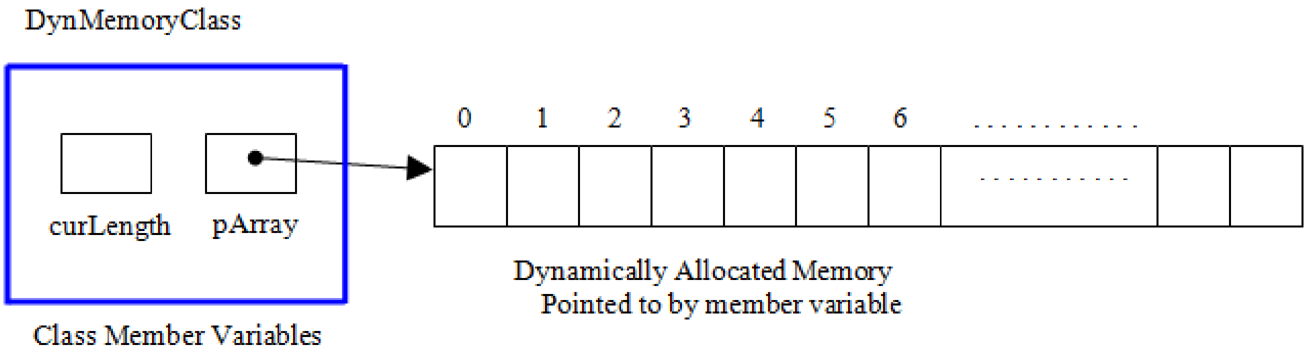
C++ Class Design with Dynamic Storage Allocation (dynamic data member)



Special methods must be specified for correct implementation of a C++ class that has pointer-based member variables that link to dynamically-allocated memory.

```
~DynMemoryClass();
// A "destructor" method
// Automatically invoked when the object is deleted.
```

```
DynMemoryClass *p = new DynMemoryClass();
. . .
   p->pArray = new char[curLength];  // internal allocation
. . .
delete p;
```

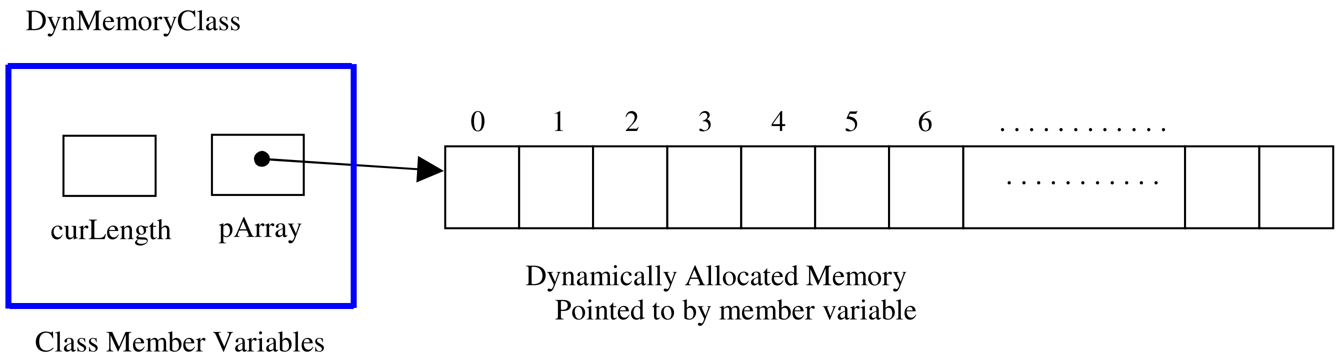
```
DynMemoryClass(const DynMemoryClass &rhs);
// A "copy constructor" method
// Must construct a "deep" copy of rhs
```

```
DynMemoryClass & operator=(const DynMemoryClass &rhs);
// An assignment statement "operator=" overload.
// Must make a "deep" copy of rhs
```

```
DynMemoryClass x;
. . .    x.pArray = new char[curLength];

DynMemoryClass y(x);    // copy constructor: y is copy of x
DynMemoryClass z;
z = x;                  // assignment operator=
```

C++ Class Design with Dynamic Storage Allocation: Destructor Method Required



Destructor method required: ~ Classname ( )

