The following shows the output for the program:

```
A programming language serves two related purposes: it provides a
vehicle for the programmer to specify actions to be executed and a
set of concepts for the programmer to use when thinking about what
can be done. The first aspect ideally requires a language that is
‘close to the machine’, so that all important aspects of a machine
are handled simply and efficiently in a way that is reasonably
obvious to the programmer. The C language was primarily designed with
this in mind. The second aspect ideally requires a language that is
‘close to the problem to be solved’ so that the concepts of a
solution can be expressed directly and concisely. The facilities
added to C to create C++ were primarily designed with this in mind.

      -- Bjarne Stroustrup

There are 129 words
There are 71 unique words
The most common word is ‘to’ and is used 9 times
```

```
9    int main (void) {
10     Hash_Table<Gen_String,int> a1;           // Declare Hash_Table variable
11     Gen_String s;                            // Temporary string variable
12     int counter = 0, max_count = 0;          // Initialize word counters
13     cout << text;                            // Output paragraph
14     text.compile (”[a-zA-Z]+”);              // Match any alphabetical word
15     while (text.find ()) {                   // While still more words
16      text.sub_string (s, text.start (), text.end ()); // Get word
17      if (h1.find (s))                        // If word already found
18        h1.put (h1.key (), h1.value ()+1);    // Update use count
19    else h1.put (s, 1);                       // Else add word
20     }
21     h1.reset ();                             // Invalidate current position
22   Iterator<Hash_Table> i1;                   // Iterator object
23   while (h1.next ()) {                       // While there are still nodes
24    counter += h1.value ();                   // Sum number of words used
25    if (h1.value () > max_count) {            // If most used word so far
26      i1 = h1.current_position ();            // Save position in list
27      max_count = h1.value ();                // And keep track of usage
28    }
29   }
30   cout << ”There are ” << counter << ” words in the paragraph\n”;
31   cout << ”There are ” << h1.length () << ” unique words in the paragraph\n”;
32   h1.current_position () = i1;               // Set position of most used word
33   cout << ”The most common word is `” << h1.key () << ”' and is used ” <<
                    h1.value () << ” times\n”;
34   exit (0);                                  // Exit with successful status
35   }
```

Lines 1 through 3 include the COOL `Hash_Table.h`, `Gen_String.h`, and `Iterator.h` class header files. Line 4 includes a statically allocated paragraph of text to be scanned by the program. Lines 5 and 6 define a container class of a hash table whose key is a string and whose value is an integer. Lines 7 and 8 define an iterator for the hash table class. Lines 10 through 13 declare various variables and print the complete paragraph. A regular expression to match sequences of alphabetical characters (that is, words) is compiled in line 14. Lines 15 through 20 contain a loop that finds each word in the paragraph and adds it to the hash table if not already there. Otherwise, the current frequency is incremented and used as the new value for the key. Line 21 resets the internal current position iterator inside the hash table object and line 22 defines an iterator for a hash table object.

Lines 23 through 29 are the heart of the program. The loop iterates through the elements of the hash table summing up the frequencies of all the words to get a total word count. In addition, if the count for a given word is the largest so far, the position in the table is saved in the iterator object. This procedure repeats until all words have been scanned. Lines 30 through 33 output the results of the word search and count. Finally, the program ends with a successful completion code.

void set_value_compare (*Hash_Value_Compare* = **NULL**);
Updates the value compare function for this instance of hash table. *Hash_Value_Compare* is a function of type **Boolean** (**Function*)(**const** *Vtype&*,**const** *Vtype&*). If no argument is provided, the **operator==** for *Vtype*, the value over which the class is parameterized, is used.

const *Vtype&* **value** ();
Returns a reference to the value of the key/value pair at the current position. If the current position is invalid, an **Error** exception is raised.

Friend Functions:

friend ostream& operator<< (*ostream& os*,
const Hash_Table<*Ktype,Vtype*>& *ht*);
Overloads the output operator for a reference to a **Hash_Table**<*Ktype,Vtype*> object. This function provides a formatted output with key/value pairs printed one per line.

inline friend ostream& operator<< (*ostream& os*,
const Hash_Table<*Ktype,Vtype*>* *ht*);
Overloads the output operator for a pointer to a **Hash_Table**<*Ktype,Vtype*> object. This function provides a formatted output with key/value pairs printed one per line.

Hash Table Example

7.9 The following program declares a hash table of strings and integers, storing each word as the key and its frequency of occurrence in a paragraph of text as the value. The hash table is traversed using the current position function of the class to determine the total number of words and the most commonly used word in the paragraph.

```
#include <cool/Hash_Table.h>            // Include Hash_Table class
#include <cool/Gen_String.h>            // Include COOL String class
#include <cool/Iterator.h>              // Include COOL Iterator class
#include "paragraph.h"                  // Include Stroustrup text

DECLARE Hash_Table<Gen_String,int>      // Declare Hash_Table type
IMPLEMENT Hash_Table<Gen_String,int>    // Implement Hash_Table type
DECLARE Iterator<Hash_Table>            // Declare Hash_Table iterator
IMPLEMENT Iterator<Hash_Table>          // Implement Hash_Table iterator
```

Boolean prev ();
Moves the current position pointer to the previous entry in the hash table and returns **TRUE**. If the current position is invalid, this function moves to the last entry and returns **TRUE**. If moving to the previous entry passes the first entry in the hash table, this function invalidates the current position and returns **FALSE**.

Boolean put (**const** *Ktype& key,* **const** *Vtype& value*);
Searches the hash table for *key* and puts the corresponding key/value pair into the hash table. If *key* is not there, the key/value pair is added and **TRUE** is returned; otherwise, if *key* is already there, this function updates the value with *value* and returns **FALSE**. If the bucket determined by the hash is full, the table grows and the key/value pairs are rehashed and inserted. This function sets the current position to the key/value pair.

