Lists and Strings

6

6.1 LIST DEFINITION

Exercises 6.1

Given the methods for lists described in this section, write functions to do each of the following tasks. Be sure to specify the preconditions and postconditions for each function. You may use local variables of types List *and* List_entry*, but do not write any code that depends on the choice of implementation. Include code to detect and report an error if a function cannot complete normally.*

E1. Error_code insert_first(**const** List_entry &x, List &a_list) *inserts entry* x *into position 0 of the* List a_list.

Answer

```
Error_code insert_first(const List_entry &x, List &a_list)
/* Post:  Entry x is inserted at position 0 of List a_list. */
{
  return a_list.insert(0, x);
}
```

E2. Error_code remove_first(List_entry &x, List &a_list) *removes the first entry of the* List a_list, *copying it to* x.

Answer

```
Error_code remove_first(List_entry &x, List &a_list)
/* Post:  A code of underflow is returned if List a_list is empty.  Otherwise, the first entry of List
          a_list is removed and reported as x. */
{
  return a_list.remove(0, x);
}
```

E3. Error_code insert_last(**const** List_entry &x, List &a_list) *inserts* x *as the last entry of the* List a_list.

Answer

```
Error_code insert_last(const List_entry &x, List &a_list)
/* Post:  Parameter x is inserted as the last entry of the List a_list. */
{
  return a_list.insert(a_list.size(), x);
}
```

E4. Error_code remove_last(List_entry &x, List &a_list) *removes the last entry of* a_list, *copying it to* x.

Answer
```
Error_code remove_last(List_entry &x, List &a_list)
/* Post: A code of underflow is returned if List a_list is empty. Otherwise, the last entry of List
         a_list is removed and reported as x. */
{
  return a_list.remove(a_list.size() - 1, x);
}
```

E5. Error_code median_list(List_entry &x, List &a_list) *copies the central entry of the* List a_list *to* x *if* a_list *has an odd number of entries; otherwise, it copies the left-central entry of* a_list *to* x.

Answer
```
Error_code median_list(List_entry &x, List &a_list)
/* Post: A code of underflow is returned if List a_list is empty. Otherwise, the median entry of
         List a_list is reported as x. */
{
  return a_list.retrieve((a_list.size() - 1)/2, x);
}
```

E6. Error_code interchange(**int** pos1, **int** pos2, List &a_list) *interchanges the entries at positions* pos1 *and* pos2 *of the* List a_list.

Answer
```
Error_code interchange(int pos1, int pos2, List &a_list)
/* Post: Any entries at positions pos1 and pos2 of List a_list are interchanged. If either entry is
         missing a code of range_error is returned. */
{
  List_entry entry1, entry2;
  Error_code outcome = a_list.retrieve(pos1, entry1);
  if (outcome == success)
    a_list.retrieve(pos2, entry2);
  if (outcome == success)
    a_list.replace(pos1, entry2);
  if (outcome == success)
    a_list.replace(pos2, entry1);
  return outcome;
}
```

E7. **void** reverse_traverse_list(List &a_list, **void** (*visit)(List_entry &)) *traverses the* List a_list *in reverse order (from its last entry to its first).*

Answer
```
void reverse_traverse_list(List &a_list,
                           void (*visit)(List_entry &))
/* Post: The List a_list is traversed, in reverse order, and the function *visit is applied to all
         entries. */
{
  List_entry item;
  for (int i = a_list.size() - 1; i >= 0; i--) {
    a_list.retrieve(i, item);
    (*visit)(item);
  }
}
```

E8. Error_code copy(List &dest, List &source) *copies all entries from* source *into* dest; source *remains unchanged. You may assume that* dest *already exists, but any entries already in* dest *are to be discarded.*

Answer

```
Error_code copy(List &dest, List &source)
/* Post: All entries are copied from from source into dest; source remains unchanged. A code
         of fail is returned if a complete copy cannot be made. */
{
  List_entry item;
  Error_code outcome = success;
  while (!dest.empty()) dest.remove(0, item);
  for (int i = 0; i < source.size(); i++) {
    if (source.retrieve(i, item) != success) outcome = fail;
    if (dest.insert(i, item) != success) outcome = fail;
  }
  return outcome;
}
```

E9. Error_code join(List &list1, List &list2) *copies all entries from* list1 *onto the end of* list2*;* list1 *remains unchanged, as do all the entries previously in* list2.

