Chapter 6

LISTS AND STRINGS

1. List Specifications
2. List Implementations
   (a) Class Templates
   (b) Contiguous
   (c) Simply Linked
   (d) Simply Linked with Position Pointer
   (e) Doubly Linked
3. Strings
4. Application: Text Editor
5. Linked Lists in Arrays
6. Application: Generating Permutations

List Definition

DEFINITION A ***list*** of elements of type T is a finite sequence of elements of T together with the following operations:

1. ***Construct*** the list, leaving it empty.
2. Determine whether the list is ***empty*** or not.
3. Determine whether the list is ***full*** or not.
4. Find the ***size*** of the list.
5. ***Clear*** the list to make it empty.
6. ***Insert*** an entry at a specified position of the list.
7. ***Remove*** an entry from a specified position in the list.
8. ***Retrieve*** the entry from a specified position in the list.
9. ***Replace*** the entry at a specified position in the list.
10. ***Traverse*** the list, performing a given operation on each entry.

Comparison with Standard Template Library:

- The STL list provides only those operations that can be implemented efficiently in a List implementation known as doubly linked, which we shall study shortly.
- The STL list does not allow random access to an arbitrary list position.
- The STL vector, does provide some random access to a sequence of data values, but not all the other capabilities we shall develop for a List.
- In this way, our study of the List ADT provides an introduction to both the STL classes list and vector.

Method Specifications

```
List :: List( );
```

Post: The List has been created and is initialized to be empty.

```
void List :: clear( );
```

Post: All List entries have been removed; the List is empty.

```
bool List :: empty( ) const;
```

Post: The function returns **true** or **false** according to whether the List is empty or not.

```
bool List :: full( ) const;
```

Post: The function returns **true** or **false** according to whether the List is full or not.

```
int List :: size( ) const;
```

Post: The function returns the number of entries in the List.

Position Number in a List

- To find an entry in a list, we use an integer that gives its *position* within the list.

- We shall number the positions in a list so that the first entry in the list has position 0, the second position 1, and so on.

- Locating an entry of a list by its position is superficially like indexing an array, but there are important differences. If we insert an entry at a particular position, then the position numbers of all later entries increase by 1. If we remove an entry, then the positions of all following entries decrease by 1.

- The position number for a list is defined without regard to the implementation. For a contiguous list, implemented in an array, the position will indeed be the index of the entry within the array. But we will also use the position to find an entry within linked implementations of a list, where no indices or arrays are used at all.

```
Error_code List::insert(int position, const List_entry &x);
```

Post: If the List is not full and $0 \leq$ position $\leq n$, where n is the number of entries in the List, the function succeeds: Any entry formerly at position and all later entries have their position numbers increased by 1, and x is inserted at position in the List.
Else: The function fails with a diagnostic error code.

Error_code List :: remove(**int** position, List_entry &x);

Post: If $0 \leq$ position $< n$, where n is the number of entries in the List, the function succeeds: The entry at position is removed from the List, and all later entries have their position numbers decreased by 1. The parameter x records a copy of the entry formerly at position.
Else: The function fails with a diagnostic error code.

Error_code List :: retrieve(**int** position, List_entry &x) **const;**

Post: If $0 \leq$ position $< n$, where n is the number of entries in the List, the function succeeds: The entry at position is copied to x; all List entries remain unchanged.
Else: The function fails with a diagnostic error code.

Error_code List :: replace(**int** position, **const** List_entry &x);

Post: If $0 \leq$ position $< n$, where n is the number of entries in the List, the function succeeds: The entry at position is replaced by x; all other entries remain unchanged.
Else: The function fails with a diagnostic error code.

void List :: traverse(**void** (*visit)(List_entry &));

Post: The action specified by function *visit has been performed on every entry of the List, beginning at position 0 and doing each in turn.

Class Templates

- A C++ ***template*** construction allows us to write code, usually code to implement a class, that uses objects of an arbitrary, generic type.

- In template code we utilize a parameter enclosed in angle brackets < > to denote the generic type.

- Later, when a client uses our code, the client can substitute an actual type for the template parameter. The client can thus obtain several actual pieces of code from our template, using different actual types in place of the template parameter.

- Example: We shall implement a **template class** List that depends on one generic type parameter. A client can then use our template to declare several lists with different types of entries with declarations of the following form:

```
List<int> first_list;
List<char> second_list;
```

- Templates provide a new mechanism for creating generic data structures, one that allows many different specializations of a given data structure template in a single application.

- The added generality that we get by using templates comes at the price of slightly more complicated class specifications and implementations.

Contiguous Implementation

```
template <class List_entry>
class List {
public:
//    methods of the List ADT
   List( );
   int size( ) const;
   bool full( ) const;
   bool empty( ) const;
   void clear( );
   void traverse(void (*visit)(List_entry &));
   Error_code retrieve(int position, List_entry &x) const;
   Error_code replace(int position, const List_entry &x);
   Error_code remove(int position, List_entry &x);
   Error_code insert(int position, const List_entry &x);

protected:
//    data members for a contiguous list implementation
   int count;
   List_entry entry[max_list];
};
```

List Size

```
template <class List_entry>
int List<List_entry> :: size( ) const
/* Post:  The function returns the number of entries in the List.  */
{
   return count;
}
```

Insertion

```
template <class List_entry>
Error_code List<List_entry> :: insert(int position, const List_entry &x)
/* Post: If the List is not full and 0 ≤ position ≤ n, where n is the number of
         entries in the List, the function succeeds: Any entry formerly at position and
         all later entries have their position numbers increased by 1 and x is inserted at
         position of the List.
         Else: The function fails with a diagnostic error code.  */

{
   if (full( ))
      return overflow;

   if (position < 0 || position > count)
      return range_error;

   for (int i = count − 1; i >= position; i−−)
      entry[i + 1] = entry[i];

   entry[position] = x;
   count++;
   return success;
}
```

Traversal

```
template <class List_entry>
void List<List_entry> :: traverse(void (*visit)(List_entry &))
/* Post:  The action specified by function (*visit) has been performed on every entry of
          the List, beginning at position 0 and doing each in turn.  */
{
   for (int i = 0;  i < count;  i++)
      (*visit)(entry[i]);
}
```

Performance of Methods

In processing a contiguous list with n entries:

- insert and remove operate in time approximately proportional to n.
- List, clear, empty, full, size, replace, and retrieve operate in constant time.