Boolean remove ();
Removes the key/value at the current position and returns **TRUE**. This function sets the current position to the entry immediately following the entry removed if in the same bucket; otherwise, this function invalidates the current position. If the current position is invalid, an **Error** exception is raised and, if the handler returns, this function returns **FALSE**.

Boolean remove (**const** *Ktype& key*);
Searches the hash table for *key*, removes the indicated key/value pair from the table, sets the current position to the old location of the key/value pair, and returns **TRUE**. If *key* is not found in the hash table, this function returns **FALSE**.

inline void reset ();
Invalidates the current position.

Boolean resize (**long** *number*);
Resizes the hash table for at least the indicated number of entries. If a growth ratio has been selected and it satisfies the resize request, the table is grown by this ratio. This function invalidates the current position. If the resize value is zero or negative, an **Error** exception is raised.

inline void set_hash (*Hash h*);
Updates the hash function for this instance of hash table. *Hash* is a function of type **unsigned long** (**Function*) (**const** *Ktype&*). If the key is of type **char***, the hash is the result of successively exclusive-or-ing each byte with the current hash value shifted left seven bits. If the key is not of type **char***, the default hash function is the computation of a 32-bit value shifted left three bits with the result then modulo the prime number of buckets. If the size of (*Ktype*) is greater than four bytes, the 32-bit value is computed by successively exclusive-or-ing 32-bit values for the length of the key.

void set_key_compare (*Hash_Key_Compare* = **NULL**);
Updates the key compare function for this instance of hash table. *Hash_Key_Compare* is a function of type **Boolean** (**Function*)(**const** *Ktype&*, **const** *Ktype&*). If no argument is provided, the **operator==** for *Ktype*, the key over which the class is parameterized, is used. If the key is a **char***, a **String**, or a **Gen_String**, the default compare function is a string comparison.

inline void set_ratio (*float*);
Updates the growth ratio for this instance of the hash table to the specified value. When a hash table needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, an **Error** exception is raised.

Boolean find (**const** *Ktype& key*);
Searches the hash table for *key* and returns **TRUE** if found; otherwise, this function returns **FALSE**. If *key* is found, this function sets the current position to the key/value entry; otherwise, this function invalidates the current position.

Boolean get (**const** *Ktype& key, Vtype& value*);
Calculates the hash value for *key* and returns the value associated with that key in the table by copying it to *value*. This function sets the current position to the key/value pair. If *key* is found, this function returns **TRUE**; otherwise, this function returns **FALSE**.

inline long get_bucket_count () **const**;
Returns the prime number of buckets currently allocated for the hash table.

inline int get_count_in_bucket (**long** *n)* **const**;
Returns the number of keys currently hashed to the zero-relative *n*th bucket.

Boolean get_key (**const** *Vtype& value, Ktype& key*);
Searches the table for *value*. If found, this function copies the associated key into *key*, sets the current position to the key/value pair, and returns **TRUE**. If *value* is not found in the hash table, this function invalidates the current position and returns **FALSE**.

inline Boolean is_empty () **const**;
Returns **TRUE** if the hash table contains no entries; otherwise, this function returns **FALSE**.

const *Ktype&* **key** ();
Returns the key of the key/value pair at the current position. If the current position is invalid, an **Error** exception is raised.

inline long length () **const**;
Returns the number of entries in the hash table.

Boolean next ();
Advances the current position pointer to the next entry in the hash table and returns **TRUE**. If the current position is invalid, this function advances to the first entry and returns **TRUE**. If advancing past the last entry in the hash table, this function invalidates the current position and returns **FALSE**.

Hash_Table<*Ktype,Vtype*>& **operator=** (**const**
    Hash_Table<*Ktype,Vtype*>& *ht*);
Overloads the assignment operator for the class and assigns one hash table object to have the value of another by duplicating the size and entries. This function invalidates the current position of the object.

Boolean operator== (**const Hash_Table**<*Ktype,Vtype*>& *ht*);
Overloads the equality operator for the hash table class. This function returns **TRUE** if the tables have the same number of buckets with the same key/value pairs; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Hash_Table**<*Ktype,Vtype*>& *ht*);
Overloads the inequality operator for the hash table class. This function returns **TRUE** if the tables have a different number of buckets or different key/value pairs; otherwise, this function returns **FALSE**.

The following shows the output for the program:

```
A programming language serves two related purposes: it provides a
vehicle for the programmer to specify actions to be executed and a
set of concepts for the programmer to use when thinking about what
can be done. The first aspect ideally requires a language that is
'close to the machine', so that all important aspects of a machine
are handled simply and efficiently in a way that is reasonably
obvious to the programmer. The C language was primarily designed with
this in mind. The second aspect ideally requires a language that is
'close to the problem to be solved' so that the concepts of a
solution can be expressed directly and concisely. The facilities
added to C to create C++ were primarily designed with this in mind.

     -- Bjarne Stroustrup

There are 129 words
There are 71 unique words
The most common word is 'to' and is used 9 times
```

Hash_Table Class

7.8 The **Hash_Table**<*Ktype,VType*> class is publicly derived from the **Hash_Table** class and implements hash tables of user-specified types for both the key and the value. This is accomplished by using the parameterized type capability of C++. The **Hash_Table** class is dynamic in nature. Its size (that is, the number of buckets in the table) is always a prime number. Each bucket holds eight items. No holes are left in a bucket; if a key/value pair is removed from the middle of a bucket, the following entries are moved up. When a hash on a key ends up in a bucket that is full, the table is enlarged.

Name: **Hash_Table**<*Ktype,Vtype*> — A dynamic, parameterized hash table

Synopsis: **#include** <COOL/Hash_Table.h>

Base Classes: **Hash_Table, Generic**

Friend Classes: None

Constructors:

Hash_Table<*Ktype,Vtype*> ();
Allocates a hash table of the default size (three buckets).

Hash_Table<*Ktype,Vtype*> (**unsigned long** *number*);
Allocates a hash table with at least enough buckets for *number* entries.

