Outline

Objectives

1. Addresses and Pointers
2. Pointers to Array Elements
3. Problem Solving Applied: El Niño-Southern Oscillation Data
4. Dynamic Memory Allocation
5. Problem Solving Applied: Seismic Event Detection
6. Common Errors Using new and delete
7. Linked Data Structures
8. The C++ Standard Template Library
9. Problem Solving Applied: Concordance of a Text File

Objectives

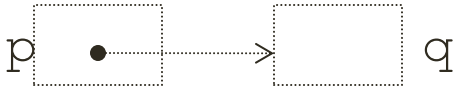
Develop problem solutions in C++ containing:

- Addresses and pointers
- Pointers to arrays
- Dynamic memory allocation
- Pointers with character strings
- new and delete
- Linked data structures
- Classes from the C++ Standard Template Library
- iterators

Addresses and Pointers

Addresses and Pointers

- A pointer is an object that holds the *memory address* of another object.
- If a variable p contains the address of another variable q, then p is said to point to q.
- If q is a variable at location 100 in memory, then p would have the value 100 (q's address).
  - Memory snapshot:



Address Operator

- The operator & is called the *address* operator.
- When the & operator is applied to an object, the result is the address of the object.

```
//Example:
int x=75;
cout << "x is " << x;
cout << "\nthe addres of x is " << &x << endl;
```

x [0x7fff8164] 75

Pointer Assignment

- Pointer types are declared using the pointer operator *, also called the dereferencing operator.
- When declaring more than one pointer variable, the * operator must precede each identifier.

Declaring Pointer Variables

Syntax:

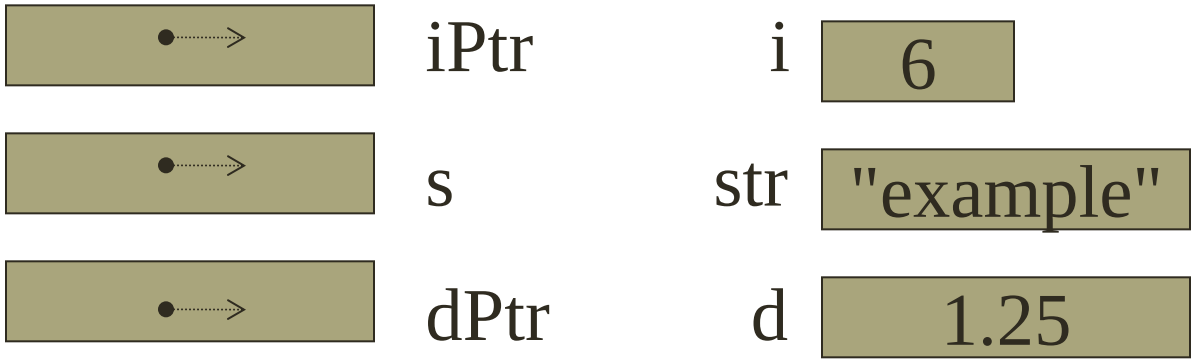
type_specifier *identifier1[, *identifier2[,...]];

Examples

```
int *iPtr; // iPtr is a pointer to int type.
int *iPtr1, *iPtr2; // iPtr1 and iPtr2 are both pointers to int type.
double *iPtr3, iPtr4; // iPtr3 is a pointer to double type.
                      // iPtr4 is a double
```

Examples

```
int *iPtr, i=6;
char* s, str[] = "example";
double *dPtr, d=1.25;
```

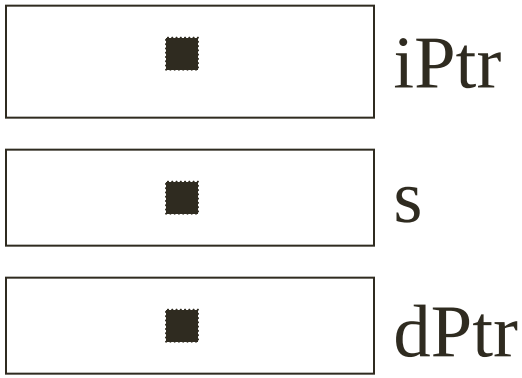


Initialization and Assignment

- Pointer types may be initialized at the time they are declared.
- Pointer types may be assigned new values using the assignment operator.

