Arrays

CS1A

- Why use arrays?
- What are arrays?
- Declaring Arrays
- Using Arrays
- Parallel Arrays

Why do we need Arrays?

```
int main ()
{
   int item0, item1,item2;
   int sum;

   cout << "Enter 3 integers: ";
   cin   >> item0 >> item1 >> item2;

   sum = item0 + item1 + item2;

   cout << "The sum of the numbers = " << sum << endl;
   cout << "the numbers in reverse order are ";
   cout <<  item2 << " " << item1 << " "<< item0 << endl;

   return 0;
}
```

What do we do if we have to store more than 1 piece of information of the same type?

We declare different variables.

What are arrays?

A collection of data of the same type
- A special group of variables

Arrays can hold
- many pieces of data
- all have the same data type and name,
- but different values.

- "Aggregate" data type
  - Means "grouping"

- Used for lists of like items
  - Test scores, temperatures, names, etc.
  - Avoids declaring multiple simple variables
  - Can manipulate "list" as one entity

Simple & Composite data types

- Simple Data Types
  - Data types that store only one piece of information
  - What we have been using thus far
  - short, int, long, float, double, char, bool
- Structured / Composite Data types
  - Each data item is a collection of other data items

Declaring an Array

Syntax
dataType arrayName[*number_of_elements*];

Declaring an array allocates the memory for the array

Example

```
int scoresAr[5];   // declares an array of 5 integers
                   // named score
```

The number of elements can be…

- a literal (e.g. 5)

```
int scoresAr[5]
```

- Or a named constant

```
const int NUMBER_OF_TESTS = 5;
int scoresAr[NUMBER_OF_TESTS];
```


Elements and Indexes

Each individual item in an array is called an element

- Each element has an index associated with it

An index is a number which indicates which value we are referring to

Example

scoresAr[0] ← the first element in our array

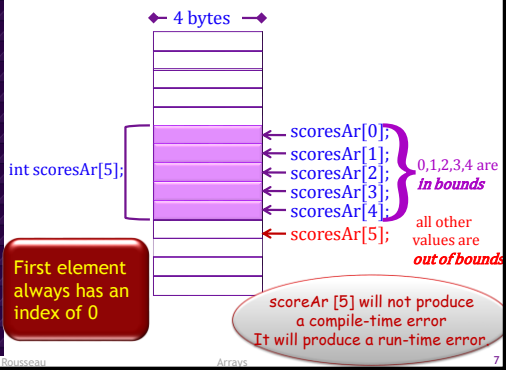
Index

The first element is ALWAYS zero

So if we have 5 elements our indexes would be 0,1,2,3,4
or scoresAr[0], scoresAr[1], scoresAr [2], scoresAr[3], score[4]

Note:
The brackets specify the size in the declaration and the subscript or index anywhere else

Memory and Arrays



Indexes

An index can be anything that evaluates to an integer

- A literal
  - e.g. scoresAr[4]
- A variable or a constant
  - e.g. scoresAr[i]
- An expression
  - e.g. scoresAr[2 * i - j]

Initializing Arrays

Simple variables can be initialized at declaration:

```
int price = 0;    // 0 is initial value of after declaration
```

This is not considered good style → do not do this in this class please

or equivalently in the code

```
int price;
price = 0;
```

- Arrays can be initialized at declaration as well:

```
int itemsAr[3] = {2, 12, 1};
```

or equivalently in the code :

```
int itemsAr[3];
itemsAr[0]  = 2;
itemsAr [1] = 12;
itemsAr [2] = 1;
```

Initializing Arrays (2)

if you have more elements than values in the list then the extras at the end default to 0

```
int itemsAr[5] = {2,12,1};   ⇔   int itemsAr[5] = {2,12,1,0,0};
```

This is not recommended!

This is okay!

You can also initialize all the elements to 0 using this method

```
int itemsAr[5] = {0};
```

This is okay!

if you have more values than elements specified then you will get a compiler error

```
int itemsAr [5] = {2,12,1,2,9,5}; → compiler error
```

if you don't specify the number of elements it will default to the number of values in the list

```
int itemsAr [] = {2,12,1,2,9,5}; → children will default to 6 elements
```

This is not recommended!

