Implementing Lists

1 Overview

This chapter explores implementations of the list abstract data type, and introduces the concept of dynamically-allocated data structures and, in particular, linked lists. We begin with an implementation of the list ADT that uses an array to store the data. For convenience, the C++ List class interface from Chapter 3 is displayed below.

Listing 1: List Class Interface

```
typedef actual_type_to_use list_item_type;

class List
{
public:
   List();   // default constructor
   ~List();  // destructor

   bool is_empty() const;
   // Determines whether a list is empty.
   // Precondition: None.
   // Postcondition: Returns true if the list is empty,
   //                otherwise returns false.

   int length() const;
   // Determines the length of a list.
   // Precondition: None.
   // Postcondition: Returns the number of items that are currently in the list.

   void insert(int new_position, list_item_type new_item,
               bool& success);
   // Inserts an item into a list.
   // Precondition: new_position indicates where the
   //               insertion should occur. new_item is the item to be inserted.
   // Postcondition: If If 1 <= position <= this->length()+1, new_item is
   //                at position new_position in the list, other items are
   //                renumbered accordingly, and success is true;
   //                otherwise success is false.

   void delete(int position, bool& success);
   // Deletes an item from a list.
   // Precondition: position indicates where the deletion should occur.
   // Postcondition: If 1 <= position <= this->length(),
   //                the item at position position in the list is
   //                deleted, other items are renumbered accordingly,
   //                and success is true; otherwise success is false.

   void retrieve(int position, list_item_type & DataItem,
                 bool& success) const;
```

```
    // Retrieves a list item by position number.
    // Precondition: position is the number of the item to be retrieved.
    // Postcondition: If 1 <= position <= this->length(),
    //                DataItem is the value of the desired item and
    //                success is true; otherwise success is false.
private:
    // this is what we are about to supply
};
```

2 Array-Based Implementation of a List

One way to implement a list is to store successive elements in an array or, in C++, a vector. Let us work with an array for simplicity and language-independence. To determine whether it is feasible to use an array, or whether it is a good idea, we have to think about how each list operation could be implemented if the data were in an array. An array is declared to be a fixed size, known at the time the list is created. Suppose the size is `N`:

```
list_item_type items[N];
```

To keep track of how many items are in the list, we need a variable, which we call `size`:

```
int size;
```

(With vectors, the vector object can keep track of this for us.)

The `List` class interface would be modified by the specification of the two private data members as follows:

```
const int MAX_LIST_SIZE = N;
class List {
public:
    // the methods from Listing 1
private:
    list_item_type items[MAX_LIST_SIZE];
    int size;
};
```

With this definition, creating a list simply means setting `size` to 0. Destroying a list resets `size` to 0. Testing a list for emptiness checks whether `size` is 0 and getting the length of the list is simply returning the value of `size`. These are easy and straightforward to implement.

```
//Constructor
List::List()
{
    size = 0;
}

// Destructor
List::~List()
{
    size = 0;
}
```

```
// Test for emptiness
bool List::is_empty() const
{
    return (0 == size);
}
```

All other list operations must enforce the assertion that the list items are stored in consecutive array entries at all times. Hence, in a non-empty list, the first item is in `items[0]` and the k^{th} item is in `items[k-1]`. The `retrieve()` member function could be implemented by the following code:

```
void List::retrieve( int position,
                     list_item_type & item,
                     bool & success ) const
{
    success = ( 1 <= position  && position <= size  );
    if ( success  ) {
        item = items[position - 1];
    }
}
```

Obviously the `retrieve()` method takes the same number of instructions regardless of the position of the item, and this number is a fixed constant that is independent of the size of the list. This is good. When an algorithm takes a constant number of steps independent of the size of the data structure or input parameters, we say it ***runs in constant-time***.

Having to keep the items in consecutive entries implies that to perform an insertion, the items in the insertion point and after it have to be shifted towards the end, and to perform a deletion, say at position `k`, all items from `items[k]` to `items[size-1]` have to be shifted down by one cell. *This is not so good.* Remember that list positions start at 1, not 0, so the item at position `k` is in cell `items[k-1]`, not `items[k]`, and deleting the k^{th} item means that the array entry `items[k-1]` must be removed.

The insertion has to be done carefully as well. Suppose `k` is the index of the array at which the new item must be inserted. Since items must be shifted towards the high end, `items[size-1]` has to be moved to `items[size]`, then `items[size-2]` moved to `items[size-1]`, and so on until `items[k]` is moved to `items[k+1]`. (You cannot start by moving `items[k]` to `items[k+1]`, then `items[k+1]` to `items[k+2]` and so on. Why not?) Then the item can be put in `items[k]`. Of course if `size` is `N` before the insertion, we cannot insert anything new, because we would overflow the allocated storage and cause a run-time exception. (If we use a vector, then we could "grow" it when this happens.)

The insertion function is therefore

```
void List::insert( int position_to_insert,
                   list_item_type new_item,
                   bool & success )
{
    if ( N == size )
        success = false;   // array is full -- cannot insert
    else if ( position_to_insert < 1 || position_to_insert > size+1 )
        success = false;   // bad value of position_to_insert
    else {
        // make room for new item by shifting all items at
        // k >= position_to_start toward the end of the list
        for (int k = size-1; k >= position_to_insert-1; k--)
             items[k+1] = Items[k];
```

```
            // insert new item
            items[position_to_insert-1] = new_item;
            size++;
            success = true;
        }  // end if
    }
```

It should be clear that the insertion takes the most time when we insert a new first element, since the shifting must be done roughly N times. It takes the least time when we insert a new last element. But in this worst case, when we insert in the front of the list, the number of instructions executed is proportional to the size of the list. We therefore say it is a ***linear-time operation*** or that its ***running time is linear in the size of the list*** because the running time is a linear function of the size of the data structure. (A linear function f of one variable x is of the form $f(x)=ax+b$.)

The deletion operation also requires shifting items in the array, roughly the reverse of insertion. We start at the point of deletion and move towards the end, i.e., we move `items[position]` to `items[position-1]`, then `items[position+1]` to `items[position]`, and so on until we get to `items[size-1]`, which we move to `items[size-2]`. Since there is no `items[size]`, we cannot move it into `items[size-1]`. Hence, the loop that shifts items must stop when the index of the entry being filled is `size-2`:

```
    void List::delete( int position,
                       bool& success)
    {
        if ( 0 == size )
            success = false;  // array is empty
        else if ( position < 1 || position > size )
            success = false;  // bad position
        else {
            // shift all items at positions position down by 1
            // toward the end of the list. In this loop
            // k is the index of the target of the move, not the
            // source
            for (int k = position-1; k <= size-2; k++)
                 items[k] = items[k+1];
            size--;
            success = true;
        }  // end if
    }
```

In this algorithm the number of loop iterations is the length of the list minus the position at which the deltion occurs. In the worst this is when the first element in the list is deleted, because in this case the number of loop iterations is the length of the list (minus one). So we see that the deletion operation also has a worst-case running time that is linear in the size of the list.