Simply Linked Implementation

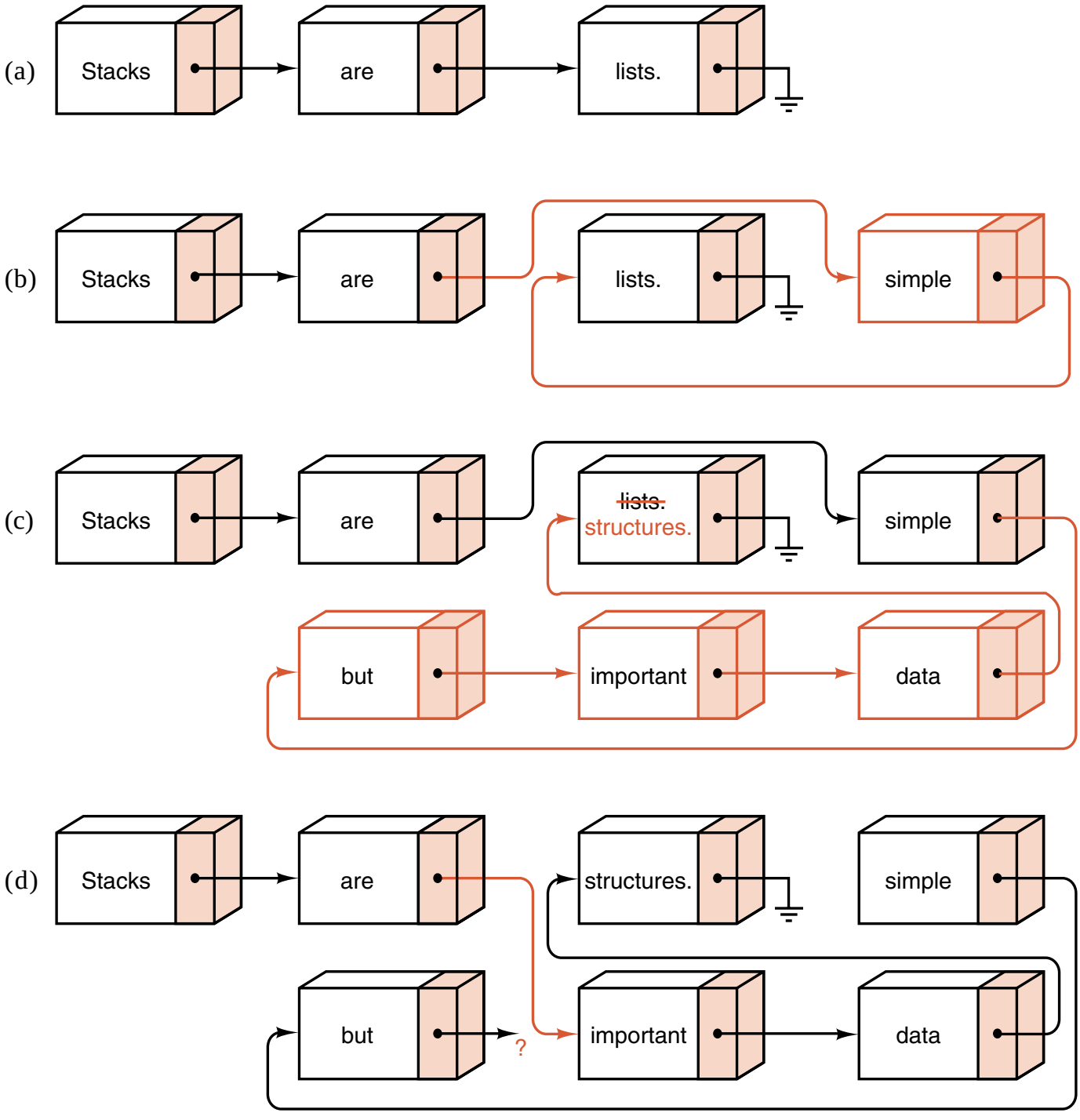
Node declaration:

```
template <class Node_entry>
struct Node {
//    data members
   Node_entry entry;
   Node<Node_entry> *next;
//    constructors
   Node( );
   Node(Node_entry, Node<Node_entry> *link = NULL);
};
```

List declaration:

```
template <class List_entry>
class List {
public:
//    Specifications for the methods of the list ADT go here.

//    The following methods replace compiler-generated defaults.
    ~List( );
   List(const List<List_entry> &copy);
   void operator = (const List<List_entry> &copy);
protected:
//    Data members for the linked list implementation now follow.
   int count;
   Node<List_entry> *head;

//    The following auxiliary function is used to locate list positions
   Node<List_entry> *set_position(int position) const;
};
```

Actions on a Linked List



Finding a List Position

- Function set_position takes an integer parameter position and returns a *pointer* to the corresponding node of the list.
- Declare the visibility of set_position as **protected**, since set_position returns a pointer to, and therefore gives access to, a Node in the List. To maintain an encapsulated data structure, we must restrict the visibility of set_position. Protected visibility ensures that it is only available as a tool for constructing other methods of the List.
- To construct set_position, we start at the beginning of the List and traverse it until we reach the desired node:

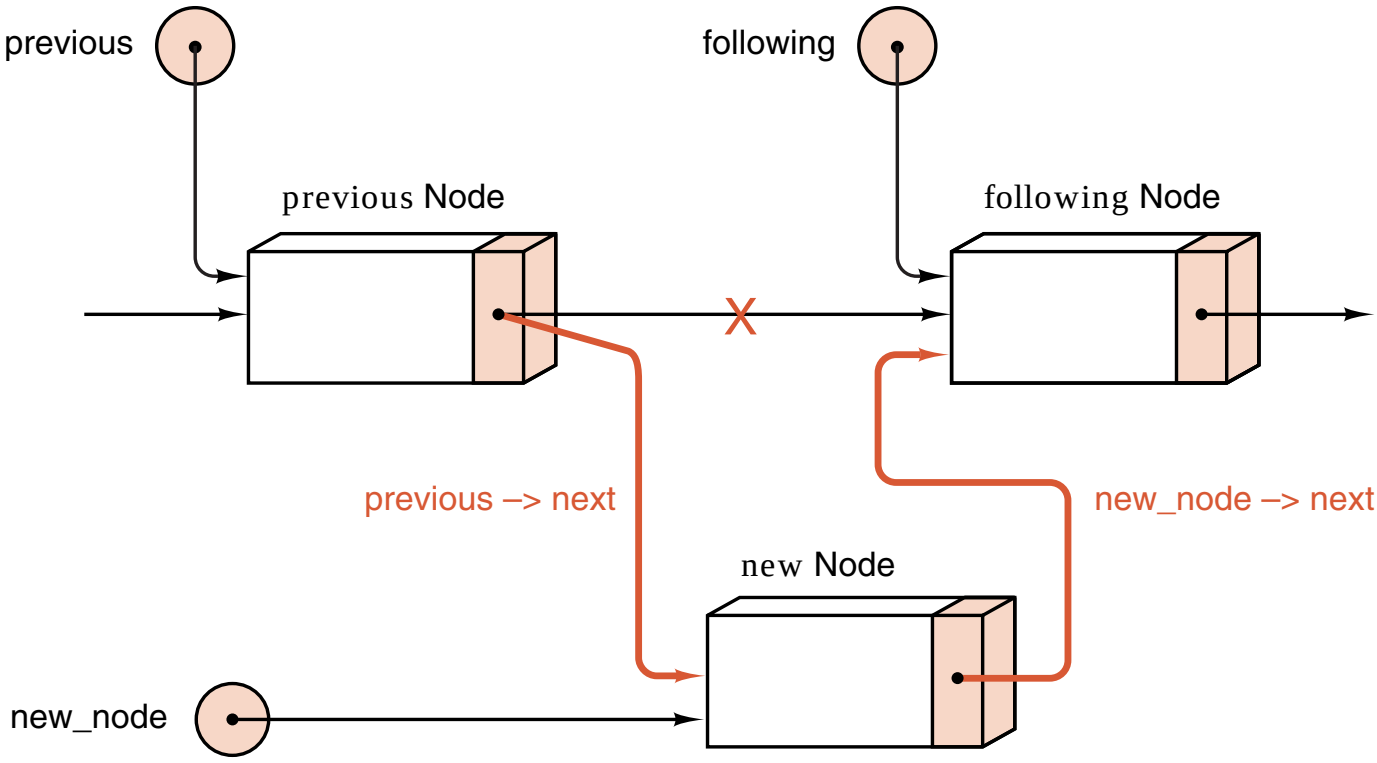
```
template <class List_entry>
Node<List_entry> *List<List_entry> :: set_position(int position) const
/* Pre:   position is a valid position in the List; 0 ≤ position < count.
   Post:  Returns a pointer to the Node in position.  */
{
   Node<List_entry> *q = head;
   for (int i = 0;  i < position;  i++) q = q->next;
   return q;
}
```

- If all nodes are equally likely, then, on average, the set_position function must move halfway through the List to find a given position. Hence, on average, its time requirement is approximately proportional to n, the size of the List.

