Chapter 3

Data Structures Used in the Present Applications

In this chapter, we describe several useful data structures, along with their C++ implementation. In particular, we present dynamic data structures, with a size that is specified in run time rather than compilation time. Furthermore, we implement lists of objects with variable size. Finally, we implement flexible linked lists, with algorithms to merge and order them. These data structures are most useful throughout the book.

3.1 Data Structures

As discussed above, a variable defined and used during the execution of a program is just a particular place in the computer memory where the corresponding datum is stored. The name of a variable is actually a way to refer to it and access the datum stored in the computer memory.

In most applications, one must use not only individual variables but also large structures that may contain many variables. The structures observe some pattern, according to which the variables are ordered or related to each other. These structures are called data structures.

The most common data structure is the array. In the array, the variables are stored one by one continuously in the computer memory. Thus, they must all be of the same type, e.g., integer, float, double, etc.

The array is considered as a particularly efficient data structure. Because the variables are stored one by one in the computer memory, it is particularly easy to scan them in loops. However, the array is often not sufficiently flexible; its size and type are fixed and cannot be changed, and it is also impossible to add more items to it or drop items from it after it is constructed.

The array is particularly useful in implementing algorithms in linear algebra. Indeed, the matrix and vector objects that are often used in linear algebra can be naturally implemented in arrays. The natural implementation of these mathematical objects provides the opportunity to implement numerical algorithms in the same spirit in which they were developed originally.

This approach is further developed in C++, where arrays are used not only to store vectors and matrices efficiently but also to actually provide complete matrix and vector objects, with all the required arithmetic operations between them. These objects can then be used exactly as in the original mathematical algorithm; the code is free of any detail about storage issues and is transparent and easy to debug.

In summary, in C++, the mathematical algorithm is completely separate from the details of storage. The arrays are hidden inside the vector and matrix classes and are manipulated only through interface functions. Once the vector and matrix objects are ready, they are used as complete mathematical objects in the required numerical algorithm.

3.2 Templates in Data Structures

As we have seen above, an array must be homogeneous: it can contain variables of only one type. This feature leads naturally to the use of templates in C++. When vectors and matrices are defined as template classes, the type of variable in them is not determined in advance but rather remains implicit until a concrete object is actually constructed. The opportunity to use templates thus saves a lot of programming effort, because it allows the definition of data structures with every possible type of variable. For example, the vector and matrix classes can be used with integer, float, double, or complex type, or any other variable with constant size.

Templates, however, provide not only an efficient and economic programming style but also a suitable approach to the implementation of data structures. Indeed, data structures are independent of the particular type of variable in use. They can be better defined with a general type, to be determined later when used in a concrete application.

Using templates in the definition of data structures is thus most natural; it takes out of the way any distracting detail about the particular type of variable and lets the programmer concentrate on the optimal implementation of the pure data structure and its mathematical features.

The mathematical features of a data structure include also the way in which variables are organized in it. These features can be rather complex: for example, they can describe the way a particular variable can be accessed from another variable. The relations between variables are best implemented in the template class, from which the particular type of variable is eliminated.

So far, we have dealt only with the simplest data structure: the vector. In this data structure, the relation between the variables is determined by their order in the vector: a particular variable is related to the previous one and the next one in the vector. This is why vectors are particularly suitable for loops. In matrices, variables are also related to variables that lie above and below them in the matrix. Therefore, matrices are also suitable for nested loops.

In the next section, we introduce an implementation of vectors in which the dimension is determined in run time rather than compilation time. This property is most important in practical applications.

3.3 Dynamic Vectors

The implementation of the "vector" object in Chapter 2, Section 18, requires that its dimension 'N' be specified in compilation time. In many cases, however, the dimension is known only in run time. For example, the dimension may be a parameter read from an external file, specified in a recursive process, or passed as an argument to a function. In such cases, the dimension is not yet known in compilation time, and the "vector" class of Chapter 2, Section 18, cannot be used. The need for dynamic vectors whose dimension is determined in run time is clear.

In this section, we provide the required dynamic implementation. In this implementation, the dimension of the "dynamicVector" object is stored in a private data member. Thus, the "dynamicVector" object contains not only the components of the vector but also an extra integer to specify the dimension of the vector. The value of this integer can be set in run time, as required.

The memory used to store a "dynamicVector" object cannot be allocated in compilation time, because the actual dimension is not yet known then. Instead, the memory allocation is done in run time, using the reserved word "new". The "new" function allocates memory for a certain object and returns the address of this memory. For example,

```
double* p = new double;
```

allocates memory for a "double" variable and uses its address to initialize a pointer-to-double named 'p'.

The present implementation is flexible and dynamic, as required. Templates are used only to specify the type of component, not the number of components. This number is indeed determined dynamically during run time.

The data fields in the "dynamicVector" class are declared "protected" to make them accessible from derived classes to be defined later. Two data fields are used: the integer "dimension" that indicates the dimension of the vector and the pointer "component" that points to the components of the vector.

Because the dimension is not yet available, the "component" field must be declared as pointer-to-T rather than array-of-T's as in the "vector" class in Chapter 2, Section 18. It is only at the actual call to the constructor of the "dynamicVector" class that the required memory is allocated and the "component" points to the concrete array of components:

```
#include<stdio.h>
template<class T> class dynamicVector{
  protected:
    int dimension;
    T* component;
  public:
    dynamicVector(int, const T&);
    dynamicVector(const dynamicVector&);
    const dynamicVector& operator=(const dynamicVector&);
    const dynamicVector& operator=(const T&);
```

The constructors and assignment operators are only declared above. The actual definition will be given later on. Next, we define the destructor:

```
    ~dynamicVector(){
      delete [] component;
    }  //  destructor
```

The "delete[]" command used in the destructor deletes the entire array "component" and frees the memory occupied by it for future use. Note that the "dimension" field doesn't have to be removed explicitly. Because it is not a pointer, it is deleted implicitly by the default destructor built in the C++ compiler, which is invoked automatically at the end of every call to the above destructor.

Because the "dimension" field is protected, we need a public function to read it from ordinary classes that are not derived from the "dynamicVector" class:

```
      int dim() const{
        return dimension;
      }  //  return the dimension
```

The "operator()" defined below returns a reference to the 'i'th component in the current "dynamicVector" object. Because this reference is nonconstant, it can be changed by the user. The "operator[]", on the other hand, returns a constant reference, so the 'i'th component can only be read by it:

```
      T& operator()(int i){
        return component[i];
      }  //  read/write ith component

      const T& operator[](int i) const{
        return component[i];
      }  //  read only ith component
```

We also declare some member arithmetic operators that operate on the current "dynamic Vector" object. The complete definition will be given later on.

```
      const dynamicVector& operator+=(const dynamicVector&);
      const dynamicVector& operator-=(const dynamicVector&);
      const dynamicVector& operator*=(const T&);
      const dynamicVector& operator/=(const T&);
  };
```

This concludes the block of the "dynamicVector" class.

We now define the functions that are only declared above. In the constructor defined below, memory for the array of components is allocated in the initialization list, provided that the dimension is nonzero: this dynamic memory allocation is done by the "new" command. The address returned by this command is placed in the "component" field for future access:

```
  template<class T>
  dynamicVector<T>::dynamicVector(
         int dim = 0, const T& a = 0)
         : dimension(dim), component(dim ? new T[dim] : 0){
     for(int i = 0; i < dim; i++)
       component[i] = a;
  }  //  constructor
```

The same approach is used in the copy constructor:

```
  template<class T>
  dynamicVector<T>::dynamicVector(const dynamicVector<T>& v)
```

```
    : dimension(v.dimension),
     component(v.dimension ? new T[v.dimension] : 0){
    for(int i = 0; i < v.dimension; i++)
      component[i] = v.component[i];
  }  //  copy constructor
```

Next, we define the assignment operator:

```
  template<class T>
  const dynamicVector<T>&
  dynamicVector<T>::operator=(const dynamicVector<T>& v){
    if(this != &v){
```

We have just entered the "if" block that makes sure that the assignment operator is not called by a trivial call of the form "u = u". Now, we need to make the dimension of the current vector the same as that of the argument vector 'v':

```
      if(dimension != v.dimension){
         delete [] component;
         component = new T[v.dimension];
      }
```

