**Linked Lists**

The most common and versatile data structures in computer science are linked lists and containers in general.  The Standard Template Library (STL) of C++17 provides various implementations, enabling developers to focus on defining the compound type that is suitable for the programming problem at hand.  The STL takes care of the memory management as well as the access to elements of the lists and containers.

This chapter introduces the Standard Template Library along with the concept of a linked list.  To illustrate the concept this chapter presents two simple examples: a stack and a queue.  A stack is a last in first out list, like a stack of plates.  A queue is a first in first out list, like a lineup at a bus stop.  The following chapter introduces containers in general.

Standard Template Library

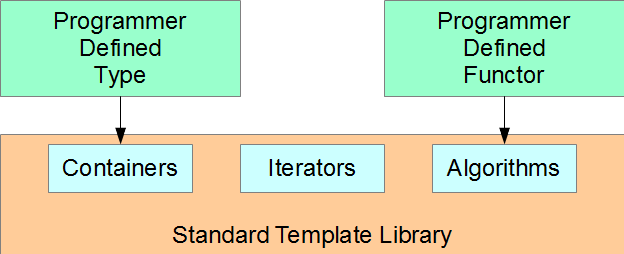
The Standard Template Library (STL) is arguably the most important part of the C++17 Standard Library.  It provides code for managing the elements of a data structure in a generic form, hiding the complex details and allowing re-use.  The STL consists of:

* container template classes
  + sequential containers
  + associative containers
  + container adaptors
* iterators
* algorithms
* function objects

A container class represents the shell of a data structure, manages the memory associated with the elements of that structure and provides member functions to access those elements.  Iterators facilitate the traversal of the data structure and provide simple access to range of elements.  Algorithms implement solutions for sequences of elements through the use of iterators and function objects.

A complete programming solution to the implementation of a data structure requires:

* the definition of the data type of each element in the data structure
* the choice of the optimal data structure to collect the elements
* the function object for the algorithm to use on the data structure
* supervisory coding to accesses the facilities of the STL



The Concept of a Linked List

A linked list is a collection of objects of *identical* type with possibly different lifetimes.  A list implemented this flexibility by allocating dynamic memory separately for each object and linking its objects together through pointers to adjacent objects in the list.  The objects are not necessarily stored contiguously in memory.

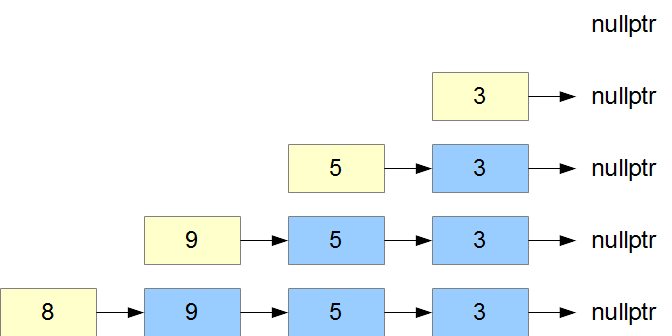
A linked list is significantly more flexible than a statically-allocated array: the list can change its size by discarding and inserting select objects.  An array is primarily suited for managing objects with identical lifetimes.

A Simple Collection of Objects

The simplest way to build a dynamic collection of objects is to link each object to the next one in the list by address.

Adding Objects

Consider a chain of elements that hold integer values.  Each element consists of a single int and a pointer to the next element in the chain.  The last element in the chain points to the nullptr address.  We call the element furthest from the nullptr address the *head* and the element that points to the nullptr address the *tail*.  We start with an empty chain and allocate memory one element at a time.  We add each new element to the head of the chain.