Initializing using a FOR loop

NOTE: For Loops are very useful when you need to access every element in an array.

```
#include <iostream.h>
int main()
{
    float gpasAr[5];  // an array holding 5 grade point averages - INP.& OUT.

    // load the array from the keyboard
    for(int index = 0; index < 5; index++)
    {
        cout << "Enter the gpa for student " << index + 1 << ": ";
        cin  >> gpasAr[index];
    }
    // output the contents of the array
    cout << "\n\nStudent Grade Point Averages\n";

    for(int ind = 0; ind < 5; ind++)
    {
        cout << "\nGPA for student " << ind + 1 <<": " << gpasAr[ind];
    }
    return 0;
}
```

This loop initializes the array

This loop outputs the array

Example

Do a desk check with Inputs → 5, 10, 15

```
int main ()
{   int itemsAr[3];
    int sum, index;

    sum=0;

    for (index = 0; index < 3 ; index++)
    {
          cout << "Enter an integer: ";
          cin  >> itemsAr[index];
          sum = sum + itemsAr [index];
    }

    cout << "The sum of the numbers = " << sum << endl;
    cout << "The numbers in reverse are:  ";
    for (index = 2 ; index > -1 ; index-- )
    {
          cout <<  itemsAr [index] << ",  ";
    }
    return 0;
}
```

itemsAr index sum

0 1 2

Output

What if we want to have 10 items? What changes will we have to make?

Defining a Constant as Array Size

- Always use defined/named constant for array size
- Example:
  const int AR_SIZE = 5;
  int scoresAr[AR_SIZE];
  - NOTE: Can't do this with a variable
- Improves readability
- Improves versatility
- Improves maintainability

Note that it must be declared as an integer!

The number of elements *must be known* at compile time

Using a constant is considered a best practice

Instead of all those changes we can use a constant and just change the constant.

```
int main ()
{
    const int AR_SIZE =10;
    int itemsAr [AR_SIZE];
    int sum, index;

    sum=0;

    for (index = 0 ; index < AR_SIZE; index++)
    {
          cout << "Enter an integer: ";
          cin   >> itemsAr[index];
          sum = sum + itemsAr[index];
    }

    cout << "The sum of the numbers = " << sum <<endl;
    cout << "The numbers in reverse order are:  ";

    for (index = AR_SIZE-1; index > -1 ; index--)
    {
          cout <<  itemsAr[index] << ", ";
    }
    return 0;
}
```

Works well if they have to enter 10 items

Note that this is AR_SIZE-1

Initializing using while loops

- Need to check for out of bounds as well as user controlled LCV

```
...
    int itemsAr[AR_SIZE];
    int index;
    int intInput;

    index = 0;

    // load the array from keyboard input
    cout << "Enter the item (enter -1 when done): " ;
    cin   >> intInput
    while (intInput != -1 && index < AR_SIZE)
    {
       itemsAr[index] = intInput;

       cout << "Enter the item (enter -1 when done): " ;
       cin   >> intInput
       index++;
    }
```

Initialize Both LCVs

Change Both LCVs

Need to make sure we don't Go out of bounds

What if we want to read in from a file?

Initializing from a File

- Need to check if we are not at the end of our input file
  while (inFile) will handle this
  - inFile will return False if it is at the end of file
- We need to check 2 things then
  - While we are not at the end of the file
  - AND while we are still within bounds of our array

```
...
    int index;
    // load the array from the keyboard
    index = 0;
    while (inFile && index < AR_SIZE)
    {
        cout << "Enter the gpa for student " << index + 1 << ": ";
        cin  >> itemsAr[index];
        index++;
    }
```

What is wrong with this code?

Should be reading in from inFile

Don't need to prompt The file

Initializing from a File

- This is more appropriate

```
...
    const int AR_SIZE = 5;

    int itemsAr [AR_SIZE] = {0};
    int index;
    ifstream inFile;

    inFile.open("input.txt");
    // load the array from a file
    index = 0;
    while (inFile && index < AR_SIZE)
    {
        inFile >> itemsAr [index];
        index++;
    }
    inFile.close();
```


Common Errors

REMEMBER: Array indexes always start with zero!