Hash_Table<*Ktype,Vtype*> (**const Hash_Table**<*Ktype,Vtype*>& *ht*);
Duplicates the size and entries of another hash table object *ht*.

Member Functions:

inline long capacity () **const**;
Returns the maximum number of entries that the hash table can hold.

void clear ();
Removes all entries from the hash table and adjusts the appropriate counts.

inline Hash_Table_state& current_position () **const;**
Returns a reference to the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of hash table.

```
9    int main (void) {
10     Association<Gen_String,int> a1;          // Declare Association variable
11     Gen_String s;                            // Temporary string variable
12     int counter = 0, max_count = 0;          // Initialize word counters
13     cout << text;                            // Output paragraph
14     text.compile ("[a-zA-Z]+");              // Match any alphabetical word
15     while (text.find ()) {                   // While still more words
16      text.sub_string (s, text.start (), text.end ()); // Get word
17      if (a1.find (s))                        // If word already found
18        ++a1.value ();                        // Increment use count
19    else a1.put (s, 1);                       // Else add word
20     }
21     a1.reset ();                             // Invalidate current position
22    Iterator<Association> i1;                 // Iterator object
23    while (a1.next ()) {                      // While there are still nodes
24     counter += a1.value ();                  // Sum number of words used
25     if (a1.value () > max_count) {           // If most used word so far
26       i1 = a1.current_position ();           // Save position in list
27       max_count = a1.value ();               // And keep track of usage
28     }
29    }
30   cout << "There are " << counter << " words\n";
31   cout << "There are " << a1.length () << " unique words\n";
32   a1.current_position () = i1;              // Set position of most used word
33   cout << "The most common word is `" << a1.key () << "' and is used " <<
                   a1.value () << " times\n";
34   exit (0);                                 // Exit with successful status
35   }
```

Lines 1 through 3 include the COOL `Association.h`, `Gen_String.h`, and `Iterator.h` class header files. Line 4 includes a statically allocated **Gen_String** object that contains a paragraph of text to be scanned by the program. Lines 5 and 6 define a container class of an association of strings and integers, and lines 7 and 8 define an iterator for the association class. Lines 10 through 13 declare various variables and print the complete paragraph. A regular expression to match sequences of alphabetical characters (that is, words) is compiled in line 14. Lines 15 through 20 contain a loop that finds each word in the paragraph and adds it to the association if not already there. Otherwise, the current frequency is incremented. Line 21 resets the internal current position iterator inside the association object, and line 22 defines an iterator for an association object.

Lines 23 through 29 are the heart of the program. The loop iterates through the elements of the association summing up the frequencies of all the words to get a total word count. In addition, if the count for a given word is the largest so far, the position in the association is saved in the iterator object. This procedure repeats until all words have been scanned. Lines 30 through 33 output the results of the word search and counting. Finally, the program ends with a successful completion code.

inline void set_alloc_size (**int** *size*);
Updates the allocation growth size to be used when the growth ratio is zero. Default allocation growth size is 100 bytes. If the size specified is negative, an **Error** exception is raised.

inline void set_growth_ratio (**float** *ratio*);
Updates the growth ratio for this instance of an association to the specified value. When an association needs to grow, the current size is multiplied by the ratio to determine the new size. If *ratio* is negative, an **Error** exception is raised.

inline void set_key_compare (*Assoc_Key_Compare* = **NULL**);
Updates the key compare function for this class of association. *Assoc_Key_Compare* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator==** for *Ktype* over which the key for the association class is parameterized is used.

inline long set_length (**long** *number*);
Specifies the *number* of elements in an association to allow random access via the overloaded **operator[]** member function. If *number* is larger than the storage allocated, this function truncates *number* to the largest value the allocated size will support. This function returns the updated number of elements.

inline void set_value_compare (*Assoc_Value_Compare* = **NULL**);
Updates the value compare function for this class of association. *Assoc_Value_Compare* is a function of type **Boolean** (**Function*)(**const** *Ktype&*, **const** *Vtype&*). If no argument is provided, the **operator==** for *Vtype* over which the value for the association class is parameterized is used.

inline Vtype& value ();
Returns a reference to the value of the key/value pair at the current position.

Friend Functions:

friend ostream& operator<< (*ostream os*,
const Association<*Ktype,Vtype*>*& assoc*);
Provides a formatted output capability for reference to an **Association**<*Ktype,Vtype*> object.

inline friend ostream& operator<< (*ostream& os*,
const Association<*Ktype,Vtype*>** assoc*);
Provides a formatted output capability for a pointer to an **Association**<*Ktype,Vtype*> object.

Association Example

7.7 The following program declares an association of strings and integers, storing each word and its frequency of occurrence in a paragraph of text in individual elements. The association of words is traversed using the current position functionality of the class to determine the total the number of words and the most commonly used word in the paragraph.

```
#include <cool/Association.h>              // Include Association class
#include <cool/Gen_String.h>               // Include COOL String class
#include <cool/Iterator.h>                 // Include COOL Iterator class
#include "paragraph.h"                     // Include Stroustrup text

DECLARE Association<Gen_String,int>        // Declare association type
IMPLEMENT Association<Gen_String,int>      //  Implement association type
DECLARE Iterator<Association>              // Declare assoc iterator
IMPLEMENT Iterator<Association>            // Implement assoc iterator
```

inline Boolean next ();

Advances the current position pointer to the next element in the association and returns **TRUE.** If the current position is invalid, this function advances to the first element and returns **TRUE**. If advancing past the last element, this function invalidates the current position and returns **FALSE**.

Association<*Ktype*,*Vtype*>& **operator=** (**const Association**<*Ktype*,*Vtype*>&);

Overloads the assignment operator for the **Association** class and assigns one association object to have the value of another by duplicating the size and element values. This function invalidates the current position. If the association is prohibited from dynamically growing as necessary, an **Error** exception is raised.

