CIS 22A
Review: Arrays

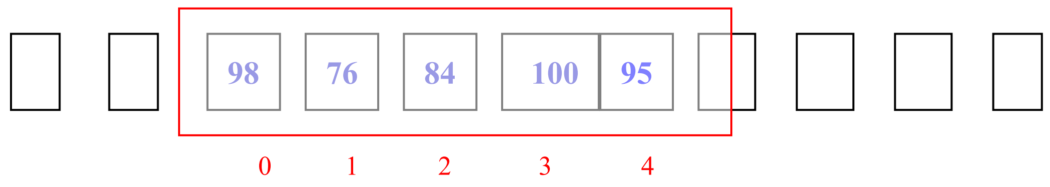
Array − an internal collection of elements of the same type, called the *base type of the array*.

Why use arrays (as opposed to individual variables)?

- group a large amount of related data under a single name
- easy processing using loops (shorter code, easier to understand and maintain)
- easy to pass to functions

A **linear list** is a list in which each element has a unique successor except the last one. One way of implementing a linear list is using an array. There are other ways of implementing linear lists, and there are lists that are not linear. (*More about these in advanced courses*

Arrays and Memory − an array occupies contiguous memory; for instance, an array of 5 integers takes `5 * sizeof(int)` bytes.



Size or **Length** of an Array – its number of elements

Fixed Length Arrays – the number of elements is a constant.

Maximum Size – the maximum number of elements the array can store; not all of them might be used all the time.

Actual Size – the actual number of elements the array holds; it has to be less than or equal to the maximum size (always to be checked!)

Declaring and Defining Arrays

```
// type, name, and size(or length): the number of elements
int    scoreList[40];
char   title[80];
double pricelist[100];
```

Another way to declare and define an array is to use a named constant instead of a literal constant for its maximum size:

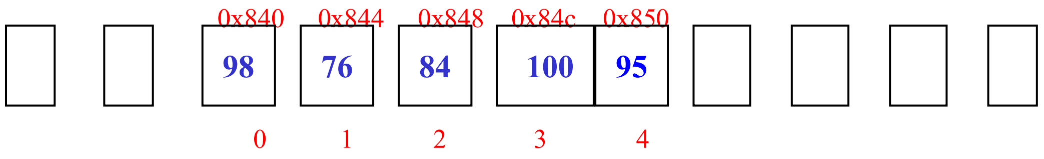
```
const int MONTH = 12;
//. . .
     string monthList[MONTH];


#define MAX_SIZE 100
// . . .
     int aList[MAX_SIZE];
```

The Name of an Array – in C++ has a special meaning. We refer to an array by using its name, for instance `aList`, but `aList` is not technically an array itself. `aList` is a constant that represents the address of the first element in the list.

```
int    aList[5] = { 98, 76, 84, 100, 95 };

cout << aList; // 0x840
```



Initializing Arrays

```
int    scoreList[5] = { 98, 76, 84, 100, 95 };
char   gradeList[4] = { 'A', 'C', 'B', 'A' };
double priceList[9] = { 65.99, 24.75, 30.00 };
                  // partial initialization: the last 6 elements are set to ZERO
int    aList[10000] = {0}; // all elements are set to ZERO
int    bList[]      = { 9, 8, 7, 5 ); //default size is 4
```

Accessing Elements in an Array – using an auxiliary variable called **index** that represents the location of the element in the array: the first one is at location 0, the second one is at location 1, and so on. The index must be of an integral type (it could be any expression as long as its value is integral and is within range).

```
int aList[6] = { 98, 76, 84, 100, 95 };
                       // the 6th element has the default value: 0
cout << aList[0];      // 98
cout << aList[4];      // 95

aList[1] += 4;         // 76 + 4
cout << aList[3 - 2];  // aList[1] is now 80
```

What happens if you use an index that is outside the range? For instance, in the above example we have 6 correct indices: 0, 1, 2, 3, 4, and 5. What happens if we use 5, or 6 or 10 or –1? Try it!

```
cout << aList[5]; // 0  - default value
cout << aList[6]; // junk
cout << aList[10];// junk
cout << aList[-1];// junk
```