This way, the array "component[]" has the same dimension as the array "v.component[]" and can be filled with the corresponding values in a standard loop:

```
      for(int i = 0; i < v.dimension; i++)
        component[i] = v.component[i];
      dimension = v.dimension;
    }
    return *this;
  }  //  assignment operator
```

This completes the definition of the assignment operator that takes a vector argument. Next, we define another assignment operator that takes a scalar argument. This scalar is simply assigned to all the components in the current dynamic vector:

```
  template<class T>
  const dynamicVector<T>&
  dynamicVector<T>::operator=(const T& a){
    for(int i = 0; i < dimension; i++)
      component[i] = a;
    return *this;
  }  //  assignment operator with a scalar argument
```

Next, we implement some useful arithmetic operators:

```
  template<class T>
  const dynamicVector<T>&
  dynamicVector<T>::operator+=( const dynamicVector<T>&v){
      for(int i = 0; i < dimension; i++)
```

```
        component[i] += v[i];
      return *this;
  }//  adding a dynamicVector to the current one

  template<class T>
  const dynamicVector<T>
  operator+(const dynamicVector<T>&u,
            const dynamicVector<T>&v){
    return dynamicVector<T>(u) += v;
  }  //  dynamicVector plus dynamicVector

  template<class T>
  const dynamicVector<T>
  operator-(const dynamicVector<T>&u){
    return dynamicVector<T>(u) *= -1.;
  }  //  negative of a dynamicVector
```

Finally, here is a function that prints a dynamic vector to the screen:

```
  template<class T>
  void print(const dynamicVector<T>&v){
    print("(");
    for(int i = 0;i < v.dim(); i++){
      printf("v[%d]=",i);
      print(v[i]);
    }
    print(")\n");
  }  //  printing a dynamicVector
```

The implementation of some arithmetic operations such as subtraction, multiplication, division by scalar, and inner product is left as an exercise. Assuming that these operators are also available, one can write all sorts of vector operations as follows:

```
  int main(){
    dynamicVector<double> v(3,1.);
    dynamicVector<double> u;
    u=2.*v;
    printf("v:\n");
    print(v);
    printf("u:\n");
    print(u);
    printf("u+v:\n");
    print(u+v);
    printf("u-v:\n");
    print(u-v);
    printf("u*v=%f\n",u*v);
    return 0;
  }
```

3.4 Lists

The vector and dynamic vector above are implemented as arrays of components of type ’T’. By definition, an array in C (as well as other programming languages) must contain components that are all of the same type and size. No array can contain components that occupy different amounts of memory.

In many applications, however, one needs to use sequences of objects that differ in size from each other. For example, one might need to use sequences of vectors whose dimensions are not yet known in compilation time. This kind of application can no longer use a standard array to store the vectors because of the different sizes they may take in the end. A more flexible data structure is needed.

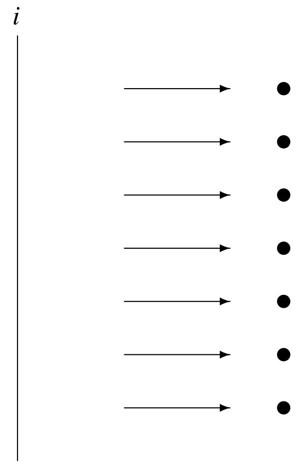

Figure 3.1. *Schematic representation of a list of objects. The arrows represent pointers, and the bullets represent constructed objects. i stands for the index in the array of pointers.*

The required data structure is implemented in the "list" class, which has an array of pointer-to-’T’ objects (see Figure 3.1). Although objects of type ’T’ may have different sizes (e.g., when ’T’ is a "dynamicVector"), their addresses are just integer numbers that occupy a fixed amount of memory and, hence, can be placed in an array. During run time, concrete objects of type ’T’ are placed in the addresses in the array. For example, if ’T’ is specified in compilation time as "dynamicVector", then the dimensions of the "dynamicVector" objects in the list are specified during run time using the constructor of the "dynamicVector" class and then placed in the addresses contained in the array in the list.

The length of the list, that is, the number of pointers-to-’T’ in the array in it, can also be determined dynamically in run time. As in the "dynamicVector" class, this is done by using an extra integer field to store the number of items. The "list" class thus contains two protected data fields: "number" to indicate the number of items and "item" to store their addresses. The detailed implementation is as follows:

```
template<class T> class list{
  protected:
    int number;
    T** item;
  public:
    list(int n=0):number(n), item(n ? new T*[n]:0){
    }  //  constructor

    list(int n, const T&t)
      : number(n), item(n ? new T*[n] : 0){
      for(int i=0; i<number; i++)
        item[i] = new T(t);
    }  //  constructor with T argument

    list(const list<T>&);
    const list<T>& operator=(const list<T>&);
```

The copy constructor and assignment operator are only declared above and will be defined later. The destructor defined below deletes first the pointers in the array in the "list" object and then the entire array itself:

```
    ~list(){
      for(int i=0; i<number; i++)
        delete item[i];
      delete [] item;
    }  //   destructor
```

Because the "number" field is protected, it cannot be read from ordinary functions. The only way to read it is by using the "size()" function:

```
    int size() const{
      return number;
    }  //  list size
```

Similarly, because the "item" field is protected, it cannot be accessed by ordinary functions. The only way to access its items is through "operator()" (read/write) or "operator[]" (read only):

```
    T& operator()(int i){
      if(item[i])return *(item[i]);
    }  //  read/write ith item

    const T& operator[](int i)const{
      if(item[i])return *(item[i]);
    }  //  read only ith item
};
```

This concludes the block of the "list" class, including the definitions of the constructor, destructor, and functions to read and access individual items in the list. Note that one should be careful to call "l(i)" or "l[i]" only for a list ’l’ that contains a well-defined ’i’th item.

The copy constructor and assignment operator are only declared in the class block above. Here is the actual definition:

```
template<class T>
list<T>::list(const list<T>&l):number(l.number),
      item(l.number ? new T*[l.number] : 0){
  for(int i=0; i<l.number; i++)
    if(l.item[i])
      item[i] = new T(*l.item[i]);
    else
      item[i] = 0;
}  //  copy constructor
```

Here is the definition of the assignment operator:

```
template<class T>
const list<T>&
list<T>::operator=(const list<T>& l){
   if(this != &l){
```

We have just entered the "if" block that makes sure that the assignment operator has not been called by a trivial call of the form "l = l". Now, we make sure that the current list contains the same number of items as the list 'l' that is passed as an argument. (In other words, the array "item[]" is modified to have the same dimension as the array "l.item[]"):

```
     if(number != l.number){
       delete [] item;
       item = new T*[l.number];
     }
```

We can now go ahead and copy the items in 'l' to the current "list" object:

```
     for(int i = 0; i < l.number; i++)
       if(l.item[i])
         item[i] = new T(*l.item[i]);
       else
         item[i] = 0;
     number = l.number;
   }
   return *this;
 }  //  assignment operator
```

Finally, we implement the function that prints the items in the list to the screen:

```
template<class T>
void print(const list<T>&l){
  for(int i=0; i<l.size(); i++){
    printf("i=%d:\n",i);
    print(l[i]);
  }
}  //  printing a list
```

3.5 Linked Lists

The "list" object in Section 3.4 is implemented as an array of pointers or addresses of objects. This array, however, is too structured: it is not easy to add new items or remove old ones. Furthermore, the number of items is determined once and for all when the list is constructed and cannot be changed afterward. These drawbacks make the "list" data structure unsuitable for many applications. We need to have a more flexible, easily manipulated, and unrestrained kind of list.

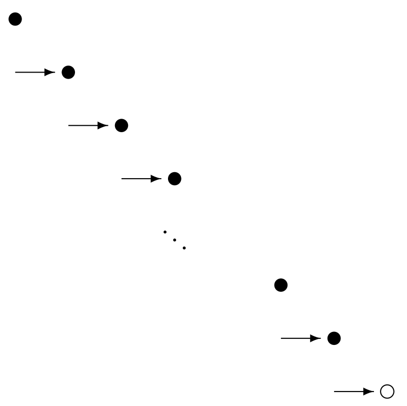

Figure 3.2. *Schematic representation of a linked list: each item (denoted by a bullet) contains a pointer (denoted by an arrow) to the next item (except the last item, which contains the null (or zero) pointer).*

Linked lists don't use arrays at all. Instead, each item also has a pointer that points to the next item in the list (see Figure 3.2). This structure allows the addition of an unlimited number of items to the linked list if necessary and also allows inserting and dropping items at every location in it.

