LECTURE 13
Reference Counting

If you have large objects that take a lot of memory (e.g. large matrices), it can really slow things down if you have to copy these objects to different parts of memory. This can occur when the copy constructor and the assignment operator are used. It is much more efficient to move pointers around. For example, if we add 2 large matrices, A and B, the sum is a large matrix D. (A + B = D) We can overload the `+` sign to do this operation. The result of the addition is stored in a temporary matrix which then gets copied into the permanent matrix D. It would be nice to avoid copying the large matrix D and just access the temporary matrix with a pointer. But the temporary matrix will disappear as soon as the addition is finished and the addition function goes out of scope. *Reference counting* is a way of getting around this. It keeps count of the number of pointers pointing at an object, and won't let the object be destroyed until there are no pointers pointing at it. Let's consider the following example where we revisit the `Array` class from chapter 4. I have added to this class a `friend` function that overloads the addition operator to allow us to add 2 matrices.

```
// ***************************This is array.h************************

class Array
    {
public:
    Array(int n);                          // Constructor:
                                           //   Create array of n elements
    Array();                               // Default constructor:
                                           //   Create array of 0 elements
    ~Array();                              // Destructor
    Array(const Array &);                  //   Copy constructor

    int numElts() const;                    // Number of elements
    Array & operator = (const Array &); // Array assignment
    Array & operator = (double);            // Scalar assignment
    void setSize(int n);                   // Change size

    double &operator[] (int i);             // Subscripting
    const double &operator[] (int i) const;   // Subscripting

    friend Array operator + (const Array &, const Array &);
                                           //Overload addition operator
private:
    int num_elts;                          // Number of elements
    double *ptr_to_data;                    // Pointer to data
    void copy(const Array & a);            // Copy in elements of a
```

```
    };

void error(const char *s);              // For bounds checking

// ***************************This is array.cc********************
#include <iostream.h>
#include "array.h"

Array::Array(int n)
    {
    num_elts = n;
    ptr_to_data = new double[n];
    }
Array::Array()
    {
    num_elts = 0;
    ptr_to_data = 0;
    }

Array::~Array()                         // Destructor
    {
    delete[] ptr_to_data;
    }

Array::Array(const Array& a)            // Copy constructor
    {
    num_elts = a.num_elts;
    ptr_to_data = new double[num_elts];
    copy(a);                            // Copy a's elements
    }

void Array::copy(const Array& a)
    {
    // Copy a's elements into the elements of *this
    double *p = ptr_to_data + num_elts;
    double *q = a.ptr_to_data + num_elts;
    while (p > ptr_to_data)
         *--p = *--q;
    }

double& Array::operator[](int i)
    {
```

```
    #ifdef CHECKBOUNDS
      if(i<0 || i>num_elts)
        error("out of bounds");
    #endif
    return ptr_to_data[i];
    }

const double& Array::operator[](int i) const
    {
    #ifdef CHECKBOUNDS
      if(i<0 || i>num_elts)
        error("out of bounds");
    #endif
    return ptr_to_data[i];
    }

int Array::numElts() const
    {
    return num_elts;
    }

Array& Array::operator=(const Array& rhs)
    {
    if (ptr_to_data != rhs.ptr_to_data)
        {
        setSize(rhs.num_elts);
        copy(rhs);
        }
    return *this;
    }

void Array::setSize(int n)
    {
    if (n != num_elts)
       {
        delete[] ptr_to_data;               // Delete old elements,
        num_elts = n;                       // set new count,
        ptr_to_data = new double[n];         // and allocate new elements
       }
    }

Array& Array::operator=(double rhs)
```

```
    {
    double *p = ptr_to_data + num_elts;
    while (p > ptr_to_data)
        *--p = rhs;
    return *this;
    }

void error(const char *s)
    {
    cerr << endl << s << endl; //1st "endl" is in case program is printing
    cout << endl << s << endl; //something out when the error occurs
    cout.flush();              //write the output buffer to the screen
                               //or wherever the output of cout goes.
    abort();
    }

Array operator + (const Array &a1, const Array &a2)
    {
     if(a1.numElts()==a2.numElts())
       {
        Array sum(a1.numElts());
        int i;
        for(i=0;i<a1.numElts();i++)
           sum[i] = a1[i] + a2[i];
        return sum;
       }
     else
       error("can't add arrays of different lengths");
     }
// *************************This is testarray.cc*************
#include <iostream.h>
#include "refarray.h"

int main()
  {
   int max=5;
   Array matrixA(max);
   Array matrixB(max);
   int i;
   for(i=0;i<max;i++)
    { matrixA[i]=i;                          //matrixA={0 1 2 3 4}
      cout << matrixA[i] << " ";
```

```
    }
      cout << endl;
      for(i=0;i<max;i++)
    { matrixB[i]=i+4.0;                          //matrixB={4 5 6 7 8}
      cout << matrixB[i] << " " ;
    }
      cout << endl;

   Array result = matrixA + matrixB;             // invokes copy constructor
   for(i=0;i<max;i++)
    { cout << result[i] << " ";}
      cout << endl;

   result = matrixA;                           // invokes assignment operator
   for(i=0;i<max;i++)
    { cout << result[i] << " ";}
      cout << endl;

   return 0;
   }
```