Examples

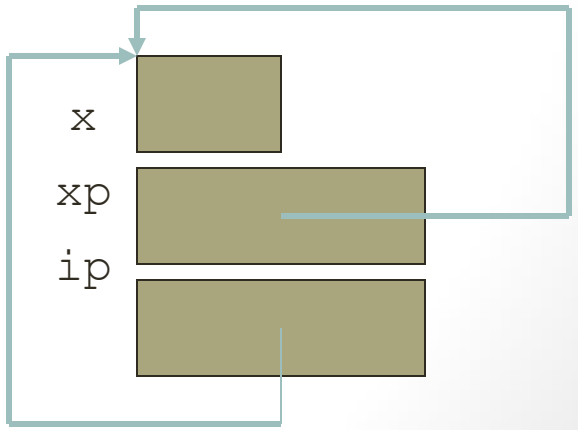
```
int *iPtr=0;
char *s=NULL;//predefined constant in iostream
double *dPtr=NULL;
```



Assignment

- The assignment operator (=) is defined for pointers of the same base type.
- The right operand of the assignment operator can be any expression that evaluates to the same type as the left operand.

```
int x, *xp, *ip;
xp = &x;
ip = xp;
```



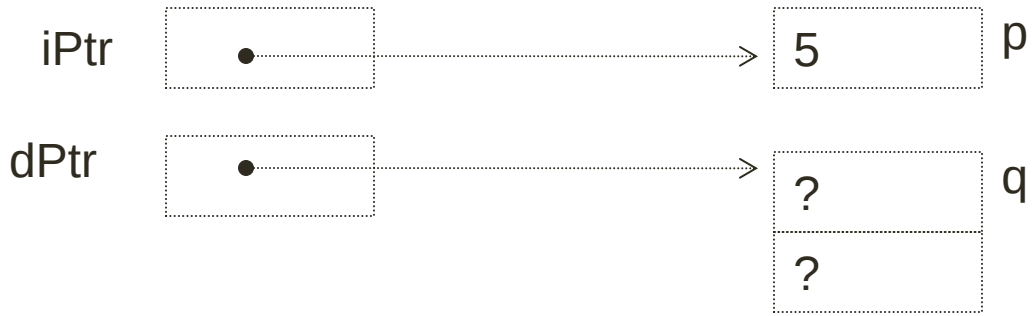
The Base Type

- The base type of a pointer refers to the type of object the pointer is referencing.
- The base type of a pointer defines the *size* of the object the pointer is referencing.
- The *size* of a pointer is independent of its base type.
  - `p` and `q` are the same (4 bytes**), but `p` points to 4 bytes** and `q` points to 8 bytes**

**compiler dependent

Examples

```
int * p(5), *iPtr=&p;
double q, *dPtr=&q;
```




```
sizeof p is 4
sizeof q is 8
sizeof iPtr is 4
sizeof dPtr is 4
```

Base Type

- A pointers base type determines how the object referenced by the pointer will be interpreted.
- The declaration:

```
int *p;
```

declares p to be a pointer to `int`. What ever p points to will be *interpreted* as an `int`, ie 4 bytes.

- Base type also defines pointer arithmetic.

Pointer Arithmetic

- Four arithmetic operations are supported:
  `+, -, ++, --`
- Arithmetic is performed relative to the base type of the pointer.
- When applied to pointers, ++ means increment pointer to point to next object.

Example: `p++;`

- if `p` is defined as `int *p, p` will be incremented by 4 (bytes).
- if `p` is defined as `double *p, p` will be incremented by 8(bytes).

Example

```
int *p;
cout << "size of char is "
     << sizeof(char) << endl;
cout << "size of int is "
     << sizeof(int) << endl;
cout << "size of double is "
     << sizeof(double) << endl;
cout << "size of float is "
     << sizeof(float) << endl;
cout << "the size of p is "
     << sizeof(p) << endl;
```

Output:

```
size of char is 1
size of int is 4
size of double is 8
size of float is 4
the size of p is 4
```

Comparing Pointers

- You may compare pointers using relational operators
- Common comparisons are:
  - check for null pointer **(p == NULL)**
  - *Note: since NULL evaluates as false, and any other pointer evaluates as true, checking for a null pointer can be done as **(!p)***
  - check if two pointers are pointing to the same object

    (p == q)
  - *Note: **(*p == *q)** means they are pointing to equivalent, but not necessarily the same data.*

Pointers to Array Elements

1D Arrays

- The name of an array is the address of the first element (i.e. a pointer to the first element).
- Arrays and pointers may often be used interchangeably.