- Zero is "first" number to computer scientists

- C++ will "let" you go out of range
  - Unpredictable results
  - Compiler will not detect these errors!
- Up to programmer to "stay in range"

More on arrays

itemsAr

0	1	2
5	10	15

- What if you want the last element in an array?

  cout << itemsAr [AR_SIZE - 1]; → this will output 15 → AR_SIZE = 3

- What if you want to know the size of the array
  - *sizeof()* outputs the # of bytes - each int is 4 bytes

  cout << sizeof(itemsAr);

  → this will output 12 for our array is itemsAr [3]

  → If you want the # of elements you need to use

  cout << sizeof(itemsAr)/sizeof(itemsAr [0]);

  → this will output 3 for our array is itemsAr [3]

NOTE: sizeof() will not work properly if you are passing an array into a function (it will give you the size of the address)
It is best to send the array size in functions → more on this later

Searching an array for one instance

```
int itemsAr[AR_SIZE] = {0};
// INPUT - read input from a file into the array
index = 0;
while (inFile && index < AR_SIZE)
{
    inFile >> itemsAr[index];
    index++;
}
// SEARCH - for searchItem in the array
searchItem = 10;
index  = 0;
found = false;
while(index  < AR_SIZE &&  !found)
{
    if (itemsAr[index] == searchItem)
    {
        found = true;
    }
    else
    {
        index++;
    }
}
```

This loop initializes the array

This loop searches the array

NOTE: we should make sure we haven't exceeded the size of our array AND if we are looking for one element we should stop searching when it is found

NOTE: When this loop terminates the index will indicate where in the array searchItem was found.
If the index == MAX_ITEMS we know it was not found

INPUT FILE
3 7 10 2 11

Searching an array for the # of instances

```
const int AR_SIZE = 6;
int itemsAr[AR_SIZE ] = {3, 7, 10, 2, 10, 12};
int index, searchItem, instances;

instances = 0;
searchItem = 10;
for(index = 0; index  <  AR_SIZE; index++)
{
    if (itemsAr[index] == searchItem)
    {
        instances++;
    }
}
```

itemsAr

0	1	2	3	4	5
3	7	10	2	10	12

AR_SIZE	index	instances	searchItem
6			

Let's do a deskcheck

NOTE: We can use a for loop because we must search the entire array.

Instances will indicate how many times it was found

No Aggregate Operations on Arrays

Aggregate Operation → any operation that manipulates the entire array as one component

Example

To copy the elements from one array to another you can't just say

```
int firstArray[5] = {1,2,3,4,5}
int secondArray[5];
secondArray = firstArray;     ← this will produce a compiler error
```

Instead you can use a loop

```
for (int index = 0; index < 5; index ++)
 {
      secondArray[index] = firstArray[index];
 }
```


Aggregate Operations – Ex 2

- Suppose you want to read in a bunch of values into your array

```
cin >> firstArray;   ← this is illegal in C++ (except c-strings)
```

Instead you would use a loop

```
while (<non-terminal value exp> &&  index < AR_SIZE)
      cin >> firstArray[index];
```

- Other aggregate operations not allowed

```
if(arrayOne == arrayTwo)   ← comparison - illegal
cout << arrayOne;   ← output - illegal (except C-strings)
arrayTwo = arrayTwo - arrayOne; ← arithmetic - illegal
return arrayOne; ← returning an entire array - illegal
```


Base Address

An array stores the address of the first element in the array → this is called the *base address*

- When you declare an array the computer remembers
  - The name of the array
  - The data type
  - The base address
  - And the number of elements
- To access item[2] the computer calculates the address of item 2
  - Base address + (4 * 2) → 4 bytes 3rd element
- This is why aggregate operations don't do what you'd expect

Using Arrays in Functions

- As arguments to functions
  - Indexed variables
    - An individual "element" of an array can be function parameter

Example

```
AddTwoInts(int num1, int num2);  // prototype
...
int intArray[5] = {1,2,3,4,5};
sum = AddTwoInts(intArray[1], intArray[2]);
```

An Array cannot be a return value in a function!