Boolean operator== (**const Association**<*Ktype*,*Vtype*>& *assoc*) **const**;

Overloads the equality operator for the **Association** class. This function returns **TRUE** if the associations have the same number of elements with the same values; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const Association**<*Ktype*,*Vtype*>& *assoc*) **const**;

Overloads the inequality operator for the **Association** class. This function returns **TRUE** if the associations have a different number of elements or different values; otherwise, this function returns **FALSE**.

inline Boolean prev ();

Moves the current position pointer to the previous element in the association and returns **TRUE**. If the current position is invalid, this function moves to the last element and returns **TRUE**. If moving to the previous element passes the first element in the association, this function invalidates the current position and returns **FALSE**.

Boolean put (**const** *Ktype& key*, **const** *Vtype& value*);

Puts the *key*/*value* pair into the association. If a pair already exists with the specified key, the value for that pair is replaced with *value*. If required and not prohibited, the association is grown. If the new pair is successfully put into the association, **TRUE** is returned; otherwise, **FALSE** is returned.

Vtype& remove ();

Removes and returns a reference to the element at the current position. This function sets the current position to the element immediately following the element removed. If the element removed is at the end of the association, this function invalidates the current position. If the current position is invalid, and **Error** exception is raised.

Boolean remove (**const** *Ktype& key*);

Searches for *key* and, if found, this function removes the pair associated with *key* and sets the current position to the element immediately following the element removed; then, the function returns **TRUE**. If *key* is found at the end of the association, this function invalidates the current position and returns **TRUE**. If *key* is not found, this function returns **FALSE**.

inline void reset ();

Invalidates the current position.

inline void resize (**long** *number*);

Resizes the association for at least *number* elements. If a growth ratio has been selected and it satisfies the resize request, the association is grown by this ratio. This function invalidates the current position. If the size specified is zero or negative, an **Error** exception is raised.

Name: **Association**<*Ktype*,*Vtype*> — A dynamic, parameterized association

Synopsis: **#include** <COOL/Association.h>

Base Classes: **Vector**<*Type*>, **Vector**, **Generic**

Friend Classes: None

Constructors:

Association<*Ktype*,*Vtype*> ();
Creates an empty association of the specified type.

Association<*Ktype*,*Vtype*> (**const Association**<*Ktype*,*Vtype*>& *assoc*);
Duplicates the size and value of an association object.

Association<*Ktype*,*Vtype*> (**unsigned long** *number*);
Allocates enough storage for an association of a specific type to hold *number* elements.

Association<*Type*> (**void*** *storage*, **unsigned long** *number*);
Creates a static-sized association object for *number* elements whose storage *storage* is provided by the user. If an object of this type attempts to grow dynamically or the programmer invokes the **resize** member function, an **Error** exception is raised.

Member Functions:

inline long capacity ();
Returns the maximum number of elements the association can contain.

inline void clear ()
Removes all elements in the object and invalidates the current position.

inline Association_state& current_position ()**;**
Returns the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of association.

Boolean find (**const** *Ktype& key*);
Searches the association for *key*. If found, this function sets the current position and returns **TRUE**; otherwise, this function resets the current position and returns **FALSE**.

Boolean get (**const** *Ktype& key*, *Vtype& value*);
Gets the associated *value* for *key*. This function returns **TRUE** and modifies *value* to contain the associated value. If *key* is not found, this function returns **FALSE** and does not modify *value*.

Boolean get_key (**const** *Vtype& value*, *Ktype& key*) **const**;
Gets the first associated *key* for *value*. This function returns **TRUE** and modifies *key* to contain the associated key. If *value* is not found, this function returns **FALSE** and does not modify *key*.

inline const *Ktype&* **key** () **const**;
Returns the key of the key/value pair at the current position.

inline long length ();
Returns the number of elements (pairs) in the association.

inline Boolean operator!= (**Pair**<*T1*,*T2*>&) **const**;
Returns **TRUE** if the pairs have different element values; otherwise, this function returns **FALSE** .

inline T2& second ();
Returns a reference to the second element of the pair.

inline void set_compare (*Pair_Compare* = **NULL**);
Updates the compare function for this class of pair. *Pair_Compare* is a function of type **Boolean** (**Function*)(**const Pair**<*T1*,*T2*>&, **const Pair**<*T1*,*T2*>&). If no argument is provided, the **operator==** for the types over which the class is parameterized are used.

inline void set_first (**const** *T1&*);
Sets the first element of the pair to the specified value.

inline void set_second (**const** *T2&*);
Sets the second element of the pair to the specified value.

Friend Functions:

friend ostream& operator<< (*ostream&*, **const Pair**<*T1*,*T2*>&);
Provides a formatted output capability for reference to a **Pair**<*T1,T2*> object.

inline friend ostream& operator<< (*ostream&*, **const Pair**<*T1*,*T2*>*);
Provides a formatted output capability for a pointer to a **Pair**<*T1*,*T2*> object.

Association Class

7.6 The **Association**<*Ktype*,*Vtype*> class is privately derived from the **Vector**<*Type*>class and implements a collection of pairs. The first of the pair is called the *key*, and the second of the pair is called the *value*. The **Association**<*Ktype*,*Vtype*> class implements a one-dimensional vector parameterized over a pair of objects. The first type specifies the type of the key, and the second type specifies the type of the value. Many of the member functions for **Association**<*Ktype*,*Vtype*> are inherited from **Vector**<*Type*> and, consequently, are inline calls to the vector member function of the same name.

The **Association**<*Ktype*,*Vtype*> class inherits the dynamic growth capability of the **Vector** class. Vectors are, by default, dynamic in nature. A static-sized vector object is selected by setting the growth allocation size to zero or by passing in a pointer to a block of user-supplied storage to the constructor. If a vector is of static size and an operation is performed that requires more storage, an **Error** exception is raised.