Example

```
int num[4] = {1,2,3,4}, *p;
p = num; //same as p = &num[0];
cout << *p <<endl;
++p;
cout << *p;
```

Output:

```
1
2
```

1D Arrays

- You can index a pointer using [] operator.

Example

```
char myString[] = "This is a string";
char *str;
str = myString;
for(int i =0; str[i]; i++)  //look for null
  cout << str[i];
```

Output:

```
This is a string
```

Arrays and Pointers

- When an array is defined, memory is allocated according to the specified size of the array.
- The name of an array is a pointer to the first element. However, the value of the pointer can not be changed. It will always point to the same memory location.
- When a pointer is defined, 4 bytes* are allocated to store a memory address.
- The value assigned to a pointer can be modified (reassigned to point to a different object).

Arrays of Pointers

- You may define arrays of pointers like any other data type in C++

Example

```
int num=8;
//declare an array of 10 pointers to int
int *iPtrs[10];
//first element is assigned a value
iPtrs[0] = &num;
//output the value of num
cout << *iPtrs[0];
```

Pointers As Arguments to Functions

- Pointers may be passed either "by value" or "by reference".
- In either case, the pointer can be used to modify the object to which it is pointing.
- Only when passed by reference can the pointer argument itself be modified.

Common Pointer Problems

- **Using uninitialized pointers**

```
int *iPtr;
*iPtr = 100;
```

  iPtr has not been initialized. The value 100 will be assigned to *some* memory location. Which one determines the error.

- **Failing to reset a pointer after altering it's value.**
- **Incorrect/unintended syntax.**

Problem Solving Applied: El Niño-Southern Oscillation Data

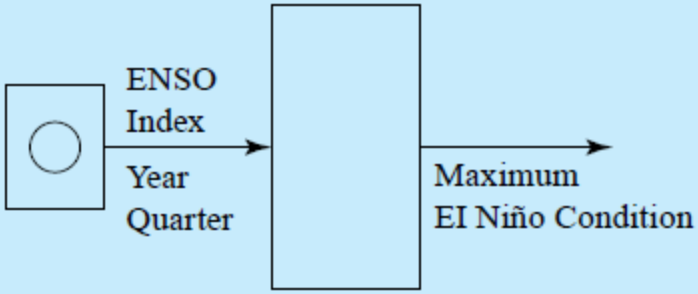
Problem Solving Applied: El Niño-Southern Oscillation Data

1. PROBLEM STATEMENT

Determine the year and quarter with the strongest El Niño conditions.

2. INPUT/OUTPUT DESCRIPTION

The I/O diagram shows the data file as the input and the year and quarter as output.



Problem Solving Applied: El Niño-Southern Oscillation Data

3. HAND EXAMPLE

Assume that the data file contained the following data:

Year	Quarter	ENSO Index
1990	1	0.6
1991	1	0.2
1992	1	1.1
1993	1	0.5
1994	1	0.1
1995	1	1.2
1996	1	−0.3
1997	1	−0.1
1998	1	2.2
1999	1	−0.7
2000	1	−1.1

The corresponding output would then be the following report:

```
Maximum El Nino Conditions in Data file
Year: 1998, Quarter: 1
```

Problem Solving Applied: El Niño-Southern Oscillation Data

4. ALGORITHM DEVELOPMENT

We first develop the decomposition outline because it divides the solution into a set of sequential steps.

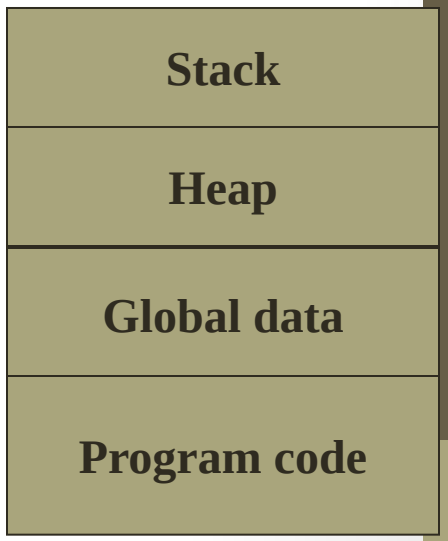
Decomposition Outline

1. Read the ENOS data into arrays and determine the maximum positive index.
2. Print the year and quarter that go with the maximum intensity.

