CSIS/COMP 1117B
Computer Programming

Arrays, Searching and Sorting

Arrays, Searching And Sorting

- Arrays
- Searching
  - sequential search
  - binary search
- Sorting
  - selection sort
  - insertion sort
- Multi-dimensional arrays

Array Declaration

- An **array** is a collections of data of the ***same*** type.
- To declare an array, we need to specify the name of the array, the type of data it contains and the total number of entries making up the array:

  <type> <identifier> ‘[‘ <length> ‘]’ ‘;’
- For example:

```
int a[20];          // a is an int array of 20 entries
double c[1000];     // c is a double array of size 1000
bool f[10];         // f is a bool array of length 10
char x[112];        // x is a char array of 112 elements
```

Array Initialization

- Similar to ordinary variables, arrays can be initialized in their declarations; for example, the following declaration declares a 3-element int array together with their initial contents:

  ```
  int p[3] = { 1, 2, 3 };   // p[0]=1, p[1]=2, p[2]=3
  ```

- Arrays containing elements of type char:

  ```
  char x[5] = { 'h', 'k', 'u' }; // x[3] & x[4] are not initialized
  char x[5] = "hku";             // similar to the declaration above
  char n[ ] = { 'h', 'k', 'u' };  // declare & initialize an 3-entry array
  char b[ ] = "This is a much longer char array";   // length of b?
  ```

Indexing Array Elements

- Individual entries of an array are identified by its position in the array, called the **index** of the entry.
- For an array of n entries, the index ranges from 0 to n–1; for example:
  - a[0], a[1], a[2], . . ., a[19] are int variables
  - c[0], c[1], . . ., c[999] are double variables
- Both the name of the array and an index are required to access an entry of the array:

  <identifier> '[' <expression> ']'
- For example:

  f[3] // 4th entry of array f, a bool variable

  x[101] // 102th entry of array x, a char variable
- Note that expression ***should*** evaluate to an integer between 0 and (size of array – 1).

Array Out of Bound

- It should be obvious that programs should not try to access non-existent elements of an array (i.e., try to use an index not within the proper range for that particular array).
- For example:

```
int t[ ] = { 2, 3, 5, 7, 11 };
for (i= 0; i <=5; i++) {
      . . . use t[i]            // t[5] does not exist!
}
```

- ***C++ does not report this***!!

Consequence . . .

- If the accessed variable is within the data area:
  - reading its contents is ***probably*** harmless
  - updating its contents could be a very subtle bug
    - a name constant could be changed to a different value – *how can you find that out*?
- If the accessed data is to the code area:
  - the hardware might prevent updates to the code area (code area is usually ***read-only***) and the operating system will typically abort the program

Application

- An array is often used to store a collection of data ***in order***. The ordering among values of the index set is used to impose an ordering on the data stored in an array.
- For example, to store data in ***ascending*** order in an array:
  - from mathematics, we know: $0 < 1 < 2 < 3 < \ldots$
  - put smallest value in a[0], put next smallest in a[1], …
- Data can also be stored in an array in ***descending*** order, i.e., the largest data is in a[0], next largest in a[1], …

Example - Finding The Smallest Element

- The following program will find the smallest element in an array:

```
double x[ ] = { 4.1, 12.0, 7.3, 3.6, 5.9 };  // size of x is 5
double min;
min = x[0];
for (int i = 1; i < 5; i++)
    if (x[i] < min) min = x[i];
```

- Re-writing it as a function:

```
double min(double x[ ], int n) {  // n is the size of x
    double m;
    m = x[0];
    for (int = 1; i < n; i++) if (x[i] < m) m = x[i];
    return m;
}
```

Searching

- An important application of computers is to manage large amounts of information in the form of **databases**.
- For example, a student database contains student records; each student record typically contains personal information on a student as well as enrollment information.
- Individual records are identified by a ***key*** which is unique among all records; for example, students are uniquely identified by a UID.
- Problem of searching: given a key, determine whether or not a record with a matching key exits in a collection of records.

Sequential Search

- The most straightforward solution is to go through the collection of records one by one until we either find it or have exhausted all the records.
- A sequential search function:

```
// search returns the position of key in x
int search(double x[ ], int n, double key) {
    for (int i = 0; i < n; i++)
        if (key == x[i]) return i;
    return -1;    // key not found
}
```

- Sequential search is very inefficient (on the order of the size of the data set being search).

