A FIRST LOOK AT ARRAYS

Today

- How we can use arrays to hold sets of related data.
- Common patterns of handing data:
  - Finding the largest element
  - Summing up elements
  - Counting elements with some feature.
- You should read these notes in conjunction with the program in `arrays.cpp`

What is an array?

- You can think of an array as a set of variables which are grouped together, all using the same identifier.
- Just as
  ```
  int a;
  ```
  declares an integer variable with the name a, then
  ```
  int b[5];
  ```
  declares an *array* of 5 integers, with the name b.
- The square brackets [ ] are the crucial bit of syntax, telling the compiler it is dealing with an array.
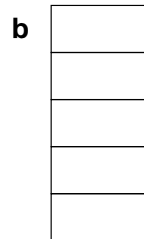

- Whereas
  ```
  int a;
  ```
  reserves space for one integer in memory and associates the name a with it:

  a ☐

the declaration

```
int b[5];
```

reserves space for five integers in memory right next to one another.

b

- Elements of the array b are just integers, and we can do exactly the same things to them that we can do to integers.
- The only difference is how we *address* them.
- While we can assign a value to a by:

  ```
  a = 5;
  ```

  To do the same to one of the *elements* of b, we have to specify which element it is. For example:

  ```
  b[1] = 5;
  ```

- Thus all of the following are legal operations:

  ```
  b[1]  += 2;
  b[2]  =   7 % 3;
  b[3]  =   b[2] - 5;
  b[4]  =   b[1]/b[3];
  ```

- One thing to be careful of is the limits on the *index*, that is the number inside the square brackets [ ].
- The first element of an array always has index 0.
- So the first element of b is:

  ```
  b[0]
  ```

  and, since b has 5 elements, the last element of b is:

  ```
  b[4].
  ```

- Every C++ programmer forgets this from time to time.

- Arrays are useful when you want to store lost of data in memory.
- If I want to use 3 integers in my progran, I'll just declare 3 different variables.
- If I want to use 30,000 integers in my program, it is a lot easier to use an array.
- Arrays also go very nicely with for loops.
- For example, here is code from arrays.cpp which reads information from a file, and puts part of it into two different arrays.
- The arrays are diseases[] and ages[]

```
for(counter = 0; counter < 7 ; counter++)
    {
        infile >> idNumber;         // Read data in from
        infile >> age;              // file.
        infile >> disease;
        infile >> zipCode;

        // Store disease and age data in arrays

        diseases[counter] = disease;
        ages[counter]     = age;

        printRecord();              // Print data
    }
```

- Once the information is in the array, we can do all kinds of stuff with it.
- We can, for example, print out the values in reverse order:

```
cout << endl << "Diseases in reverse order" << endl;

for(counter = 6; counter >= 0;  counter--)
    {
        cout << diseases[counter] << endl;
    }
```

- Another thing we can do is to count up how many times we find some value in the array.
- The value we are looking for is in the variable disease

```
for(counter = 0; counter < 7;  counter++)
    {
      if(diseases[counter] == disease)
        {
          numberOfDiseases += 1;
        }
    }
```

- This is one common pattern of using a loop to *summarize* some information.

- A slightly different summarization would be look look for oldest patient. (The patient with the largest age.)
- We collect the largest age in the variable oldest:

```
int oldest = 0;

for(counter = 0; counter < 7;  counter++)
    {
      if(ages[counter] > oldest)
        {
          oldest = ages[counter];
        }
    }
```

- We could also store the value of counter that corresponds to the highest age, and then we could look at other aspects of the oldest patient.

- A third, and final summarization is to add up, and then compute the average of, the patient ages:

```
int sumOfAges = 0;

for(counter = 0; counter < 7;  counter++)
    {
      sumOfAges += ages[counter];
    }

cout << endl << "Average age is ";
cout << endl << sumOfAges / 7.0 << endl;
```

- We divide by 7.0 in order to force the result to be a decimal fraction.

- If we didn;t do this, we would be dividing one integer by another, and we'd get an integer result.

Variable precision

- This last example has us printing out a decimal fraction.

- In order to print out decimal fractions neatly, we can specifcy how many decimal places we want to print out.

- In this example:

```
cout.setf(ios::fixed, ios::floatfield);
cout.precision(2);

cout << endl << "Average age is ";
cout << endl << sumOfAges / 7.0 << endl;
```

- The first two lines tell cout to print floating point numbers (in other words decimal fractions) with fixed width, and to use two decimal places.

Constants

- One thing to notice with all of these arrays is that we have been writing 7, the size of the array, a lot.

- What would happen if we decided we now needed the array to hold 10 elements?

- Well, we'd have to make lots of changes.

- Each change gives us the chance to make a mistake.

- There is a way to reduce the number of changes, and that is to use a *constant*.

- We define a constant using the keyword const:

```
const int LENGTH = 7;
```

- By convention we give a constant a name that is all capitals.

- We can then use LENGTH wherever we need the number 7:

```
int diseases[LENGTH];

for(counter = 0; counter < LENGTH; counter++)
    {
      sumOfAges += ages[counter];
    }
```

and so on.

Summary

- In this lecture we talked about four things.

- First we briefly introduced arrays.

- (We will talk about arrays a lot more in the future).

- Second we talked about some common patterns of programming.

- Third we talked about formatting output, in particular controlling the precision of numbers.

- Finally we talked about constants.