2.1 Weaknesses of the Array Implementation of a List

Arrays are fixed size, so if the list needs to grow larger than the size of the array, either the implementation will fail, in the sense that it cannot be used in this case, or it has to provide a means to allocate a new, larger array when that happens. Even if a C++ vector is used, this behavior occurs, because the C++ vector class automatically resizes the vector when it becomes too small. (In fact it doubles the vector capacity.) Alternatively, the array can be made so large that it will always be big enough, but that means it will have many unused entries most of the time and be wasteful of storage. So this is one major weakness.

On average, an insertion requires moving half of the list each time, and a deletion, on average, also requires moving half of the list each time. As noted above, the amount of time to do an insertion or a deletion in the worst case (or in the average case) is proportional to the length of the list. As the size of the data set grows, the time to insert and delete increases. This is an even more serious weakness than the capacity problem. Thus, in summary, the problems with arrays as a data structure for implementing a list are that

- they are fixed size – they cannot grow if size exceeded, or if they are auto-resized, it is very time-consuming;
- if they are sized large enough, there is wasted space, and
- inserting and deleting cause significant movement of data and take too much time.

The alternative is to create an implementation

- in which insertions and deletions in the middle are efficient,
- which can grow as needed, and
- which uses no more space than is actually required.

To do this, we need to be able to allocate memory as needed, and release that memory when no longer needed. For this we need pointers[1].

3 Review of Pointers

A pointer variable is a variable that stores the address of a memory cell. The memory cell might be the start of any object or function, whether a simple variable, constant, object of a class, and so on.

3.1 Usage

Declaring pointer variables:

```
int * intp;         // declares a pointer to an integer memory cell
char *pc;           // pointer to char
int* a[10];         // a is an array of pointers to int
int (*b)[10];       // b is a pointer to an array of 10 ints
int* f(int x);      // f is a function returning a pointer to an int
int (*pf)(int x);   // pf is a pointer to a function returning an int
int* (*pg)(int x);  // pg is a pointer to a function returning a pointer to an int
```

The `*` is a **pointer declarator** operator. In a declaration, it turns the name to its right into a pointer to the type to the left. It is a unary, **right associative** operator. It does not matter whether you leave space to the right or left of it, as demonstrated above.

```
int * p, q;        // p is a pointer not q
int *p, *q;        // p and q are both int pointers
```

A `typedef` is useful for declaring types that are pointers to things:

[1]At least in languages such as C and C++ we do.

```
typedef    int*  intPtrType;
intPtrType intp, intq;        // p and q are both pointer to int
typedef    bool (*MessageFunc) (char* mssge);
// This defines a MessageFunc to be a pointer to a function with a
// null-terminated string as an argument, returning a bool.
// It can be used as follows:
void handle_error( MessageFunc f, char* error_mssge);
// Here, f is a function with a char* argument returning a bool.
```

The preceding declarations do not give values to the pointers. None of the pointers declared above have any valid addresses stored in their memory cells. To define a pointer, i.e., to give it a value, you give it an address. The & operator is the **address-of** operator. It "returns" the address of its operand.

```
int x = 12;
int* p = &x;    // p is a pointer to the int x.
*p = 6;         // this changes x to store 6 and not 12
(*p)++;         // * has lower precedence than the post-increment ++
// so this increments x
int a[100] = {0};
int *q = &a[0]; // q is the address of a[0]
q++;            // q is the address sizeof(int) bytes after a[0], which is a[1]
*q = 2;         // now a[1] contains a 2!
```

3.2 Dynamic Memory Allocation

Pointers are the means to perform dynamic memory allocation. The new operator allocates memory:

```
int* p;       // p is a pointer to int
p = new int;  // creates an unnamed int cell in memory; p points to it
```

Arrays can be allocated dynamically:

```
int *q;             // q is a pointer to int
q = new int[100];   // 100 ints are allocated and q points to the first of them
for (int i = 0; i < 100; i++)
    q[i] = i*i;
```

Now q can be treated like the name of an array, but it is different, because unlike an array name, q is not a const pointer. It can be assigned to a different memory location. Note that *q is the first element of the array, i.e., q[0], and *(q+1) is the same as q[1]. More generally,

*(q+k) is the same as q[k].

The delete operator frees memory allocated with new. To delete a single cell, use

```
delete p; // delete the cell pointed to by p
```

If q is a dynamically allocated array, you need to use the delete[] operator, as in

```
delete[] q;
```

3.3 Potential Errors: Insufficient Memory, Leaks and Dangling Pointers

There are several things that you can easily do wrong when using pointers, and they are the most dangerous part of C and C++. (This is why Java does not have pointers.) The first common mistake is failing to check the result of using the `new` operator. Most people do not think about the fact that `new` may fail. It can, if your program has run out of memory, so you must always check that it did not. If `new` fails, the value it returns is `NULL`. Good programmers always check this, as in the following code snippet:

```
p = new int;
if ( NULL == p ) {
    // bail out
    cerr << “Out of memory\n”;
    exit(1);
}
*p = ...;
```

Now consider the following lines of code:

```
p = new int;
if ( NULL == p ) {
    // bail out
    cerr << “Out of memory\n”;
    exit(1);
}
*p = 5;
q = p;
delete q;
cout << *p; // error - the memory is gone - dangling pointer
```

Memory is allocated and a pointer to it is given to `p`. The location is filled with the number 5. Then `q` gets a copy of the address stored in `p` and `delete` is called on `q`. That means that `p` no longer points to an allocated memory cell, so the next line causes an error known as a **dangling pointer** error.

One must use caution when multiple pointers point to same cells.

A different problem is a **memory leak**. Memory leaks occur when allocated memory is not freed by the program. For example, the following function allocates an array:

```
void someTask( int n)
{
    int *array = new int[n];
    // do stuff with the array here
}
```

Notice that the function did not delete the array. If this function is called repeatedly by the program, it will accumulate memory and never release it. The program will start to use more memory than might be available for it, and it will start to get slower and slower as the operating system tries to make room for it in memory. If things get really out of hand, it can slow down the entire machine.