The **Association**<K*type, Vtype*> class implements the notion of a current position. This is useful for iterating through the elements of a vector. The current position is maintained in a data member of type **Association_state** and is set or reset by all member functions affecting elements in the class. Member functions are provided to reset the current position, move to the next and previous elements, find an element, and get the value at the current position. The **Iterator**<*Type*> class provides a mechanism to save and restore the state associated with the current position, thus allowing the programmer to use multiple iterators over the same instance of an association object.

```
A programming language serves two related purposes: it provides a
vehicle for the programmer to specify actions to be executed and a
set of concepts for the programmer to use when thinking about what
can be done. The first aspect ideally requires a language that is
'close to the machine', so that all important aspects of a machine
are handled simply and efficiently in a way that is reasonably
obvious to the programmer. The C language was primarily designed with
this in mind. The second aspect ideally requires a language that is
'close to the problem to be solved' so that the concepts of a
solution can be expressed directly and concisely. The facilities
added to C to create C++ were primarily designed with this in mind.

     -- Bjarne Stroustrup

There are 129 words
There are 71 unique words
The most common word is 'to' and is used 9 times
```

Pair Class

7.5 The parameterized **Pair**<*T1*,*T2*> class implements an association between one object and another. The objects may be of different types, with the first representing the *key* of the pair and the second representing the *value* of the pair. The **Pair**<*T1*,*T2*> class is used by the **Association**<*Ktype*,*Vtype*> class to implement an association list (that is, a vector of pairs of associated values).

Name: **Pair**<*T1*,*T2*> — A parameterized pair

Synopsis: **#include** <COOL/Pair.h>

Base Classes: None

Friend Classes: None

Constructors:

Pair<*T1*,*T2*> ();
Creates an empty pair of the specified types.

Pair<*T1*,*T2*> (**const Pair**<*T1*,*T2*>&);
Duplicates the size and value of a pair object.

Pair<*T1*,*T2*> (**const** *T1&*, **const** *T2&*);
Creates a pair from the two specified elements.

Member Functions:

inline const *T1&* **get_first** () **const**;
Returns a constant reference to the first element of the pair.

inline const *T2&* **get_second** () **const**;
Returns a constant reference to the second element of the pair.

inline T1& first ();
Returns a reference to the first element of the pair.

Pair<*T1*,*T2*>& **operator=** (**Pair**<*T1*,*T2*>&);
Assigns one pair object to have the value of another by duplicating element values.

Boolean operator== (**Pair**<*T1*,*T2*>&) **const**;
Returns **TRUE** if the pairs have the same element values; otherwise, this function returns **FALSE**.

```
26      while (l1.next ())                   // While there are still nodes
27        if (l1.value () == cur_word)       // If word appears in list
28                counter++;                 // Increment usage count
29      if (counter > max_count) {           // If most used word so far
30        max_count = counter;               // Update maximum count
31        s = cur_word;                      // And save word
32      }
33      l1.current_position () = i1;         // Restore old current position
34    }
35    cout << "There are " << l1.length () << " words\n";
36    l1.remove_duplicates (); // Remove duplicate words
37    cout << "There are " << l1.length () << " unique words\n";
38    cout << "The most common word is `" << s << "' and is used " <<
                max_count << " times\n";
39    exit (0);                              // Exit with successful status
40  }
```

Lines 1 through 3 include the COOL `List.h`, `Gen_String.h`, and `Iterator.h` class header files. Line 4 includes a statically allocated **Gen_String** object that contains a paragraph of text to be scanned by the program. Lines 5 and 6 define a container class of a list of strings and lines 7 through 8 define an iterator for the list class. Lines 10 through 13 declare various variables and print the complete paragraph. A regular expression to match sequences of alphabetical characters (that is, words) is compiled in line 14. Lines 15 through 18 contain a loop that finds each word in the paragraph and pushes it onto the list. Line 18 resets the internal current position iterator inside the list object.

Lines 20 through 34 are the heart of the program. The loop iterates through the elements of the list, assigning each word to a current word variable and the current position to a list iterator object. An inner loop uses the current position functionality to loop through the elements of the list counting the number occurrences of the current word. If this count is the largest so far, both the word and the count are saved. When the inner loop terminates, the outer loop establishes the previous current position maintained by the iterator object and the procedure is repeated again until all words have been scanned. Lines 35 through 38 output the results of the word search and counting. Finally, the program ends with a successful completion code.

The following shows the output for the program:

Boolean sublist (**List**<*Type*>& *l1*, **const List**<*Type*>& *l2*);
Searches for *l2* in the object. If *l2* is a sublist of the object, this function sets the current position of the object to the first element of *l2*, sets *l1* to the sublist within the object (starting at the new current position), and returns **TRUE**; otherwise, this function sets *l1* list to **NIL** (an empty list) and returns **FALSE.**

void tail (**List**<*Type*>& *l*, **int** *n* = 1);
Sets *l* to point to the *n*th tail of the object. The *n*th tail is a list whose first element is the *n*th (zero-relative) element of the object. When it has no second argument, **tail** sets *l* to point to the first tail of the object. *n*=1 sets *l* to all of the elements in the object. This function sets *l* to **NIL** if *n* is greater than or equal to the number of elements in the object. This function sets the current position to the *n*th element of the object.

inline *Type&* **value** ();
Returns the element at the current position in the object. An **Error** exception is raised if the current position is invalid.

Friend Functions:

friend ostream& operator<< (*ostream& os*, **const List**<*Type*>& *l*);
Provides a formatted output capability for a reference to a list.

inline friend ostream& operator<< (*ostream& os*, **const List**<*Type*>* *l*);
Provides a formatted output capability for a pointer to a list.