- **The default destructor (supplied by compiler) will only deallocate the explicit class member variables (e.g., curLength, pArray).**
- **The default destructor never automatically deallocates the dynamically allocated components.**
- **Relying on default destructor will result in a memory leak.**
- **Proper class design requires declaration and implementation of a destructor method.**

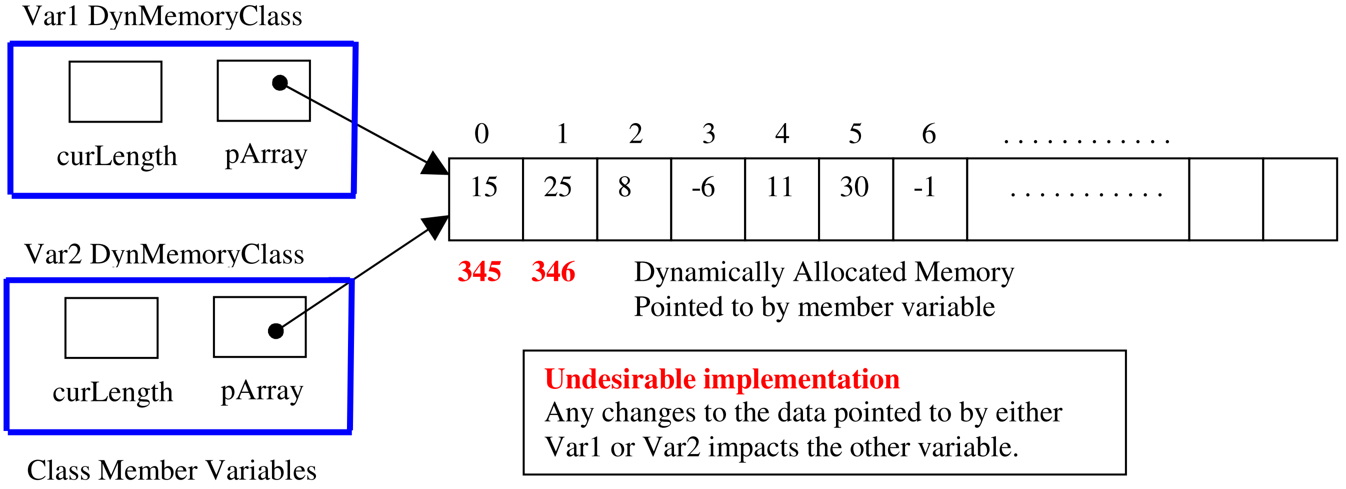
Example:

```
~DynMemoryClass()
{
        if(pArray != NULL)
        {
           delete [ ] pArray;
        }
}
```

C++ Class Design
with Dynamic Storage Allocation:
Copy-Constructor & Assignment methods

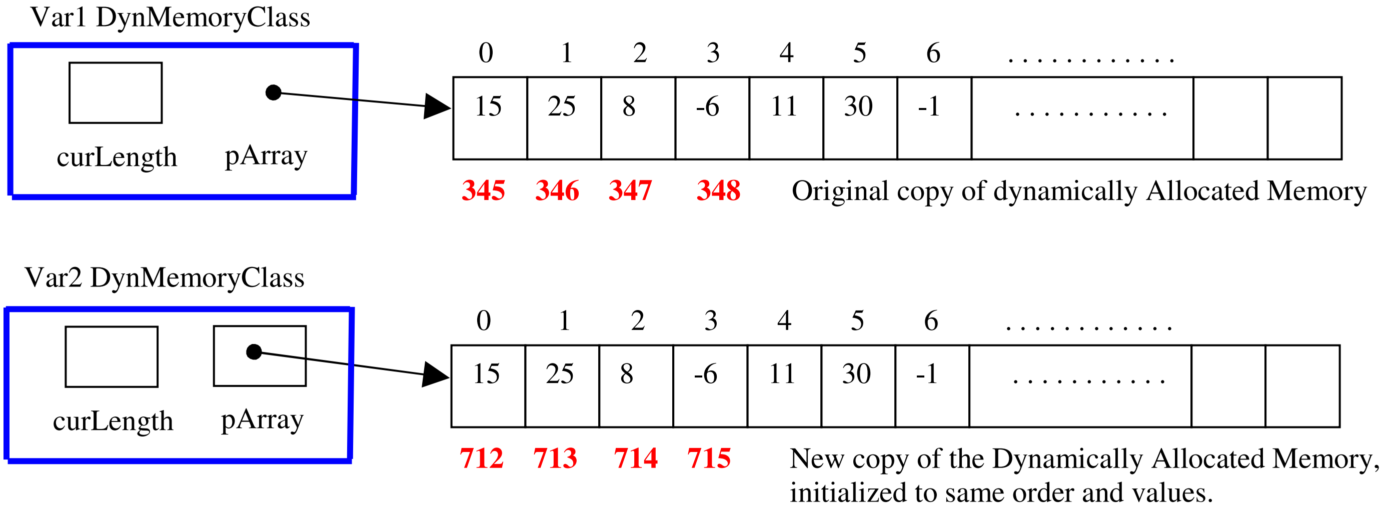
What does it mean to create a copy of a variable of type DynMemoryClass? **e.g.,** **DynMemoryClass Var2(Var1);**
or
Var2 = Var1; // assignment

