects of any classes, not only for inherited from special base class. Well, this is the weakness of the SmartPtr too. Imagine that you have a piece of code like this:

```
LPSOMETHING    p1 = new CSomething;
CSomething*    p2 = (CSomething*)p1;
LPSOMETHING    p3 = p2;
```

As result, at the end of this piece of code you think you will have p1 and p3 point to the same object. Yes, they will. But they will have different representation objects in p1 and p3. So when p3 is destroyed the underlying CSomething object will be destroyed too and p1 will point to invalid memory. So I have a simple **tip: If you use SmartPtr never use real pointers to underlying objects** like in the code above. But if you want you can still use real pointers when dealing with other objects.

Let's look at this code:

```
LPSOMETHING    p1 = new CSomething;
SomeFunc( p1 );

void    SomeFunc( CSomething* pSth ) {
    LPSOMETHING    p3 = pSth;
}
```

p3 will create its own representation, ie. its own reference counter and when SomeFunc exits, p3 will free the memory pointed by p1. Instead of this function definition you'd better define it this way.

```
void    SomeFunc( LPSOMETHING pSth );
```

I talk about these to notify you that calling SmartPtr *constructor* or *operator =* with T\* parameter creates new representation object and this will lead to "strange" behaviour of SmartPtr.

The mentioned above weakness of the SmartPtr behaviour could be solved if we add static table of associations - T\* <-> CxxxxRep\* and every time SmartPtr gets T\* we could search in this table and get appropriate CxxxRep object if it is already created. But this will lead to more memory and CPU overhead. And since now I think it is better to follow mentioned above rule than overhauling.

## SmartPtr is as efficient as regular pointer and integer values.

Here is a table with some sizeof() results:

	Size in bytes
sizeof( SmartPtr )	4
sizeof( CSomething* )	4
sizeof( int )	4

SmartPtr objects have the same size as pointers and int values. So, passing a SmartPtr objects as function (method) arguments or returning SmartPtr object as function result is as efficient as doing the same with regular pointer or int value.

## SmartPtr objects can accept different pointer types.

SmartPtr class has a non template base class SmartPtrBase, which is made argument of various constructors and assignment operators. Thus a code like this is possible to be used

```
class B {
    ....
};

class A : public B {
    ....
};

class C {
    ....
};

RefCountPtr<A> a = new A;
RefCountPtr<B> b = a;
b = a;
```

Even this is very good feature in cases where you need a pointer to base class to hold objects of inherited classes you may have a code like this

```
RefCountPtr<A> a = new A;
RefCountPtr<B> b = new B;
SyncPtr<C> c = a;
a = b;
a = c;
c = a;

c = b;
```

And the compiler will not warn you there are implicit type conversions. Note that there is no relation between **class C** and **class B**.

In fact, SmartPtr implicitly passes underlying representation objects in such assignments. The behaviour of the pointer depends on the underlying representation object. So, no matter what is the intention of the holder pointer (in the example above "c" is a synchronization pointer) the pointer behaviour depends on the type of the representation object creator.

In the example above, after the last line (c = b;), "c" will hold the object pointed by "b", and even "c" is declared as "synchronization" pointer when it deals with its object it will behave as "reference counting" pointer, because the original creator is "b" which is of such type.

The enclosed demo project is a simple console application and demonstrates different situations and usage of SmartPtr. It is created by VC ++ 5.0. SmartPtr.h can be compiled with warning level 4.

That's all.