List Example

7.4 The following program declares a list of strings and stores the words in a paragraph of text in individual nodes. The list of words is traversed using a parameterized iterator, and the nodes are manipulated to determine the total number of words, the number of unique words, and the most commonly used word in the paragraph. These results are sent to the standard output, and the program then ends.

```
#include <cool/List.h>                    // Include list header file
#include <cool/Gen_String.h>              // Include COOL String class
#include <cool/Iterator.h>                // Include COOL Iterator class
#include "paragraph.h"                    // Include Stroustrup text

DECLARE List<Gen_String>                  // Declare list type
IMPLEMENT List<Gen_String>                // Implement list type
DECLARE Iterator<List>                    // Declare list iterator type
IMPLEMENT Iterator<List>                  // Implement list iterator type

int main (void) {
  List<Gen_String> l1;                    // Declare list variable
  Gen_String s;                           // Temporary string variable
  int max_count = 0;                      // Temporary counting variable
  cout << text;                           // Output paragraph
  text.compile ("[a-zA-Z]+");             // Match any alphabetical word
  while (text.find ()) {                  // While still more words
   text.sub_string (s, text.start (), text.end ()); // Get word
   l1.push (s);                           // And add to list
  }
  l1.reset ();                            // Invalidate current position
  while (l1.next ()) {                    // While there are still nodes
    int counter = 0;                      // Initialize counter
    Gen_String cur_word;                  // Temporary string variable
    Iterator<List> i1 = l1.current_position (); // Save current position
    cur_word = l1.value ();               // Get word to be counted
    l1.reset ();                          // Invalidate current position
```

Type& remove ();
Removes the element at the current position in the object and returns a reference to the element. This function sets the current position to the element immediately following the element removed. If the current position is invalid, an **Error** exception is raised.

inline Boolean remove (**const** *Type& value*);
Removes the first occurrence of *value* in the object. If the element is found, this function removes it from the object, sets the current position to the element immediately following the element removed, and returns **TRUE**. If the element is not found, this function returns **FALSE**.

Boolean remove_duplicates ();
Removes any duplicate elements from the object. This function returns **TRUE** if any elements were removed; otherwise, this function returns **FALSE**. This function invalidates the current position.

inline Boolean replace (**const** *Type& value1*, **const** *Type& value2*);
Replaces the first occurrence of *value1* in the object with *value2* and sets the current position to this element. If *value1* is found, this function returns **TRUE**; otherwise, this function returns **FALSE**.

inline Boolean replace_all (**const** *Type& value1*, **const** *Type& value1*);
Replaces all occurrences of *value1* in the object with *value2*. This function returns **TRUE** if any elements were replaced; otherwise, this function returns **FALSE**. This function invalidates the current position.

inline void reset ();
Invalidates the current position in the object.

void reverse ();
Reverses the order of the elements in the object. This function invalidates the current position.

Boolean search (**const List**<*Type*>& *l*);
Searches for the sublist *l* in the object. If *l* is a sublist in the object, this function sets the current position in the object to the first element of the sublist and returns **TRUE**; otherwise, this function returns **FALSE**.

inline void set_compare (*List_Compare* = **NULL**);
Updates the compare function for the object. *List_Compare* is a function of type **Boolean** (**Function*) (**const** *Type&*, **const** *Type&*). If no argument is provided, the **operator=** for the type over which the class is parameterized is used.

Boolean set_tail (**const List**<*Type*>& *l*, **int** *n* = 1);
Sets the *n*th tail of the object to *l*. This function sets the current position of the object to the first element of the *n*th tail. This function returns **TRUE** if the object has an *n*th tail; otherwise, this function returns **FALSE**.

inline void sort (*List_Predicate p*);
Sorts the elements of the object by using *p* for determining the collating sequence. *List_Predicate* is a function of type **int** (**Function*) (**const** *Type&*, **const** *Type&*) that returns –1 if the first argument should precede the second, zero if they are equal, and 1 if the first argument should follow the second.

Type pop ();
Removes and returns the first element in the object. This function invalidates the current position. If there is nothing in the object, an **Error** exception is raised.

Boolean pop (*Type& value*);
Copies the first element in the object to *value* and removes it from the object. This function invalidates the current position. If there is nothing in the list, an **Error** exception is raised.

int position ();
Returns the current position as a zero-relative index into the object that can be used with the overloaded **operator[]**.

int position (**const** *Type& value*);
Searches the object for *value*. If the element is found, this function sets the current position to this element and returns the zero-relative index of this element; otherwise, this function returns –1.

inline Boolean prepend (**const List**<*Type*>& *l*);
Adds the elements of *l* to the beginning of the object and returns **TRUE**. This function sets the current position to the first element added. This function returns **FALSE** if the specified list argument is **NIL**.

Boolean prev ();
Moves the current position to the previous element in the object and returns **TRUE**. If the current position is invalid, this function sets the current position to the last element and returns **TRUE**. If the current position is the first element in the object, this function invalidates the current position and returns **FALSE**.

Boolean push (**const** *Type& value*);
Adds *value* to the beginning of the object and returns **TRUE**. This function sets the current position to the first element of the object. This function returns **FALSE** when heap memory is exhausted.

inline Boolean push_end (**const** *Type& value*);
Adds *value* to the end of the object and returns **TRUE**. This function sets the current position to the last element of the object. This function returns **FALSE** when heap memory is exhausted.

inline Boolean push_end_new (**const** *Type& value*);
Adds *value* to the end of the object (if it is not already in the object) and sets the current position to this element. This function returns **TRUE** if the element is added to the object; otherwise, this function returns **FALSE**.

inline Boolean push_new (**const** *Type& value*);
Adds *value* to the beginning of the object (if it is not already in the object) and sets the current position to this element. This function returns **TRUE** if the element is added to the object; otherwise, this function returns **FALSE**.