Answer

```
Error_code join(List &list1, List &list2)
/* Post: All entries from list1 are copied onto the end of list2. A code of overflow is returned if
         list2 is filled up before the copying is complete. */
{
  List_entry item;
  for (int i = 0; i < list1.size(); i++) {
    list1.retrieve(i, item);
    if (list2.insert(list2.size(), item) != success)
      return overflow;
  }
  return success;
}
```

E10. void reverse(List &a_list) *reverses the order of all entries in* a_list.

Answer

```
void reverse(List &a_list)
/* Post: Reverses the order of all entries in a_list. A code of fail is returned in case the reversal
         cannot be completed. */
{
  List temp;
  List_entry item;
  Error_code outcome = success;
  for (int i = 0; i < a_list.size(); i++) {
    a_list.retrieve(i, item);
    if (temp.insert(i, item) != success)
      outcome = fail;
  }
  for (int j = 0; j < a_list.size(); j++) {
    temp.retrieve(j, item);
    a_list.replace(a_list.size() - 1 - j, item);
  }
}
```

E11. Error_code split(List &source, List &oddlist, List &evenlist) *copies all entries from* source *so that those in odd-numbered positions make up* oddlist *and those in even-numbered positions make up* evenlist. *You may assume that* oddlist *and* evenlist *already exist, but any entries they may contain are to be discarded.*

Answer

```
Error_code split(List &source, List &oddlist, List &evenlist)
/* Post: Copies all entries from source so that those in odd-numbered positions make up oddlist
         and those in even-numbered positions make up evenlist. Returns an error code of
         overflow in case either output list fills before the copy is complete. */
{
   List_entry item;
   Error_code outcome = success;
   for (int i = 0; i < source.size(); i++) {
      source.retrieve(i, item);
      if (i % 2 != 0) {
         if (oddlist.insert(oddlist.size(), item) == overflow)
            outcome = overflow;
      }
      else
         if (evenlist.insert(evenlist.size(), item) == overflow)
            outcome = overflow;
   }
   return outcome;
}
```

6.2 IMPLEMENTATION OF LISTS

Exercises 6.2

E1. *Write C++ functions to implement the remaining operations for the contiguous implementation of a* List, *as follows:*

(a) *The constructor* List

Answer

```
template <class List_entry>
List<List_entry> :: List()
/* Post: The List is initialized to be empty. */
{
   count = 0;
}
```

(b) clear

Answer

```
//    clear: clear the List.
/* Post: The List is cleared. */
template <class List_entry>
void List<List_entry> :: clear()
{
   count = 0;
}
```

(c) empty

Answer

```
//    empty: returns non-zero if the List is empty.
/* Post: The function returns true or false according as the List is empty or not. */
template <class List_entry>
bool List<List_entry> :: empty() const
{
   return count <= 0;
}
```

E3. *Write implementations for the remaining* String *processing functions.*

(a) `void strcpy(String &copy, const String &original);`

Answer

```
void strcpy(String &s, String &t)
/* Post: Copies the value of String t to String s. */
{
  s = t;
}
```

(b) `void strncpy(String &copy, const String &original, int n);`

Answer

```
void strncpy(String &s, const String &t, unsigned len)
/* Post: Copies the first len characters of String t to make String s. */
{
  const char *temp = t.c_str();
  char *copy = new char[len + 1];
  strncpy(copy, temp, len);
  copy[len] = 0;
  s = copy;
  delete [ ]copy;
}
```

(c) `int strstr(const String &text, const String &target);`

Answer

```
int strstr(String s, String t)
/* Post: Returns the index of the first copy of String t as a substring of String s. Else: Return
         -1. */
{
  int answer;
  const char * content_s = s.c_str();
  char *p = strstr((char *) content_s, t.c_str());
  if (p == NULL) answer = -1;
  else answer = p - content_s;
  return answer;
}
```

E4. *A* ***palindrome*** *is a string that reads the same forward as backward; that is, a string in which the first character equals the last, the second equals the next to last, and so on. Examples of palindromes include* 'radar' *and*

palindromes

'ABLE WAS I ERE I SAW ELBA'.

Write a C++ function to test whether a String *object passed as a reference parameter represents a palindrome.*

Answer

```
bool palindrome(String &to_test)
/* Post: Returns true if to_test represents a palindrome. */
{
  const char *content = to_test.c_str();
  int l = strlen(content);
  for (int i = 0; i < l/2; i++)
    if (content[i] != content[l - 1 - i]) return false;
  return true;
}
```