



chapter

22

Dynamic Members

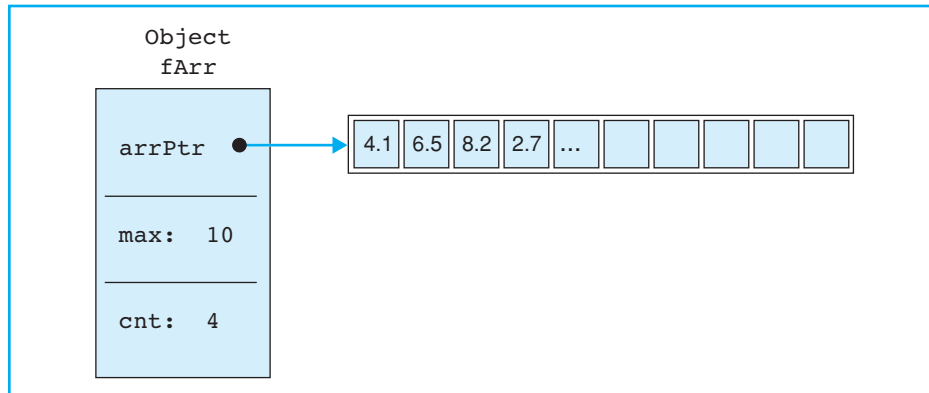
This chapter describes how to implement classes containing pointers to dynamically allocated memory. These include

- your own copy constructor definition and
- overloading the assignment operator.

A class designed to represent arrays of any given length is used as a sample application.

MEMBERS OF VARYING LENGTH

An object of class `FloatArr` in memory



Data members of class `FloatArr`

```
// A class representing dynamic arrays of floats.
// -----
class FloatArr
{
private:
    float* arrPtr;    // Dynamic member
    int max;         // Maximum quantity without
                    // reallocating new storage.
    int cnt;         // Number of present elements

public:
    // Public methods here
};
```

□ Dynamic Members

You can exploit the potential of dynamic memory allocation to leverage existing classes and create data members of variable length. Depending on the amount of data an application program really has to handle, memory is allocated as required while the application is running. In order to do this the class needs a pointer to the dynamically allocated memory that contains the actual data. Data members of this kind are also known as *dynamic members* of a class.

When compiling a program that contains arrays, you will probably not know how many elements the array will need to store. A class designed to represent arrays should take this point into consideration and allow for dynamically defined variable length arrays.

□ Requirements

In the following section you will be developing a new version of the `FloatArr` class to meet these requirements and additionally allow you to manipulate arrays as easy as fundamental types. For example, a simple assignment should be possible for two objects `v1` and `v2` in the new class.

Example: `v2 = v1;`

The object `v2` itself—and not the programmer—will ensure that enough memory is available to accommodate the array `v1`.

Just as in the case of fundamental types, it should also be possible to use an existing object, `v2`, to initialize a new object, `v3`.

Example: `FloatArr v3(v2);`

Here the object `v3` ensures that enough memory is available to accommodate the array elements of `v2`.

When an object of the `FloatArr` is declared, the user should be able to define the initial length of the array. The statement

Example: `FloatArr fArr(100);`

allocates memory for a maximum of 100 array elements.

The definition of the `FloatArr` class therefore comprises a member that addresses a dynamically allocated array. In addition to this, two `int` variables are required to store the maximum and current number of array elements.