The result of the addition is a temporary matrix `sum` that will disappear when the addition operator goes out of scope. So we copy the result into `result`. This copying is time consuming. Rather than copying the result of adding 2 matrices into `result`, let's give `result` a pointer to the sum. This saves time and space in memory. To do this we use reference counting. This keeps count of the number of pointers pointing to each array, including `result`. Reference counting won't let a matrix be destroyed until we don't need it anymore, i.e., until there are no more pointers pointing at it. So let's modify the `Array` class to include reference counting:

```
// **********************This is refarray.h**********************

class Array
    {
public:
    Array(int n);                          // Constructor:
                                           //   Create array of n elements
    Array();                               // Default constructor:
                                           // Create array of 0 elements
    ~Array();                              // Destructor
    Array(const Array &);                  // Copy constructor

    int numElts() const;                    // Number of elements
```

```
    Array & operator = (const Array &);  // Array assignment
    Array & operator = (double);         // Scalar assignment
//    void setSize(int n);               // Change size; comment out
    double &operator[] (int i);          // Subscripting
    const double &operator[] (int i) const;   // Subscripting

    friend Array operator + (const Array &, const Array &);
                                         //Overload addition operator
private:
    int num_elts;                       // Number of elements
    double *ptr_to_data;                 // Pointer to data
//    void copy(const Array & a);         // Copy elements of a; comment out

    int *refcnt;                        // Pointer to integer that counts
                                        // number of pointers pointing to
                                        // object
    void decrement();                   // decrement refcnt
    };                                  // destroy object if refcnt=0

void error(const char *s);              // For bounds checking
//**********************This is refarray.cc************************
#include <iostream.h>
#include "refarray.h"

Array::Array(int n)
    {
    num_elts = n;
    ptr_to_data = new double[n];
    refcnt = new int(1);              //initialize *refcnt to 1
    }
Array::Array()
    {
    num_elts = 0;
    ptr_to_data = 0;
    refcnt = new int(1);
    }
Array::~Array()                          // Destructor
    {
    decrement();
    }
void Array::decrement()                 // decrement refcnt
    {                                   // destroy object if refcnt=0
```

```
    (*refcnt)--;
    if(*refcnt == 0)
      {
       delete[] ptr_to_data;
       delete refcnt;
      }
    }
Array::Array(const Array& a)             // Copy constructor
    {
    num_elts = a.num_elts;
//    ptr_to_data = new double[num_elts];
      ptr_to_data = a.ptr_to_data;         // Shallow copy of ptr_to_data
//    copy(a);                             // Copy a's elements
    refcnt = a.refcnt;                   // Shallow copy of refcnt
                               //original and copy point to same refcnt
    (*refcnt)++;                 // increment refcnt
    }

/* void Array::copy(const Array& a)        //Comment out "copy" function
    {
    // Copy a's elements into the elements of *this
    double *p = ptr_to_data + num_elts;
    double *q = a.ptr_to_data + num_elts;
    while (p > ptr_to_data)
         *--p = *--q;
    }
*/

double& Array::operator[](int i)
    {
    #ifdef CHECKBOUNDS
      if(i<0 || i>num_elts)
        error("out of bounds");
    #endif
    return ptr_to_data[i];
    }

const double& Array::operator[](int i) const
    {
    #ifdef CHECKBOUNDS
      if(i<0 || i>num_elts)
        error("out of bounds");
```

```
    #endif
    return ptr_to_data[i];
    }

int Array::numElts() const
    {
    return num_elts;
    }

Array& Array::operator=(const Array& rhs)
    {
    if (ptr_to_data != rhs.ptr_to_data)
        {
        (*rhs.refcnt)++;                //increment *refcnt for rhs
        decrement();                    //decrement *refcnt for lhs
        refcnt = rhs.refcnt;
        ptr_to_data = rhs.ptr_to_data;  //shallow copy of ptr_to_data
        num_elts=rhs.num_elts;

//        setSize(rhs.num_elts);        //comment out setSize
//        copy(rhs);              //comment out copying
        }
    return *this;
    }

/* void Array::setSize(int n)          //comment out setSize
    {
    if (n != num_elts)
       {
        delete[] ptr_to_data;              // Delete old elements,
        num_elts = n;                      // set new count,
        ptr_to_data = new double[n];        // and allocate new elements
       }
    }
*/
Array& Array::operator=(double rhs)
    {
    double *p = ptr_to_data + num_elts;
    while (p > ptr_to_data)
        *--p = rhs;
    return *this;
    }
```

```
void error(const char *s)
    {
    cerr << endl << s << endl; //1st "endl" is in case program is printing
    cout << endl << s << endl; //something out when the error occurs
    cout.flush();              //write the output buffer to the screen
                               //or wherever the output of cout goes.
    abort();
    }

Array operator + (const Array &a1, const Array &a2)
    {
     if(a1.numElts()==a2.numElts())
       {
        Array sum(a1.numElts());
        int i;
        for(i=0;i<a1.numElts();i++)
           sum[i] = a1[i] + a2[i];
        return sum;
       }
     else
       error("can't add arrays of different lengths");
     }
// **********************This is testarray.cc*********************
// testarray.cc is the same as before.
#include <iostream.h>
#include "refarray.h"

int main()
  {
   int max=5;
   Array matrixA(max);
   Array matrixB(max);
   int i;
   for(i=0;i<max;i++)
    { matrixA[i]=i;                                  //matrixA={0 1 2 3 4}
      cout << matrixA[i] << " ";
    }
      cout << endl;
      for(i=0;i<max;i++)
    { matrixB[i]=i+4.0;                             //matrixB={4 5 6 7 8}
      cout << matrixB[i] << " " ;
```

```
    }
      cout << endl;

   Array result = matrixA + matrixB;            // invokes copy constructor
   for(i=0;i<max;i++)
    { cout << result[i] << " ";}
      cout << endl;

   result = matrixA;                        // invokes assignment operator
   for(i=0;i<max;i++)
    { cout << result[i] << " ";}
      cout << endl;

   return 0;
   }
```