Using Arrays in functions

Sending the entire array as a parameter

- All array elements can be as "one entity"
- Arrays can be passed by reference ONLY
  - ← value would take too much memory
  - Since pass by reference is the only option we don't use the &
  - The size of the array is omitted
- You cannot return an array
  - You can modify the value of the elements in an array
- When an array is used as a parameter the base address is sent

Example

```
void InitializeIntArray(int listAr[], const int LIST_SIZE)
{
   int count;
   for (count = 0; count < LIST_SIZE; count++)
        listAr[count] = 0;
}  // This function will initialize an int array of any size
```

If you do not want your array to be changed in a function how should you pass it?

Using Arrays in functions (2)

If you don't want your array modified by a function
→ send it by constant reference

Example

Note: const here

```
int SumArray(const int LIST_AR[], const int LIST_SIZE)
{
      int index;
      int sum;

      for (index = 0; index < LIST_SIZE; index++)
      {
         sum = sum + LIST_AR[index];
      }
      return sum;
}
```

Passing a constant when you don't need to change the array is considered a best practice

C-Strings are special arrays

- C++ treats arrays of type char a little differently
- 'A' ≠ "A"
  - 'A' ← represents the character A
  - "A" ← represents 2 characters A & \0

char name[16] = {'P', 'e', 't', 'e', '\0'}; ⇔ char name[16]="Pete";
char name[16] = "Pete"; ≠ char name[]= "Pete";

- No aggregate operations with c-strings → they are arrays
  - name = "Pete"; ← this is illegal

Special C-String Operations

- strcpy(s1, s2)
  - Copies the string s2 into the string variable s1
  - The length of s1 should be *at least* as large as s2
- strncpy(s1, s2)
  - Same as strcpy, but checks the size of the destination string
  - Stores either the length of s2 unless it is too large then stores what will "fit" into s1
- strcmp(s1, s2)

Return value	if ASCII VALUES are such that
0	s1 == s2
Integer < 0	s1 < s2
Integer > 0	s1 > s2

- strlen(s)
  - Returns the length of string s (excluding the null terminator)

Examples

```
strncpy(name, "Pete McBride");
   //puts the value "Pete McBride" into the c-string name

strncpy (name2, name);
    //puts the value of the c-string name to into the c-string name2

int val;
val = strlen("Happy camper");
   // returns the value 12 and stores it in val (doesn't count \0)

val = strcmp("Pete ","Steve");
   // returns a value < 0

val = strcmp("Steve", "Pete ")
   // returns a value > 0
```

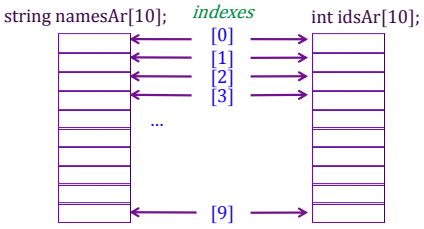
Parallel Arrays

CS1A

Parallel Arrays

- When you have more than one data type to track in arrays, but they are related
- For example, if you want to track related data such as a
  - Name and id
- These are different data types, but related data
- We could create 2 parallel arrays to represent this data

Parallel Arrays Example

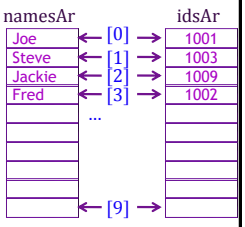


Parallel Arrays Example

So if we had data like this:

Name	ID#
Joe	1001
Steve	1003
Jackie	1009
Fred	1002

We would first declare two arrays of the same size but different data types




```
const int AR_SIZE = 10;

string  namesAr[AR_SIZE];
int     idsAr[AR_SIZE];
```

Now our arrays can be associated by their index numbers

Reading in Parallel Arrays

```
const int AR_SIZE = 10;

string namesAr[AR_SIZE];
int    idsAr[AR_SIZE];
int    index;

index = 0;

while (inFile && index < AR_SIZE)
{
    getline(inFile, namesAr[index]);
    inFile >> idsAr[index];
    inFile.ignore(1000, '\n');
    index++;
}
```

Remember you should use getline with strings

What do we need to add to use files?