■ CLASSES WITH A DYNAMIC MEMBER

First version of class FloatArr

```
// floatArr.h : Dynamic array of floats.
// -----
#ifndef _FLOATARR_
#define _FLOATARR_
class FloatArr
{
private:
    float* arrPtr;    // Dynamic member
    int max;         // Maximum quantity without
                    // reallocation of new storage.
    int cnt;        // Number of array elements
public:
    FloatArr( int n = 256 );    // Constructor
    FloatArr( int n, float val);
    ~FloatArr();              // Destructor
    int length() const { return cnt; }
    float& operator[](int i); // Subscript operator.
    float operator[](int i) const;
    bool append(float val);   // Append value val.
    bool remove(int pos);     // Delete position pos.
};
#endif // _FLOATARR_
```

Creating objects with dynamic members

```
#include "floatArr.h"
#include <iostream>
using namespace std;
int main()
{
    FloatArr v(10);    // Array v of 10 float values
    FloatArr w(20, 1.0F); // To initialize array w of
                        // 20 float values with 1.0.

    v.append( 0.5F);
    cout << " Current number of elements in v: "
         << v.length() << endl; // 1
    cout << " Current number of elements in w: "
         << w.length() << endl; // 20
    return 0;
}
```

The next question you need to ask when designing a class to represent arrays is what methods are necessary and useful. You can enhance `FloatArr` class step by step by optimizing existing methods or adding new methods.

The first version of the `FloatArr` class comprises a few basic methods, which are introduced and discussed in the following section.

□ Constructors

It should be possible to create an object of the `FloatArr` class with a given length and store a `float` value in the object, if needed. A constructor that expects an `int` value as an argument is declared for this purpose.

```
FloatArr(int n = 256);
```

The number 256 is the default argument for the length of the array. This provides for a default constructor that creates an array with 256 empty array elements.

An additional constructor

```
FloatArr( int n, int val );
```

allows you to define an array where the given value is stored in each array element. In this case you need to state the length of the array.

Example: `FloatArr arr(100, 0.0F);`

This statement initializes the 100 elements in the array with a value of 0.0.

□ Additional Methods

The `length()` method allows you to query the number of elements in the array. `arr.length()` returns a value of 100 for the array `arr`.

You can overload the subscript operator `[]` to access individual array elements.

Example: `arr[i] = 15.0F;`

The index `i` must lie within the range 0 to `cnt-1`.

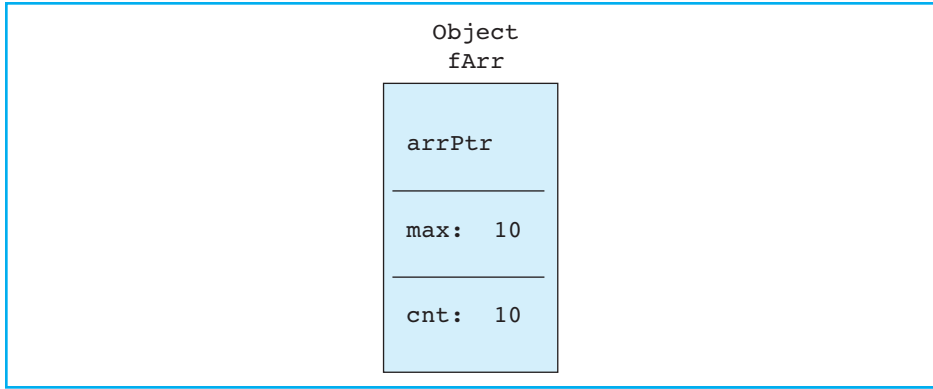
The `append()` method can be used to append a value to the array. The number of elements is then incremented by one.

When you call the `remove()` method it does exactly the opposite of `append()`—deleting the element at the stated position. This reduces the current count by one, provided a valid position was stated.