Accessing items through pointers as above is also called "indirect indexing." Clearly, it is less efficient than the direct indexing used in arrays, because the items are no longer stored continuously in the computer memory. Nevertheless, its advantages far exceed this disadvantage. Indeed, the freedom to insert and remove items is essential in implementing many useful objects (see Chapter 4). Furthermore, the slight inefficiency in computer resources may be well worth it for the sake of better use of human resources: programmers who use linked lists can benefit from their special recursive structure to implement complex objects, functions, and algorithms.

The linked list is implemented in the template class "linkedList" defined below. (The items in the linked list are of type 'T', to be defined later in compilation time.) The "linkedList" class contains two data fields: "item", which contains the first item in the linked list, and "next", which contains the address of the rest of the linked list. Both data fields are declared "protected", so they can be accessed from derived classes later on.

The definition of the "linkedList" object is recursive: the shorter linked list that contains all items but the first one is defined recursively as a "linkedList" object as well and is placed in the address in the field "next". This recursive pattern is useful in many basic operations.

Here is the full implementation of the "linkedList" class:

```
template<class T> class linkedList{
  protected:
    T item;
    linkedList* next;
  public:
    linkedList():next(0){
    }  //  default constructor

    linkedList(T&t, linkedList* N=0)
      : item(t),next(N){
    }  //  constructor
```

The data fields "item" and "next" can also be read (although not changed) from ordinary functions by the public member functions "operator()" and "readNext()" defined below:

```
    const T& operator()() const{
      return item;
    }  //  read item field

    const linkedList* readNext() const{
      return next;
    }  //  read next
```

The assignment operator is only declared here, and will be defined later on. The recursive pattern of the linked list is particularly useful in the copy constructor. In fact, it needs only to copy the "item" field and then be applied recursively to the shorter linked list that contains the rest of the items:

```
    const linkedList& operator=(const linkedList&);
    linkedList(const linkedList&l):item(l()),
        next(l.next ? new linkedList(*l.next):0){
    }  //  copy constructor
```

Here, all the work is done in the initialization list: the "item" field in the constructed object is initialized to be "l()", which is just the first item in 'l' (accessed by the "operator()" defined above). The "next" field in the constructed object is then initialized by the "new" command and a recursive call to the copy constructor itself.

The recursive structure is also useful in defining the destructor. In fact, the destructor should contain only one command that deletes the field "next", which points to the shorter linked list that contains all but the first item. When this pointer is destroyed, the same destructor is invoked automatically to destroy its content (the shorter linked list) and free it for further use. This way, the rest of the items in the linked list are destroyed as well, with no need to write any explicit code:

```
    ~linkedList(){
      delete next;
      next = 0;
    }  //  destructor
```

The first field in the "linkedList" object, "item", needs no explicit command to destroy it, because it is not a pointer. It is destroyed automatically right after "next" is destroyed.

Further, we define recursive functions that return a reference to the last item in the linked list and the number of items in it. These functions are then used to append a new item at the end of the linked list:

```
      linkedList& last(){
        return next ? next->last() : *this;
      }  //  last item

      int length() const{
        return next ? next->length() + 1 : 1;
      }  //  number of items

      void append(T&t){
        last().next = new linkedList(t);
      }  //  append item
```

The following functions insert new items at different locations in the linked list. The function "insertNextItem()" places the new item right after the first item in the linked list:

```
      void insertNextItem(T&t){
        next = new linkedList(t,next);
      }  //  insert item in second place
```

The function "insertFirstItem()" places the new item just before the first item in the linked list:

```
      void insertFirstItem(T&t){
        next = new linkedList(item,next);
        item = t;
      }  //  insert item at the beginning
```

Finally, we also declare some more functions, including functions that drop certain items from certain places in the linked list. (The full definition of these functions will be given later.)

```
      void dropNextItem();
      void dropFirstItem();
      const linkedList& operator+=(linkedList&);
      linkedList& order(int);
      template<class S>
        const S truncateItems(double, const S&);
      template<class S>
        const S dropPositiveItems(
            int, const S&, double);
      template<class S>
        const S maskItems(
            const dynamicVector<int>&, const S&);
  };
```

Note that the three latter functions also use an extra template symbol ’S’ to serve as the type of the returned object. When a user calls any of these functions, he/she must pass to it (by reference) a concrete argument to specify ’S’.

This concludes the block of the "linkedList" class; the functions that are only declared above will be defined below.

One may rightly ask why the ’T’ argument in the above constructor and other functions like "insertNextItem" and "insertFirstItem" has not been declared constant. After all, the current "linkedList" object is the one that is being changed in these functions, and surely there is no need to change the ’T’ argument as well. Declaring it as a constant could protect the code from compilation- and run-time errors, couldn’t it?

The answer is that usually it is indeed a good idea to declare the argument as a constant. Here, however, we plan to derive from the "linkedList" class an unusual class with a constructor that changes its argument as well. This unusual class is described in Chapter 13.

Next, we introduce the assignment operator. This operator also benefits from the recursive pattern of the linked list. In fact, after the first item has been assigned a value, the assignment operator is applied recursively to the rest of the linked list:

```
template<class T>
const linkedList<T>&
linkedList<T>::operator=(const linkedList<T>&L){
  if(this != &L){
    item = L();
    if(next){
      if(L.next)
        *next = *L.next;
      else{
        delete next;
        next = 0;
      }
    }
    else
      if(L.next)next = new linkedList(*L.next);
 }
 return *this;
}  //  assignment operator
```

The main advantage of linked lists is the opportunity to insert and drop items. Here is the function that drops the second item in the linked list. (The next item can be dropped by applying the same function to the shorter linked list of which it is the second item.)

```
template<class T>
void linkedList<T>::dropNextItem(){
   if(next){
     if(next->next){
       linkedList<T>* keep = next;
       next = next->next;
```

```
        keep->item.~T();
      }
      else{
        delete next;
        next = 0;
      }
    }
    else
      printf("error: cannot drop nonexisting next item\n");
}  //  drop the second item from the linked list
```

The above function is also used in the following function to drop the first item in the linked list:

```
template<class T>
void linkedList<T>::dropFirstItem(){
    if(next){
      item = next->item;
      dropNextItem();
    }
    else
      printf("error: cannot drop first item; no next.\n");
}  //  drop the first item in the linked list
```

Note that when a linked list contains only one item, it is never dropped. If an attempt is made to drop it, then an error message is printed to the screen.

3.6 Recursive Truncation

Here we implement the "truncateItems" member function declared above. This function drops from the current linked list items that are relatively small in magnitude. Furthermore, we highlight a slight inefficiency problem in the recursive implementation of the function, and show how it can be fixed.

It is assumed that the ’T’ class has a "getValue()" function that returns the value of the ’T’ object. If this value is smaller (in absolute value) than the prescribed threshold, then the item is dropped from the linked list. If, on the other hand, the ’T’ class has no "getValue()" function, then the present function cannot be called.

The detailed implementation is as follows:

```
template<class T>
void linkedList<T>::truncateItems(double threshold){
  if(next){
    if(abs(next->item.getValue()) <= threshold){
      dropNextItem();
      truncateItems(threshold);
    }
```

In the above inner "if" block, the second item in the linked list is considered for dropping. If it is indeed dropped, then the third item replaces it as the new second item and is then considered for dropping in the recursive call. If, on the other hand, it is not dropped, then the third item remains in its original place and is considered for dropping by a recursive call applied to the "tail" of the current linked list (the shorter linked list that starts from the second item):

```
      else
        next->truncateItems(threshold);
    }
```

Finally, the first item is also considered for dropping, provided that it is not the only item in the linked list:

```
    if(next&&(abs(item.getValue()) <= threshold))
        dropFirstItem();
  }  //  truncate small items
```

Note how the recursive pattern of the linked list is used in the above implementation. First, the second item in the linked list is considered for dropping. Then, the function is called recursively for the remaining items.