Binary Search

- If the data is sorted (e.g., in ascending order), searching can be performed is a much more efficient manner:
  - given a subset a[u] … a[v] of an array a[n]
  - compute m = (u + v) / 2 (midpoint of the range [u,v])
  - if a[m] = key, return m
  - if key < a[m], look for key in the ***lower*** half;
    - impossible for key to be in the upper half since they are ***all larger*** than a[m]
  - otherwise, look for key in the ***upper*** half
- Binary search is more efficient than sequential search because the data set to be searched in the ***next*** step is reduced by ***half*** at each step.

A Simple Binary Search Program

```
#include <iostream>
using namespace std;
char c[ ] = “0123456789abcdefghijklmnopqrstuvwxyz”;
int b_search(char c[ ], int n, char key) {
    int  m, u, v;
    u = 0; v = n – 1;                       // u, v index the two ends of c
    while (u <= v) {                        // there are entries between u & v
        m = (u + v) / 2;                    // compute index of midpoint
        if (c[m] == key) return m;          // found a match
        else if (c[m] > key) v = m – 1;     // continue in lower
        else u = m + 1;                     // continue in upper
    }
    return -1;                              // not found
}
int main() {
    char k;
    cout << “Type a letter to be searched: “;
    cin >> k;
    cout << k << “ is in position: “ << b_search(c, sizeof(c), k) << endl;
}
```

Recursive Binary Search

- Binary search is *naturally* recursive: if not found, apply binary search on a selected half of the original data set.

```
#include <iostream>
using namespace std;
char c[ ] = “0123456789abcdefghijklmnopqrstuvwxyz”;
int b_search(char c[ ], int u, int v, char key) {
    int m;
    if (u > v) return –1;            // check for termination, not found
    m = (u + v) / 2;
    if (c[m] == key) return m;    // found key
    else if (c[m] > key) return b_search(c, u, m – 1, key);   // continue in lower
    else return b_search(c, m + 1, v, key);                   // continue in upper
}
int main() {
   char k;
   cout << “Input a letter to be searched: “;
   cin >> k;
   cout << k “ is in position “ << b_search(c, 0, sizeof(c) – 1, k) << endl;
}
```

Sorting

- One very simple method for sorting data in sequence is:
  1. starting from one end, look for the data that should reside at this end and put it there
     - may need to interchange the data at this end with some data in another position

     ```
     // swap 2 integers a, b
     void swap(int &a, int &b) {
         int t;
         t = a; a = b; b = t;
     }
     ```
  2. repeat above on the remaining data until the data set contains only 1 piece of unsorted data (which is in order!)

Selection Sort

- Suppose we want to sort an int array in ascending order:
  1. find the largest value and interchange with the last value if necessary
  2. *remove* the last element and repeat the above step until there is only 1 element left

```
// find position of largest value in array a[n]
int max(int a[ ], int n) {
        int k;
        k = n – 1;
        for (int i = k – 1; i > = 0; i--)
             if (a[i] > a[k]) k = i;
        return k;
}
```

Selection Sort - Program

```
// swap 2 integers a, b
void swap(int &a, int &b) {
      int t;
      t = a; a = b; b = t;
}
// find position of max in array a[0..n–1]
int max(int a[ ], int n) {
      int k;
      k = n – 1;
      for (int i = k – 1; i > = 0; i--)
           if (a[i] > a[k]) k = i;
      return k;
}
// selection sort on array a[0..n-1]
void s_sort(int a[ ], int n) {
      int  t;
      for (int k = n; k > 1; ) {
                t = max(a, k);     // max returns a value between 0 and k – 1
                if (t != --k) swap(a[t], a[k]);
      }
}
```

Some Remarks on Selection Sort

- The number of comparisons performed by selection sort is of the order the square of the size of the data set being sorted.
- Selection sort operates mainly on the *unsorted* part of the set – going through it repeatedly looking for the largest value remaining in the set.
- *What if we shift our attention to the sorted part of the set, would there be any gain?*
  - yes and no!

