Automatic Garbage Collection in C++ using Smart pointers

By **abc876** | 18 May 2003

Licence
First Posted **18 May 2003**
Views **61,106**
Downloads **654**
Bookmarked **27 times**

VC6 VC7 Win2K WinXP MFC Dev Intermediate

This article explains how to prevent your programs from memory leaks, by incorporating Garbage Collector in your class.

- Download source files - 4.25 Kb
- Download demo project - 2.81 Kb

Introduction

C++ is the most popular language around. Although many people have shifted to other high level languages like VB and Java, it is still the language of choice for system programming and the situations where performance can never be compromised.

C++ offers great features for dynamic memory allocation and de-allocation, but you can hardly find any C++ programmer who hasn't been bugged by memory leaks in C++ programs. The reason is that in C++, you have to de-allocate memory yourself which is, at times, bug provoking.

Some high level languages like Java and C# provide automatic memory de-allocation facility. These languages have a built in Garbage Collector which looks for any memory which has no further reference in the program and de-allocates that memory. So programmers don't have to worry about memory leaks. **How about having Garbage Collection facility in C++ programs?**

Well, we can implement simple garbage collection facility in our programs using smart pointers, but it comes at some cost. There is a small overhead of an extra integer and a character variable per instance of classes for which you implement garbage collection. All source code for these garbage collection classes are provided with this article.

What are smart pointers?

I said we can implement garbage collection using "smart" pointers. But actually what are "smart" pointers?

C++ allows you to create "smart pointer" classes that encapsulate pointers, and override pointer operators to add new functionality to pointer operations. Template feature of C++ allows you to create generic wrappers to encapsulate pointers of almost any kind.

An example of smart pointers with templates could be a class which encapsulates double dimension arrays in C++.

```
template< class T> class ArrayT
{
  T** m_ptr; // for double dimension
  int sx,sy; // size of dimensions of array
  public:
  ArrayT( int dim1, int dim2) // in case of ArrayT<float> b(10,20)
  {
    sx=dim1;sy=dim2;
    m_ptr= new T*[dim1];
    for( int i=0;i<dim1;i++)
      m_ptr[i]=new T[dim2];
  }
  T* operator[] ( < pan class=cpp-keyword>int index)
      { return m_ptr[index]; } // to add [] operator to ArrayT

  ~ArrayT()
  {
    for( int d=0;d<sx;d++)
      delete [] m_ptr[d];
    delete m_ptr;
  }
};
```

This class encapsulates the functionality of double dimension arrays for any object in C++. You can extend the functionality of 2D arrays in this manner. Using the same technique, you can also implement STL style resizable collection classes like vector.

Now, coming back to garbage collection using smart pointers, how can we use smart pointers for garbage collection within the class?

What our Garbage Collector does?

We are embedding the garbage collection feature within a particular class. This simple garbage collector de-allocates memory when an object is no longer referenced, hence preventing memory leaks. This is really simple to implement.

We are using reference counting mechanism. Whenever there is a new reference to an object, we increment the reference count, and when it is no longer referenced, i.e. reference count=0, we de allocate the memory.

Implementation

The template class `gcPtr<T>` implements a garbage collecting pointer to any class derived from `RefCount` class.

```
template <class T>class RefCount
{
  protected:

  int refVal;
  T* p;

  public:

  RefCount(){ refVal=0; p=(T*)this;}
  void AddRef() { refVal++; }
```

```
  void ReleaseRef()
  {
    refVal--;
    if(refVal == 0)
      delete [] this;
  }

  T* operator->(void) { return p; }
  // Provide -> operator for class inheriting

  // RefCount
  // Similarly you can add other overloaded
  // operators in this class //so that
  // you dont have to implement them again
  // and again. //Once you have added
  // these operators, they will be available
  // to all classes // inheriting from RefCount
  // to incorporate Garbage Collection.
};
```

This class implements simple reference counting. Any class which wishes to implement garbage collection should derive from this class. Note that `RefCount` takes a template parameter. This parameter is used to overload `->` operator for the class which inherits from `RefCount` class to implement garbage collection. I.e.,

```
    class foo : public RefCount<foo><FOO>
    {
    };
```

`gcPtr` is a template class using smart pointers which encapsulates all garbage collection processes.

```
template <class T> class gcPtr
{
  T* ptr;
  char c;

  public:

  gcPtr()
  {
    c='0'; // called when variable declare // as gcPtr<foo> a;
  }
  gcPtr(T* ptrIn)
  {
    ptr=ptrIn;
    // called when variable declared
    // as gcPtr<foo> a=new foo;
    ptr->AddRef();
    c='1';
  }
  // assuming we have variable gcPtr<foo> x
  operator T*(void) { return ptr; } //for x[]
  T& operator*(void) { return *ptr; } // for *x type operations
  T* operator->(void){ return ptr; } // for x-> type operations
  gcPtr& operator=(gcPtr<T> &pIn)
  // for x=y where y is also gcPtr<foo> object
  {
    return operator=((T *) pIn);
  }
```

```
  gcPtr& operator=(T* pIn)
  {
    if(c=='1')
    // called by gcPtr& operator=(gcPtr<T>&pIn) in case of
    { // assignment
      // Decrease refcount for left hand side operand
      ptr->ReleaseRef();
      // of '=' operator
      ptr = pIn;
      pIn->AddRef(); // Increase reference count for the Right Hand
      // operand of '=' operator
      return *this;
    }
    else
    // if c=0 i.e variable was not allocated memory when // it was declared
    { // like gcPtr<foo> x. in this case we //allocate memory using new
      // operator, this will be called
      ptr=pIn;
      ptr->AddRef();
      return *this
    }
  }
  ~gcPtr(void) { ptr->ReleaseRef(); } // Decrement the refcount
};
```