Insertion Method

```
template <class List_entry>
Error_code List<List_entry> :: insert(int position, const List_entry &x)
/* Post: If the List is not full and 0 ≤ position ≤ n, where n is the number of
         entries in the List, the function succeeds: Any entry formerly at position and
         all later entries have their position numbers increased by 1, and x is inserted at
         position of the List.
         Else: The function fails with a diagnostic error code. */
{
   if (position < 0 || position > count)
      return range_error;
   Node<List_entry> *new_node, *previous, *following;
   if (position > 0) {
      previous = set_position(position − 1);
      following = previous->next;
   }
   else following = head;
   new_node = new Node<List_entry>(x, following);
   if (new_node == NULL)
      return overflow;
   if (position == 0)
      head = new_node;
   else
      previous->next = new_node;
   count++;
   return success;
}
```

Insertion into a Linked List



In processing a linked List with n entries:

- clear, insert, remove, retrieve, and replace require time approximately proportional to n.
- List, empty, full, and size operate in constant time.

Variation: Keeping the Current Position

- Suppose an application processes list entries in order or refers to the same entry several times before processing another entry.
- *Remember* the last-used position in the list and, if the next operation refers to the same or a later position, start tracing through the list from this last-used position.
- Note that this will not speed up *every* application using lists.
- The method retrieve is defined as **const**, but its implementation will need to alter the last-used position of a List. To enable this, we use the **mutable** qualifier. ***Mutable*** data members of a **class** can be changed, even by constant methods.

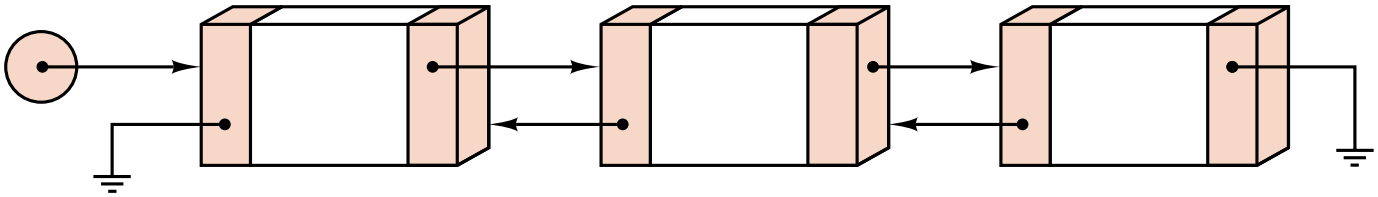
```
template <class List_entry>
class List {
public:
//  Add specifications for the methods of the list ADT.
//  Add methods to replace the compiler-generated defaults.

protected:
//  Data members for the linked-list implementation with
//  current position follow:
  int count;
  mutable int current_position;
  Node<List_entry> *head;
  mutable Node<List_entry> *current;

//  Auxiliary function to locate list positions follows:
  void set_position(int position) const;
};
```

- All the new class members have protected visibility, so, from the perspective of a client, the class looks exactly like the earlier implementation.
- The current position is now a member of the **class** List, so there is no longer a need for set_position to return a pointer; instead, the function simply resets the pointer current directly within the List.

```
template <class List_entry>
void List<List_entry> :: set_position(int position) const
/* Pre:   position is a valid position in the List: 0 ≤ position < count.
   Post:  The current Node pointer references the Node at position.  */
{
  if (position < current_position) {  //     must start over at head of list
    current_position = 0;
    current = head;
  }
  for (; current_position != position; current_position++)
    current = current->next;
}
```

- For repeated references to the same position, neither the body of the **if** statement nor the body of the **for** statement will be executed, and hence the function will take almost no time.
- If we move forward only one position, the body of the **for** statement will be executed only once, so again the function will be very fast.
- If it is necessary to move backwards through the List, then the function operates in almost the same way as the version of set_position used in the previous implementation.

Doubly Linked Lists



Node definition:

```
template <class Node_entry>
struct Node {
//    data members
   Node_entry entry;
   Node<Node_entry> *next;
   Node<Node_entry> *back;
//    constructors
   Node( );
   Node(Node_entry, Node<Node_entry> *link_back = NULL,
                    Node<Node_entry> *link_next = NULL);
};
```

List definition:

```
template <class List_entry>
class List {
public:
//    Add specifications for methods of the list ADT.
//    Add methods to replace compiler generated defaults.

protected:
//    Data members for the doubly-linked list implementation follow:
   int count;
   mutable int current_position;
   mutable Node<List_entry> *current;

//    The auxiliary function to locate list positions follows:
   void set_position(int position) const;
};
```

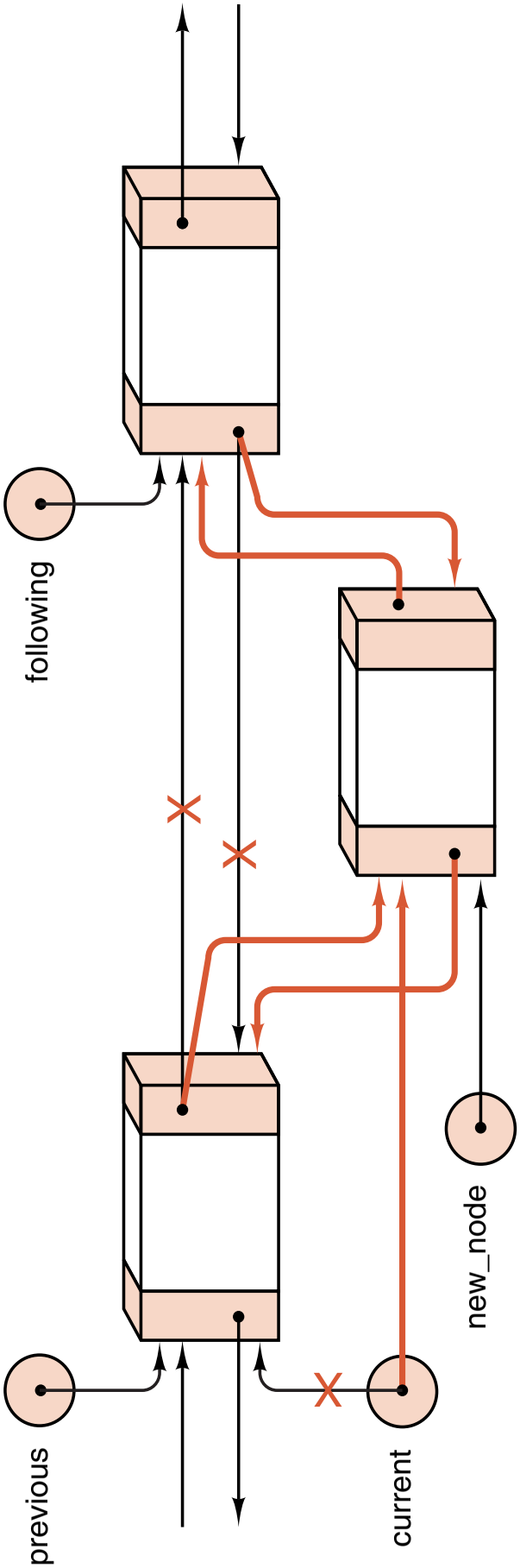
- We can move either direction through the List while keeping only one pointer, current, into the List.

- We do not need pointers to the head or the tail of the List, since they can be found by tracing back or forth from any given node.

- To find any position in the doubly linked list, we first decide whether to move forward or backward from the current position,and then we do a partial traversal of the list until we reach the desired position.

```
template <class List_entry>
void List<List_entry> :: set_position(int position) const
/* Pre:   position is a valid position in the List: 0 ≤ position < count.
   Post:  The current Node pointer references the Node at position.  */
{
   if (current_position <= position)
      for ( ;  current_position != position;  current_position++)
         current = current->next;
   else
      for ( ;  current_position != position;  current_position--)
         current = current->back;
}
```

- The cost of a doubly linked list is the extra space required in each Node for a second link, usually trivial in comparison to the space for the information member entry.