4 Linked Lists

Linked lists are lists in which each data item is contained in a **node** that also has a **link** to the next node in the list, except for the last, which does not link to anything. A **link** is simply a reference to a node. Links are implemented as pointers in a language such as C++ or C, but they may be represented by other means as well. Conceptually, a link is something that can be dereferenced to get the contents of the node to which it points.

4.1 Singly-linked Linked List

In a ***singly-linked*** list, each node has a link to the next node in the list, except the last node, which does not. We will use a pointer-based solution to this problem. A pointer-based linked list is a linked list in which the links are implemented as pointers to nodes. A general description of a node is

```
typedef int list_item_type;  // for simplicity assume node data is type int
struct node {
    list_item_type data;  // the node's data element
    node*          next;  // the pointer to the next node
};

typedef node* link;       // link is short for “pointer to node”
```

This can be written more succinctly as follows:

```
typedef struct node * link;  // in C++ it can be written typedef node * link;
struct node {
    list_item_type data;
    link           next;
};
```

Notice in the above that a `link` is defined as another name for a pointer to a `node` structure even though a `node` structure has not yet been defined at that point in the code. This is not incorrect – it will compile, because the only thing the compiler needs to “know” to compile it is how much storage the `link` needs and what type it is. A pointer is always a fixed number of bytes, and the compiler sees that a `link` is only allowed to point to things of type `node` structure, so these requirements are met.

The actual data in a node will vary from one list to another, depending on how the `itemType` is defined. The key point is how the pointers are used. Each pointer points to the next item in the list, unless it is in the last node, in which case it is a null pointer. A linked list is a sequence of these nodes. The list itself is accessed via a special link called the **head**. The head of a list is a pointer to the first node in the list. The **tail** of a list is the last node. Figure 1 depicts this.

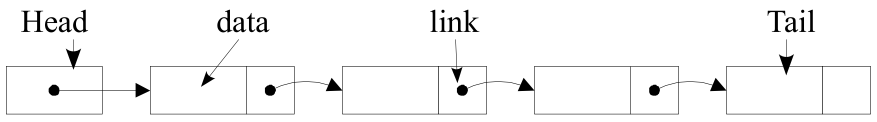


Figure 1: Linked list.

It is convenient to create a special variable to store the head of the list, with the declaration

```
link head;
```

which creates a pointer named `head` that can be used to point to a list. Of course at this point `head` has only garbage in it; it has no address and there is no list. By itself this declaration does not create a list!

It is also convenient to keep track of the length of the list with an integer variable that stores its size. Therefore, a linked list's private data must include a `head` pointer and *should* include a `size` variable:

```
class List
{
public:
    // the member functions defined in Listing 1
private:
    link  head;        // pointer to the first node in the list
    unsigned int size; // number of nodes of the list
};
```

We can now run through how to implement the various operations (methods) defined in a `List` ADT using this pointer-based representation, assuming that `itemType` is a `typedef` for `int` in the above definitions.

4.1.1 Creating an Empty List

To create an empty list is trivial:

```
head = NULL; // head points to nothing
size = 0;    // list has zero items
```

An empty list is a list with no nodes, so setting `head` to `NULL` makes `head` a pointer to an empty list, and setting size to 0 says it has no nodes. This implies that the default constructor is simply

```
List::List()
{
    head = NULL;
    size = 0;
}
```

4.1.2 Creating a Single List Node

To create a node that is not yet part of a list and put data into it, we need the `new` operator to create the node and then we assign the data to the `data` member and we set the `next` member to `NULL`:

```
link p;
p = new node;             // p points to a new node without data.
if ( NULL == p ) exit(1); // ALWAYS handle this possible error
p->data = 6;              // p->data contains 6
p->next = NULL;           // p->next points nowhere
```

This arbitrarily puts the value 6 into the `data` member of the node. Right now this node is not linked to anything.

4.1.3 Writing the Contents of a List to a Stream

Most container classes, such as lists, should have a method that can write their contents onto an output stream, such as a display device or a file. This method was not mentioned when we introduced lists in Chapter 3, but now that we are implementing them, it is worthwhile to develop such a method because it is certainly useful, and it is instructive to implement it. Because it needs access to the list private data, it should be either a friend function or a member of the list class. The approach we use here is to create a private method that can be called by a public method in the class interface. The private function prototype is

```
// precondition:  ostream out is open and list is initialized
// postcondition: ostream out has the contents of list appended to it,
//                one data item per line
void write( ostream & out, link list);
```

This private method can print the list contents starting at any node to which it is given a pointer. The public method would be implemented as

```
void List::write( ostream & out )
{
    write( out, head);
}
```

where `write()` would be a private method that actually writes the list data onto the stream named `out`.

The private `write` function logic is the following:

1. Set a new pointer named `current` to point to whatever `list` points to.
2. Repeatedly do the following:
   (a) if `current` is not a `NULL` pointer, write the data in the node that `current` points to and then advance `current` to point to the next node;
   (b) if `current` is a `NULL` pointer, then exit the loop.

Writing the data means writing it to the output stream followed by a newline. This can be written as follows:

```
void write( ostream & out, link list)
{
    link current = list;

    while ( NULL != current ) {
        out << current->data << "\n";
        current = current->next;
    }
}
```

The instruction

```
current = current->next;
```

is guaranteed to succeed because `current` is not `NULL`, so it points to a node, and the `next` pointer in that node exists. This function is general enough that it can write to the terminal or to a file, so we have a way to save a list to a file.

4.1.4 Getting the Size of a Linked List

It is trivial to implement the `length()` function:

```
int List::length() const
{
    return size;
}
```

4.1.5 Testing for Emptiness

This is also trivial, and there are two ways to do it. One way is simply

```
bool List::is_empty() const
{
    return size > 0 ? false : true;
}
```

4.1.6 Inserting a Node at a Specified Position in a Linked List

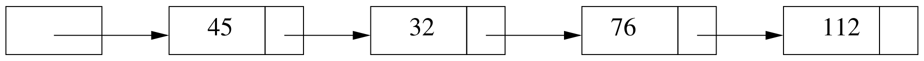
The insert method is a bit more challenging than the preceding ones. Recall that its prototype is:

```
void insert( int position_to_insert,
             list_item_type new_item,
             bool & success );
```

in which 1 <= `position_to_insert` <= `size`+1.