Boolean put (**const** *Type& value*, **int** *n* = 0);
Replaces the *n*th (zero-relative) element in the object with *value*. This function returns **TRUE** if the *n*th element exists; otherwise, this function returns **FALSE**. If *n* is negative, an **Error** exception is raised.

inline List<*Type*>& **operator+** (**const List**<*Type*>& *l*);
Returns a reference to a new list containing the concatenation of the object and *l*. Since the new list is allocated from heap memory, you must delete its storage.

inline List<*Type*>& **operator–** (**const List**<*Type*>& *l*);
Returns a reference to a new list containing the difference of the object and *l*. Since the new list is allocated from heap memory, you must delete its storage.

inline List<*Type*>& **operator^** (**const List**<*Type*>& *l*);
Returns a reference to a new list containing the exclusive-or of the object and *l*. Since the new list is allocated from heap memory, you must delete its storage.

inline List<*Type*>& **operator&** (**const List**<*Type*>& *l*);
Returns a reference to a new list containing the intersection of the object and *l*. Since the new list is allocated from heap memory, you must delete its storage.

inline List<*Type*>& **operator|** (**const List**<*Type*>& *l*);
Returns a reference to a new list containing the union of the object and *l*. Since the new list is allocated from heap memory, you must delete its storage.

inline List<*Type*>& **operator=** (**List**<*Type*>& *l*);
Assigns the object (the list on the left-hand side of the assignment) to point to the same set of elements as *l* (on the right-hand side of the assignment) and returns a reference to the updated object. Also, this function invalidates the current position.

inline List<*Type*>& **operator+=** (**const List**<*Type*>& *l*);
Sets the object (the list on the left-hand side of the operator) to the concatenation of the object and *l* (the list on the right-hand side of the operator) and returns a reference to the updated object. Also, this function invalidates the current position.

inline List<*Type*>& **operator–=** (**const List**<*Type*>& *l*);
Returns a reference to the modified object containing the difference of the object and *l*.

inline List<*Type*>& **operator^=** (**const List**<*Type*>& *l*);
Returns a reference to the modified object containing the exclusive-or of the object and *l*.

inline List<*Type*>& **operator&=** (**const List**<*Type*>&);
Returns a reference to the modified object containing the intersection of the object and *l*.

inline List<*Type*>& **operator|=** (**const List**<*Type*>&);
Returns a reference to the modified object containing the union of the object and *l*.

Boolean operator== (**const List**<*Type*>& *l*) **const**;
Returns **TRUE** if the elements of the two lists have the same values; otherwise, this function returns **FALSE**.

inline Boolean operator!= (**const List**<*Type*>& *l*) **const**;
Returns **TRUE** if the elements of the two lists have different values; otherwise, this function returns **FALSE**.

Type& operator[] (**int** *n*);
Returns a reference to the the *n*th (zero-relative) element in the object. This function sets the current position to this element. If the index is negative or is greater than the number of nodes in the list, an **Error** exception is raised.

inline Boolean is_empty ();

Returns **TRUE** if the object does not have any elements; otherwise, this function returns **FALSE**.

void last (**List**<*Type*>& *l*, **int** *n* = 1);

Sets *l* to point to the last *n* elements of the object. When it has no second arguments, **last** sets *l* to point to the last element of the object. If *n* is equal to the number of elements in the object, this function sets *l* to point to all the elements of the object. If *n* is greater than the number of elements in the object or *n* is zero, this function sets *l* to **NIL**, a list with no elements. This function sets the current position to the first of the last *n* elements of the list object.

int length ();

Returns the number of elements in the object.

void lunion (**const List**<*Type*>& *l*);

Sets the object to contain everything that is an element of either the object or *l*. This function invalidates the current position of the list object.

inline Boolean member (**List**<*Type*>& *l*, **const** *Type& value*);

Searches the object for *value*. If the element is found, this function sets the current position to this element, sets *l* to the sublist within the object starting with the desired element, and returns **TRUE**. If the *value* is not found, this function sets *l* to **NIL** and returns **FALSE**.

inline void merge (**const List**<*Type*>& *l*, *List_Predicate p*);

Merges the elements of *l* into the object by using the supplied predicate *p* for determining the collating sequence. *List_Predicate* is a function of type **Boolean** (**Function*)(**const** *Type&*, **const** *Type&*).

inline Boolean next ();

Advances the current position to the next element in the object and returns **TRUE**. If the current position is invalid, this function sets the current position to the first element and returns **TRUE**. If the current position is the last element of the object, this function invalidates the current position and returns **FALSE**.

Boolean next_difference (**const List**<*Type*>& *l*);

Sets the current position to the next element in the difference of the object and *l* and returns **TRUE**. If there are no more elements in the difference, this function invalidates the current position and returns **FALSE**.

Boolean next_exclusive_or (**const List**<*Type*>& *l*);

Sets the current position to the next element in the exclusive-or of the object and *l* and returns **TRUE**. If there are no more elements in the exclusive-or, this function invalidates the current position and returns **FALSE**.

Boolean next_intersection (**const List**<*Type*>& *l*);

Sets the current position to the next element in the intersection of the object and *l* and returns **TRUE**. If there are no more elements in the intersection, this function invalidates the current position and returns **FALSE**.

Boolean next_lunion (**const List**<*Type*>& *l*);

Sets the current position to the next element in the union of the object and *l* and returns **TRUE**. If there are no more elements in the union, this function invalidates the current position and returns **FALSE**.

void copy (**List**<*Type*>& *l*) **const**;
Sets *l* to a copy of the object. This function invalidates the current position of *l*.

inline List_state& current_position () **const;**
Returns a reference to the state information associated with the current position. This function should be used with the **Iterator**<*Type*> class to save and restore the current position, thus facilitating multiple iterators over an instance of list.

void describe (*ostream& os*);
Provides a formatted output capability for displaying the internal structure of the list including reference counts and values for each node.

void difference (**const List**<*Type*>& *l*);
Removes from the object the elements that also appear in *l*. This function invalidates the current position of the list object.

void exclusive_or (**const List**<*Type*>& *l*);
Performs an exclusive-or operation by setting the object to contain all the elements in the object that are not in *l*, and all the elements in *l* that are not in the object. This function invalidates the current position of the list object.