Insertion into a Doubly Linked List

```
template <class List_entry>
Error_code List<List_entry> :: insert(int position, const List_entry &x)
/* Post: If the List is not full and 0 ≤ position ≤ n, where n is the number of
         entries in the List, the function succeeds: Any entry formerly at position and
         all later entries have their position numbers increased by 1 and x is inserted at
         position of the List.
         Else: the function fails with a diagnostic error code. */

{
   Node<List_entry> *new_node, *following, *preceding;

   if (position < 0 || position > count) return range_error;

   if (position ==  0) {
      if (count ==  0) following = NULL;
      else {
         set_position(0);
         following = current;
      }
      preceding = NULL;
   }

   else {
      set_position(position − 1);
      preceding = current;
      following = preceding->next;
   }
   new_node = new Node<List_entry>(x, preceding, following);
```

```
    if (new_node ==  NULL) return overflow;
    if (preceding != NULL) preceding->next = new_node;
    if (following != NULL) following->back = new_node;
    current = new_node;
    current_position = position;
    count++;
    return success;
}
```

Comparison of Implementations

Contiguous storage is generally preferable

- when the entries are individually very small;
- when the size of the list is known when the program is written;
- when few insertions or deletions need to be made except at the end of the list; and
- when random access is important.

Linked storage proves superior

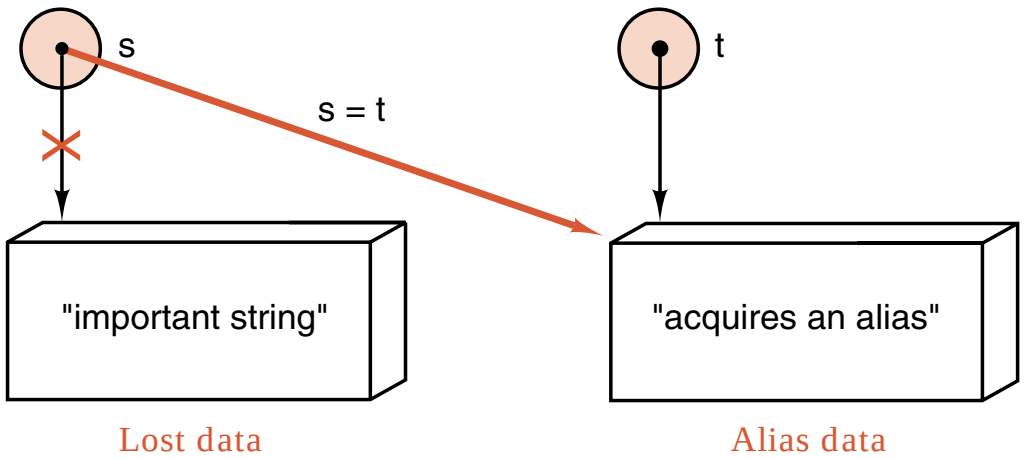
- when the entries are large;
- when the size of the list is not known in advance; and
- when flexibility is needed in inserting, deleting, and rearranging the entries.

To choose among linked list implementations, consider:

- Which of the operations will actually be performed on the list and which of these are the most important?
- Is there *locality* of reference? That is, if one entry is accessed, is it likely that it will next be accessed again?
- Are the entries processed in sequential order? If so, then it may be worthwhile to maintain the last-used position as part of the list structure.
- Is it necessary to move both directions through the list? If so, then doubly linked lists may prove advantageous.

Strings in C

- A ***string*** is defined as a sequence of characters.
- Examples: "This is a string" or "Name?", where the double quotes are not part of the string. The ***empty string*** is "".
- A string ADT is a kind of list, but the operations are usually quite different from other lists.
- The first implementation of strings is found in the C subset of C++. We call these ***C-strings***. C-strings reflect the strengths and weaknesses of the C language:
  - C-strings are widely available.
  - C-strings are very efficient.
  - C-string objects are not encapsulated.
  - C-strings are easy to misuse, with consequences that can be disastrous.
  - It is easy for a client to create either garbage or aliases for C-string data. For example:



C-String Definitions

- Every C-string has type **char** *. Hence, a C-string references an address in memory, the first of a contiguous set of bytes that store the characters making up the string.

- The storage occupied by the string must terminate with the special character value '\0'.

- The standard header file <cstring> (or <string.h>) contains a library of functions that manipulate C-strings.

- In C++, the output operator ≪ is overloaded to apply to C-strings, so that a simple instruction cout ≪ s prints the string s.

- In C++, it is easy to use encapsulation to embed C-strings into safer class-based implementations of strings.

- The standard template library includes a safe string implementation in the header file <string>. This library implements a class called std::string that is convenient, safe, and efficient.

Standard C-String Library

char *strcpy(**char** *to, **char** *from);

Pre: The string from has been initialized.

Post: The function copies string from to string to, including '\0'; it returns a pointer to the beginning of the string to.

char *strncpy(**char** *to, **char** *from, **int** n);

Pre: The string from has been initialized.

Post: The function copies at most n characters from string from to string to; it returns a pointer to the beginning of the string to. If from has less than n characters, the remaining positions are padded with '\0's.

char *strcat(**char** *to, **char** *from);

Pre: The strings from and to have been initialized.

Post: The function copies string from to the end of string to, including '\0'; it returns a pointer to the beginning of the string to.

char *strncat(**char** *to, **char** *from, **int** n);

Pre: The strings from and to have been initialized.

Post: The function copies at most n characters from string from to the end of string to, and terminates to with '\0'; it returns a pointer to the beginning of the string to.

size_t strlen(**char** *s);

Pre: The string s has been initialized.

Post: The function returns the length of the string s, not including the null byte '\0' at the end of the string s.

int strcmp(**char** *s1, **char** *s2);

Pre: The strings s1 and s2 have been initialized.

Post: The function compares string s1 to string s2; it returns < 0 if s1 < s2, 0 if s1 == s2, or > 0 if s1 > s2.

int strncmp(**char** *s1, **char** *s2);

Pre: The strings s1 and s2 have been initialized.

Post: The function compares at most n characters of string s1 to string s2; it returns < 0 if s1 < s2, 0 if s1 == s2, or > 0 if s1 > s2.

char *strchr(**char** *s, **char** c);

Pre: The string s has been initialized.

Post: The function returns a pointer to the first occurrence of the character c in the string s, or it returns NULL if c is not present in s.

char *strrchr(**char** *s, **char** c);

Pre: The string s has been initialized.

Post: The function returns a pointer to the last occurrence of the character c in the string s, or it returns NULL if c is not present in s.

size_t strspn(**char** *s1, **char** *s2);

Pre: The strings s1 and s2 have been initialized.

Post: The function returns the length of the prefix of s1 that consists of characters that appear in s2. The type size_t is defined in the standard header file stdlib.h and it usually means **unsigned int**

size_t strcspn(**char** *s1, **char** *s2);

Pre: The strings s1 and s2 have been initialized.

Post: The function returns the length of the prefix of s1 that consists of characters that do not appear in s2. The type size_t is defined in the standard header file stdlib.h and it usually means **unsigned int**.

char *strpbrk(**char** *s1, **char** *s2);

Pre: The strings s1 and s2 have been initialized.

Post: The function returns a pointer to the first occurrence in the string s1 of any character of the string s2, or it returns NULL if no character of s2 appears in s1.

char *strstr(**char** *s1, **char** *s2);

Pre: The strings s1 and s2 have been initialized.

Post: The function returns a pointer to the first occurrence of the string s2 in the string s1, or it returns NULL if the string s2 is not present in s1.