Dynamic Memory Allocation

Allocation of Memory

- Memory for variables is allocated from one of several classes:
  - *Run-time stack*
    - Local variables
    - Formal parameters
    - Managed by the compiler
  - *Heap*
    - Dynamic storage
    - Managed by storage allocator
  - *Global Data*
    - Static (global) variables
    - Managed by the compiler



Dynamic Memory Allocation

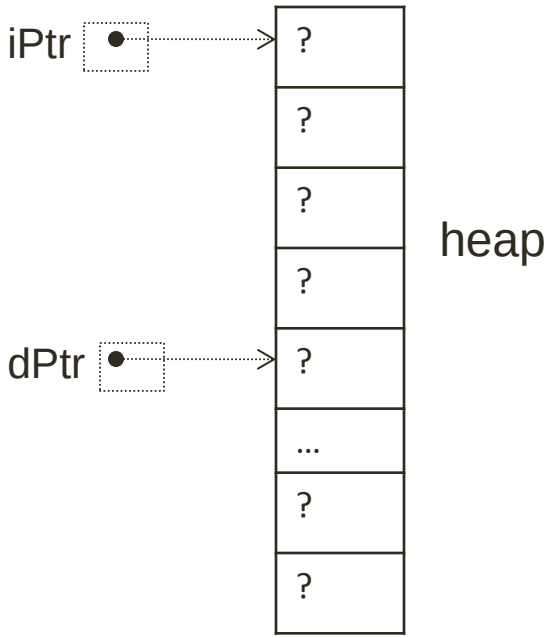
- Dynamically allocated memory is defined at *runtime.*
- Dynamic allocation of memory allows for more efficient use of a finite resource.
- Dynamic allocation is often used to support dynamic data structures such as stacks, queues, linked lists and binary trees.
- Dynamically allocated memory should be freed during execution when it is no longer needed.

Operator new

- Replaces *malloc*() used in C
- Allocates a block of memory from the heap.
- Returns the address of the first byte.
- If allocation fails, new throws an exception that terminates the program (or returns NULL on older compilers).

Example

```
int *iPtr;
iPtr = new int;// 4 bytes are allocated
               // iPtr points to 1st byte
double *dPtr;
dPtr = new double[20]; // 160 bytes allocated
                       // dPtr points to 1st byte
```



Initializing Dynamically Allocated Memory

- To initialize a dynamically allocated object, the initial value is provided inside parentheses following the type.

Example:

```
int *iPtr;
iPtr = new int(100);   //4 bytes allocated
                       //initial value: 100
```

Operator delete

- Replaces free() used in C.
- The `delete` operator frees memory allocated by `new`.
- Using `delete` to attempt to free any other type of address will result in errors.

Example

```
int *iPtr;
iPtr = new int(100);
cout << *ptr;
```


```
delete iPtr;
```


```
//memory manager is now free to reallocate
//freed memory to other programs
//accessing freed memory may or may not result
//in errors. Good idea to set pointer to null
iPtr = NULL;
```

Example: Dynamically Allocated Arrays

```
double *dptr;
const int SIZE = 10;
dptr = new double[SIZE];//80 bytes
for(int i=0; i<SIZE; ++i)
  cin >> dptr[i];
fun1(dptr, SIZE); // pass array to fun1
delete [] dptr;   //free all 10 elements
```