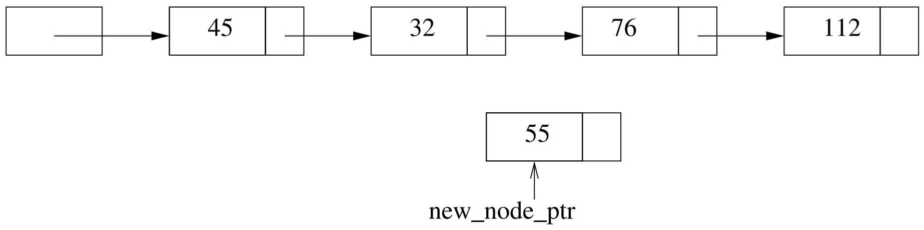
How do we insert into a linked list? What steps must we follow? First consider the case in which the new data is inserted between two existing list nodes. Inserting in the very front of the list or after the last node will be handled afterward.

Assume the list looks like this to start:

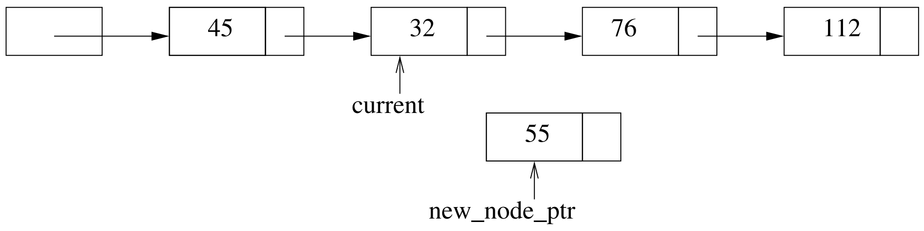


and that we want to insert a node containing the value 55 at position 3. This means that the node must be inserted after 32 and before 76. The steps we must follow are

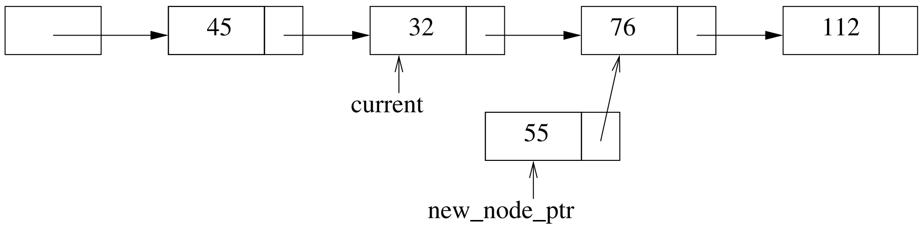
1. Create a new node and fill it with the value to be inserted. Suppose `new_node_ptr` is a pointer to this new node.



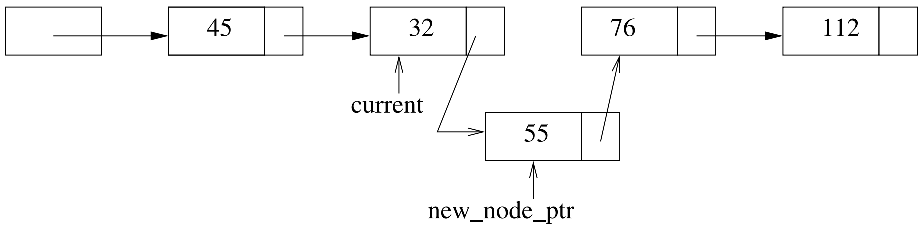
2. Position a pointer variable, which we will name `current`, so that it points to the node containing 32. We will figure out how to do this afterward.



3. Make `new_node_ptr`'s `link` member point to what `current->link` points to. In other words, copy `current->link` into `new_node_ptr->link`.



4. Make `current->link` point to what `new_node_ptr` points to. In other words, copy `new_node_ptr` into `current->link`.



You can see that the new node is now in position 3 in the list, and that the nodes after it were shifted by one position towards the end of the list.

The question that remains is how to position a pointer at a specific position in the list. Because this is an operation that will need to be done for insertion, deletion, and retrieval, we make it a private helper function in our implementation. It is just a matter of advancing a pointer along the list and counting while we do it, like stepping along stones in a pond until we get to the k^{th}stone. The following function does this:

```
// precondition: 1 <= pos <= list length
// postcondition: none
// returns a pointer to the node at position pos
link List::get_ptr_to(int pos) const
{
    // sanity check to prevent a crash:
    if ( ( pos < 1 ) || ( pos > size ) )
        return NULL;
    else  { // pos is within the required range
        // count from the beginning of the list
        link current = head;

        for (int i = 1; i < pos; i++ )
             // loop invariant: current points to ith node
          current = current->next;
        // i == pos and therefore current points to pos'th node
        return current;
   }
}
```

Notice that this function will position a pointer at any node in the list, including the last. Also notice that it starts not by pointing to the head node, but by pointing to what `head` points to, namely the first node.

Now what happens if the list is empty? Then `size == 0` and `head == NULL`, so this will return `NULL` because it fails the sanity check.

So tentatively, we can put the preceding steps into a first attempt at an implementation of the insert method:

Listing 2: Linked list insert method

```
 1 void List::insert(int new_position, list_item_type new_item,
 2                       bool& success)
 3 {
 4     link prev_ptr, new_node_ptr;
 5
 6     int new_size = size + 1;
 7     success = (new_position >= 1) && (new_position <= new_size) ;
 8     if (success) {
 9         // create a new node and place new_item into it
10         new_node_ptr = new node;
11         success = ( NULL != new_node_ptr );
12         if ( success ) {
13             size = new_size;
14             new_node_ptr->data = new_item;
15             prev_ptr = get_ptr_to ( new_position-1 );
16
17             // insert new node after node to which Prev points
18             new_node_ptr->next = prev_ptr->next;
19             prev_ptr->next = new_node_ptr;
20         }
21     }
22 } // end insert
```

This will work, except in one situation: what if we need to insert a node at position 1? The problem is in line 15. When `new_position` == 1, we call `get_ptr_to(new_position-1)` but `get_ptr_to()` does not accept an argument of 0; it makes no sense to have a pointer to the 0th node. Unfortunately, we have to "break out" a special case to handle insertion in the first position, given how we have defined a linked list[2].