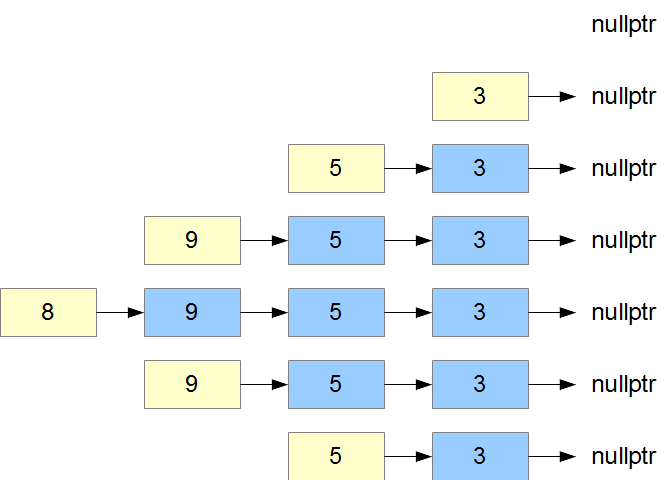
The following code implements this sequential allocation.  We deallocate elements in opposite order to that of construction:

|  |  |
| --- | --- |
| // Linked Lists - Adding Objects to a Chain  // chain.cpp  #include <iostream>  struct Element {  int data;  Element\* next;  Element(int d, Element\* n) : data(d), next(n) {}  };  int main () {  Element\* head = nullptr;  // Add one element at a time to the head of the chain  head = new Element(3, head);  head = new Element(5, head);  head = new Element(9, head);  head = new Element(8, head);  // Display elements from head to tail  for (Element\* p = head; p; p = p->next)  std::cout << p->data << ' ';  std::cout << std::endl;  // Deallocate memory one element at a time  while (Element\* p = head) {  head = head->next;  delete p;  }  } | 8 9 5 3 |

This list displays its elements naturally from head to tail; that is, in reverse order to that of their addition.

Removing Elements

To remove the first two elements from the head of the chain, we reset head to the address of the third element and deallocate the memory used by the first two elements:



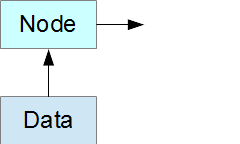
|  |  |
| --- | --- |
| // Linked Lists - Removing Elements from a Chain  // removeElement.cpp  #include <iostream>  struct Element {  int data;  Element\* next;  Element(int d, Element\* n) : data(d), next(n) {}  };  int main () {  Element\* head = nullptr;  // Add one element at a time to the head of the list  head = new Element(3, head);  head = new Element(5, head);  head = new Element(9, head);  head = new Element(8, head);  // Remove first two elements  Element\* remove = head;  head = head->next;  delete remove;  remove = head;  head = head->next;  delete remove;  // Display elements from head to tail  for (Element\* p = head; p; p = p->next)  std::cout << p->data << ' ';  std::cout << std::endl;  // Deallocate one element at a time  while (Element\* p = head) {  head = head->next;  delete p;  }  } | 5 3 |

In this design, we can only remove the element at the head.  First, we change the address of the head to that of the element pointed to by the element at the head.  Then, we deallocate the memory for the removed element.

Nodes

A standard linked list uncouples the data structure from the data values themselves.  The structure collects the objects through a system of *node*s.  Each node refers to a single object and holds at least one pointer to another node.  Accessing a particular object in a list involves starting with the node for a known object and stepping through the list node by node.

Referring to the example above, let us uncouple each data value from the pointer to the next element.  We do so by introducing a node for each element.  The node contains the object and at least one pointer to another node.  The object itself only holds the data value stored in the element.



For example,

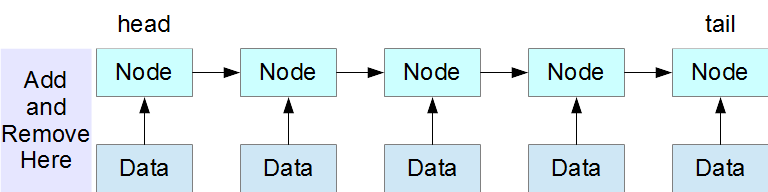
|  |  |
| --- | --- |
| // Linked Lists - Nodes  // nodes.cpp  #include <iostream>  class Data {  int data;  public:  Data(int d) : data(d) {}  int out() const { return data;}  };  struct Node {  Data data;  Node\* next;  Node (const Data& d, Node\* n) : data(d), next(n) {}  };  int main () {  Node\* head = nullptr;  // Add one nodes at a time to the head of the list  head = new Node(3, head);  head = new Node(5, head);  head = new Node(9, head);  head = new Node(8, head);  // Remove the head node  Node\* remove = head;  head = head->next;  delete remove;  // Display elements from head to tail  for (Node\* p = head; p; p = p->next)  std::cout << p->data.out() << ' ';  std::cout << std::endl;  // Deallocate one node at a time  while (Node\* p = head) {  head = head->next;  delete p;  }  } | 9 5 3 |

The Node constructor passes the data to its Data subobject and stores the address of the next node in its pointer member.  Both the Data object and the pointer are publicly accessible.