Problem Solving Applied: Seismic Event Detection

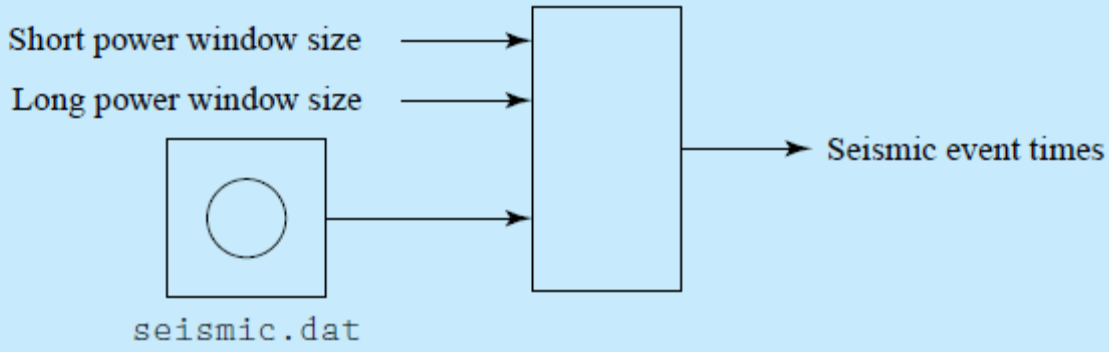
Problem Solving Applied: Seismic Event Detection

1. PROBLEM STATEMENT

Determine the locations of possible seismic events using a set of seismometer measurements from a data file.

2. INPUT/OUTPUT DESCRIPTION

The inputs to this program are a data file named *seismic.dat* and the number of measurements to use for short-time power and longtime power. The output is a report giving the times of potential seismic events.



3. HAND EXAMPLE

Suppose that a data file contains the following data, which includes number of points to follow (11) and time interval between points (0.01), followed by the 11 values that correspond to a sequence of values $x_0, x_1, \ldots x_10$:

```
11 0.01
1    2    1    1    1    5    4    2    1    1    1
```

If the short-time power measurement is made using two samples and the longtime power measurement is made using five measurements, then we can compute power ratios, beginning with the rightmost point, in a window:

```
1 2 1 1 1 5 4 2 1 1 1
        ⌊⌋

short window
⌊_____⌋

long window

Point x4:    Short-time power = (1 + 1)/2 = 1

Long-time power= (1 + 1 + 1 + 4 + 1 )/5 = 1.6
Ratio = 1/1.6 = 0.63

1 2 1 1 1 5 4 2 1 1 1
          ⌊⌋

short window
  ⌊_____⌋

long window
```

Point x5: Short-time power = (25 + 1)/2 = 13

Long-time power = (25 + 1 + 1 + 1 + 4)/5 = 6.4

Ratio = 13/6.4 = 2.03

1 2 1 1 1 5 4 2 1 1 1

short window

long window

Point x6: Short-time power = (16 + 25)/2 = 20.5

Long-time power = (16 + 25 + 1 + 1 + 1)/5 = 8.8

Ratio = 20.5/8.8 = 2.33

1 2 1 1 1 5 4 2 1 1 1

short window

long window

Point x7: Short-time power = (4 + 16)/2 = 10

Long-time power = (4 + 16 + 25 + 1 + 1)/5 = 9.4

Ratio = 10/9.4 = 1.06

1 2 1 1 1 5 4 2 1 1 1
short window

long window

Point x8: Short-time power = (1 + 4)/2 = 2.5

Long-time power = (1 + 4 + 16 + 25 + 1)/5 = 9.4

Ratio = 2.5/9.4 = 0.27

1 2 1 1 1 5 4 2 1 1 1

short window

long window

Problem Solving Applied: Seismic Event Detection

```
Point x9:    Short-time power = (1 + 1)/2 = 1

Long-time power = (1 + 1 + 4 + 16 + 25 )/5 = 9.4

Ratio = 1/9.4 = 0.11

1 2 1 1 1 5 4 2 1 1 1

short window

long window

Point x10:    Short-time power = (1 + 1)/2 = 1

Long-time power = (1 + 1 + 1 + 4 + 16 )/5 = 4.6

Ratio = 1/4.6 = 0.22
```

By using the previous ratios computed, possible seismic events occurred at points x5 and x6. Because the time interval between points is 0.01 second, the times that correspond to the seismic events are 0.05 and 0.06 second. (We assume that the first point in the file occurred at 0.0 second.)

Problem Solving Applied: Seismic Event Detection

4. ALGORITHM DEVELOPMENT

We first develop the decomposition outline because it divides the solution into a series of sequential steps.