inline Boolean find (**const** *Type& value*, **List_state** *start* = **NULL**);
Searches the object for *value* beginning at the current position specified. If a starting point is not provided, this function begins the search at the head of the list. If *value* is found, this function sets the current position to this element and returns **TRUE**; otherwise, this function returns **FALSE**. If *value* is not found, this function returns **FALSE** and resets the current position of the list object.

inline Type& get (**int** *n* = 0);
Returns a reference to the *n*th (zero-relative) element in the object. This function sets the current position to this element. If the index is negative or is greater than the number of nodes in the list, an **Error** exception is raised.

inline Boolean insert_after (**const** *Type& value1*, **const** *Type& value2*);
Inserts *value1* after *value2* in the object, sets the current position to this new element, and returns **TRUE**. If *value2* is not in the object, this function returns **FALSE**.

inline Boolean insert_after (**const** *Type& value*);
Inserts *value* after the element at the current position in the object, sets the current position to this new element, and returns **TRUE**. If the current position is invalid, an **Error** exception is raised.

inline Boolean insert_before (**const** *Type& value1*, **const** *Type& value2*);
Inserts *value1* before *value2* in the object, sets the current position to this new element, and returns **TRUE**. If *value2* is not in the object, this function returns **FALSE**.

inline Boolean insert_before (**const** *Type& value*);
Inserts *value* before the element at the current position in the object, sets the current position to this new element, and returns **TRUE**. If the current position is invalid, an **Error** exception is raised.

void intersection (**const List**<*Type*>& *l*);
Sets the object to contain only the elements that exist in both the object and *l*. This function invalidates the current position of the list object.

List Class

7.3 The **List**<*Type*> class implements Common Lisp-style lists that provide a rich collection of member functions for list manipulation and management. A list consists of a collection of nodes, each of which contains a reference count, a pointer to the next node in the list, and a data element of a user-specified type. The **List**<*Type*> class uses reference counting to allow more efficient sharing and reuse of list node objects. The reference count indicates the number of list or node objects pointing to a node and ensures that the node and the data are deallocated when the node is no longer referenced.

Considerable attention has been paid to performance and efficiency concerns in the **List**<*Type*> class. The **Base_List** class implements generic list functionality required by the parameterized **List** class. The **Base_List** class is not usable as a stand-alone class, but used to derive the **List**<*Type*> class. By providing generic operations in a base class, the quantity of code generated for each implementation of a parameterized class is reduced considerably. Consequently, most member functions for **List**<*Type*> are inline calls to the generic equivalent function in the base **List** class.

Name: **List**<*Type*> — A parameterized list

Synopsis: **#include** <COOL/List.h>

Base Classes: **List, Generic**

Friend Classes: None

Constructors:

List<*Type*> ();
Creates an empty list of the specified type.

List<*Type*> (**const** *Type& value*);
Creates a list with one element of the specified type and value.

List<*Type*> (**int** *number*, *Type&*, ...);
Creates a list of *number* elements of the specified type initialized with the optional values provided.

List<*Type*> (**List**<*Type*>& *l*);
Creates a list from *l* of the specified type.

List<*Type*> (**const** *Type& value*, **List**<*Type*>& *l*);
Creates a list of the specified type with *value* as the first element and *l* as the tail.

Member Functions:

Boolean append (**const List**<*Type*>& *l*);
Adds the elements of *l* to the end of the object and returns **TRUE**. This function sets the current position to the first element added. This function returns **FALSE** if a new node cannot be created.

void but_last (**List**<*Type*>& *l*, **int** *n* = 1);
Sets *l* to point to all but the last *n* elements of the object. When no second argument is specified, **but_last** sets *l* to point to all but the last element of the object. A second argument whose value is zero sets *l* to point to all of the elements in the object. This function sets *l* to **NIL** if the second argument is greater than or equal to the number of elements in the object. This function invalidates the current position of *l*.

void clear ();
Removes all elements in the object and invalidates the current position.

UNORDERED SEQUENCE CLASSES 7

Introduction

7.1 The unordered sequence classes are a collection of basic data structures that implement random-access data structures as parameterized classes, thus allowing the user to customize a generic template to create a specific user-defined class. The following classes are discussed in this section:

- **List**<*Type*>
- **Pair**<*T1, T2*>
- **Association**<*Ktype, Vtype*>
- **Hash_Table**<*Ktype,Vtype*>

The **List**<*Type*> class implements dynamic, Common Lisp-style lists supporting such functions as insert, delete, replace, search, reverse, print, and sort. The **Pair**<*T1,T2*> class implements a simple object that contains two other objects, primarily for use in the **Association**<*Ktype, Vtype*> class. The **Association**<*Ktype, Vtype*> class maintains a collection of associated objects. The **Hash_Table**<K*type, Vtype*> class implements dynamic hash tables with the option for user-defined hashing functions. The **List**<*Type*>, **Hash_Table**<*Ktype, Vtype*>, and **Association**<K*type, Vtype*> classes support the notion of a current position. The example programs in this section solve the same problem using different data structures, allowing the reader to compare the different features of each.

In order to achieve successful compilation and usage, certain operations must be supported by any user-specified type over which an unordered sequence class is parameterized. The member functions **operator=**, **operator<**, **operator>**, **operator<<**, and **operator==** must be overloaded for any class object used as the type. Note that built-in types already have these functions defined.

NOTE: The unordered sequence classes use **operator=** of the parameterized type when copying elements. You should be careful when parameterizing an unordered sequence class over a pointer to a type, since the default pointer assignment operator usually copies the pointer, not the value pointed at.

Requirements

7.2 This section discusses the parameterized unordered sequence container classes. It assumes that you have read and understood Section 5, Parameterized Templates. In addition, no attempt is made to discuss the concepts and algorithms for the data structures discussed. You should refer to a general data structures or computer science text for this information.