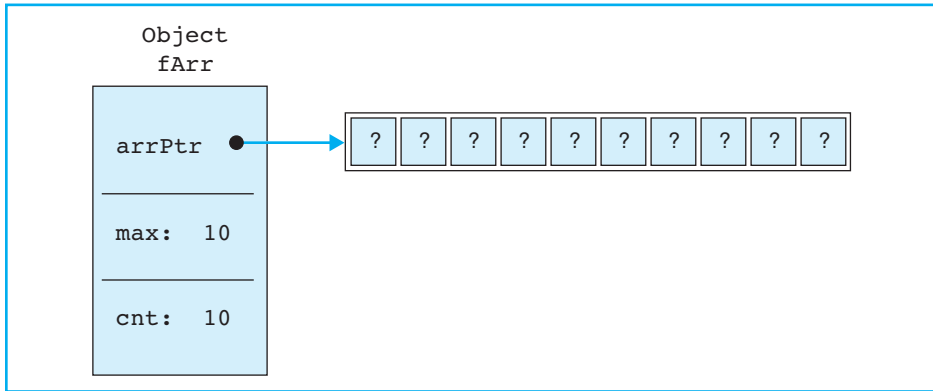
■ CREATING AND DESTROYING OBJECTS

Effects of the declaration `FloatArr fArr(10, 1.0F);`

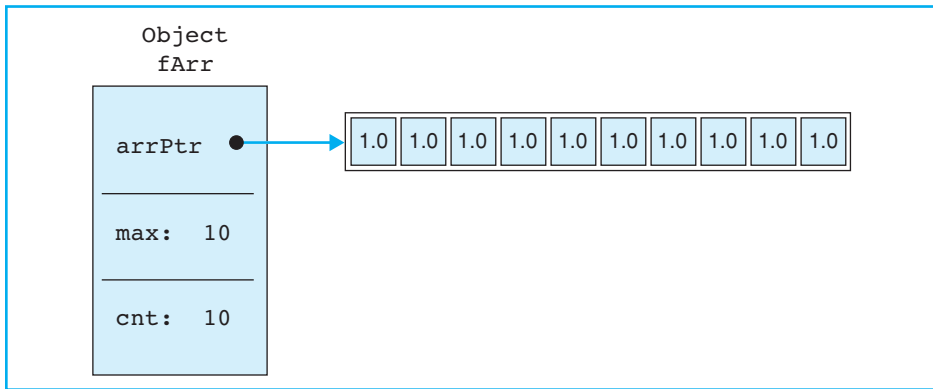
First, memory is allocated for the data members:



Then storage is allocated for 10 array elements and the variables `max` and `cnt` are set to 10:



Finally, a value of 1.0 is used to initialize the array elements:



The memory for the array elements is not contained in a `FloatArr` object and must be allocated dynamically by the constructor. The object itself only occupies the memory required for the data members `arrPtr`, `max`, and `cnt`. Thus, `sizeof(FloatArr)` is a constant value that defaults to 12 bytes for 32 bit computers.

The additional dynamic memory allocation may need to be adjusted to meet new requirements, for example, if an assignment is made. Finally, the memory has to be released explicitly when an object is destroyed.

□ Constructing an Object

The first constructor in the `FloatArr` class is defined as follows:

```
FloatArr::FloatArr( int n )
{
    max = n;    cnt = 0;
    arrPtr = new float[max];
}
```

This allocates memory for `n` array elements. The current number of array elements is set to 0.

The second constructor fills the array with the supplied value and is therefore defined as follows:

```
FloatArr::FloatArr(int n, float val)
{
    max = cnt = n;
    arrPtr = new float[max];
    for( int i=0; i < cnt; ++i)
        arrPtr[i] = val;
}
```

The opposite page shows how memory is allocated for the object `fArr` and how this object is initialized.

□ Destroying an Object

When an object is destroyed the dynamic memory the object occupies must be released. Classes with dynamic members will *always* need a destructor to perform this task.

The `FloatArr` class contains a dynamic array, so memory can be released by a call to the `delete[]` operator.

```
FloatArr::~FloatArr()
{
    delete[] arrPtr;
}
```