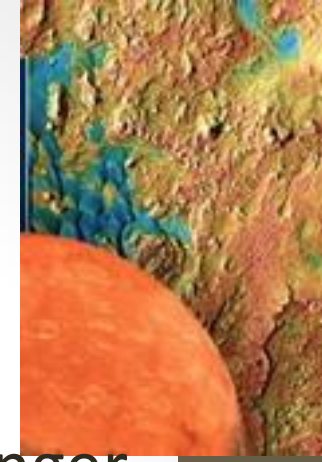
Decomposition Outline

1. Read data header and allocate memory.
2. Read seismic data from the data file and read numbers of measurement for power from the keyboard.
3. Compute power ratios and print possible seismic event times.

Step 3 involves computing power ratios and comparing them to the threshold to determine whether a possible event occurred. Because we need to compute two power measurements for each possible event location, we implement the power measurement as a function.

Common Errors Using new and delete

Memory Management



- Dynamically allocated memory should always be returned to the heap once it is no longer needed to allow for the reuse of the memory.
- Careful tracking of pointers to this memory is required.
- Strategies for managing pointers often involve establishing an owner of each an every piece of dynamic memory.
  - The owner is responsible for deleting the memory when it is no longer needed.

Common Dynamic Memory Errors

- Referencing a pointer to dynamically allocated memory after delete has been called to return the memory to the heap.
- Failing to return memory when it is no longer being used.
  - Often called a **memory leak**.
- Using the delete operator with a pointer that does not reference memory that has been dynamically allocated using the new operator.
- Omitting the square brackets when using delete to free a dynamically allocated array.

Linked Data Structures

Why Linked Data Structures?

- Array-based data structures (C++ arrays, vector, string) are efficient for accessing elements.
  - Offset-based access to elements from the base address is very fast.
- The restriction that memory is a contiguous block has consequences.
  - STL makes this less painful for the programmer by encapsulating memory management into the objects; however there are still consequences.
- Insertion and deletion of elements are expensive

Linked Data Structures

- Linked data structures avoid the restriction that the memory of a data structure be contiguous.
- Pointers connect (i.e. link) pieces of the data structure together.
- Simple linked data structures include:
  - Linked lists
  - Stacks
  - Queues

Linked List

- A linked list is a data structure in which each element of the list consists of the data stored in the element and a pointer to the next element in the list.
  - Usually keep a pointer to the first element of the list (the head of the list)
- Inserting an element to a particular place in the list is very efficient, but traversing the list is not.
  - This is the exact opposite characteristics of array operations!

Stack

- A queue is a sequential container like the linked list and queue.
- Last-In, First-Out (LIFO) data structure.
  - Thus one may only remove (pop) from one ‘end’ of the data structure, and only insert (push) to the ***same*** ‘end’.
- Essential in the design of design of compilers, operating systems, etc.

Queue

- A queue is a sequential container like the linked list.
- First-In, First-Out (FIFO) data structure.
  - Thus one may only remove (pop) from one ‘end’ of the data structure, and only insert (push) to the ***other*** ‘end’.
- Often used in processing of job requests.
  - E.g. printer queues.

The C++ Standard Template Library (STL)

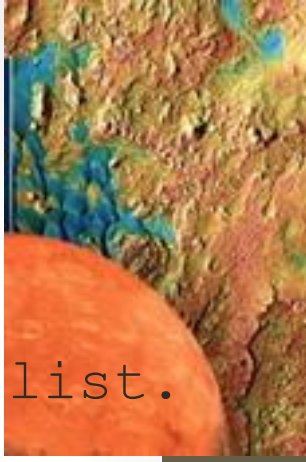
Sequential STL Containers

- List
- Stack
- Queue

STL List

- A list is a data structure organized as a collection of elements, or nodes, that are linked by pointers.
- Elements can be added at any position in a list in constant time by reassigning pointers.
- List have many useful applications in the organization large amounts of data.

The STL list Class



- `list` is a `class template` defined in the header file `list`.

Example

```
#include <string>
#include <list>

using namespace std;

list<string> word; //list of strings
```

Methods of list Class

- Elements can be added to and removed from any position in a list using the appropriate method and an iterator.
- An iterator is similar to a pointer, but is usually implemented as a class object.
- Common Methods of List Class:

```
bool empty()
iterator insert()
iterator remove()
iterator begin()
iterator end()
```

Example

```
list<string> wordList;
list<string>::iterator iter =

  wordList.begin();
iter = wordList.insert(iter,"hello");
wordList.insert(iter,"world");
for (iter=wordList.begin();
    iter!= wordList.end();
    iter++)
  cout << *iter << " ";
```

Output:

```
hello world
```

The STL queue Class

- `queue` is a `class template` defined in the header file `queue`.