The Data object accepts information through its constructor and exposes that information through its out() query.

Stack

A *stack* is a special kind of linked list in which each node adds to the head and removes from the head.  In a stack, the last element in is the first element out (LIFO).



Let us redesign our Nodes example in the form of a Stack class.  A Stack object adds a node to its head through a push() member function and removes a node from its head through a pop() member function.

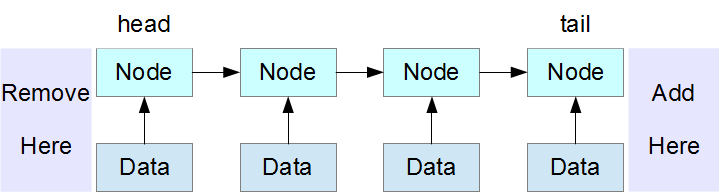
|  |  |
| --- | --- |
| // Linked Lists - Stack  // stack.cpp  #include <iostream>  class Data {  int data;  public:  Data(int d = 0) : data(d) {}  int out() const { return data;}  };  struct Node {  Data data;  Node\* next;  Node (const Data& d, Node\* n) : data(d), next(n) {}  };  class Stack {  Node\* head;  public:  Stack() : head(NULL) {}  ~Stack() {  while (Node\* p = head) {  head = head->next;  delete p;  }  }  void push(int d) { head = new Node (d, head);}  Data pop() {  Data data;  if (head) {  Node\* p = head;  data = head->data;  head = head->next;  delete p;  }  return data;  }  bool empty() { return head == NULL;}  };  int main () {  Stack s;  // Push Data Onto Stack  s.push(3);  s.push(5);  s.push(9);  s.push(8);  // Remove first Node  s.pop();  // Pop Data Off Stack  while (!s.empty())  std::cout << s.pop().out() << ' ';  std::cout << std::endl;  } | 9 5 3 |

The only instance variable in our Stack class is a pointer to the head node.  Given the address of this node, we can locate any data value on the stack by progressing through the elements one by one.

Our Stack class completely hides its implementation details from the application itself.

Queue

A *queue* is a special kind of linked list that adds nodes to the tail and remove nodes from the head.  A queue operates on a first in, first out principle (FIFO).



Let us redesign our Nodes example in the form of a Queue class.  Our Queue object adds a node to its head through a push() member function and removes a node from its tail through a pop() member function.

|  |  |
| --- | --- |
| // Linked Lists - Queue  // queue.cpp  #include <iostream>  class Data {  int data;  public:  Data(int d = 0) : data(d) {}  int out() const { return data;}  };  struct Node {  Data data;  Node\* next;  Node (const Data& d, Node\* n) : data(d), next(n) {}  };  class Queue {  Node\* head;  Node\* tail;  public:  Queue() : head(NULL), tail(NULL) {}  ~Queue() {  Node\* current;  while (current = head) {  head = head->next;  delete current;  }  }  void push(int d) {  Node\* p = new Node(d, 0);  if (head)  tail->next = p;  else  head = p;  tail = p;  }  Data pop() {  Data data;  if (head) {  Node\* p = head;  data = head->data;  head = head->next;  delete p;  if (!head) tail = NULL;  }  return data;  }  bool empty() { return head == NULL;}  };  int main () {  Queue q;  // Push Data onto the Queue  q.push(3);  q.push(5);  q.push(9);  q.push(8);  // Remove First Node  q.pop();  // Pop Data Off the Queue  while (!q.empty())  std::cout << q.pop().out() << ' ';  std::cout << std::endl;  } | 5 9 8 |

Our Queue class has two instance variables: a pointer to the head node and a pointer to the tail node. A Queue object displays its data values from head to tail in the same order as it has added nodes.