■ IMPLEMENTING METHODS

New version of class `FloatArr`

```
// FloatArr.cpp:
// Implementing the methods of class FloatArr.
// -----
#include "floatArr.h"
#include <iostream>
using namespace std;

// Constructors and destructor as before.
// Subscript operator for objects that are not const:
float& FloatArr::operator[]( int i )
{
    if( i < 0 || i >= cnt )           // Range checking
    {
        cerr << "\n class FloatArr: Out of range! ";
        exit(1);
    }
    return arrPtr[i];
}

float FloatArr::operator[]( int i ) const
{
    // Else as before.
}

bool FloatArr::append( float val )
{
    if( cnt < max )
    {
        arrPtr[cnt++] = val; return true;
    }
    else
        return false;           // Enlarge the array!
}

bool FloatArr::remove( int pos )
{
    if( pos >= 0 && pos < cnt )
    {
        for( int i = pos; i < cnt-1; ++i )
            arrPtr[i] = arrPtr[i+1];
        --cnt;
        return true;
    }
    else
        return false;
}
```

□ Read and Write Access Using the Subscript Operator

The subscript operator can be overloaded to allow easy manipulation of array elements.

```
Example: FloatArr v(5, 0.0F);  
           v[2] = 2.2F;  
           for( int i=0; i < v.length(); ++i)  
             cout << v[i];
```

The operator allows both read and write access to the array elements and cannot be used for constant objects for this reason. However, you will need to support read-only access to constant objects.

The `FloatArr` class contains two versions of the operator function `operator[]()` for this purpose. The first version returns a reference to the *i*-th array element and thus supports write access. The second, read-only version only supports read access to the array elements and is automatically called by the compiler when accessing constant objects.

The implementation of these versions is identical. In both cases range checking is performed for the index. If the index lies within the valid boundaries, an array element—or simply a value in the case of the read-only version—is returned.

□ Appending and Deleting in Arrays

The `FloatArr` class comprises the methods `append()` and `remove()` for appending and deleting array elements.

In the first version, the `append()` only works if there is at least one empty slot in the array. In the exercises, `append()` is used to extend the array as required. This also applies for a new method, `insert()`, which you will write as an exercise in this chapter.

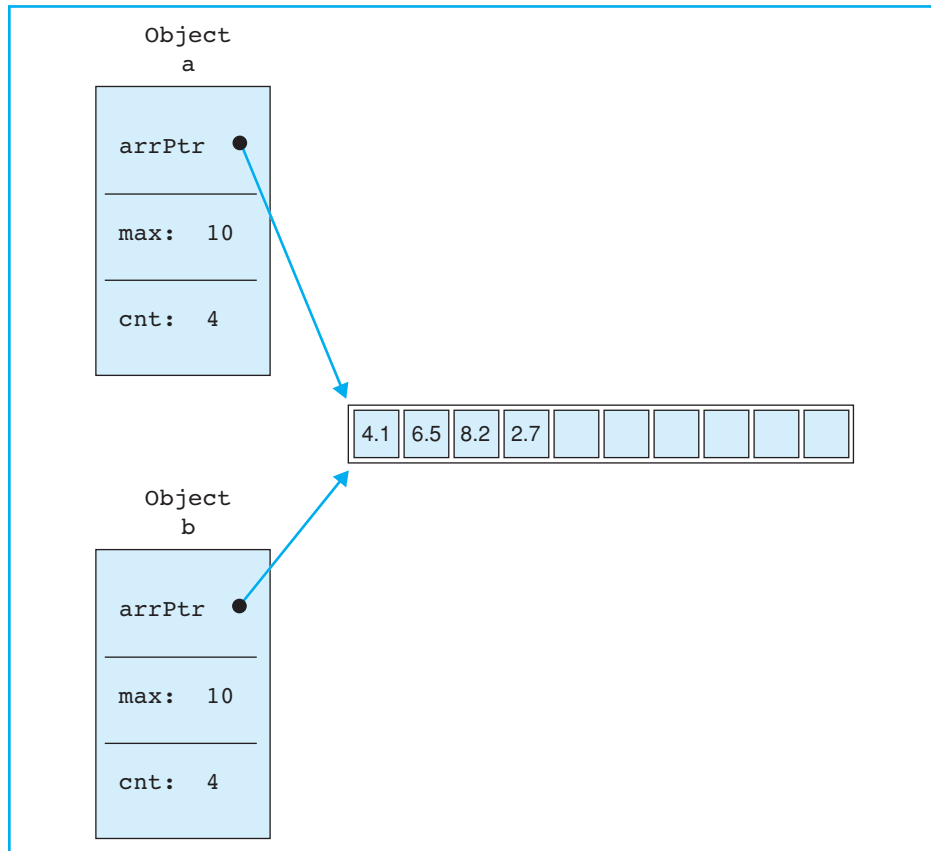
When the `remove()` method is used to delete an element, the elements following the deleted element move up one place, preserving the original order. The current count is decremented by one. What was formerly the last element in the array is not deleted but overwritten when a new element is inserted.