Safe Implementation of Strings

To create a safer string implementation, we embed the C-string representation as a member of a **class** String. Features:

- Include the string length as a data member in the String class.
- The String class avoids the problems of aliases, garbage creation, and uninitialized objects by including an overloaded assignment operator, a copy constructor, a destructor, and a constructor.
- Include overloaded versions of the Boolean comparison operators <, >, <= , >= , == , != .
- Include a constructor that uses a parameter of type **char** * and translates from C-string objects to String objects.
- Include a constructor to convert from a List of characters to a String.
- Include a String method c_str( ) that converts String objects to corresponding C-string objects.
- The resulting String class is a fully encapsulated ADT, but it provides a complete interface both to C-strings and to lists of characters.

String Class Specification

```
class String {
public:                                  //    methods of the string ADT
   String( );
    ~String( );
   String (const String &copy);          //    copy constructor
   String (const char * copy);           //    conversion from C-string
   String (List<char> &copy);            //    conversion from List

   void operator = (const String &copy);
   const char *c_str( ) const;           //    conversion to C-style string

protected:
   char *entries;
   int length;
};



bool operator ==  (const String &first, const String &second);
bool operator >   (const String &first, const String &second);
bool operator <   (const String &first, const String &second);
bool operator >=  (const String &first, const String &second);
bool operator <=  (const String &first, const String &second);
bool operator !=  (const String &first, const String &second);
```

String Constructors

```
String :: String (const char *in_string)
/* Pre:   The pointer in_string references a C-string.
   Post:  The String is initialized by the C-string in_string.  */
{
   length = strlen(in_string);
   entries = new char[length + 1];
   strcpy(entries, in_string);
}

String :: String (List<char> &in_list)
/* Post:  The String is initialized by the character List in_list.  */
{
   length = in_list.size( );
   entries = new char[length + 1];
   for (int i = 0;  i < length;  i++) in_list.retrieve(i, entries[i]);
   entries[length] = '\0';
}

const char*String :: c_str( ) const
/* Post:  A pointer to a legal C-string object matching the String is returned.  */
{
   return (const char *) entries;
}

bool operator ==  (const String &first, const String &second)
/* Post:  Return true if the String first agrees with String second.  Else:  Return
          false.  */
{
   return strcmp(first.c_str( ), second.c_str( )) ==  0;
}
```

Samples of Further String Operations

```
void strcat(String &add_to, const String &add_on)
/* Post: The function concatenates String add_on onto the end of String add_to.
         */
{
   const char *cfirst = add_to.c_str( );
   const char *csecond = add_on.c_str( );
   char *copy = new char[strlen(cfirst) + strlen(csecond) + 1];
   strcpy(copy, cfirst);
   strcat(copy, csecond);
   add_to = copy;
   delete [ ]copy;
}
String read_in(istream &input)
/* Post: Return a String read (as characters terminated by a newline or an end-of-file
         character) from an istream parameter. */
{
   List<char> temp;
   int size = 0;
   char c;
   while ((c = input.peek( )) != EOF && (c = input.get( )) != '\n')
      temp.insert(size++, c);
   String answer(temp);
   return answer;
}
```

Overloaded version:

```
String read_in(istream &input, int &terminator);
```

Post: Return a String read (as characters terminated by a newline or an end-of-file character) from an istream parameter. The terminating character is recorded as the output parameter terminator.

```
void write(String &s)
/* Post:  The String parameter s is written to cout.  */
{
   cout << s.c_str( ) << endl;
}
```

void strcpy(String &copy, **const** String &original);

Post: The function copies String original to String copy.

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

Post: The function copies at most n characters from String original to String copy.

int strstr(**const** String &text, **const** String &target);

Post: If String target is a substring of String text, the function returns the array index of the first occurrence of the string stored in target in the string stored in text. Else: The function returns a code of -1.

Text Editor Operations

'R' Read the text file (name in command line) into the buffer. Any previous contents of the buffer are lost. At the conclusion, the current line will be the first line of the file.

'W' Write the contents of the buffer to an output file. Neither the current line nor the buffer is changed.

'I' Insert a single new line typed in by the user at the current line number. The prompt 'I:' requests the new line.

'D' Delete the current line and move to the next line.

'F' Find the first line, starting with the current line, that contains a target string that will be requested from the user.

'L' Show the length in characters of the current line and the length in lines of the buffer.

'C' Change the string requested from the user to a replacement text, also requested from the user, working within the current line only.

'Q' Quit the editor; terminates immediately.

'H' Print out help messages explaining all the commands. The program will accept '?' as an alternative to 'H'.

'N' Next line: advance one line through the buffer.

'P' Previous line: back up one line in the buffer.

'B' Beginning: go to the first line of the buffer.

'E' End: go to the last line of the buffer.

'G' Go to a user-specified line number in the buffer.

'S' Substitute a line typed in by the user for the current line. The function should print out the line for verification and then request the new line.

'V' View the entire contents of the buffer, printed out to the terminal.

The Main Program

```
int main(int argc, char *argv[ ])  //    count, values of command-line arguments
/* Pre:   Names of input and output files are given as command-line arguments.
   Post:  Reads an input file that contains lines (character strings), performs simple
          editing operations on the lines, and writes the edited version to the output file.
   Uses:  methods of class Editor  */
{
   if (argc != 3) {
      cout << "Usage:\n\t edit  inputfile  outputfile" << endl;
      exit (1);
   }

   ifstream file_in(argv[1]);        //    Declare and open the input stream.
   if (file_in ==  0) {
      cout << "Can't open input file " << argv[1] << endl;
      exit (1);
   }

   ofstream file_out(argv[2]);     //    Declare and open the output stream.
   if (file_out ==  0) {
      cout << "Can't open output file " << argv[2] << endl;
      exit (1);
   }

   Editor buffer( &file_in,  &file_out);
   while (buffer.get_command( ))
      buffer.run_command( );
}
```

The Editor Class Specification

```
class Editor: public List<String> {
public:
   Editor(ifstream *file_in, ofstream *file_out);
   bool get_command( );
   void run_command( );
private:
   ifstream *infile;
   ofstream *outfile;
   char user_command;

//    auxiliary functions
   Error_code next_line( );
   Error_code previous_line( );
   Error_code goto_line( );
   Error_code insert_line( );
   Error_code substitute_line( );
   Error_code change_line( );
   void read_file( );
   void write_file( );
   void find_string( );
};
```

Constructor

```
Editor :: Editor(ifstream *file_in, ofstream *file_out)
/* Post:  Initialize the Editor members infile and outfile with the parameters.  */
{
   infile = file_in;
   outfile = file_out;
}
```

Receiving a Command

```
bool Editor :: get_command( )
/* Post:  Sets member user_command; returns true unless the user's command is q.
   Uses:  C library function tolower.  */
{
   if (current != NULL)
      cout  << current_position << " : "
            << current->entry.c_str( ) << "\n??" << flush;
   else
      cout  << "File is empty.\n??" << flush;

   cin >> user_command;              //    ignores white space and gets command
   user_command = tolower(user_command);
   while (cin.get( ) != '\n')
      ;                              //    ignore user's enter key
   if (user_command == 'q')
      return false;
   else
      return true;
}
```