Now, let's see what happening in `gcPt` class template. There are two constructors. `gcPtr()` is for cases when you are just declaring the variable but not assigning memory to it using `new` operator. `gcPtr(T*)` is for cases when you are assigning memory to `gcPtr` variable on declaration using `new` operator. We have overloaded `T*` operator to provide support for array notation `[]`. This returns us the `T*` type pointer which our class is encapsulating. `->` operator is also overloaded to provide support for pointer operations like `x->func()`. This operator also returns the `T*` pointer. An interesting thing is happening in the case of assignments like `x=y`. `gcPtr& operator=(gcPtr<T> &pIn)` function is called, which in turn calls `gcPtr& operator=(T* pIn)` function, and only the '`if`' block of this function is executed. Check this code:

```
gcPtr& operator=(gcPtr<T> &pIn) // for x=y where y is also gcPtr<foo> object
{
  return operator=((T *) pIn);
}
```

We are type casting the input `gcPtr` parameter to `T*`. If `x=y` was the assignment (where `x` and `y` are variables of `gcPtr<FOO>`), this means we are type casting `y` to `foo*` and calling `gcPtr& operator=((foo*) y)` function. Now, let's see how garbage collection mechanism is implemented in this function. Check this '`if`' block code:

```
  ptr->ReleaseRef();
  // Decrement the reference count for 'x'. Now if the
  //reference count is zero, memory for x will
  // be de-allocated

  ptr = pIn; // assign (foo*) y to ptr which is
         //of type foo

  pIn->AddRef(); // Since we have made a new reference
                 //to (foo*) y, increment
                //Reference count for (foo*)y
```

```
return *this; // return the x variable by reference.vv
```

So, this explains how garbage collection is done when an object is no longer referenced.

How to use Garbage collection in your project?

Follow these steps to implement this simple garbage collection mechanism in your programs. It's really simple.

1. Add `RefCount` and `gcPtr` classes to your project.
2. Derive your class from `RefCount`, passing the name of your class to `RefCount` as template parameter.

   For example:

   ```
   class foo: public RefCount<FOO><FOO>
   {
      public:
      void func() { AfxMessageBox("Hello World") };
   };
   ```

3. Now, to declare a pointer to your class, use `gcPtr <T>` class, passing the name of your class as template parameter. `gcPtr<T>` is a "smart" pointer. After allocating memory using `new` operator in one of the methods shown below, you can use `gcPtr` just like your class pointer. It supports all pointer operations. You don't need to de-allocate memory using `delete` operator. It will be handled by `gcPtr` class.

   ```
   //First Method

   gcPtr<FOO> obj; // simple declaration
   obj=new foo; // Assign memory to obj
   obj->func();
   //Second Method
   gcPtr<FOO> obj=new obj;
   obj->func();
   ```

4. You can also declare an array of your class using `gcPtr<T>` class:

   ```
   gcPtr<FOO>obj= new obj[5];
   obj[2]->func();
   ```

   => Memory will be de-allocated when your object is no longer referenced. For example:

   ```
   gcPtr<FOO> obj1=new obj;
   // increment reference count for obj1 i.e refcount=1
    gcPtr<FOO> obj2=new obj; // increment reference count for obj2
   // i.e refcount=1
    obj1=obj2; // decrement reference count for obj1
   //i.e refcount=0 and increment
    // reference count for obj2 i.e refcount=2 . Memory will
   //be de-allocated
    // for obj1. Now obj1 to the same memory as obj2
    // When destructor for obj1 is called, it will decrement
   //the referencecount to 1, and
   ```

```
//when destructor for obj2 is called, it will decrement it
// to 0 and memory is de-allocated.
```

That's it. Isn't it too simple to incorporate garbage collection and prevent memory leaks in C++? Of course; **yes**. What all you need is, to derive your class from `Refcount`.

How to implement garbage collection for built in data types?

To implement garbage collection for built-in data types like `int`, `float` etc., you can write a simple wrapper for them by overloading the operators.

```
class MyInt : public gcPtr<MYINT>
    {
      public:
        int* val;
    ..... // overloaded operators like ++,--,> etc
    ..... // or overload these operators in
          //RefCount base class
// so that they will be available to any class inheriting
// incorporating garbage collection
};
```

Instead of every time adding all operator overloads in your class, you can once modify `RefCount` class and provide overloaded operators for `T` data type, as one overloaded operator is already provided there: i.e., `T* operator->(void) { return p; }`.

Garbage collection in Action

Now, let's see Garbage Collection in action in a simple MFC Dialog Box application. Run your program in Debug mode and start debugging. If I am using garbage collection, here is the output from the output window:

```
    Loaded 'C:.DLL', no matching symbolic information found.
    Loaded 'C:.DLL', no matching symbolic information found.
    The thread 0x744 has exited with code 0 (0x0).
    The program 'D:.exe' has exited with code 0 (0x0).
```

There is no memory leak. Now, if I am neither doing garbage collection, nor I am manually deleting the object, here is the output:

```
    Loaded 'C:.DLL', no matching symbolic information found.
    Loaded 'C:.DLL', no matching symbolic information found.
    Detected memory leaks!
    Dumping objects ->
    {65} normal block at 0x002F2C80, 16 bytes long.
    Data: < ,/ > 00 00 00 00 80 2C 2F 00 CD CD CD CD CD CD CD CD
    Object dump complete.
    The thread 0x50C has exited with code 0 (0x0).
    The program 'D:.exe' has exited with code 0 (0x0).
```

You can see the memory dumps. There is memory leak of 16 bytes.

I hope you enjoyed this article J. This is my first article on Code Project. Feel free to email me if you have any suggestions or if you have any problems using these classes + **don't forget to vote for me if you find this article useful :).**