Class `Array` now includes reference counting. Notice that the copy constructor and assignment operators no longer copy all the array elements. They just handle the pointers pointing to the objects and make sure that there is a proper count of the number of references to each object. The destructor decreases by one the number of pointers pointing at an object and only destroys an object if there are no pointers pointing at it any longer.

In chapter 14 of Barton and Nackman, there is a class called `ReferenceCount` that keeps track of the bookkeeping for reference counting.

```
// **********************ReferenceCount.h************************
#ifndef ReferenceCountH
#define ReferenceCountH

class ReferenceCount {
public:

  ReferenceCount();                                    // Create with count of 1
  ReferenceCount(const ReferenceCount&);               // Copy and increment count
  ReferenceCount& operator=(const ReferenceCount&);  // Assign;
                                                 //decrement lhs count, increment rhs
  ~ReferenceCount();                             // Decrement count; delete if 0
  bool unique() const;                           // True if count is 1
private:
  unsigned int* p_refcnt;                        // Pointer to actual count
  void decrement();                              // Decrement count; delete if 0
};
inline bool ReferenceCount::unique() const
  {return *p_refcnt == 1; }
```

```
inline ReferenceCount::ReferenceCount() : p_refcnt(new unsigned int(1)) {}

inline ReferenceCount::ReferenceCount(const ReferenceCount& anRC) :
    p_refcnt(anRC.p_refcnt) { ++*p_refcnt; }

inline void ReferenceCount::decrement() {
    if (unique()) delete p_refcnt;
    else --*p_refcnt;
}

inline ReferenceCount::~ReferenceCount() { decrement(); }

inline ReferenceCount&
         ReferenceCount::operator=(const ReferenceCount& rhs) {
    ++*rhs.p_refcnt;
    decrement();
    p_refcnt = rhs.p_refcnt;
    return *this;
}
#endif
```