Another technique would be to copy the last element to the position of the element that needs to be deleted, simply overwriting that element. Of course, this technique is quicker and preferable for cases where the order of the elements is not significant.

■ COPY CONSTRUCTOR

Effect of the standard copy constructor

```
FloatArr b(a);           // Creates a copy of a.
```



A self-defined copy constructor for class `FloatArr`

```
//floatArr.cpp: Implementing the methods.
// -----
FloatArr::FloatArr(const FloatArr& src)
{
    max = src.max;    cnt = src.cnt;
    arrPtr = new float[max];

    for( int i = 0; i < cnt; i++ )
        arrPtr[i] = src.arrPtr[i];
}
```

□ Initializing with an Object

The next step is to ensure that an existing object can be used to initialize a new object. Given an array, `a`, the following statement should be valid:

Example: `FloatArr b(a);`

The array `b` should now be the same length as the array `a` and the array elements in `b` should contain the same values as in `a`.

The `FloatArr` class needs a *copy constructor* to perform this task. The constructor has a reference to a constant array as a parameter.

Prototype: `FloatArr(const FloatArr&);`

□ Standard Copy Constructor

If a class does not contain a copy constructor, the compiler will automatically create a minimal version, known as the *standard copy constructor*. This constructor copies the data members of the object passed to it to corresponding data members of the new object.

A standard copy constructor is normally sufficient for a class. However, simply copying the data members would serve no useful purpose for objects containing dynamic members. This would merely copy the pointers, meaning that the pointers of several different objects would reference the same place in memory. The diagram on the opposite page illustrates this situation for two `FloatArr` class objects.

This scenario would obviously mean trouble. Imagine releasing memory allocated for an object dynamically. The pointer for the second object would reference a memory area that no longer existed!

□ Proprietary Version of the Copy Constructor

Clearly you will need to write a new copy constructor for classes with dynamic members, ensuring that the live data and not just the pointers are copied from the dynamically allocated memory.

The example on the opposite page shows the definition of the copy constructor for the `FloatArr` class. Calling `new[]` creates a new array and the array elements of the object passed to the method are then copied to that array.

■ ASSIGNMENT

New declarations in class FloatArr

```
// FloatArr.h : Dynamic arrays of floats.
// -----
class FloatArr
{
private:
    // . . . Data members as before
public:
    // . . . Methods as before and
    FloatArr(const FloatArr& src);    // Copy constructor
    FloatArr& operator=( const FloatArr&); // Assignment
};
```

Defining the assignment

```
// In file floatArr.cpp
// The operator function implementing "=".
// -----
FloatArr& FloatArr::operator=( const FloatArr& src )
{
    if( this != &src )           // No self assignments!
    {
        max = src.max;
        cnt = src.cnt;
        delete[] arrPtr;         // Release memory,
        arrPtr = new float[max]; // reallocate and
        for( int i=0; i < cnt; i++) // copy elements.
            arrPtr[i] = src.arrPtr[i];
    }
    return *this;
}
```

Sample calls

```
#include "FloatArr.h"
int main()
{
    FloatArr v;                // Default constructor.
    FloatArr w(20, 1.0F);     // Array w - 20 float values
                                // with initial value 1.0.
    const FloatArr kw(w);    // Use copy constructor
                                // to create an object.
    v = w;                   // Assignment.
}
```

Each class comprises four implicitly defined default methods, which you can replace with your own definitions:

- the default constructor and the destructor
- the copy constructor and the standard assignment

In contrast to initialization by means of the copy constructor, which takes place when an object is defined, an assignment always requires an existing object. Multiple assignments, which modify an object, are possible.