Shallow Copy: only copy member variables (default)



Undesirable implementation
Any changes to the data pointed to by either Var1 or Var2 impacts the other variable.

Deep Copy: replicate the entire data structure
Requires explicit Copy-Constructor and operator= methods



Queue implemented using a Dynamic Array

A Queue is a FIFO list-data-structure where items are inserted at the end of the list and removed from the front of the list.

Using a Dynamic Array allows the data storage for the list to be allocated with the precise size required.

The DynArrayQ class requires a Destructor, a Copy-Constructor, and an assignment operator= method.

```
// DynArrayQ.h      CSC 2430  Mike Tindall
// FIFO First-In First-Out Queue

#ifndef _DYNARRAYQ_H
#define _DYNARRAYQ_H

class DynArrayQ
{
public:
   // Construct empty Queue
   DynArrayQ();

   // Copy Constructor.  Create (deep) copy of s
   DynArrayQ(const DynArrayQ& s);

   // Assignment operator= overload.  Make (deep) copy of rhs
   DynArrayQ& operator=(const DynArrayQ& rhs);

   // Destructor
   ~DynArrayQ();

   // Return current length of Queue
   const int length() const;

   // Return whether Queue is empty or not
   const bool isEmpty() const;

   // Add value to end of Queue.
   void Add(const int value, bool &Success);

   // Remove and return first value from Queue.
   void Remove(int &value, bool &Success);

private:
   int curLength;
   int *pArray;
};

#endif
```

```cpp
// main.cpp  DynArrayQ Example  CSC 2430  Mike Tindall
#include <iostream>
using namespace std;
#include "DynArrayQ.h"

void PrintOutQue(DynArrayQ& q)      // by reference
{
    bool success;
    int val;

    while(!q.isEmpty())
    {
        q.Remove(val, success);

        if(success)
            cout << "Removed: " << val << endl;
        else
            cout << "Remove() failed." << endl;
    }
}

int main()
{
    DynArrayQ myQue;            // Invoke Default constructor
    bool success;

    for(int i = 1; i <= 10; ++i)
    {
        myQue.Add(i*i, success);
        if(!success)
            cout << "Add() failed." << endl;
    }

    cout << "myQue size: " << myQue.length() << endl;

    DynArrayQ newQue1(myQue);   // Invoke Copy constructor
    DynArrayQ newQue2;
    newQue2 = myQue;            // Invoke assignment operator=

    PrintOutQue(myQue);
    PrintOutQue(newQue1);

    newQue2.Add(1000, success);
    newQue2.Add(2000, success);
    newQue2.Add(3000, success);
    newQue2.Add(4000, success);
    newQue2.Add(5000, success);

    for(int i=0; i<10000; ++i)
        newQue2.Add(i, success);

    cout << "newQue2 size: " << newQue2.length() << endl;

    int val;
    for(int i=0; i<9995; ++i)
        newQue2.Remove(val, success);

    PrintOutQue(newQue2);

    return(0);
}                               // Invoke ~Destructor
```

```
myQue size: 10
Removed: 1                       // myQue
Removed: 4
Removed: 9
Removed: 16
Removed: 25
Removed: 36
Removed: 49
Removed: 64
Removed: 81
Removed: 100

Removed: 1                       // newQue1
Removed: 4
Removed: 9
Removed: 16
Removed: 25
Removed: 36
Removed: 49
Removed: 64
Removed: 81
Removed: 100

newQue2 size: 10015              // newQue2
Removed: 9980
Removed: 9981
Removed: 9982
Removed: 9983
Removed: 9984
Removed: 9985
Removed: 9986
Removed: 9987
Removed: 9988
Removed: 9989
Removed: 9990
Removed: 9991
Removed: 9992
Removed: 9993
Removed: 9994
Removed: 9995
Removed: 9996
Removed: 9997
Removed: 9998