Here one may ask: why not consider the first item for dropping by the "dropFirstItem" function, and then apply the function recursively to the remaining items? The reason for not using this approach is that the last item in the linked list that is dropped (if it is indeed small enough) by the innermost recursive call can never be dropped by the "dropFirstItem" function, but rather by the "dropNextItem" function called from the previous item. Indeed, the "dropFirstItem" function cannot drop the only item in a linked list. Therefore, the recursion must always look ahead and truncate the next item, rather than the current item.

This is why the first item in the linked list is considered for dropping last, at the end of the above code segment. Still, there is an inefficiency problem in the above implementation. Indeed, the first item in the linked list is checked for dropping twice: once in the last code line in the function, and another time in the recursive call (if the second item should indeed be dropped, so the inner "if" block above is indeed entered). Furthermore, this problem is even enhanced in inner recursive calls, which may needlessly check the first item in the linked lists to which they are applied. To avoid this redundant work, the last code segment can be separated from the above function and placed in another function, which is carried out only once at the end of the process. In other words, the above function can be split into two functions as follows. The first function, "truncateTail", drops small items only from the "tail" of the current linked list (that is, from the second item onward). For this purpose, it uses the same recursion as in the original function:

```
  template<class T>
  void linkedList<T>::truncateTail(double threshold){
    if(next){
      if(abs(next->item.getValue()) <= threshold){
        dropNextItem();
        truncateTail(threshold);
      }
      else
        next->truncateTail(threshold);
    }
  }  //  truncate small items from the tail only
```

Furthermore, the next function, "truncateFirstItem", truncates only the first item (that is, if it is indeed sufficiently small in magnitude and is not the only item in the current linked list):

```
template<class T>
void linkedList<T>::truncateFirstItem(double threshold){
  if(next&&(abs(item.getValue()) <= threshold))
      dropFirstItem();
}  //  truncate the first item only
```

The required "truncateItems" function can now be reimplemented more efficiently as a combination of the above two functions:

```
template<class T>
void linkedList<T>::truncateItems(double threshold){
  truncateTail(threshold);
  truncateFirstItem(threshold);
}  //  truncate small items
```

The recursive structure of the linked list is also useful to print. In fact, after the first item has been printed, the "print" function is applied recursively for the rest of the items in the linked list:

```
template<class T>
void print(const linkedList<T>&l){
  printf("item:\n");
  print(l());
  if(l.readNext())print(*l.readNext());
}  //  print a linked list
```

Here is how linked lists are actually used in a program:

```
int main(){
  linkedList<double> c(3.);
  c.append(5.);
  c.append(6.);
  c.dropFirstItem();
  print(c);
  return 0;
}
```

3.7 Nonrecursive Truncation

Here we provide a slightly more efficient nonrecursive version of the above "truncateItems" function. Furthermore, this version also returns the sum of the dropped items.

The type of this sum is the same as the type returned by the "getValue" function of the 'T' class. At this point, this type is still unspecified. Therefore, it is denoted by the template symbol 'S', to be specified later by the user who actually calls the function.

In our applications, 'S' is always "double"; in more advanced applications, however, it may also stand for a complex number.

The particular interpretation of the symbol 'S' depends on the argument "compare", passed to the function by reference. This argument is used to determine which items in the current linked list are too small and should be dropped from it:

```
template<class T>
template<class S>
const S linkedList<T>::truncateItems(double threshold,
        const S& compare){
  S sum=0.;
  if(next){
    int dropped = 0;
```

The following loop uses the pointer-to-linked-list "runner" to advance from item to item in the current linked list. In the loop, not the item pointed at by "runner" but rather the item that follows it is considered for dropping by the "dropNextItem" function. If this item should indeed be dropped, then the local integer variable "dropped" takes the value 1, which means that "runner" has already advanced to the next item. If, on the other hand, this item shouldn't be dropped, then "dropped" takes the value 0, which means that "runner" should be advanced manually to the next item to continue the loop:

```
    for(linkedList<T>* runner = this;
        runner->next;
        runner = dropped ? runner : runner->next){
      dropped = 0;
      if(fabs(runner->next->item.getValue())
          <= threshold * fabs(compare)){
        sum += runner->next->item.getValue();
        runner->dropNextItem();
        dropped = 1;
      }
    }
  }
```

Finally, the first item in the current linked list should also be considered for dropping:

```
  if(next&&(fabs(item.getValue())
          <= threshold * fabs(compare))){
    sum += item.getValue();
    dropFirstItem();
  }
  return sum;
}  //  truncate small items
```

Here are two other functions that are very similar to the above function. In fact, they differ from it only in the criterion for dropping items. They assume that the ’T’ class has also a "getColumn" function that returns an integer that characterizes the place of the item in the linked list. In fact, the integer number returned by the "getColumn" function may be viewed as the (possibly discontinuous) index of the item in the linked list: the first item in the linked list has the minimal index, the next item has a (possibly much) larger index, and so on, until the last item in the linked list, which has the maximal index.

The following "dropPositiveItems" function takes two extra arguments: the integer "diag" and the ’S’-object "center". The criteria for dropping an item are whether its index returned from the "getColumn" function is different from "diag", and whether its magnitude is small or its sign is the same as that of "center":

```
template<class T>
template<class S>
const S linkedList<T>::dropPositiveItems(
        int diag, const S&center, double threshold){
  S sum=0.;
  if(next){
    int dropped = 0;
    for(linkedList<T>* runner = this;
        runner->next;
        runner = dropped ? runner : runner->next){
      dropped = 0;
      if((double(runner->next->item.getValue() / center)
              >= -threshold)&&
          (runner->next->item.getColumn() != diag)){
        sum += runner->next->item.getValue();
        runner->dropNextItem();
        dropped = 1;
      }
    }
  }
  if(next&&(double(item.getValue() / center) >= -threshold)
          &&(item.getColumn() != diag)){
    sum += item.getValue();
    dropFirstItem();
  }
  return sum;
}  //  drop positive or small items
```

Another function that is very similar to the above functions is the "maskItems" function. In this function, the items in the linked list are "masked" by a vector of integers that is passed to it by reference. This way, only items whose "getColumn()" indices correspond to nonzero components in this vector remain in the current linked list, whereas all the other items are dropped from it.

The second argument that is passed to the function is never actually used in the code. Its only purpose is to specify the type ’S’ when the function is actually called by the user. This type is then used to define the local variable "sum", which contains the sum of the dropped items:

```
template<class T>
template<class S>
const S linkedList<T>::maskItems(
    const dynamicVector<int>& mask, const S&){
  S sum=0.;
  if(next){
    int dropped = 0;
    for(linkedList<T>* runner = this;
        runner->next;
        runner = dropped ? runner : runner->next){
      dropped = 0;
      if(!mask[runner->next->item.getColumn()]){
        sum += runner->next->item.getValue();
        runner->dropNextItem();
        dropped = 1;
      }
    }
  }
  if(next&&(!mask[item.getColumn()])){
    sum += item.getValue();
    dropFirstItem();
  }
  return sum;
}  //  masking items
```

3.8 The Merging Problem

Users of linked lists may want to manipulate them in many ways. For example, they may want to insert items into them or drop items from them. Furthermore, they would like to be able to do this without dealing with storage issues such as pointers and addresses. The functions that perform these tasks are described above.

Users of linked lists may want to do another operation: merge two ordered linked lists with each other while preserving the order. Completing this task efficiently is called the merging problem.

The power of templates is apparent here. Indeed, it is particularly convenient to deal with an unspecified type 'T', provided that it supports some order. This way, we can concentrate on the data structure under consideration rather than the type used in the variables in it. It is only when the merging function is called that the concrete type is specified.

In the following, we describe a function that merges two linked lists into a single linked list while preserving the order. It is assumed that the type 'T' of the items in the linked list supports a complete priority order; that is, every two 'T' objects can be compared by the '<', '>', and "==" binary operators.