`ReferenceCount` is a private member of a pointer class `CountedBuiltInPtr` that keeps track of the object pointed to. This is an example of a smart pointer. "Smart pointers" are pointer classes which help manage objects. Their star data member is a pointer.

```
#ifndef CountedObjPtrH
#define CountedObjPtrH

#include "ReferenceCount.h"

template<class T>
class CountedBuiltInPtr {
public:
    CountedBuiltInPtr() : the_p(0) {}       // Construct as null pointer
    CountedBuiltInPtr(T* just_newed) : the_p(just_newed) {}
                            // Construct pointing at heap object
    CountedBuiltInPtr<T>& operator=(const CountedBuiltInPtr<T>&);
                           // Adjust counts
    CountedBuiltInPtr<T>& operator=(T*);
                          // Decrease lhs count
    ~CountedBuiltInPtr();                   //Decrease count, destroy if 0
    bool unique() const;                 // Is count one ?
```

```
    T& operator*() const { return *the_p; }   // Access object
    bool isNull() const { return the_p == 0; }


    friend bool operator==(const CountedBuiltInPtr<T>& lhs,
                           const CountedBuiltInPtr<T>& rhs) {
        return lhs.the_p == rhs.the_p;
    }
    friend bool operator!=(const CountedBuiltInPtr<T>& lhs,
                           const CountedBuiltInPtr<T>& rhs) {
        return lhs.the_p != rhs.the_p;
    }
protected:
    T* the_p;
private:
    ReferenceCount refCount; // Number of pointers to heap object
};
```

Notice that no copy constructor is needed. The default copy constructor will do since we just want a shallow copy of the pointers. The default copy constructor for `CountedBuiltInPtr` will call the copy constructor for `ReferenceCount`. The class `CountedBuiltInPtr` is fine for built–in types. For pointing to class objects we need a member selection operator. This is done in the class `CountedObjPtr`. The template class `CountedBuiltInPtr` is inherited by the class `CountedObjPtr`.

```
template<class T>
class CountedObjPtr :
    public CountedBuiltInPtr<T> {
public:
    CountedObjPtr() {}
    CountedObjPtr(T* just_newed) : CountedBuiltInPtr<T>(just_newed) {}
    CountedObjPtr<T>& operator=(T* rhs) {
        CountedBuiltInPtr<T>::operator=(rhs);
        return *this;
    }
    CountedObjPtr<T>& operator=(const CountedObjPtr<T>& rhs) {
        CountedBuiltInPtr<T>::operator=(rhs);
        return *this;
    }
    T* operator->() const { return the_p; }    // member selection operator
};
```

We have defined constructors and assignment operators for `CountedObjPtr` because they are not inherited from `CountedBuiltInPtr`.

```
// ******definitions of template member functions from CountedObjPtr.cc

template<class T>
CountedBuiltInPtr<T>&
   CountedBuiltInPtr<T>::operator=(const CountedBuiltInPtr<T>& rhs) {
    if (the_p != rhs.the_p) {
        if (unique()) delete the_p;
        the_p = rhs.the_p;
        refCount = rhs.refCount;
    }
    return *this;
}
template<class T>
CountedBuiltInPtr<T>& CountedBuiltInPtr<T>::operator=(T* just_newed) {
    if (unique()) delete the_p;
    the_p = just_newed;
    refCount = ReferenceCount();          //invokes default constructor
    return *this;
}
template<class T>
CountedBuiltInPtr<T>::~CountedBuiltInPtr() {
    if (refCount.unique()) delete the_p;
}
template<class T>
bool CountedBuiltInPtr<T>::unique() const {
    return refCount.unique();
}
#endif
```

This makes reference counting easy to use. You just put the class that you want to have reference counting as the type `T` in `CountedObjPtr<T>`. For example, if you have a class `Vector` that you want to have reference counting, you would write

```
int main()
{
   Vector* velocity = new Vector;
   CountedObjPtr<Vector> ref_velocity(velocity);  //calls constructor of
                                                  //CountedObjPtr
  // other stuff ...
}
```