A C++ DYNAMIC ARRAY

C++ does not have a dynamic array inbuilt, although it does have a template in the Standard Template Library called vector which does the same thing. Here we define a dynamic array as a class, first to store integers only, and then as a template to store values of any type.

First we define the required functions and operations:

```
class Dynarray {
private:
    int *pa;                        // points to the array
    int length;                     // the # elements
    int nextIndex;                  // the next highest index value
public:
    Dynarray();                     // the constructor
    ~Dynarray();                    // the destructor
    int& operator[](int index);     // the indexing operation
    void add(int val);              // add a new value to the end
    int size();                     // return length
};
```

The class declares an integer pointer, pa, that will point to the array itself. length is the number of elements in the array, and nextIndex is the next available (empty) element. The class will have a default constructor which will initialize the variables, a destructor, which will do clean-up, and four member functions. We will overload the index operator [] so that we can index our array just like normal arrays, and provide a function for adding a new value at the end of the array. size will return the length of the array.

The class declration goes in a file Dynarray.h. Function definitions will go in a file Dynarray.cc.

The constructor

```
Dynarray::Dynarray() {
    pa = new int[10];
    for (int i = 0; i < 10; i++)
        pa[i] = 0;
    length = 10;
    nextIndex = 0;
}
```

The constructor creates a default size array (10 elements) using new and assigns it to pa. The for loop initializes the array elements to zero. length is set to 10 and nextIndex to 0. The constrcutor is called when an object is created by e.g.

```
Dynarry da;
```

The destructor

```
Dynarray::~Dynarray() {
    delete [] pa;
}
```

When a object of type Dynarray is created on the stack, as it will be by the above declaration, then care must be taken to clean-up any memory allocation when the object is destroyed (when its activation record is popped off the execution stack). This avoids memory leakage. The memory is recovered for re-use by using delete. The [] after delete indicate that an array is being recovered, not just a single variable.

The indexing operation

The heart of the class is the indexing operation. It must be capable of being used on the right of an assignment and on the left. E.g.

```
int x = da[5];
```

```
da[6] = 12;
```

In other words, it must produce a left-hand value as well as a right-hand value. We overload the [] operator (the array indexing operation) and return a reference to an integer. This can serve as both left and right-hand values:

```
int& Dynarray::operator[](int index) {
    int *pnewa;                                 // pointer for new array
    if (index >= length) {                      // is element in the array?
        pnewa = new int[index + 10];            // allocate a bigger array
        for (int i = 0; i < nextIndex; i++)     // copy old values
                pnewa[i] = pa[i];
        for (int j = nextIndex; j < index + 10; j++) // initialize remainder
                pnewa[j] = 0;
        length = index + 10;                    // set length to bigger size
        delete [] pa;                           // delete the old array
        pa = pnewa;                             // reassign the new array
    }
    if (index > nextIndex)                      // set nextIndex past index
        nextIndex = index + 1;
    return *(pa + index);                       // a reference to the element
}
```

The test is to make sure that the element begin indexed is in the array. If it is not, then we extend the array with the following sequence: create a new bigger array to include the element at index, copy the elements from the old, shorter array to the new array. Initialize any elements in the new array to zero that are past the end of the old array, delete the old array, reassign pa to the new array, and finally return a reference to the element at index. Set nextIndex 1 past index if necessary. The return type of int& is the reference and is obtained by dereferencing the pointer pa, incremented by index. If we returned a pointer, it would have to be dereference it in the code that uses it. Returning a reference avoids this.

The function add

```
void Dynarray::add(int val) {
    int *pnewa;
    if (nextIndex == length) {
        length = length + 10;
        pnewa = new int[length];
        for (int i = 0; i < nextIndex; i++)
                pnewa[i] = pa[i];
        for (int j = nextIndex; j < length; j++)
                pnewa[j] = 0;
        delete [] pa;
        pa = pnewa;
    }
    pa[nextIndex++] = val;
}
```

add does something similar to operator[], but works at the end of the array, making it bigger if necessary. Whereas we overloaded the operator [] for the main indexing operation, this is just an ordinary member function, and is called by. E.g.

```
da.add(15);
```

The function size

```
int Dynarray::size() {
    return length;
}
```

size just returns the current length of the array.

Sample program

A sample program using all of the features of the class is:

```
int main() {
    Dynarray da;                             // create an array object, size 10
    da.add(1);                               // add values to the end
    da.add(2);
    da.add(3);
    da[3] = 4;                               // use LHV for assignment
    for (int i = 0; i < da.size(); i++)      // get length of array using size()
        cout << da[i] << endl;               // print out using RHV
    da[12] = 5;                              // assign element past end of array
    for (int j = 0; j < da.size(); j++)      // size is now 22
        cout << da[j] << endl;               // print out all elements again in
    return 0;                                // bigger array
}
```

A template version

This array only stores integers. To make a dynamic array that stores any values, we can turn the class into a template:

```
template <class T>
class Dynarray
{
private:
    T *pa;
    int length;
    int nextIndex;
public:
    Dynarray();
    ~Dynarray();
    T& operator[](int index);
    void add(int val);
    int size();
};
```

The parameter for the template is T, i.e. any type can be passed in to instantiate the template. Of course, T* must be the type of the array and T& the return type for operator[]. Each function should also be turned into a templated version, even when T is not used. T has been substituted wherever we need the type of the array. Note that Dynarray<T> is now the prefix for each function, rather than just Dynarray as it was before.

```
template <class T>
Dynarray<T>::Dynarray() {
    pa = new T[10];
    for (int i = 0; i < 10; i++)
        pa[i] = 0;
    length = 10;
    nextIndex = 0;
}

template <class T>
Dynarray<T>::~Dynarray() {
    delete [] pa;
}

template <class T>
T& Dynarray<T>::operator[](int index) {
    T *pnewa;
    if (index >= length) {
        pnewa = new T[index + 10];
        for (int i = 0; i < nextIndex; i++)
```

```
                pnewa[i] = pa[i];
            for (int j = nextIndex; j < index + 10; j++)
                pnewa[j] = 0;
            length = index + 10;
            delete [] pa;
            pa = pnewa;
        }
        if (index > nextIndex)
            nextIndex = index + 1;
        return *(pa + index);
    }

    template <class T>
    void Dynarray<T>::add(int val) {
        T *pnewa;
        if (nextIndex == length) {
            length = length + 10;
            pnewa = new T[length];
            for (int i = 0; i < nextIndex; i++)
                pnewa[i] = pa[i];
            for (int j = nextIndex; j < length; j++)
                pnewa[j] = 0;
            delete [] pa;
            pa = pnewa;
        }
        pa[nextIndex++] = val;
    }

    template <class T>
    int Dynarray<T>::size() {
        return length;
}
```

Now we can do the following:

```
    Dynarry<int> da1;              // an array of integers
    Dynarray<float> da2;           // an array of floats
    Dynarray<Dynarray<int>> da3;   // and array of arrays of integers
```