This priority order may take all sorts of different interpretations. Indeed, if the '<', '>', and "==" operators of the 'T' class have the usual meaning, then the items in the linked list should be ordered in increasing-value order, from the smallest to the largest. If, on the other hand, these operators are interpreted in the 'T' class in terms of absolute value, then the items in the linked list should be ordered in increasing-magnitude order, from the smallest in magnitude to the largest in magnitude (see exercises in Section 3.22).

In the applications used later in the book, however, neither of these interpretations is used. In fact, the above priority order is induced in terms of the "getColumn()" function available in the 'T' class: a 'T' object is prior to ("smaller" than) another one not if it is smaller than it in (absolute) value, but rather if its "getColumn()" index is smaller than that of the other one.

It is also assumed that the current "linkedList" object and the "linkedList" argument that is merged into it are well ordered in the sense that each item is prior to ("smaller" than) the next item.

The purpose is to merge the linked list that is passed as an argument into the current linked list while preserving the correct priority order. This operation is most useful, particularly in the sparse matrix implemented in Chapter 16.

It is also assumed that the 'T' class supports a "+=" operator. With these assumptions, the "+=" operator that merges a linked list into the current linked list can also be defined.

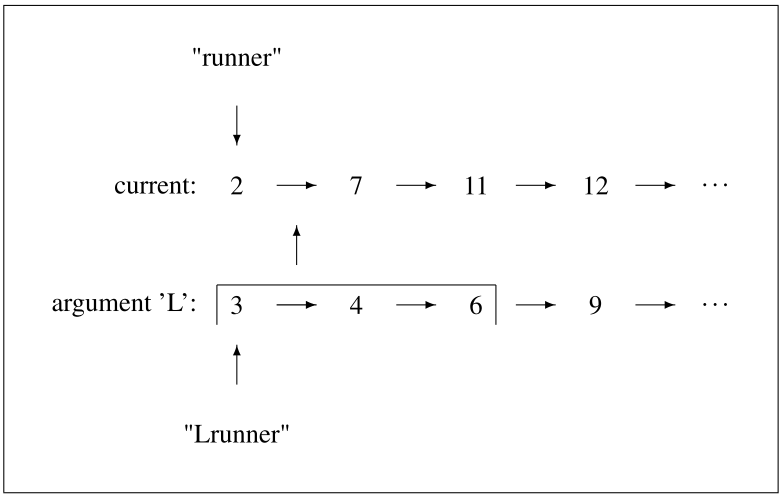
"runner"
current: 2 → 7 → 11 → 12 → ···
argument 'L': 3 → 4 → 6 → 9 → ···
"Lrunner"

Figure 3.3. *Merging two linked lists while preserving order. The items in the top linked list (the current object) are scanned by the pointer "runner" in the outer loop. The items in the bottom linked list 'L' (the argument) are scanned by the pointer "Lrunner" in the inner loop and inserted in the right place.*

The code uses two “runners” to scan the items in the linked lists (see Figure 3.3). The main runner scans the items in the current linked list. The room between the item pointed at by this runner and the item that follows it should be filled by items from the second linked list that is passed as an argument, provided that they indeed belong there in terms of order. For this purpose, a secondary runner, called "Lrunner", is used to scan the items in the second linked list that indeed belong in the location pointed at by the main runner. These items are then inserted one by one into the current linked list in their correct places.

In case an item in the second linked list has the same priority order '<' as an existing item in the current linked list, that is, it is equal to it in terms of the "==" operator of the 'T' class, then it is added to it using the "+=" operator of the 'T' class. This approach is indeed suitable for the present applications, since the "getColumn()" index remains unchanged under the "$+ =$" operator of the ’T’ class, so the original order is preserved. In the other example mentioned above, on the other hand, in which the priority order in the ’T’ class is induced by (absolute) value, the calls to the "$+ =$" operator of the ’T’ class must be removed from the code below, so each item in the second linked list is placed right after the item smaller than or equal to it in the current linked list. (The code line that follows such a call can be made conditioned, to avoid repetition of the same item in the resulting list.)

```
template<class T>
const linkedList<T>&
linkedList<T>::operator+=(linkedList<T>&L){
  linkedList<T>* runner = this;
  linkedList<T>* Lrunner = &L;
```

Initially, "Lrunner" points to the linked list ’L’ that is passed to the function as an argument. However, in order to start the merging process, we must first make sure that the first item in the current linked list is prior (according to the priority order ’<’) to the first item in ’L’, "L.item". If this is not the case, then "L.item" must first be placed at the beginning of the current linked list and "Lrunner" advanced to the next item in ’L’:

```
  if(L.item < item){
    insertFirstItem(L.item);
    Lrunner = L.next;
  }
  for(; runner->next; runner=runner->next){
```

Here we enter the main loop. Initially, "runner" points to the entire current linked list. Then, it is advanced gradually to point to subsequent sublists that contain fewer and fewer items. The loop terminates when "runner" points to the sublist that contains only the last item in the original linked list.

We are now ready to add an item from ’L’ to the current linked list. If the item pointed at by "Lrunner" has the same priority as the item pointed at by "runner", then it is added to it using the "+=" operator available in the ’T’ template class:

```
    if(Lrunner&&(Lrunner->item == runner->item)){
      runner->item += Lrunner->item;
      Lrunner = Lrunner->next;
    }
    for(; Lrunner&&(Lrunner->item < runner->next->item);
            Lrunner = Lrunner->next){
```

We now enter the inner loop, in which the items in ’L’ pointed at by "Lrunner" are added one by one to the current linked list and placed in between the item pointed at by "runner" and the item that follows it. Once an item from ’L’ has been added to the current linked list, the "runner" pointer must be advanced to skip it:

```
      runner->insertNextItem(Lrunner->item);
      runner = runner->next;
    }
  }
```

The inner and outer loops are now complete. However, 'L' may still contain more items that should be placed at the end of the current linked list. Fortunately, "runner" and "Lrunner" were defined before the loops started, so they still exist. In fact, at this stage, "runner" points to the last item in the current linked list, and "Lrunner" points to the remaining items in 'L', if any. These items are appended to the current linked list as follows:

```
    if(Lrunner&&(Lrunner->item == runner->item)){
      runner->item += Lrunner->item;
      Lrunner = Lrunner->next;
    }
    if(Lrunner)runner->next=new linkedList<T>(*Lrunner);
    return *this;
  }  //  merge two linked lists while preserving order
```

This completes the merging of 'L' into the current linked list while preserving order, as required.

3.9 The Ordering Problem

The ordering problem is as follows: find an efficient algorithm to order a given disordered list of items. Naturally, this algorithm requires changing the order in the original list. Thus, the linked list implemented above, which easily accepts every required change, is the obvious candidate for the data structure for the required algorithm.

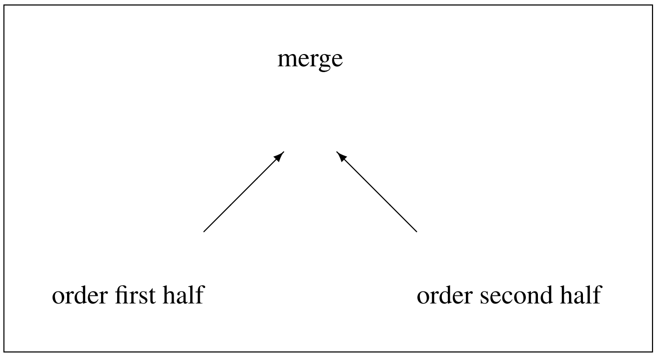

Figure 3.4. *The "order()" function that orders a disordered linked list: the original list is split into two sublists, which are ordered recursively and merged (while preserving order).*

Here, we use the "+=" operator that merges two ordered linked lists to define the "order()" template function that orders a disordered linked list (see Figure 3.4). The algorithm for doing this uses recursion as follows. Divide the disordered linked list into two sublists. Apply the "order()" function separately to each of these sublists. Then, merge them into a single, well-ordered list using the above "+=" operator.