□ Default Assignment

Given that `v1` and `v2` are two `FloatArr` class objects, the following assignment is valid:

Example: `v1 = v2;` `// Possible, but ok?`

Default assignment is performed member by member. The data members of `v2` are copied to the corresponding data members of `v1` just like the copy constructor would copy them. However, this technique is not suitable for classes with dynamic members. This would simply point the pointers belonging to different objects at the same dynamic allocated memory. In addition, memory previously addressed by a pointer of the target object will be unreferenced after the assignment.

□ Overloading the Assignment Operator

In other words, you need to overload the default assignment for classes containing dynamic members. Generally speaking, if you need to define a copy constructor, you will also need to define an assignment.

The operator function for the assignment must perform the following tasks:

- release the memory referenced by the dynamic members
- allocate sufficient memory and copy the source object's data to that memory.

The operator function is implemented as a class method and returns a reference to the target object allowing multiple assignments. The prototype of the operator function for the `FloatArr` class is thus defined as follows:

```
FloatArr& FloatArr::operator=( const FloatArr& src)
```

When implementing the operator function you must avoid self assignment, which would read memory areas that have already been released.

exercises

■ EXERCISES

New methods of class `List`

```
// Copy constructor:  
List::List(const List&);  
// Assignment:  
List& List::operator=( const List&);
```

New methods of class `FloatArr`

```
// Methods to append a float or an  
// array of floats:  
void append( float val);  
void append( const FloatArr& v);  
FloatArr& operator+=( float val);  
FloatArr& operator+=( const FloatArr& v);  
  
// Methods to insert a float or an  
// array of floats:  
bool insert( float val, int pos);  
bool insert( const FloatArr& v, int pos );  
  
// In any case, more memory space must be allocated  
// to the array if the current capacity is  
// insufficient.
```

Exercise 1

Complete the definition of the `List` class, which represents a linked list and test the class.

First, modify your test program to create a copy of a list. Call the default assignment for the objects in the `List` class. Note how your program reacts.

A trial run of the program shows that the class is incomplete. Since the class contains dynamic members, the following tasks must be performed:

- Define a copy constructor for the `List` class.
- Overload the assignment operator.

Exercise 2

Add the methods shown opposite to the `FloatArr` class. In contrast to the existing method

```
bool append( float val );
```

the new method must be able to allocate more memory as required. As this could also be necessary for other methods, write a private auxiliary function for this purpose

```
void expand( int newMax );
```

The method must copy existing data to the newly allocated memory.

Overload the operator `+=` so it can be used instead of calling the function `append()`.

The `insert()` method inserts a `float` value or a `FloatArr` object at position `pos`. Any elements that follow `pos` must be pushed.

Also overload the shift operator `<<` to output an array using the field width originally defined to output the array elements.

NOTE

The `width()` method in the `ostream` class returns the current field width, if you call the method without any arguments.

Now add calls to the new methods to your test program and output the results after each call.