In case `new_position` is 1, instead of lines 15, 18, and 19, we need

```
    new_node_ptr->next = head;
    head = new_node_ptr;
```

Putting this together, we get the correct version of the insertion method:

Listing 3: Linked list insert() method

```
void List::insert(int new_position, list_item_type new_item,
                      bool& success)
{
    link prev_ptr, new_node_ptr;

    int new_size = size + 1;
    success = (new_position >= 1) && (new_position <= new_size) ;
    if (success) {
        // create a new node and place new_item into it
        new_node_ptr = new node;
        success = ( NULL != new_node_ptr );
        if ( success ) {
            size = new_size;
```

[2]There are alternative implementations that use a "dummy" first node. A dummy node is never used to hold data. It exists just to remove this special case from the code, but it also makes the other operations different. This is a design trade-off that you can explore on your own.

```
                new_node_ptr->data = new_item;
                if ( 1 == new_position ) {
                    new_node_ptr->next = head;
                    head = new_node_ptr;
                }
                else {
                    prev_ptr = get_ptr_to ( new_position-1 );
                    new_node_ptr->next = prev_ptr->next;
                    prev_ptr->next = new_node_ptr;
                }
            }
        }
} // end insert
```

4.1.7 Retrieving the Item in a Given Position

We handle this method next because it is much easier than the `delete()` function and the destructor. To retrieve the item at a given position we check that the position is valid, and if so, we use the `get_ptr_to()` function to position a pointer at the node. Then we just extract the value from that node.

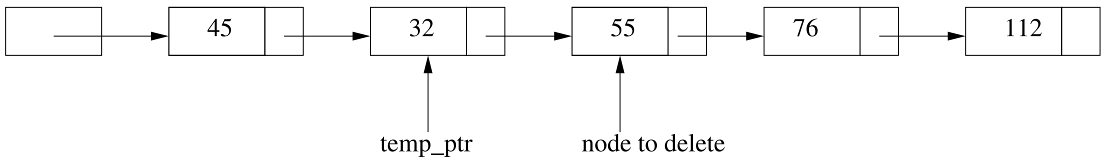
```
    void retrieve(int position, list_item_type & DataItem,
                  bool& success) const;
    {
        link current;
        success = (position >= 1) &&  (position <= size);
        if (success)  {
            // get pointer to node, then data in node
            current = get_ptr_to(position);
            DataItem = current->data;
        }
    }
```

4.1.8 Deleting an Item from a Linked List

The function that we now need to implement has the prototype

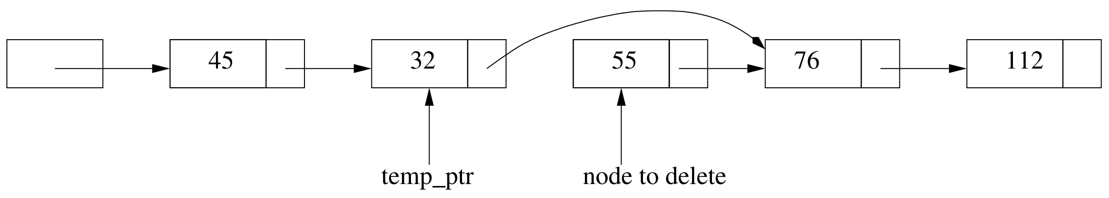
```
    void delete(int position, bool& success);
```

Thus, we are given a position and we need to delete the node at that position, assuming it is a valid position. Unlike the insertion method, this requires that we get a pointer to the node ***before*** the node to be deleted. To understand this, consider the following figure.

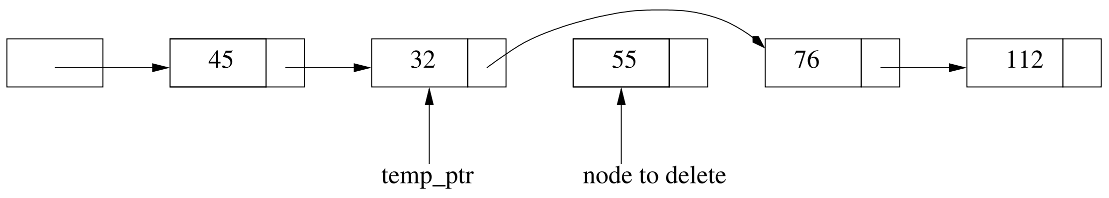


To delete the node containing the item 55, we need to make the `next` link in the previous node, pointed to in the diagram by `temp_ptr`, point to the node ***after*** the node to delete! This means that we need to get a pointer to the node ***before*** the node to be deleted. This is not a problem because we already have the function `get_ptr_to()`, but again we have a special case at the front of the list, which we discuss after the general case. The steps from this point forward are:

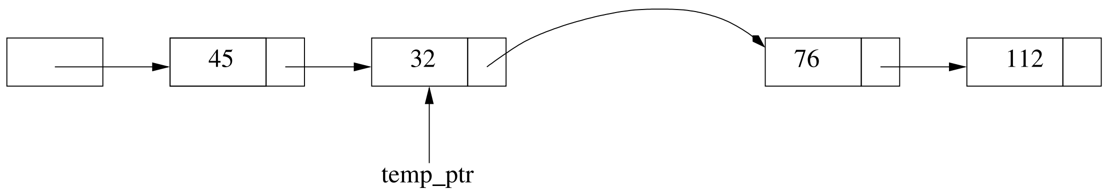
1. Unlink `temp_ptr->next` and relink it so that it points to the node after the node to delete. This is a single assignment statement: `temp_ptr->next = node_to_delete->next`.



2. Unlink the `next` link of the node being deleted, using `node_to_delete->next = NULL`.



3. Release the storage allocated to the node, using the `delete` operator: `delete node_to_delete`.



The preceding steps will not work if the node to delete is the first node. In this case we need to execute the following instructions instead of the ones just described:

```
temp_ptr = head;
head     = head->next;
delete temp_ptr;
```

This is assuming that the list is not empty of course. If the list is not empty, then the head pointer does point to a node, so there will not be an error trying to dereference that pointer to get to the node's `next` member. We put all of this together in the following function.

```
void List::delete(int position,  bool& success)
{
    link node_to_delete, temp_ptr;

    // the following will set success to false if the list is empty and position > 0
    success = (position >= 1) &&  (position <= size) ;

    if (success)  {
        // if we make it here, position >= 1 and we are definitely going to delete
        // something, so we decrement list size.
        --size;
        if (1 == position)  {  // delete the first node from the list
            node_to_delete = head;   // save pointer to node
            head           = head->next;
            }
            else  {
                // Because position must be greater than 1 here, we can safely call
                // get_ptr_to() now.
                temp_ptr = get_ptr_to(position-1);

                // delete the node after the node to which temp_ptr points
                node_to_delete = temp_ptr->next;  // save pointer to node
                temp_ptr->next = node_to_delete->next;
            }
            // node_to_delete points to node to be deleted
            node_to_delete->next = NULL;  // always safe to do this
            delete node_to_delete;
            node_to_delete  = NULL;       // and this
        }
    }
```

Notice in the above function that we purposely set the `next` link in the node we are about to delete to `NULL`. Even though we are returning it to the system to be reused, we break the link from this node to our list, just in case somehow this storage gets reused without changing its bits. This is just safe programming. Similarly, we set `node_to_delete` to `NULL`, even though it is just on the program's run-time stack. Perhaps it is overly cautious.