Example

```
#include <queue>

using namespace std;

queue<int> word; //queue of integers
```

Methods of queue Class

```
bool empty();
void pop(); //remove from front
void push(data_type); //add to end
data_type front(); //return front
data_type back(); //return end
```

Example

```
#include <queue>
#include <iostream>
using namespace std;
…
queue<int> theQueue;
theQueue.push(10);
theQueue.push(20);
theQueue.push(30);
while (!theQueue.empty() ) {
  cout << theQueue.front() << endl;
  theQueue.pop();
}
```

Output:

```
10
20
30
```

The STL stack Class

- `stack` is a `class template` defined in the header file `stack`.

Example

```
#include <stack>

using namespace std;

stack<int> word; //stack of integers
```

Methods of stack Class

```
bool empty();
void pop(); //remove from top
void push(data_type); //add to top
data_type top(); //return top
```

Example

```
#include <stack>
#include <iostream>
using namespace std;
…
stack<int> theStack;
theStack.push(10);
theStack.push(20);
theStack.push(30);
while (! theStack.empty() ) {
  cout << theStack.top() << endl;
  theStack.pop();
}
```

Output:

```
30
20
10
```

Problem Solving Applied: Concordance of a Text File

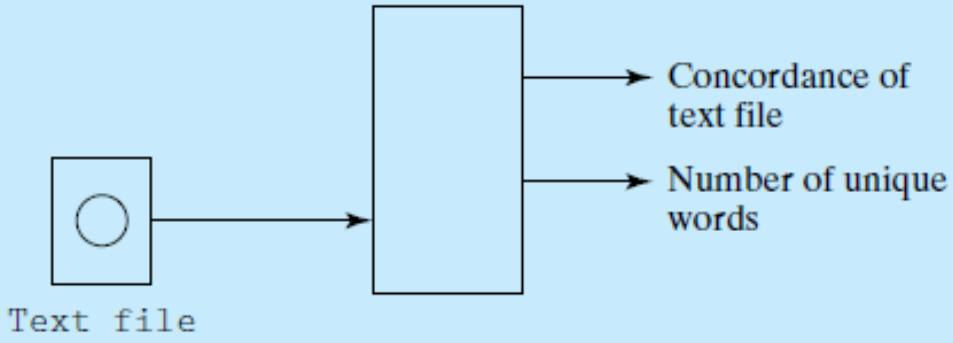
Problem Solving Applied: Concordance of a Text File

1. PROBLEM STATEMENT

Build a concordance of a text file. Write the concordance, along with a count of the unique words in the text file, to an output file.

2. INPUT/OUTPUT DESCRIPTION

The following diagram shows that the input to the program is a text file, and that the output is a concordance of the text file along with the count of unique words. We will use the `list` class to define a list, where each element in the list is of type `string`.



Problem Solving Applied: Concordance of a Text File

3. HAND EXAMPLE

Suppose our text file contained only the following text:

```
A concordance of a text file is an alphabetical list of
the unique words in the text file.
```

Our program would generate the following output file:

```
There are 13 distinct words in the text file:
a
an
alphabetical
concordance
file
in
is
list
of
text
the
unique
words
```

4. ALGORITHM DEVELOPMENT

We first develop the decomposition outline to break the solution into a series of sequential steps:

Decomposition Outline

1. Open input and output files.
2. Read a word from the input file.
3. Insert word if the word is not already in the list.
4. Alphabetize list of unique words.
5. Write the size and contents of the list to output file.

Step 2 in the decomposition outline involves a loop that reads the text file one character at a time until a nonalpha character is reached. A nonalpha character will signal the end of a word. Other than their use as delimiters between words, all nonalpha characters will be ignored. All alpha characters will be converted to lowercase. We will use a function to perform this task. Step 3 requires that we insert a word if it is not already in the list. We will use a function that utilizes member functions of the *list* class. Step 4 involves sorting the list in ascending order. We will use the generic *sort()* function. Since the list may be long, step 5 will print the list three words per line. We will use a function to perform this task.