Removed: 9999
Press any key to continue
```

```
// DynArrayQ.cpp    CSC 2430  Mike Tindall
// FIFO First-In First-Out Queue
// (Inefficient array-reallocation implementation)

#include <stdlib.h>           // for NULL
#include "DynArrayQ.h"

// Construct empty Queue
DynArrayQ::DynArrayQ()
{
   curLength = 0;
   pArray     = NULL;
}

// Return current length of Queue
const int DynArrayQ::length() const
{
   return(curLength);
}

// Return whether Queue is empty or not
const bool DynArrayQ::isEmpty() const
{
   return bool(curLength == 0);
}

// Add value to end of Queue.
void DynArrayQ::Add(const int value, bool &Success)
{
   int *pOrig = pArray;   // original array

   pArray = new int[curLength + 1]; // Add one more element
   Success = bool(pArray != NULL);

   if(Success)
   {
      for(int i=0; i<curLength; ++i)
         pArray[i] = pOrig[i];

      pArray[curLength] = value;// Add new value to end
      ++curLength;
   }
   else      // Allocation failure: reset to empty state
   {
      curLength = 0;
      pArray = NULL;
   }

   // All finished with original array -- deallocate it
   if(pOrig != NULL)
      delete [] pOrig;
}
```

```
// Remove and return first value from Queue.
void DynArrayQ::Remove(int &value, bool &Success)
{
   Success = !isEmpty();

   if(Success)
   {
      value = pArray[0];      // Value to remove and return

      int *pOrig = pArray;    // original array

      if(curLength == 1)      // Last item removed?
      {
         curLength = 0;
         pArray = NULL;
      }
      else
      {
         pArray = new int[curLength - 1];
         Success = bool(pArray != NULL);

         if(Success)
         {
            for(int i=1; i<curLength; ++i)    // Copy, removing first item
               pArray[i-1] = pOrig[i];

            --curLength;
         }
         else       // Allocation failure: reset to empty state
         {
            pArray = NULL;
            curLength = 0;
         }
      }

      if(pOrig != NULL)
         delete [] pOrig;     // Delete old array
   }
}
```

```
// Destructor
DynArrayQ::~DynArrayQ()
{
   if(pArray != NULL)
      delete [] pArray;
}

// Copy Constructor.  Create (deep) copy of s
DynArrayQ::DynArrayQ(const DynArrayQ& s)
{
   // Make deep copy of s
   if(s.curLength == 0)
   {                             // this: builtin pointer to current object
      curLength = 0;             // this->curLength = 0;  // alternate form
      pArray = NULL;             // this->pArray = NULL;
   }
   else
   {
      curLength = s.curLength;
      pArray = new int[curLength];  // Allocate new array

      // Copy array values from s
      for(int i=0; i<curLength; ++i)
         pArray[i] = s.pArray[i];     // this->pArray[i] = s.pArray[i];
   }
}

// Assignment operator= overload.  Make (deep) copy of rhs
DynArrayQ& DynArrayQ::operator=(const DynArrayQ& rhs)
{
   if(this != &rhs)    // Ignore for  s = s; self-assignment
   {
      if(pArray != NULL)
         delete [] pArray;         // Delete old array

      // Make deep copy of rhs
      if(rhs.curLength == 0)
      {
         curLength = 0;
         pArray = NULL;
      }
      else
      {
         curLength = rhs.curLength;
         pArray = new int[curLength];  // Allocate new array

         // Copy array values from rhs
         for(int i=0; i<curLength; ++i)
            pArray[i] = rhs.pArray[i];
      }
   }

   return(*this);        // return a copy of "this" object
}
```