4.1.9 Implementing a Destructor

For the first time we really do need a non-default destructor in a class. If the list is not empty, then it contains one or more nodes that have been allocated from the heap using the `new` operator, and these must be returned to the operating system so that the memory can be re-used if needed. A default constructor will free up the `head` pointer and the `size` member, but it cannot travel down a list deleting the nodes in it! It is our job to do this.

One approach is to use some kind of loop that traverses the list, deleting one node after the other, but if we do it the wrong way, we will detach the list before we delete its nodes. Another way to do this that uses a bit more overhead and is therefore a bit slower is the following:

```
    List::~List()
    {
        bool success;
        while ( size > 0 )
            delete(1, success);
    }
```

This destructor uses the public `delete()` member function to keep deleting the first node in the list as long as size is greater than zero. Obviously it will delete all of the nodes. It is a bit slower because it repeatedly makes a function call for each node in the list. Function calls have setup times. We can use a faster implementation by replacing the function call by inline code that does the same thing.

```
    List::~List()
    {
        link node_to_delete;
        while ( size > 0 ) {
            node_to_delete       = head;       // save pointer to first node
            head                 = head->next; // detach first node from list
            node_to_delete->next = NULL;        // clear its next link
            delete node_to_delete;              // release its storage
            size--;                             // decrement size
    }
```

The loop body basically does what the `delete()` function does.

4.1.10 Anything Else?

We implemented all of the public methods of the `List` ADT, plus a `write()` function as well. We should add a copy constructor, if we want to be able to create one list from an existing list. We might want a `read()` function, which would read from an input stream and create a list from it. These are good programming exercises.

4.2 Variations on the Linked List

There are several variations on linked lists. We already mentioned using a dummy head, or first, node. A dummy node has no data in it. It exists just to eliminate the need for special cases in operations such as insertions and deletions. The test for emptiness changes when a dummy head node is used. Later we shall explore doubly-linked lists, in which each node has a pointer to its predecessor and successor. One can also keep a pointer to the tail of a list, to make appending a new item at the end more efficient, otherwise one has to travel the entire list each time to do this, which is inefficient.

4.3 Sorted Linked List

A sorted list is really a different type of object, as we discussed in the third chapter. If the linked list is kept in sorted order, then inserting, deleting, and retrieving operations must be different than if it is unsorted. The other operations are the same and they are not discussed here.

4.3.1 Inserting into a Sorted List

The insertion algorithm must advance through the list, comparing each node's item with the item to insert. While the new item is greater than the list item, the pointer is advanced. If the new item is equal to the current list node item, searching stops and the algorithm reports that nothing was inserted. If the new item is less than the list node's item, it must inserted before that list node, because it was greater than all nodes before it in the list and smaller than the current one.

The problem with this idea is that, in the course of discovering where the node must be inserted, the pointer is advanced too far. We only know where to insert the node after we passed the spot! The pointer points to the first node whose data is larger than the new item, but in order to insert the new item before that node we need a pointer to the node preceding it. One solution is to advance two pointers through the loop, one trailing the other. This complicates the algorithm because we have to be careful how to handle the empty list and the case when the insertion must happen at the beginning. The following code uses this approach and solves these problems.

First, we define a private function, `insert_at()`, to simplify the solution. This is a helper function that

1. creates a new node,
2. fills it with the given data,
3. inserts it at the given position, setting call-by-reference parameter `prev` to contain its address.

```
    bool insert_at( link & prev, list_item_type new_item )
    /**
     * @pre prev != NULL
     * @post prev points to a new node containing new_item and the
     *       next link of the new node points to what prev previously
     *       pointed to.
     */
    {
        link temp_ptr = new node;
        if ( NULL == temp_ptr )
            return false;
        temp_ptr->data = new_item;
        temp_ptr->next = prev;
        prev           = temp_ptr;
        return true;
    }
```

We use this function in the `insert()` method shown in Listing 4.

Listing 4: Sorted list insertion method

```
bool SortedList::insert( list_item_type new_item)
{
    link current, previous;
    bool success;

    // If list is empty, or if it is not empty and new_item is smaller
    // than the first item in the list, insert it at the front
    if ( NULL == head || head->data > new_item ) {
        success = insert_at( head, new_item);
        if (success)
            size++;
        return success;
    }

    // The list is not empty and first item is not larger than new_item
    // Check if the 1st item is equal to new_item
    if ( head->data == new_item )
        // quit since it is in the list already
        return false;
    }

    // head->data < new_item, so we start at 2nd node if it exists
    previous = head;
    current  = head->next;   // which must exist if we make it here
    while ( current != NULL  ) {
        if ( current->data < new_item ) {
            // advance both pointers
            previous = current;
            current = current->next;
        }
        else if ( current->data > new_item ) {
            // previous points to an item that was smaller and *current
            // is bigger so new_item should be inserted after previous
            if ( success = insert_at(prev->next, new_item) )
```

```
                size++;
            return success;
        }
        else  // found a duplicate, so do not insert
            return false;
    } //end while

    // if we reach here, the item is bigger than everything in list
    // and previous points to last node
    if ( success = insert_at(previous->next, new_item) )
        size++;
    return success;
}
```

4.3.2 Searching a Sorted List

The problem of finding an item in a list is not called retrieving; it is called **searching**, or **finding**, because the algorithm is searching for a particular value, not a particular position. Usually we search for keys in order to return the data associated with them. For example, a node might contain records that represent driving records, and we search for a driver's license number in order to retrieve the entire record. If a node contains nothing but a key, then there is nothing to return other than true or false – it was found or it was not found. To generalize the algorithm, we will assume that the item is a complex structure.