Performing Commands

```
void Editor :: run_command( )
/* Post:  The command in user_command has been performed.
   Uses:  methods and auxiliary functions of the class Editor, the class String, and the
          String processing functions.  */

{
   String temp_string;
   switch (user_command) {

   case 'b':
      if (empty( ))
         cout  << " Warning: empty buffer " << endl;
      else
         while (previous_line( ) ==  success)
            ;
      break;

   case 'c':
      if (empty( ))
         cout  << " Warning: Empty file" << endl;
      else if (change_line( )  != success)
         cout  << " Error: Substitution failed " << endl;
      break;

   case 'd':
      if (remove(current_position, temp_string)  != success)
         cout  << " Error: Deletion failed " << endl;
      break;
```

```
case 'e':
   if (empty( ))
      cout  << " Warning: empty buffer " << endl;
   else
      while (next_line( ) ==  success)
         ;
   break;

case 'f':
   if (empty( ))  cout << " Warning: Empty file" << endl;
   else find_string( );
   break;

case 'g':
   if (goto_line( )  != success)
      cout << " Warning: No such line" << endl;
   break;

case '?':  case 'h':
   cout  << "Valid commands are: b(egin) c(hange) d(el) e(nd)"
         << endl
         << "f(ind) g(o) h(elp) i(nsert) l(ength) n(ext) p(rior) " << endl
         << "q(uit) r(ead) s(ubstitute) v(iew) w(rite) " << endl;

case 'i':
   if (insert_line( )  != success)
      cout << " Error: Insertion failed " << endl;
   break;

case 'l':
   cout << "There are " << size( ) << " lines in the file." << endl;
   if (!empty( ))
      cout  << "Current line length is "
            << strlen((current->entry).c_str( )) << endl;
   break;
```

```
    case 'n':
      if (next_line( ) != success)
        cout << " Warning: at end of buffer" << endl;
      break;

    case 'p':
      if (previous_line( ) != success)
        cout << " Warning: at start of buffer" << endl;
      break;

    case 'r':
      read_file( );
      break;

    case 's':
      if (substitute_line( ) != success)
        cout << " Error: Substitution failed " << endl;
      break;

    case 'v':
      traverse(write);
      break;

    case 'w':
      if (empty( ))
        cout << " Warning: Empty file" << endl;
      else
        write_file( );
      break;

    default :
      cout << "Press h or ? for help or enter a valid command: ";
    }
}
```

Reading a File

```
void Editor :: read_file( )
/* Pre:   Either the Editor is empty or the user authorizes the command.
   Post:  The contents of *infile are read to the Editor. Any prior contents of the Editor
          are overwritten.
   Uses:  String and Editor methods and functions.  */


{
   bool proceed = true;
   if ( ! empty( )) {
      cout << "Buffer is not empty; the read will destroy it." << endl;
      cout << " OK to proceed? " << endl;
      if (proceed = user_says_yes( )) clear( );
   }

  int line_number = 0, terminal_char;
  while (proceed) {
      String in_string = read_in(*infile, terminal_char);
      if (terminal_char ==  EOF) {
         proceed = false;
         if (strlen(in_string.c_str( )) > 0) insert(line_number, in_string);
      }
      else insert(line_number++, in_string);
   }
}
```

Inserting a Line

```
Error_code Editor :: insert_line( )
/* Post:  A string entered by the user is inserted as a user-selected line number.
   Uses: String and Editor methods and functions.  */
{
   int line_number;
   cout  << " Insert what line number? " << flush;
   cin    >> line_number;
   while (cin.get( ) != '\n');
   cout  << " What is the new line to insert? " << flush;
   String to_insert = read_in(cin);
   return insert(line_number, to_insert);
}
```

Searching for a String

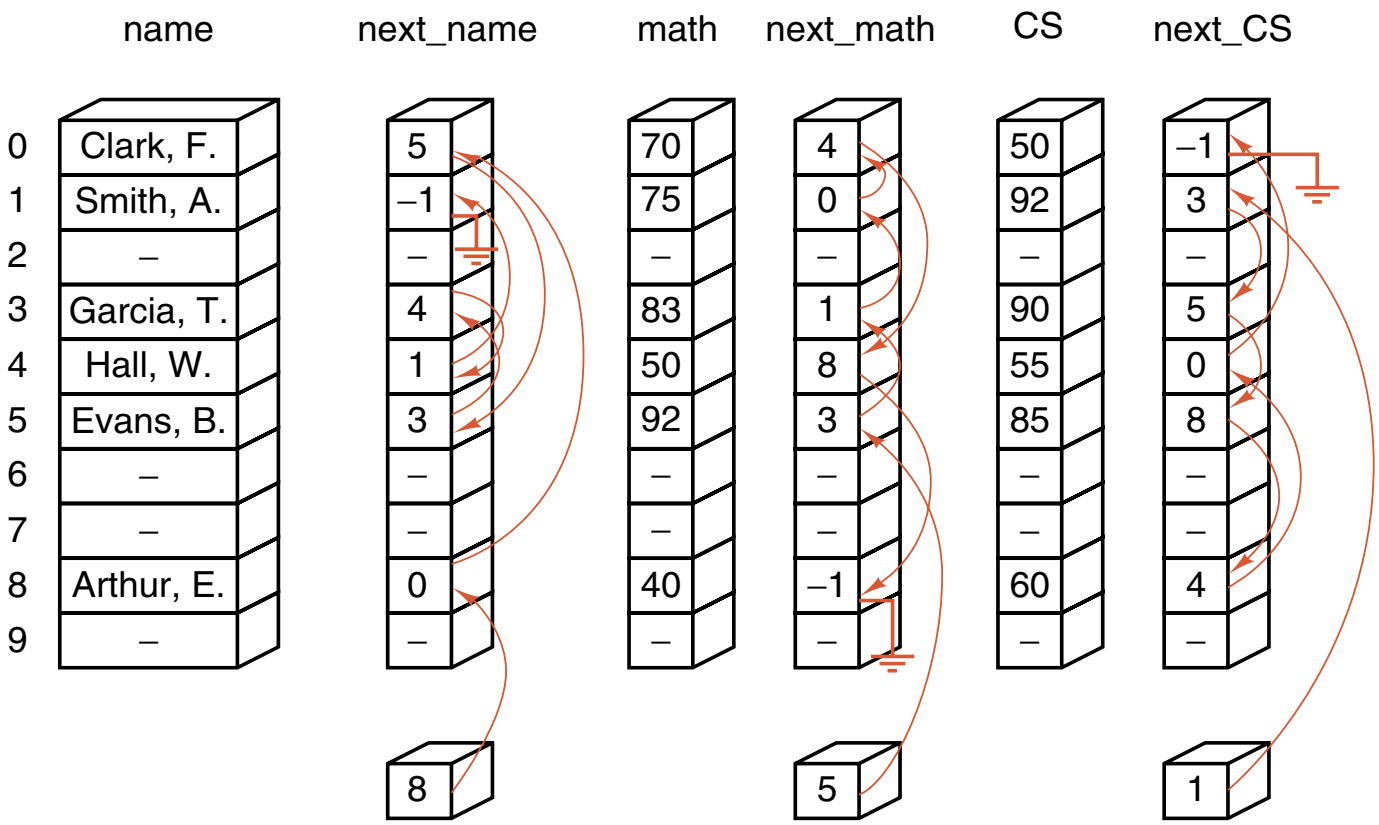
```
void Editor :: find_string( )
/* Pre:   The Editor is not empty.
   Post:  The current line is advanced until either it contains a copy of a user-selected
          string or it reaches the end of the Editor.  If the selected string is found, the
          corresponding line is printed with the string highlighted.
   Uses:  String and Editor methods and functions.  */
{
   int index;


   cout << "Enter string to search for:" << endl;
   String search_string = read_in(cin);
   while ((index = strstr(current->entry, search_string)) ==  -1)
      if (next_line( ) != success) break;
   if (index ==  -1) cout << "String was not found.";
   else {
      cout << (current->entry).c_str( ) << endl;
      for (int i = 0;  i < index;  i++)
         cout << " ";
      for (int j = 0;  j < strlen(search_string.c_str( ));  j++)
         cout << "^";
   }
   cout << endl;
}
```