Insertion Sort

- Insertion sort operates by picking one data from the unsorted part and trying to insert it into the sorted part ***in sequence***:
  1. the sorted set consists of just the last data (since a data set of size 1 is already sorted)
  2. consider the last data in the unsorted set
  3. find its position in the sorted set and put it in place
  4. (note that the unsorted set has 1 less data because of the previous steps) repeat steps 2 and 3 above until the unsorted set is empty

Finding the Insertion Point

- Consider an int array a of size n,
  - suppose a[k+1] thru a[n–1] had been sorted
  - and a[0] thru a[k] are not yet sorted
  1. let j range over k+1 and n–1
  2. if a[j] < a[k], then move a[j] down 1 place (to make room for a[k])
     - repeat step 2 until j $\geq$ n (until the last data has been checked)
  3. if (j –1) $\neq$ k, then put a[k] in a[j –1]

Insertion Sort - Program

```
// insertion sort on array a[0..n–1]
void i_sort(int a[ ], int n) {
    int t, j;
     for (int k = n – 2; k >= 0; k--) {
          t = a[k];
          for (j = k + 1; j < n; j++)
               if (a[j] >= t) break;   // found position for a[k]
               else a[j–1] = a[j];     // make room for a[k]
          if (--j != k) a[j] = t;         // put a[k] in place if needed
}
```

- *Why is this given as a single function, not 3 as in the case of selection sort?*

Some Remarks on Insertion Sort

- If the array is already sorted, insertion sort will go through it once and the effort is of the order of the size of the data set, not square of the size!
- If there appears subsequences of sorted data in parts of the array being sorted, very little effort will be spent on sorting these sorted subsequences.
  - unlike selection sort, which will always incur the same amount of effort irrespective of whether or not parts or the entire array are already sorted
- However, a substantial amount of effort is spent on moving data around for the insertion to take place!
  - in selection sort, exactly n–1 data swaps for sorting an array of size n

Multi-dimensional Arrays

- An array can be made up of objects of ***any type*** – including arrays
- For example, a two-dimensional int array a can be declared by:

  int a[3][4];

  – a[0], a[1], and a[2] are int arrays each with 4 entries
  – a[0][0], a[0][1], a[0][2], a[0][3], …, a[2][3] are int variables
- Some more examples:

```
char x[10][20][10];          // a 3-dimensional char array
double p[20][20][20][20];    // a 4-dimensional double array
bool k[2][3][4][5][6];       // a 5-dimensional bool array
```

Multi-dimensional Array Initialization

- Multi-dimensional arrays can also be initialized in their declarations:

  int a[3][4] = { {1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6} };

  int a[ ][4] = { {1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6} };

- *Partial* initialization is also allowed, in the following, the first entry of the first row, the first two entries of the second row, and the first three entries of the third row are initialized to the given values and remaining entries are initialized to 0 (by default):

  int a[3][4] = { {1}, {2, 3}, {3, 4, 5} };

Tables

- Tables are often represented as a 2-dimensional array with the first index representing rows and the second index representing columns.
- For example,

  char c[5][7]; // a 5 by 7 char array

  – c[i][j] refers to the char at row i, column j
- 2-dimensional arrays are also often used to represent matrices:

  double m[10][10]; // a 10 by 10 double matrix

  – m[j+1][k] refers to the entry at row (j + 1), column k

Matrix Multiplication

- A m by n matrix multiplied by a n by k matrix will produce a m by k matrix:

```
void multiply(double a[ ][10 ], double b[ ][10 ], double c[ ][10 ],
                    int m, int n, int k)
{
    for (int i = 0; i < m; i++)
        for (int j = 0; j < k; j++) {
            double s = 0;
            for (t = 0; t < n; t++) s = s + a[i][t] * b[t][j];
            c[i][j] = s;
         }
}
```

Storing a Multi-dimensional Array

- A 1-dimensional array is a contiguous block of entries; e.g., a[0], a[1], …, a[n–1]
  - address of a[1] = address of a[0] + size of an entry
  - address of a[k] = address of a[0] + k * size of an entry
- A 2-dimensional array is a contiguous sequence of arrays; e.g., a[0][0], a[0][1], …, a[0][m–1], a[1][0], a[1][1], …, a[1][m–1], …, a[n–1][0], a[n–1][1], …, a[n–1][m–1]
  - the size of the second index is needed to compute the size of each 1-dimensional array; ***hence, only the size of the first dimension is optional.***