The following search procedure is designed to find all occurrences of the item within the list, as opposed to just the first. Because it might find multiple items, it uses an output stream as the means for returning them to the calling function. If it were to return a single item, it could just make the item its return value. Another way to return multiple items is to construct a second list whose members are all matching nodes. That is a bit more complex, so we handle it this way here.

```
    void  SortedList::find (ostream & output, list_item_type  item) const
    {
        NodePtr          current;
        list_item_type   temp;

        // If list is empty, return
        if ( NULL == head )
            return;

        // list is not empty
        current  = head;
        while ( current != NULL ) {
            temp = current->item;
            if ( temp  < item ) {
                current = current->next;
            }
            else if ( temp > item ) { // item is not in remainder of list
                return;
            }
            else  { // found the item
                output << current->item;
                current = current->next;
            }
        }
    }
```

4.3.3 Deleting from a Sorted List

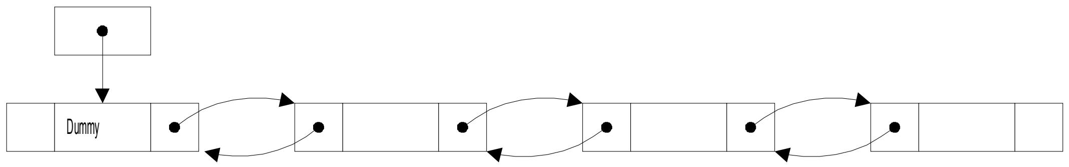
Deleting from a sorted list is similar to finding, except that items are removed as they are found. The following procedure deletes all occurrences of matching items, and returns the number of items deleted. As with deleting from an unsorted list, handling the first node is a special case that must be checked within the code. The deletion algorithm uses two pointers, one "chasing" the other. The `prev` pointer is always one node behind the `current` pointer, except when it starts, when `prev` is `NULL`. This is how the code can detect if the node to be deleted is the first node, because `prev` is still `NULL`.

```
int SortedList::delete( list_item_type item )
{
    link current, prev;

    int  count = 0;
    // If the list is empty, there is nothing to do
    if ( NULL != head ) {
        current = head;
        prev = NULL;
        while ( current != NULL ) {
            if ( current->item < item ) {
                // item is bigger than current item so advance pointers
                prev = current;
                current = current->next;
            }
            else if ( ( current->item > item ) {
                // item is smaller than current item, so item cannot be in
                // remainder of the list and we can stop searching
                break;
            }
            else { // current->item == item
                if ( NULL == prev ) {
                    // first node: handle separately
                    head          = current->next; // detach from list
                    current->next = NULL;          // clear next link
                    delete current;                // free node
                    current = head;
                    count++;
                }
                else { // not first node, so prev can be dereferenced
                    current = current->next;     // advance current
                    prev->next->next = NULL;     // clear next link in node
                    delete prev->next;           // free node
                    prev->next = current;        // detach node from list
                    count++;
                }
            }
        }  // end while
    } // end if head != NULL
    size -= count;
    return count;
}
```

4.4 Doubly-Linked Lists

In a **doubly-linked list**, each node contains a pointer to the node preceding it in the list as well a a pointer to the node following it. Because of this, there is no need to have a pointer to the previous node in the list to do insertions and deletions. On the other hand, it also makes insertions and deletions a bit more work to do. Doubly-linked lists make it possible to traverse a list in reverse order. Many lists are implemented as doubly-linked lists with dummy nodes. The use of dummy nodes eliminates special cases for operations at the front of the list. The following diagram shows what a doubly-linked list with a dummy node looks like.



Rather than going through all of the operations and how they are implemented, we can just look at a few of the things that make them different. For example, to delete the node to which a link named `cur` points, it is sufficient to do the following:

```
(cur->prev)->next = cur->next; // make the previous node point to the one after cur
(cur->next)->prev = cur->prev; // make the one after cur point to the one before it
cur->next = NULL;              // clear its links and delete it
cur->prev = NULL;
delete cur;
```

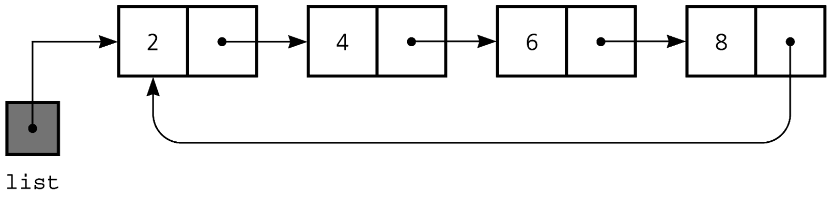
To insert a new node pointed to by `newPtr` before the node pointed to by `cur`

```
newPtr->next = cur;
newPtr->prev = cur->prev;
cur->prev    = newPtr;
(newPtr->prev)->next = newPtr;
```

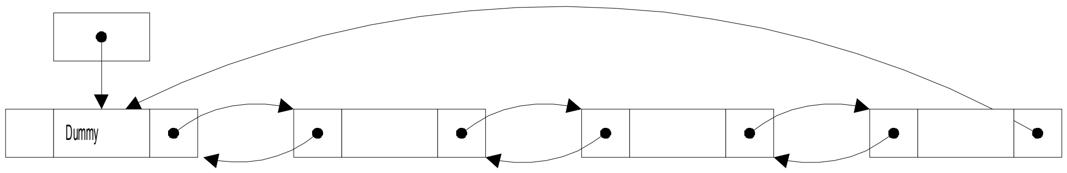
The operations that do not modify a list do not need to be changed. The constructor needs to set up the dummy node.

4.5 Circular Linked Lists and Circular Doubly Linked Lists

Ordinary singly-linked lists have the disadvantage that one must always start from the beginning of the list to do any type of traversal. In a **circular list**, the tail node points back to the first node. This eliminates need for special cases for first and last node insertions and deletions, and it also makes it possible to start searches where one left off. The diagram below illustrates a circular, singly-linked list.



In a **circular doubly-linked list**, the tail points to the dummy node, not the first node. Circular doubly-linked lists are also very useful for many applications. A circular doubly-linked list is illustrated below.



A C++ Exceptions

This section is an excerpt of my exception handling notes in C++ in the Resources page of the website, which has a more thorough explanation of it. An ***exception*** is an event that is invoked when something "goes wrong" during the execution of a program. Exceptions can be ***raised*** by hardware or by software. Dividing by zero, running out of memory, and attempting to access protected parts of memory, are examples of events typically caught by the hardware and then ***handled*** by the operating system. Usually the operating system will terminate the program that caused it. Software can also raise exceptions if the language run-time environment supports it, which C++ does.