It depends on what else is there in your program, but the program is more likely going to display junk values. If the program changes an element that's outside the range: `aList[10] += 4;` it could cause a crash. <u>It is the programmer's responsibility to keep the index within the range of the array!</u>

Exchanging Two Elements in an Array – using an auxiliary variable

```
int list[5] = { 98, 76, 84, 100, 95 };
int hold;

hold    = list[2]; // 84 is copied to hold
list[2] = list[0]; // 98 overwrites 84 in the array
list[0] = hold;    // 84 from hold overwrites 98
```

Printing an Array – one element at a time

```
int scoreList[5] = { 98, 76, 84, 100, 95 };

for ( int i = 0; i < 5; i++ )            // first to last
    cout << scoreList[i];

for ( int i = 4; i >= 0; i-- )           // last to first
    cout << scoreList[i];
```

Reading an Array – one element at a time (it is a little bit trickier than printing it!)

```
int aList[SIZE];
int n;

// ASSUME you know the actual number of elements
cout << "How many numbers? ";
cin >> n;
if ( n > SIZE ) // there's not enough memory to store them all
{
    cout << "Only " << SIZE
         << " values can be stored!\n";
    n = SIZE;
}
for ( int i = 0; i < n; i++ )
{
    cout << "Enter element at index " << i << ": ";
    cin >> aList[i];
}
```

Another approach would be to write a function that reads and validates the actual size of the array (using a loop).

The above approach is better when getting data from a file. However, instead of changing the size to the maximum size, you could write an appropriate error message and terminate the program, or get the user's input on what should be done.

```
// ASSUME you DO NOT know the actual number of elements
// ASSUME you DO NOT know the actual number of elements: stop either
// when the array is full or when the user chooses to stop (EOF)
                 // to stop entering data press CTRL^Z or CTRL^C

i = 0;
while ( i < SIZE && cin >> num )
{
     aList[i] = num; // more validation could be added here
     i++;
}
if( i == SIZE )
    cout << "The array is full!\n";
```

Copying an Array to Another Array – one element at a time (assume the two arrays have the same size)

```
int a[5] = { 98, 76, 84, 100, 95 };
int b[5];

for( int i = 0; i < 5; i++ )
    b[i] = a[i];
```

Writing `b = a;` does not work. Why?

In C++, the name of an array represents the address of its first element, and that is a constant. Since these addresses are constants, they may be used but not changed. However, we do not wish to change the address of `b`, just to copy the contents of `a` to `b`, and this must be done one element at a time.

Arrays and Functions

Passing one element of an array – by value

```
int a[5] = { 98, 76, 84, 100, 95 };

fun( a[2] );
cout << a[2];
// Output: 85 84
```

```
void fun( int num )
{
    num++;
    cout << num;
}
```

Passing one element of an array – by reference

```
int a[5] = { 98, 76, 84, 100, 95 };

fun( &a[2] );
cout << a[2];
// Output: 85 85
```

```
void fun( int &num )
{
    num++;
    cout << num;
}
```

Passing the whole array // not really! Arrays are always reference parameters:
// the function has access to the original array

```
int a[5] = { 98, 76, 84, 100, 95 };

fun( a );
for( int i = 0; i < 5; i++ )
   cout << list[i];

// Output: 98 76 84 100 95
//         99 77 85 101 96
```

```
void fun( int list[] )
{
    for( int i = 0; i < 5; i++ )
    {
        cout << list[i] << " ";
        list[i]++;
    }
    cout << endl;
}
```

Remember:
a is the address of the first element!