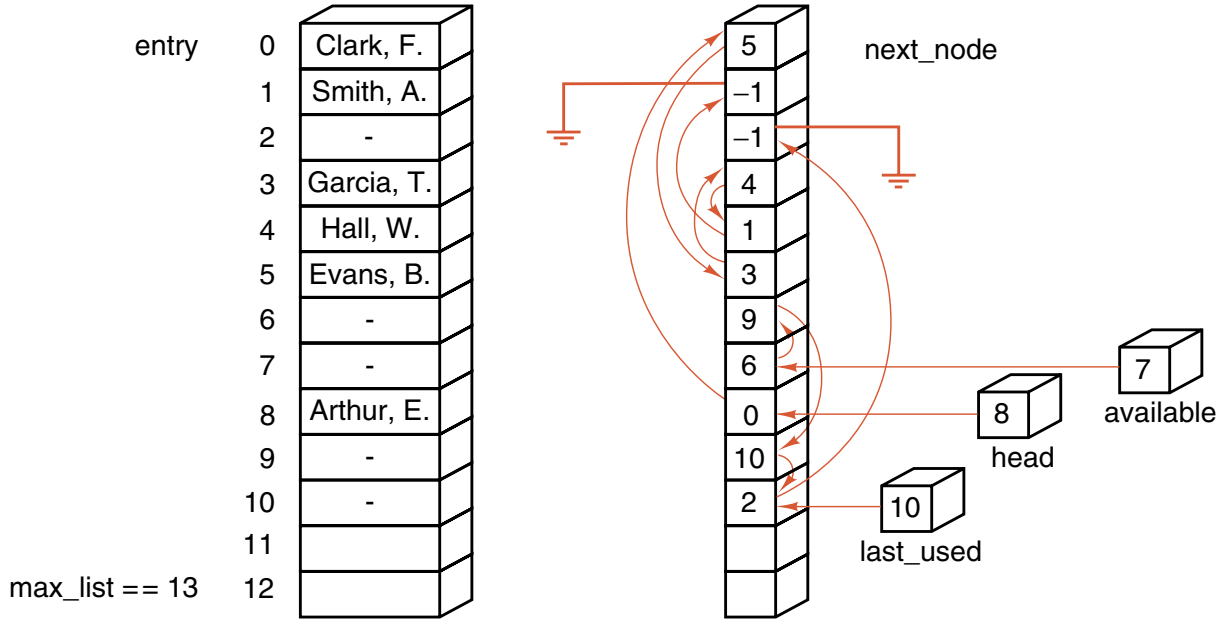
Changing One String to Another

```
Error_code Editor :: change_line( )
/* Pre:    The Editor is not empty.
   Post:   If a user-specified string appears in the current line, it is replaced by a new
           user-selected string. Else: an Error_code is returned.
   Uses:   String and Editor methods and functions.  */
{
   Error_code result = success;
   cout << " What text segment do you want to replace? " << flush;
   String old_text = read_in(cin);
   cout << " What new text segment do you want to add in? " << flush;
   String new_text = read_in(cin);


   int index = strstr(current->entry, old_text);
   if (index ==  -1) result = fail;
   else {
      String new_line;
      strncpy(new_line, current->entry, index);
      strcat(new_line, new_text);
      const char *old_line = (current->entry).c_str( );
      strcat(new_line, (String)(old_line + index + strlen(old_text.c_str( ))));
      current->entry = new_line;
   }
   return result;
}
```

Linked Lists in Arrays

- This section shows how to implement linked lists using only integer variables and arrays.
- Begin with a large workspace array and regard the array as our allocation of unused space.
- Set up our own functions to keep track of which parts of the array are unused and to link entries of the array together in the desired order.
- The one feature of linked lists that we must invariably lose in this implementation method is the dynamic allocation of storage.
- Applications where linked lists in arrays may prove preferable are those where
    - the number of entries in a list is known in advance,
    - the links are frequently rearranged, but relatively few additions or deletions are made, or
    - the same data are sometimes best treated as a linked list and other times as a contiguous list.





Class Declaration, Linked Lists in Arrays

```
template <class List_entry>
class Node {
public:
   List_entry entry;
   index next;
};
template <class List_entry>
class List {
public:
//    Methods of the list ADT
   List( );
   int size( ) const;
   bool full( ) const;
   bool empty( ) const;
   void clear( );
   void traverse(void (*visit)(List_entry &));
   Error_code retrieve(int position, List_entry &x) const;
   Error_code replace(int position, const List_entry &x);
   Error_code remove(int position, List_entry &x);
   Error_code insert(int position, const List_entry &x);
protected:
//    Data members
   Node<List_entry> workspace[max_list];
   index available, last_used, head;
   int count;
//    Auxiliary member functions
   index new_node( );
   void delete_node(index n);
   int current_position(index n) const;
   index set_position(int position) const;
};
```

New and Delete

```
template <class List_entry>
index List<List_entry> :: new_node( )
/* Post:  The index of the first available Node in workspace is returned; the data
          members available, last_used, and workspace are updated as necessary.
          If the workspace is already full, −1 is returned.  */
{
   index new_index;
   if (available != −1) {
      new_index = available;
      available = workspace[available].next;
   } else if (last_used < max_list − 1) {
      new_index = ++last_used;
   } else return −1;
   workspace[new_index].next = −1;
   return new_index;
}
template <class List_entry>
void List<List_entry> :: delete_node(index old_index)
/* Pre:   The List has a Node stored at index old_index.
   Post:  The List index old_index is pushed onto the linked stack of available space;
          available, last_used, and workspace are updated as necessary.  */
{
   index previous;
   if (old_index ==  head) head = workspace[old_index].next;
   else {
      previous = set_position(current_position(old_index) − 1);
      workspace[previous].next = workspace[old_index].next;
   }
   workspace[old_index].next = available;
   available = old_index;
}
```

Other Operations

index List<List_entry> :: set_position(**int** position) **const;**

Pre: position is a valid position in the list; $0 \leq$ position $<$ count.

Post: Returns the index of the node at position in the list.

int List<List_entry> :: current_position(index n) **const;**

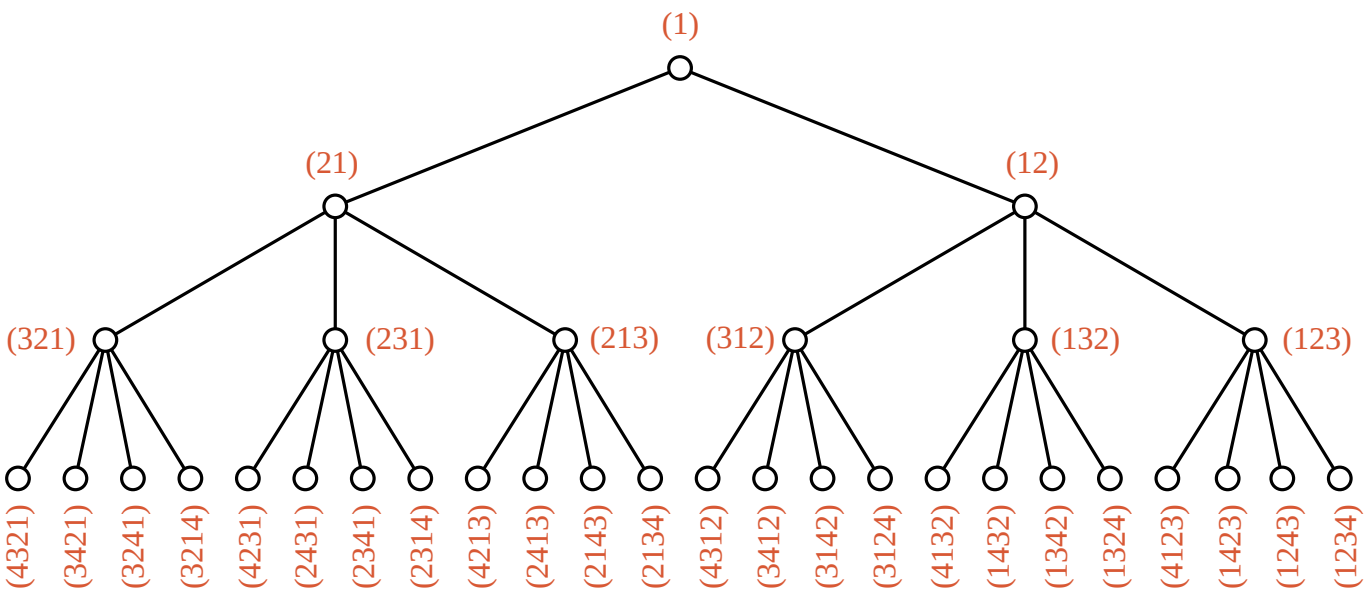
Post: Returns the position number of the node stored at index n, or -1 if there no such node.

```
template <class List_entry>
void List<List_entry> :: traverse(void (*visit)(List_entry &))
/* Post: The action specified by function *visit has been performed on every entry of
         the List, beginning at position 0 and doing each in turn.  */
{
   for (index n = head;  n != -1;  n = workspace[n].next)
      (*visit)(workspace[n].entry);
}
```