solutions

■ SOLUTIONS

Exercise I

```
// -----
// List.h
// Definition of classes ListEl and List
// representing a linked list.
// -----
#ifndef _LIST_H_
#define _LIST_H_
#include "Date.h"
#include <iostream>
#include <iomanip>
using namespace std;

class ListEl
{
    // Unchanged as in Chapter 21
};
// -----
// Definition of class List
class List
{
private:
    ListEl* first, *last;
public:
    // New methods:
    List(const List&); // Copy constructor
    List& operator=( const List&); // Assignment
    // Otherwise unchanged from Chapter 21
};
#endif // _LIST_H_

// -----
// List.cpp
// Implements those methods of class List,
// that are not defined inline.
// -----
#include "List.h"
// Copy constructor:
List::List(const List& src)
{
    // Appends the elements of src to an empty list.
    first = last = NULL;
    ListEl *pEl = src.first;
    for( ; pEl != NULL; pEl = pEl->next )
        pushBack( pEl->date, pEl->amount);
}
```

```
// Assignment:
List& List::operator=( const List& src)
{
    // Release memory for all elements:
    ListEl *pEl = first,
            *next = NULL;
    for( ; pEl != NULL; pEl = next)
    {
        next = pEl->next;
        delete pEl;
    }
    first = last = NULL;

    // Appends the elements of src to an empty list.
    pEl = src.first;
    for( ; pEl != NULL; pEl = pEl->next )
        pushBack( pEl->date, pEl->amount);

    return *this;
}

// All other methods unchanged.

// -----
// List_t.cpp
// Tests the class List with copy constructor and
// assignment.
// -----
#include "List.h"

int main()
{
    cout << "\n * * * Testing the class List * * *\n"
          << endl;
    List list1; // A list
    cout << list1 << endl; // The list is still empty.

    Date date( 11,8,1999); // Insert 3 elements.
    double amount( +1234.56);
    list1.pushBack( date, amount);

    date.setDate( 1, 1, 2002);
    amount = -1000.99;
    list1.pushBack( date, amount);

    date.setDate( 2, 29, 2000);
    amount = +5000.11;
    list1.pushBack( date, amount);
}
```

```
cout << "\nThree elements have been inserted!"
      "\nContent of the list:" << endl;
cout << list1 << endl;

cout << "\nPress return to continue! "; cin.get();

List list2( list1);
cout << "A copy of the 1st list has been created!\n"
      "Contents of the copy:\n" << endl;
cout << list2 << endl;

cout << "\nRemove the first element from the list:\n";

ListEl *ptrEl = ptrEl = list1.front();
if( ptrEl != NULL)
{
    cout << "To be deleted: " << *ptrEl << endl;
    list1.popFront();
}
cout << "\nContent of the list:\n";
cout << list1 << endl;

list1 = list2;          // Reassign the copy.

cout << "The copy has been assigned to the 1st list!\n"
      "Contents after assignment:\n" << endl;
cout << list1 << endl;

return 0;
}
```