In C++, instead of saying that the software *raised* an exception, we say that software ***throws*** an exception. The three operators that make exception handling work are: `throw`, `try`, and `catch`, which are keywords in C++. The idea is that you create a *try-block*, which is the `try` keyword followed by a statement block, which is then followed by *exception handlers*. If within that block, an exception is thrown, then control can pass to one of the exception handlers that follow the block, if they have been set up properly. An exception handler is a function that is executed when an exception is thrown, and is introduced by the keyword `catch`. The basic form is

```
try
{
     a statement sequence containing function calls to functions that
     might throw exceptions, or to functions that might  call functions
     that might call functions that might, and so on ... throw exceptions
}
catch ( exception_declaration_1 )
{
     statements to handle exceptions of type exception_declaration_1
}
catch (exception_declaration_2 )
{
     statements to handle exceptions of type exception_declaration_2
}
catch (exception_declaration_3 )
{
     statements to handle exceptions of type exception_declaration_3
}
...
catch (exception_declaration_N )
{
     statements to handle exceptions of type exception_declaration_N
}
```

If an exception is thrown by some function that is called directly or indirectly from within the try-block, execution immediately jumps to the first exception handler whose exception declaration matches the raised exception. If there are no handlers that can catch the exception, the try-block behaves like an ordinary block, which means it will probably terminate.

An example demonstrates the basic idea.

```
struct Ball
{
    int b;
    Ball(int x = 0) : b(x) { } // initializes member variable b
};

void test ( )
{
    /*
        ...
    */
    throw Ball(6);
    // This is a throw-statement. It is throwing an anonymous object
    // of Ball structure, initialized by a call to the constructor
}

int main ( )
{
    /* ... stuff here */
    try
    {
        test();
    }
    catch (Ball c)
    {
        cerr << " test() threw ball " << c.b << "\n";
    }
...
}
```

A `throw` statement is, as shown above, of the form

```
throw expression;
```

The thrown expression is any expression that could appear on the right hand side of an assignment statement. The most likely candidate is a call to a constructor, such as I showed above. The safest way to handle exceptions is to define unique types for the distinct exceptions. One way to guarantee that the types are unique is to enclose them in a namespace. This is also a good place for an error count variable.

```
// The Error namespace defined below declares three types: Towel, Game, and Ball,
// an int variable that counts errors, and a function named update() .
namespace Error {
    struct Towel {  };  // empty struct, default generated constructor
    struct Game { };    // empty struct
    struct Ball {
        int b;
    Ball(int x = 0) : b(x) { }
    };

    int count = 0;      // counts errors thrown so fa
    void update()  { count++; }  // increments error count
}
```

A program can use these error types as follows. There are three functions, `Pitcher()`, `Boxer()`, and `PoorSport()`. Each might throw an exception of some type if their input meets some specific set of conditions. For simplicity, the exception is thrown if the single integer parameter is a multiple of a fixed number. Notice that to throw an exception in the Error namespace, the function must use the syntax `Error::` followed by the type being thrown.

```
void Pitcher ( int n )
{
    /* ... */
    if ( n % 2 == 0 )
    throw Error::Ball(6);
}
void Boxer ( int n )
{
/* ... */
    if ( n % 3 == 0 )
    throw Error::the_Towel();
}
void PoorSport ( int n )
{
/* ... */
    if ( n % 11 == 0 )
    throw Error::the_Game();
}

// The main program calls Pitcher(), Boxer(), and PoorSport(). Because these might throw
// exceptions and main () does not want to be terminated because of an unhandled
// exception, if places the calls in a try-block that is followed by exception handlers.
int  main ( )
```

```
{
    /* ... stuff here */
    int n;
    cin >> n;
    try {
        Pitcher( n );
        Boxer( n );
        PoorSport( n );
    }
    catch (Error::Ball c)
    {
        Error::update();
        cerr <<Error::count << ": threw ball " << c.b << "\n";
    }
    catch (Error::Towel )
    {
        Error::update();
        cerr <<Error::count << ": threw in the towel\n";
    }
    catch (Error::Game )
    {
        Error::update();
        cerr << Error::count << ": threw the game\n";
    }
    /* ... */
    return 0;
}
```

There is a special notation that you can use for the parameter of an exception handler when you want it to catch all exceptions.

```
catch (...)
```

means catch every exception. It acts like the default case of a switch statement when placed at the end of the list of handlers.

This is just a superficial overview of the exception handling mechanism. To give you a feeling for why there is much more to learn, consider some of the following complexities.

- Since any type is a valid exception type, suppose it is publicly derived class of a base type. Do the handlers catch it if they catch the base type? What if it is the other way around?
- Can a handler throw an exception? If so, can it throw an exception of the same type as it handles? If so, what happens? Does it call itself? (Fortunately, although the answer is "yes" to the first question, it is "no" to the third.)
- What if the exception is thrown as a reference, but the handler declares it as a `const` parameter?
- Are there standard exceptions, and if so, what are they? Which library functions can raise them? The answer to this is that there is a class hierarchy of exceptions, with, not surprisingly, a root named `exception`. All exceptions derive from `exception`. There are `logic_error` exceptions, `bad_cast`, `bad_alloc` (thrown by `new()`), `underflow_error`, and `out_of_range` (thrown by `at( )`) exceptions, to name a few.

Lastly, you may see function declarations that look like

```
void f ( int a) throw ( err1, err2);
```

which specifies that `f()` is only allowed to throw exceptions of types `err1` and `err2`, and any exceptions that may be derived from these types. Why would you want to do this? Because it is a way of telling the reader of your interface that the function promises to throw only those exceptions, and so the caller does not have to handle any exceptions except those. So

```
void f( int a) throw ( );
```

means that `f()` throws no exceptions. If `f()` does throw a different exception, the system will turn it into an `std::unexpected()` call, which results in a terminate call.

B The Standard Exception Hierarchy

The set of standard exceptions forms a hierarchy in C++. The language defines a base class named `exception` from which other exceptions are derived. Some of these are defined in the `<stdexcept>` header file; others are defined in `<typeinfo>`, `<new>`, `<ios>`, and `<exception>`. The standard exceptions are listed below. The indentation indicates sub-classing. To the right of the exception name is the header file that contains its definition.

Exception Type			**Header File**
`exception`			`<exception>`
	`bad_alloc`		`<new>`
	`bad_cast`		`<typeinfo>`
	`bad_exception`		`<exception>`
	`bad_typeid`		`<typeinfo>`
	`ios_base::failure`		`<ios>`
	`logic_error`		`<stdexcept>`
		`length_error`	`<stdexcept>`
		`domain_error`	`<stdexcept>`
		`out_of_range`	`<stdexcept>`
		`invalid_argument`	`<stdexcept>`
	`runtime_error`		`<stdexcept>`
		`overflow_error`	`<stdexcept>`
		`range_error`	`<stdexcept>`
		`underflow_error`	`<stdexcept>`