Insertion

```
template <class List_entry>
Error_code List<List_entry> :: insert(int position, const List_entry &x)
/* Post: If the List is not full and 0 ≤ position ≤ n, where n is the number of
         entries in the List, the function succeeds: Any entry formerly at position and
         all later entries have their position numbers increased by 1 and x is inserted at
         position of the List.
         Else: the function fails with a diagnostic error code.  */

{
   index new_index, previous, following;

   if (position < 0 || position > count) return range_error;

   if (position > 0) {
      previous = set_position(position − 1);
      following = workspace[previous].next;
   }
   else following = head;
   if ((new_index = new_node( )) ==  −1) return overflow;
   workspace[new_index].entry = x;
   workspace[new_index].next = following;

   if (position ==  0)
      head = new_index;
   else
      workspace[previous].next = new_index;

   count++;
   return success;
}
```

Generating Permutations



Take a given permutation of $\{1, 2, \ldots, k-1\}$ and put its entries into a list. Insert k, in turn, into each of the k possible positions in this list, thereby obtaining k distinct permutations of $\{1, 2, \ldots, k\}$.

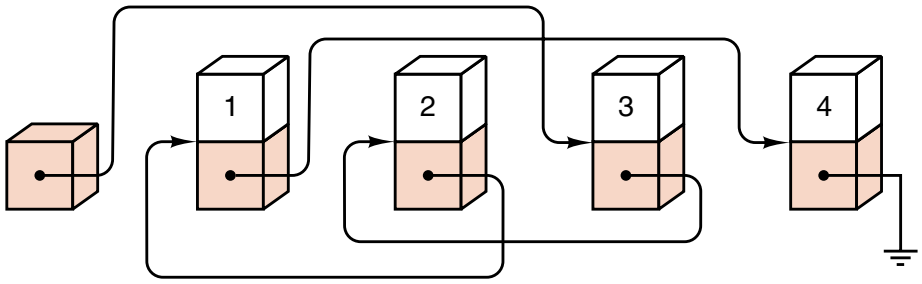
```
void permute(int k, int n)
/* Pre:   1 through k − 1 are already in the permutation list;
   Post:  inserts the integers from k through n into the permutation list */
{
   for                                  //   each of the k possible positions in the list
   {
      //   Insert k into the given position.
      if (k ==  n) process_permutation;
      else permute(k + 1, n);
      //   Remove k from the given position.
   }
}
```

The General Procedure

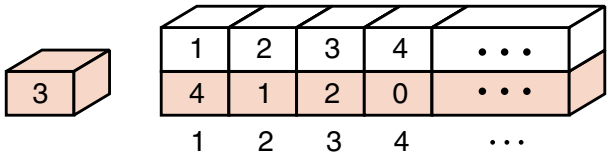
```
void permute(int new_entry, int degree, List<int> &permutation)
/* Pre:   permutation contains a permutation with entries in positions 1 through
          new_entry − 1.
   Post:  All permutations with degree entries, built from the given permutation, have
          been constructed and processed.
   Uses:  permute recursively, process_permutation, and List functions.  */
{
  for (int current = 0;  current < permutation.size( ) + 1;  current++) {
    permutation.insert(current, new_entry);
    if (new_entry ==  degree)
      process_permutation(permutation);
    else
      permute(new_entry + 1, degree, permutation);
    permutation.remove(current, new_entry);
  }
}
```

Representation of permutation (3214):

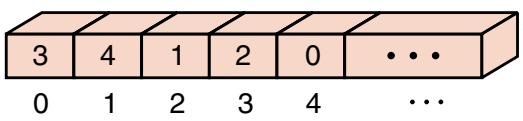
As linked list in order of creation of nodes:



Within an array with separate header:



Within reduced array with artificial first node as header:



Optimized Permute Function

```
void permute(int new_entry, int degree, int *permutation)
/* Pre:   permutation contains a linked permutation with entries in positions 1 through
          new_entry - 1.
   Post:  All permutations with degree entries, built from the given permutation, have
          been constructed and processed.
   Uses:  Functions permute (recursively) and process_permutation.  */

{
   int current = 0;
   do {
      permutation[new_entry] = permutation[current];
      permutation[current] = new_entry;
      if (new_entry ==  degree)
         process_permutation(permutation);
      else
         permute(new_entry + 1, degree, permutation);
      permutation[current] = permutation[new_entry];
      current = permutation[current];
   } while (current != 0);
}
```

Main program:

```
main( )
/* Pre:   The user specifies the degree of permutations to construct.
   Post:  All permutations of a user-supplied degree are printed to the terminal.  */
{
   int degree;
   int permutation[max_degree + 1];

   cout  << "Number of elements to permute? ";
   cin    >> degree;
   if (degree < 1 || degree > max_degree)
      cout  << "Number must be between 1 and " << max_degree
            << endl;
   else {
      permutation[0] = 0;
      permute(1, degree, permutation);
   }
}
```

Processing a permutation:

```
void process_permutation(int *permutation)
/* Pre:   permutation is in linked form.
   Post:  The permutation has been printed to the terminal.  */
{
   int current = 0;
   while (permutation[current]  != 0) {
      cout << permutation[current] << " ";
      current = permutation[current];
   }
   cout << endl;
}
```

Pointers and Pitfalls

1. Use C++ templates to implement generic data structures.

2. Don't confuse contiguous lists with arrays.

3. Choose your data structures as you design your algorithms, and avoid making premature decisions.

4. Always be careful about the extreme cases and handle them gracefully. Trace through your algorithm to determine what happens when a data structure is empty or full.

5. Don't optimize your code until it works perfectly, and then only optimize it if improvement in efficiency is definitely required. First try a simple implementation of your data structures. Change to a more sophisticated implementation only if the simple one proves too inefficient.

6. When working with general lists, first decide exactly what operations are needed, and then choose the implementation that enables those operations to be done most easily.

7. In choosing between linked and contiguous implementations of lists, consider the necessary operations on the lists. Linked lists are more flexible in regard to insertions, deletions, and rearrangement; contiguous lists allow random access.

8. Contiguous lists usually require less computer memory, computer time, and programming effort when the items in the list are small and the algorithms are simple. When the list holds large data entries, linked lists usually save space, time, and often programming effort.

9. Dynamic memory and pointers allow a program to adapt automatically to a wide range of application sizes and provide flexibility in space allocation among different data structures. Static memory (arrays and indices) is sometimes more efficient for applications whose size can be completely specified in advance.

10. For advice on programming with linked lists in dynamic memory, see the guidelines in Chapter 4.

11. Avoid sophistication for sophistication's sake. Use a simple method if it is adequate for your application.

12. Don't reinvent the wheel. If a ready-made class template or function is adequate for your application, consider using it.