Using `const` Array Parameters - to prevent a function from changing the array that is passed to it as an argument

```
void printList( const int list[], int listSize )
{
    for( int i = 0; i < listSize; i++ )
    {
        cout << list[i] << " ";
    }
    cout << endl;
}
```

Calculate the sum of the elements in an array:

```
double calcSum( const double list[], int listSize )
{
    double sum = 0;
    for( int i = 1; i < listSize; i++ )
        sum += list[i];
    return sum;
}
```

Find the location of the smallest/largest element in an array:

```
int getMin( const int list[], int listSize )
{
    int min = 0; // index of the smallest element
    for( int i = 1; i < listSize; i++ )
    {
        if (list[i] < list[min])
            min = i;
    }
    return min;
}
```

Searching an Array – find the location of a target in an array.

Sequential Search – also called **Linear Search** simply checks each element in the array in sequence, until either a match is found or the end of the array is reached. There are different variations of this algorithm.

C++ Implementation of the Sequential Search Algorithm

```
/*  Look for a target in an unordered list of size elements.
        Pre:  list – an unordered list
              size – the number of elements in the list (>=0)
              target – data to be located
        Post:  if found, return the first matching index, otherwise return -1
*/
int sequentialSearch ( int list[], int size, int target)
{
     int pos = -1; // not found
     int i = 0;
     while( i < size && target != list[i] )
          i++;
     if ( i < size )
         pos = i;
     return pos;
}
```

// Here is another implementation of this algorithm (it uses two return statements)

```
int sequentialSearch ( int list[], int size, int target)
{
     int i = 0;
     while( i < size )
     {
          if ( target == list[i] )
              return i; // found!
          i++;
     }
     return -1;  // not found!
}
```

An example of using/calling the sequential search function:

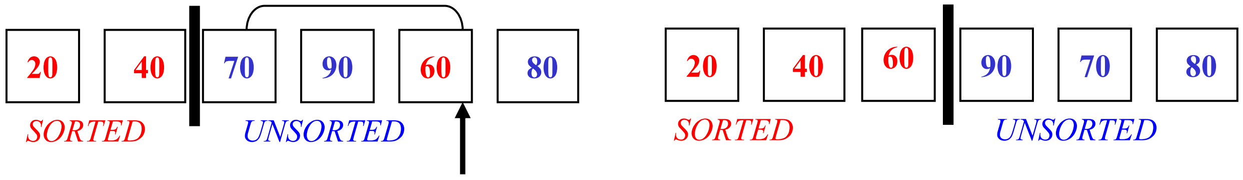
```
    int scoreList[100];
    int numScores, target, pos;

    getScores( scoreList, numScores );
    cin >> target;
    pos = sequentialSearch( scoreList, numScores, target);
    if( pos != -1 )
        cout << target << " found at index ", pos << endl;
    else
        cout << target << " not found\n";
```

Sorting an Array – rearrange the elements of an array in a certain order. For instance a list of scores can be sorted from smallest to largest (ascending order) or from largest to smallest (descending order).

Simple Sorting Algorithms – Let's say we have to sort a list of **`n`** numbers in ascending order. During the sorting process, the list is logically divided in two parts: the first one, that is sorted, and the second one that is not sorted. In the beginning the sorted list is empty and the unsorted list contains the rest of the elements. At each step in the sorting process we take one element from the unsorted list and move it into the sorted list. How do we do this depends on which algorithm we want to use.

Selection Sort – at each step we choose the smallest element in the unsorted part of the list and swap it with the first element in the unsorted part of the list.



C++ Implementation of the Selection Sort Algorithm

```
/*      Sorts by selecting smallest element in unsorted portion of array and exchanging it
        with element at the  beginning of the unsorted list.
        Pre:    ary  - has data; size - number of elements in array
        Post:   ary rearranged smallest to largest
*/
void selectionSort ( int ary[], int size )
{
        int last = size - 1;
        int curr, i, small; //  indices
        int hold;

        for (curr = 0; curr < last; curr++)
        {
            //  look for the location of the smallest value
            small = curr;
            for (i = curr + 1; i <= last; i++)
               if (ary[i] < ary[small])
                   small = i;

            //  swap
            hold        = ary[small];
            ary[small] = ary[curr];
            ary[curr]   = hold;
        }
}
```