The complete code is as follows:

```
template<class T>
linkedList<T>&
linkedList<T>::order(int length){
  if(length>1){
    linkedList<T>* runner = this;
    for(int i=0; i<length/2-1; i++)
      runner = runner->next;
```

At the end of this short loop, "runner" points to the item that lies at the middle of the current linked list. The "runner" pointer is now being used to define the pointer "second", which points to the second half of the original linked list:

```
    linkedList<T>* second = runner->next;
    runner->next = 0;
```

We are now ready for the recursion. The "order()" function is applied recursively to its first and second halves, before they are merged by the "+=" operator:

```
    order(length/2);
    *this += second->order(length-length/2);
  }
  return *this;
}  //  order a disordered linked list
```

This completes the ordering of the current linked list.

3.10 Vectors vs. Lists

So far, we have discussed four different kinds of vectors and lists: vector, dynamic vector, list, and linked list. The pros and cons of these objects are summarized in Table 3.1.

The "vector" object uses a standard array, which allows the most efficient loops. However, the components stored in the array must all have the same size, and the dimension of the array must be determined in compilation time. The "dynamicVector" object improves on "vector" by allowing the dimension to be set at run time rather than compilation time. However, the components in the array must still have the same size. The "list" object improves the situation further by allowing the items in the list to have different sizes. However, this comes at the price of using indirect indexing, which may slow down loops over the list. The "linkedList" object improves on all the others by having extra flexibility in inserting new items and removing old ones. However, this comes at the price of using not only indirect indexing but also expensive recursion to manipulate the recursively defined linked list. Nevertheless, the flexible nature of the linked list makes it most useful in this book and elsewhere (see Chapter 4).

Below, we show how the "linkedList" object can be improved further to have other desirable features. Naturally, this comes at the price of using extra storage and more complicated definitions and functions. In this book, the "linkedList" object is sufficient, and the more complex data structures are actually never used. They are discussed here briefly only for the sake of completeness.

Table 3.1. *Different kinds of vectors and lists and their pros and cons.*

	vector	dynamicVector	list	linkedList
Storage	fixed array	dynamic array	array of pointers	recursive addressing
Pros	efficient storage	efficient storage	variable-size items	variable-size items
	efficient loops	efficient loops		high flexibility
Cons	same-size items	same-size items	indirect indexing	indirect indexing
	fixed dimension			expensive recursion

3.11 Two-Sided Linked Lists

A data structure is characterized by the way one can access data about variables in it. For example, in a linked list, each item has access to the next item only. An item cannot access the item that points to it or the previous item in the linked list.

In some algorithms, this may cause a problem. There may be a need to loop on the linked list in the reverse direction or to know what the previous item in standard loops is. In such cases, one must use a two-sided linked list.

The "twoSidedLinkedList" class can be derived from the "linkedList" class by adding one field of type pointer-to-"twoSidedLinkedList". This field, named "previous", should contain the address of the previous item in the linked list. Because we don't actually use this class in this book, we omit the details.

3.12 Trees

In linked lists, each item contains a pointer to the next item. But what if it contained two pointers? We would then have a binary tree (see Figure 3.5 and Chapter 18 in [69]):

```
template<class T> class binaryTree{
  protected:
    T item;
    binaryTree* left;
    binaryTree* right;
  public:

      ...

};
```

In a linked list, access to the individual items is not always easy. In fact, accessing the last item requires stepping along a path whose length is the length of the entire list. In a tree, on the other hand, this task is much easier and requires a path whose length is at most the logarithm of the number of items.

In object-oriented programming, however, there is little need to access individual items in the linked list. One treats the linked list as a whole and tries to avoid accessing individual items in it. Thus, we use linked lists rather than trees in this book.

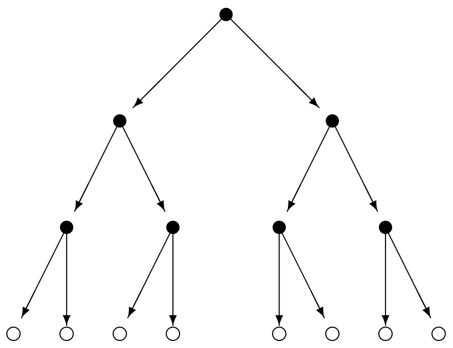

Figure 3.5. *Schematic representation of a binary tree with three levels. The arrows represent pointers, and the bullets represent constructed objects. The circles at the lowest level stand for the null (zero) pointer.*

Furthermore, binary trees lack the natural order of items available in linked lists. Although one can order the items recursively (e.g., left, middle, right), this order can be much more complicated than the natural order in linked lists.

One can also define more general trees by replacing the "left" and "right" pointers by a pointer to a linked list of trees:

```
template<class T> class tree{
  protected:
    T item;
    linkedList<tree<T> >* branch;
  public:

    ...

};
```

This tree may contain any number of subtrees in a linked list. This linked list is pointed at by the private member "branch". Note that the '>' symbols in the definition of "branch" are separated by a blank space to distinguish them from the string ">>", which has a completely different meaning in the "iostream.h" standard library.

In the above data structure, it is particularly easy to remove existing subtrees and add new ones. Two-sided trees can also be derived by adding a pointer field named "parent" to contain the address of the parent tree.

3.13 Graphs

In the above trees, an item can point only to new items; it cannot point to parent or ancestor trees or even sibling subtrees. Thus, a tree cannot contain circles.

In a graph, on the other hand, an item can point not only to new items but also to existing ones; be they ancestors, siblings, or even itself.

In fact, the mathematical definition of a graph is a set of nodes numbered $1,2,3,\dots,N$ and a set of edges (pairs of nodes). The set of edges is commonly denoted by E.

There are two kinds of graphs. In oriented graphs, each edge is an ordered pair of nodes, so the pair (i,j) (also denoted by $i \to j$) is not the same as (j,i) (also denoted by $j \to i$). It may well be that $(i,j) \in E$ but $(j,i) \notin E$, or vice versa (see Figure 3.6).

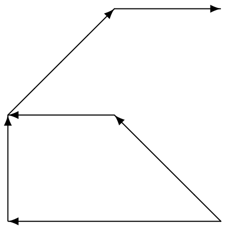

Figure 3.6. *Schematic representation of an oriented graph.*

In nonoriented graphs, on the other hand, a pair has no order in it: (i,j) is the same as (j,i), and either both are in E or both are not in E (see Figure 3.7).

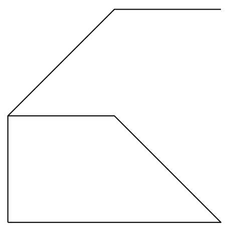

Figure 3.7. *Schematic representation of a nonoriented graph.*

In a graph, a set of k edges of the form

$$(i_1,i_2),\ (i_2,i_3),\ (i_3,i_4),\ldots,\ (i_{k-1},i_k),\ (i_k,i_1)$$

is called a circle of k edges. For example, the triplet (i,j), (j,k), and (k,i) forms a circle of three edges, the pair of edges (i,j) and (j,i) forms a circle of two edges, and even the single edge (i,i) by itself forms a circle of one edge only. These circles, although not allowed in trees, must be allowed in the implementation of graphs.

Nevertheless, the above implementation of trees is also suitable for graphs, provided that the rules are changed so that circles are allowed. This means that the constructor in the "graph" class must be different from that in the "tree" class. The tree constructor must use the "new" command in every subtree in the "$\star$branch" linked list, whereas the graph constructor may also use existing addresses in this linked list.

The fact that circles are allowed means that there is no natural order in the items in the graph. This is a drawback in the above implementation, because there is no natural way to loop over the nodes. In the next chapter, we'll see more practical implementations for graphs in which the original order $1,2,3,\ldots,N$ is preserved.

3.14 Dynamic Matrices

For the same reasons as in dynamic vectors, one would also like to have dynamic matrices, whose dimensions (width and height) are specified in run time rather than in compilation time. This class is implemented below.

The "dynamicMatrix" class is derived from the base "dynamicVector" class by introducing an extra integer data field to indicate the width of the matrix:

```
template<class T>
class dynamicMatrix : public dynamicVector<T>{
    int N;
  public:
    dynamicMatrix(int, int, const T&);
```

The constructor declared above will be defined later on.