Exercise 2

```
// -----  
// floatArr.h : Dynamic arrays of floating-point numbers.  
// -----  
#ifndef _FLOATARR_  
#define _FLOATARR_  
  
#include <iostream>  
using namespace std;  
  
class FloatArr  
{  
private:  
    float* arrPtr;    // Dynamic member  
    int max;         // Maximum quantity without  
                    // reallocating new storage.  
    int cnt;        // Number of present array elements  
  
    void expand( int newMax);    // Helps enlarge the array  
  
public:  
    // Constructors , destructor,  
    // assignment, subscript operator, and method length()  
    // as before in this chapter.  
  
    // Methods to append a floating-point number  
    // or an array of floating-point numbers:  
    void append( float val);  
    void append( const FloatArr& v);  
    FloatArr& operator+=( float val)  
    {  
        append( val);    return *this;  
    }  
  
    FloatArr& operator+=( const FloatArr& v)  
    {  
        append(v);    return *this;  
    }  
  
    // Methods to insert a floating-point number  
    // or an array of floating-point numbers:  
    bool insert( float val, int pos);  
    bool insert( const FloatArr& v, int pos );  
  
    bool remove(int pos);    // Delete at position pos.  
    // To output the array  
    friend ostream& operator<<( ostream& os,  
                                const FloatArr& v);  
};  
#endif    // _FLOATARR_
```

```
// -----  
// FloatArr.cpp  
// Implements the methods of FloatArr.  
// -----  
  
#include "floatArr.h"  
  
// Constructors, destructor, assignment,  
// and subscript operator unchanged.  
  
// --- The new functions ---  
  
// Private auxiliary function to enlarge the array.  
void FloatArr::expand( int new)  
{  
    if( newMax == max)  
        return;  
    max = newMax;  
    if( newMax < cnt)  
        cnt = newMax;  
    float *temp = new float[newMax];  
    for( int i = 0; i < cnt; ++i)  
        temp[i] = arrPtr[i];  
  
    delete[] arrPtr;  
    arrPtr = temp;  
}  
  
// Append floating-point number or an array of floats.  
void FloatArr::append( float val)  
{  
    if( cnt+1 > max)  
        expand( cnt+1);  
  
    arrPtr[cnt++] = val;  
}  
  
void FloatArr::append( const FloatArr& v)  
{  
    if( cnt + v.cnt > max)  
        expand( cnt + v.cnt);  
  
    int count = v.cnt;           // Necessary if v == *this  
  
    for( int i=0; i < count; ++i)  
        arrPtr[cnt++] = v.arrPtr[i];  
}
```

```
// Insert a float or an array of floats
bool FloatArr::insert( float val, int pos)
{
    return insert( FloatArr(1,val), pos);
}

bool FloatArr::insert( const FloatArr& v, int pos )
{
    if( pos < 0 || pos >= cnt)
        return false;           // Invalid position

    if( max < cnt + v.cnt)
        expand(cnt + v.cnt);
    int i;
    for( i = cnt-1; i >= pos; --i)    // Shift up
        arrPtr[i+v.cnt] = arrPtr[i]; // starting at pos
    for( i = 0; i < v.cnt; ++i)     // Fill gap.
        arrPtr[i+pos] = v.arrPtr[i];
    cnt = cnt + v.cnt;
    return true;
}

// To delete
bool FloatArr::remove(int pos)
{
    if( pos >= 0 && pos < cnt)
    {
        for( int i = pos; i < cnt-1; ++i)
            arrPtr[i] = arrPtr[i+1];
        --cnt;
        return true;
    }
    else
        return false;
}

// Output the array
ostream& operator<<( ostream& os, const FloatArr& v)
{
    int w = os.width();           // Save field width.
    for( float *p = v.arrPtr; p < v.arrPtr + v.cnt; ++p)
    {
        os.width(w);    os << *p;
    }
    return os;
}
```

```
// -----  
// FloatV_t.cpp  
// Tests the class FloatArr.  
// -----  
#include "FloatArr.h"  
#include <iostream>  
#include <iomanip>  
using namespace std;  
  
int main()  
{  
    FloatArr v(10);           // Array v for 10 float values  
    FloatArr w(15, 1.0F);    // Initialize the array w of  
                             // 15 float values with 1.0.  
  
    cout << " Current total of elements in v: "  
          << v.length() << endl;  
    cout << " Current total of elements in w: "  
          << w.length() << endl;  
  
    float x = -5.0F;         // Append values.  
    for( ; x < 6 ; x += 1.7F)  
        v.append(x);  
  
    v += v;                  // Also possible!  
  
    cout << "\nThe array elements after appending:"  
          << endl;  
    cout << setw(5) << v << endl;  
  
    const FloatArr cv(v);    // Copy constructor  
                             // creates const object.  
    cout << "\nThe copy of v has been created.\n";  
    cout << "\nThe array elements of the copy:\n"  
          << setw(5) << v << endl;  
  
    w.remove(3);            // Erase the element at  
                           // position 3.  
    w.append(10.0F);        // Add a new element.  
    w.append(20.0F);        // And once more!  
  
    v = w;  
    cout << "\nAssignment done.\n";  
    cout << "\nThe elements after assigning: \n"  
          << setw(5) << v << endl;  
    v.insert( cv, 0);  
    cout << "\nThe elements after inserting "  
          << " the copy at position 0: \n"  
          << setw(5) << v << endl;  
    return 0;  
}
```