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 programs in `arrays.cpp` and the patient record example in `patient.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

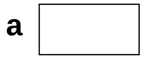
  ```
  int b[5];
  ```

  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;
  ```

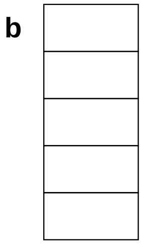
  reserves space for one integer in memory and associates the name `a` with it:



the declaration

`int b[5];`

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



- 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]`.
- *In computer science we start counting at 0*.
- Every C++ programmer forgets this from time to time.

Declaring and initialising

- We already talked about how to declare an array.
- For example:

  `int age[8];`

  declares an array of 8 `int`s called `age`.
- We can have arrays of any data type. For example:

```
double numbers[5];
char   sentence[30];
```

declares an array of 5 `double`s called `number`, and an array of 30 `char`s called `sentence`.

- We can initialise these arrays when we declare them, just as we can for other kinds of variable:

```
int age[8] = {5, 10, 15, 20, 25, 30, 35, 40}
```

- Initialising like means that:

```
cout << age[0] << endl;
cout << age[6] << endl;
```

will produce:

```
5
35
```

since, as we mentioned before, the first element in the array has subscript 0.

- If instead we initialised with:

```
int age[10] = {5, 10, 15, 20, 25, 30}
```

then

```
cout << age[0] << endl;
cout << age[6] << endl;
```

would produce:

```
5
0
```

since any values we do not explicitly assign in the initialisation will be set to 0.

- Declaring:

```
double number[5]    = {5.0, 10.1, 20.2, 30.3};
```

would set `number[4]` to 0.

- Declaring:

```
char   sentence[30] = {’H’, ’e’, ’l’, ’l’, ’o’}
```

would set all elements of `sentence` after the `’o’` to `’ ’`.

Handling arrays

- Arrays are commonly handled using `for` loops.
- Thus, to print out the array `age`, we might use:

```
for(counter = 0; counter < 8; counter++)
    {
      cout << age[counter] << "  ";
    }
```

- This will take each element of `age` in turn and send it to `cout`.
- Note that the maximum value of the subscript `counter` is 7.

- In C++ it is very important to be careful with the maximum value of the subscript of an array.
- If you *overflow* an array, for example by doing:

  ```
  age[10] = 30;
  ```

  this will not generate an error.
- However, it may cause your program to crash in an unexpected way.

- Often, when handling arrays, we want to use the same subscript to access two or more arrays.
- For example:

  ```
  for(counter = 0; counter < 5; counter++)
    {
      number[counter] = 2 * age[counter];
    }
  ```
- This replaces each element of `number` with double the corresponding element of `age`

Modifying elements in an array

- Of course, since each element of `age` is an integer, we can do to each element, exactly what we can do to an integer.
- All we have to do is remember how to address each element of the array, using a subscript.
- For example:

  ```
  for(counter = 0; counter < 8; counter++)
    {
      age[counter]++;
      age[counter] = (age[counter] * 2)/3;
    }
  ```
- What does this do to each element of the array `age`?

Functions and array elements

- Since elements of `age` are integers, we can call functions on them
- If we have the function

  ```
  int timesTwo(int number)
  {
    return 2 * number;
  }
  ```

  we can call this on the third member of `age` like so:

  ```
  timesTwo(age[2]);
  ```

Manipulating subscripts

- The subscript that we use to identify elements of an array is also just an integer.
- So we can use arithmetic expressions as subscripts, *so long as they evaluate to integers*.
- For example

```
cout << age[2+1];
cout << age[counter - 2];
cout << age[age[0]];
cout << age[timesTwo(age[2])];
```

A bigger example

- The program `patient.cpp` is a larger example of using arrays.
- The program is a simple patient record system, which reads information on patients in from a file and puts part of it into two different arrays.
- The program then manipulates the arrays in a few different ways.
- The arrays are `diseases[]` and `ages[]`.
- In this example, they are both 7 elements long

```
int age[7];
int diseases[7];
```

```
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.

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

- This lecture has looked in some detail at arrays.
- We examined the declaration and initialization of arrays.
- We looked at handling arrays using `for` loops, and by playing with subsscript values.
- We looked at different things one can do with array elements.
- In particular, we looked at some different common things we do with arrays.
- Finally we looked at a common use for `const`.