The elements in the "dynamicMatrix" object can be accessed by "operator()" with two integer arguments to indicate the location of the element in the matrix. Indeed, thanks to the fact that the array "component" is defined only as "protected" (rather than "private") in the base "dynamicVector" class, it can be accessed from the derived "dynamicMatrix" class as well:

```
    T& operator()(int i, int j){
      return component[i*N+j];
    }  //  access (i,j)th component
```

With this operator, once the dynamic matrix 'A' is well defined in some code, its (i, j)th element $A_{i,j}$ can be accessed (for reading or writing) as "A(i,j)".

Another version of "operator()" can only read (but not change) the (i, j)th element of a dynamic matrix. To distinguish this version from the previous one, it takes a dummy string argument:

```
    const T& operator()(int i, int j, char*) const{
      return component[i*N+j];
    }  //  (i,j)th component (read only)
```

To call this version, one should add just any string to the list of arguments. For example, "A(i,j,"read")" reads the (i, j)th element in the dynamic matrix 'A'.

The following member functions return the dimensions of the matrix: height (or length) and width.

```
    int height() const{
      return dim()/N;
    }  //  height of grid

    int length() const{
      return dim()/N;
    }  //  length of grid

    int width() const{
      return N;
    }  //  width of grid
```

Since the derived class contains an extra data field that is absent in the base class, it makes no sense to convert the base-class object into a derived-class object, because the extra data field would remain undefined. This is why arithmetic operators like "*=" and "+=" must be redefined in the derived class: because the original base-class versions return a base-class object that cannot be converted to the required derived-class object, they cannot be simply inherited, and must be overriden by new versions. The present versions rewritten here indeed return derived-class objects, as required:

```
      const dynamicMatrix& operator+=(const dynamicMatrix&);
      const dynamicMatrix& operator-=(const dynamicMatrix&);
      const dynamicMatrix& operator*=(const T&);
      const dynamicMatrix& operator/=(const T&);
  };
```

The detailed definitions of arithmetic operators are left as an exercise, with a solution in Section A.7 in the Appendix. This completes the block of the "dynamicMatrix" class.

Because the data fields in the base "dynamicVector" class are declared as "protected" rather than "private", they can be accessed and indeed changed in the derived "dynamicMatrix" class. This property is useful in the constructor that takes integer arguments. Indeed, this constructor first implicitly calls the default constructor of the base "dynamicVector" class, which creates a trivial "dynamicVector" object with no components at all. The data fields in this dummy object are then reconstructed by the present constructor of the "dynamicMatrix" class to assume meaningful values. This can be done thanks to the access privilege that derived classes have to "protected" members of base classes:

```
template<class T>
dynamicMatrix<T>::dynamicMatrix(
    int m=0, int n=0, const T&t=0){
  dimension = n*m;
  N = n;
  component = dimension ? new T[dimension] : 0;
  for(int i=0; i<dimension; i++)
    component[i] = t;
}  //  constructor
```

The copy constructor doesn't have to be defined, because the default copy constructor works just fine: it first implicitly invokes the copy constructor of the base "dynamicVector" class to copy the inherited data fields, and then copies the remaining data field 'N'. The same is true for the assignment operator.

3.15 Minors

The dynamic matrices defined above are now used to compute the inverse matrix, using Cramer's formula:

$$(A^{-1})_{i,j} = (-1)^{i+j}\frac{\det(A^{(j,i)})}{\det(A)},$$

where $A^{(j,i)}$ is the (j,i)th minor of A, namely, the smaller matrix obtained from A by dropping the jth row and the ith column from it.

To implement this formula, we first write the function that computes the (k,l)th minor of A, where k and l are input parameters:

```
template<class T>
const dynamicMatrix<T>  minor(int k, int l,
    const dynamicMatrix<T>&m){
  dynamicMatrix<T> minorkl(m.height()-1,m.width()-1);
  int ii=-1;
  for(int i=0; i<m.height(); i++)
    if(i!=k){
      ii++;
      int jj=-1;
      for(int j=0; j<m.width(); j++)
        if(j!=l)
          minorkl(ii,++jj) = m(i,j,"read");
    }
  return minorkl;
}  //  (k,l)th minor of m
```

3.16 Determinant of a Small Matrix

Using the above "minor" function, one can now compute the determinant of a matrix recursively, as follows:

```
template<class T>
const T det(const dynamicMatrix<T>&A){
  if(A.height()==1)
    return A(0,0,"read");
  T detA = 0.;
  int sign = 1;
  for(int j=0; j<A.height(); j++){
    detA += sign * A(0,j,"read") * det(minor(0,j,A));
    sign *= -1;
  }
  return detA;
}  //  determinant using recursion
```

This implementation follows from the recursive definition of the determinant of a matrix. However, its cost grows exponentially with the order of the matrix. Furthermore, the extra calls to the "minor" function require exponentially large memory to store the minors. This is why this implementation is suitable for small matrices only; for larger ones, one must turn to the more efficient LU factorization, as discussed below.

3.17 Inverse of a Small Matrix

For small matrices, one may use Cramer's formula to compute the inverse matrix:

```
template<class T>
const dynamicMatrix<T>
inverse(const dynamicMatrix<T>&A){
  dynamicMatrix<T> Ainverse(A.height(),A.width());
  for(int i=0; i<A.height(); i++)
    for(int j=0; j<A.width(); j++)
      Ainverse(i,j) = power(-1,i+j) * det(minor(j,i,A));
  return Ainverse/det(A);
}  //  inverse using Cramer's rule
```

(In the above code line, it is assumed that the operator that divides a dynamic matrix by a scalar is available, as in Section A.7 in the Appendix.)

As discussed above, this algorithm is too expensive for larger matrices. Therefore, we turn to more efficient algorithms based on the LU factorization of a matrix, as in Section 2.21 above.

3.18 LU Factorization

To implement the LU factorization of a matrix, we must first define the function that switches the jth and kth columns in a matrix:

```
template<class T>
void switchColumns(dynamicMatrix<T>&A,int j, int k){
  if(j != k)
    for(int i=0; i<A.height(); i++){
      T keep = A(i,k,"read");
      A(i,k) = A(i,j,"read");
      A(i,j) = keep;
    }
}  //  switch columns j and k
```

This function will be used below to form the permutation matrix P in Section 2.21. In fact, P is the product of so-called "switch" matrices. More precisely, the ith switch matrix interchanges the ith column of a matrix with a column that lies to the right of it and its ith component is maximal in magnitude. Once such switches are applied to the matrix U in Section 2.21, stability (no division by zero or too small numbers) is guaranteed. The following function finds the index of the column that should be used in this switch:

```
template<class T>
int maxRowI(const dynamicMatrix<T>&A,int i){
  int maxI = i;
  for(int j=i+1; j<A.width(); j++)
    if(fabs(A(i,j,"read"))>fabs(A(i,maxI,"read")))
      maxI = j;
  return maxI;
}  //  column of maximal element in row i
```

We are now ready to implement the algorithm in Section 2.21 to have the LU factorization $A = LUP$. Since in the end U is upper triangular and L is lower triangular with main-diagonal elements that are all equal to 1, both L and U can be stored in a single dynamic-matrix object, named "LU". Furthermore, since the permutation matrix P is the product of switch matrices, it can be stored in a vector of integers, whose ith component contains the index of the column that is interchanged with the ith column in U in the ith switch matrix. The vector of integers that stores this information is passed to the function by a nonconstant reference, so the column indices calculated throughout the function are saved in it for future use.

```
template<class T>
const dynamicMatrix<T>
LUP(const dynamicMatrix<T>&A, dynamicVector<int>&P){
  dynamicMatrix<T> LU = A;
  for(int j=0; j<A.width(); j++){
    P(j) = maxRowI(LU,j);
    switchColumns(LU,j,P[j]);
```

In the outer loop on the columns j, the jth switch matrix is defined by finding the column with the maximal jth component. The index of this column is then stored in the jth component of the vector of integers 'P', and the column is also interchanged with the jth column. This way, the (j,j)th element in "LU" ($U_{j,j}$ in Section 2.21) becomes sufficiently large in magnitude.

Next, an inner loop uses the index 'i' to calculate the factor $L_{i,j}$, store it in "LU(i,j)", and use it to eliminate the original element $A_{i,j}$:

```
    for(int i=j+1; i<A.height(); i++){
      LU(i,j) = LU(i,j,"read") / LU(j,j,"read");
```

This elimination is done by subtracting a multiple of the jth row in U from the ith row in U:

```
      for(int k=j+1; k<A.width(); k++)
        LU(i,k) -= LU(i,j,"read") * LU(j,k,"read");
    }
  }
  return LU;
}  //  LU factorization
```

3.19 Determinant of a Larger Matrix

The determinant of the original matrix can now be calculated most efficiently as the product of the main-diagonal elements in U, stored in "LU" (Section 2.21):

```
template<class T>
const T
det(const dynamicMatrix<T>&LU,
    const dynamicVector<int>&P){
  T detLU = 1;
```

```
    for(int i=0; i<LU.height(); i++){
      detLU *= LU(i,i,"read");
      if(P[i] != i)
        detLU *= -1;
    }
    return detLU;
  }  //  determinant using LU factorization
```

3.20 Inverse of a Larger Matrix

Furthermore, the above LU factorization can be used to "invert" A for a particular right-hand side b, or to solve the linear system

$$Ax = LUPx = b$$

for the unknown vector x. Indeed, since

$$x = P^{-1}U^{-1}L^{-1}b,$$

x is obtained from b by forward elimination in L,

```
  template<class T>
  const dynamicVector<T>
  invert(const dynamicMatrix<T>&LU,
      const dynamicVector<int>&P, const dynamicVector<T>&v){
    dynamicVector<T> x = v;
    for(int i=1; i<v.dim(); i++)
      for(int j=0; j<i; j++)
        x(i) -= LU(i,j,"read") * x[j];
```

followed by back substitution in U,

```
    for(int i=v.dim()-1; i>=0; i--){
      for(int j=v.dim()-1; j>i; j--)
        x(i) -= LU(i,j,"read") * x[j];
      x(i) /= LU(i,i,"read");

    }
```

followed by applying the switch matrices to x to switch the 'i'th component in it with the 'P[i]'th component:

```
    for(int i=v.dim()-2; i>=0; i--)
      if(i!=P[i]){
        T keep = x[i];
        x(i) = x[P[i]];
        x(P[i]) = keep;
      }
    return x;
  }  //  invert for some right-hand-side vector
```

Furthermore, by setting the right-hand side b to be the jth column in the identity matrix (stored in "ej" in the code below), the above function produces the jth column in the inverse matrix A^{-1}:

```
template<class T>
const dynamicMatrix<T>
invert(const dynamicMatrix<T>&LU,
    const dynamicVector<int>&P){
  dynamicMatrix<T> inverse(LU.height(),LU.width());
  dynamicVector<T> zero(LU.height(),0.);
  for(int j=0; j<LU.width(); j++){
    dynamicVector<T> ej = zero;
    ej(j) = 1.;
    dynamicVector<T> columnJ = invert(LU,P,ej);
    for(int i=0; i<LU.height(); i++)
      inverse(i,j) = columnJ[i];

  }
  return inverse;
}  //  invert a matrix using its LU factorization
```

This algorithm avoids the expensive recursion used in Cramer's formula, and thus can be used for larger matrices as well.

3.21 Using the Heap Memory Efficiently

The "new" command used in the above dynamic objects allocates memory (during run time) in the so-called “heap" memory, which is the most easily accessed part of the secondary memory, and returns a pointer to it, which is stored in the so-called “stack" memory, which is a part of the primary memory. This memory should be released at the end of the block in which such a dynamic object is defined. This should be done by the C++ compiler, which invokes the relevant destructor implicitly.

The C++ compiler, however, may often decline to call the destructor at the end of the relevant block. In fact, the destruction can be delayed until there are many objects to destroy. Unfortunately, destroying many objects together may be far more expensive than destroying them one by one.

Indeed, the most time-consuming part of the destruction is to locate the memory occupied by the object in the heap memory. Because the C++ compiler is originally programmed to serve the users of the Windows operating system and give them real-time responses, it doesn't care to complete one task before moving ahead to the next task. Instead, it completes only a small portion of the first task, then starts the next task, then goes back to the first task to complete another small portion of it, and so on.

This strategy may make sense for operating systems to let the user have simultaneous real-time progress in every respect of the project that he/she wants to accomplish, but not for running C++ programs as required here. Indeed, because the most time-consuming part in the destruction task is to find the heap memory that should be released, it makes much more sense to look for it only once and release it in one go, rather than halting and then having to look for it once again.

In order to make sure that the compiler indeed destroys the entire object before going ahead to destroy the next object, one should better invoke the destructor explicitly at the end of the block in which the object is defined. This way, the compiler is forced to complete the destruction of the entire object in one go, including the entire memory it occupies, avoiding the need to locate this memory again and again.

In the present applications in this book, however, we don't bother to write these extra code lines. They are left as an exercise to the careful reader.

3.22 Exercises

1. Complete the missing arithmetic operators in the implementation of the dynamic vector in Section 3.3, such as subtraction and multiplication and division by scalar. The solution is given in Section A.6 of the Appendix.

2. Using the "list" object in Section 3.4, write the function "Taylor()" that takes as arguments the real number h and the list of N derivatives of a function $f(x)$ (calculated at some point x) and produces as output the Taylor approximation to $f(x+h)$:

$$f(x+h) \doteq \sum_{n=0}^{N} \frac{f^{(n)}(x)h^n}{n!},$$

where $f^{(n)}$ denotes the nth derivative of $f(x)$. The solution can be found in Chapter 5, Section 11.

3. Write the function "deriveProduct()" that takes as input two lists that contain the derivatives of two functions $f(x)$ and $g(x)$ at some point x and returns the list of derivatives of the product $f(x)g(x)$, using the formula

$$(f(x)g(x))^{(n)} = \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) f^{(k)}(x)g^{(n-k)}(x).$$

The solution can be found in Chapter 5, Section 12.

4. Implement Pascal's triangle, in Chapter 1, Section 19, as a list of diagonals. The diagonals are implemented as dynamic vectors of increasing dimension. (The first diagonal is of length 1, the second is of length 2, and so on.) The components in these vectors are integer numbers. Verify that the sum of the entries along the nth diagonal is indeed 2^n and that the sum of the entries in the first, second, ..., nth diagonals is indeed $2^{n+1}-1$.

5. Define the template class "triangle<T>" that is derived from a list of dynamic vectors of increasing dimension, as above. The components in these vectors are of the unspecified type 'T'. Implement Pascal's triangle as a "triangle<int>" object. Verify that the sum of the entries in the nth diagonal is indeed 2^n.

6. Use the above "triangle<double>" object to store the (mixed) partial derivatives of a function of two variables $f(x,y)$. In particular, use the (k,l)th cell in the triangle to

store the value

$$\frac{\partial^{k+l} f}{\partial^k x \partial^l y}(x,y)$$

at some point (x,y). This way, derivatives of order up to n are stored in the first n diagonals in the triangle.

7. Write a function "Taylor2()" that takes the above triangle and two small "double" parameters h_x and h_y as arguments and produces the two-dimensional Taylor approximation of $f(x+h_x, y+h_y)$ according to the formula

$$f(x+h_x, y+h_y) = \sum_{n=0}^{\infty} \frac{1}{n!} \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) \frac{\partial^n f}{\partial^k x \partial^{n-k} y}(x,y) h_x^k h_y^{n-k}.$$

The solution is indicated in the exercises at the end of Chapter 5.

8. Apply the "order()" function in Section 3.9 to a linked list of integer numbers and order it with respect to absolute value. For example, verify that the list

$$(-5, 2, -3, 0, \ldots)$$

is reordered as

$$(0, 2, -3, -5, \ldots).$$

9. Complete the implementation of the "binaryTree" class, with constructors, destructor, and assignment operator.

10. Complete the implementation of the "tree" class, with constructors, destructor, and assignment operator. The solution can be found in Chapter 18 in [69].

11. Modify your "tree" class to obtain the "graph" class.

12. Implement the arithmetic operators with dynamic matrices, including addition, subtraction, multiplication, and division by scalar. The solution can be found in Section A.7 in the Appendix.

13. Verify that the three versions of the "det()" function in Sections 2.21, 2.23, and 3.19 indeed produce the same result.