C++ Week 13

Two-dimensional vectors, typedef, C-style arrays and C-strings, character-level I/O, precision of doubles

Two-dimensional vectors

A two-dimensional vector in C++ is just a vector of vectors. For example, you could define a two-dimensional vector of integers as follows:

```
vector<vector<int> >   v2;
```

Note the space between <int> and the second >. If you have them next to each other, as in >>, the compiler interprets it an operator and flags it as an error.

This definition gives you an empty two-dimensional vector. If you wanted to grow it with `push_back`, you would have to push back a one-dimensional vector, not a single `int`. For example:

```
vector<int> v(5);
v2.push_back(v);
```

To initialize a two-dimensional vector to be of a certain size, you can first initialize a one-dimensional vector and then use this to initialize the two-dimensional one:

```
vector<int> v(5);
vector<vector<int> >   v2(8,v);
```

or you can do it in one line:

```
vector<vector<int> >    v2(8, vector<int>(5));
```

You can picture v2 as a two-dimensional vector consisting of eight rows with five integers in each row

You refer to individual elements of a two-dimensional vector by using two subscripts. (Since v2[0], for example, is itself a vector, you can subscript it – v2[0][0].) The following sets the third element of the fifth row of v2 to 99:

```
v2[4][2] = 99;  // remember that the first element of the first row is v2[0][0]
```

The following procedure would display a two-dimensional vector of `int`:

```
void display (const vector<vector<int> >& vy)
{  for (int i = 0; i < vy.size(); i++)       // loops through each row of vy
   {  for (int j = 0; j < vy[i].size(); j++) // loops through each element of each row
          cout << " " << vy[i][j];           // prints the jth element of the ith row
      cout << endl;
   }
}
```

Defining datatypes with typedef

You can give your own names to datatypes using the `typedef` keyword. The following code creates a datatype called `Words`, which is a vector of `string`.

```
typedef vector<string> Words;  //note the capitalisation: this is a convention
```

This looks just like a definition of a variable called `Words` except that it has the word `typedef` on the front. This means that we have *not* defined a variable - we do not now have a vector called `Words`. We have defined a datatype. As well as being able to define variables of type `int` or `string` or `vector` and so on, we can now also define variables of type `Words`.

Once you have defined a datatype in this way you can use it in the same way as any other datatype to declare variables of that type. For example:

```
 Words w; // creates a vector of strings, called w
```

Defining new types can help to simplify the notation of a multidimensional vector. Let's use a `typedef` to create a 2-D vector of integers:

```
typedef vector<int> Vint;   // creates the datatype Vint
Vint v(5);                  // creates a vector v of five integers, all zero
vector<Vint> vx(4, v);      // creates a vector of 4 Vints called vx

vx[1][2] = 99;   // vx[1] is a vector, so we can subscript it
```

	v[0]	v[1]	v[2]	v[3]	v[4]
vx[0]	0	0	0	0	0
vx[1]	0	0	99	0	0
vx[2]	0	0	0	0	0
vx[3]	0	0	0	0	0

```
vector<Vint> vy(10);                     // creates a vector vy of 10
                                         // Vint vectors, all empty
Vint    v(5);
for (int i = 0; i < 10; i++)
   vy[i] = v;                            // makes each vy a vector of
```

```
                                       // 5 zeroes - note that you can use assignment with vectors
vy.push_back(v);                       // You can push a Vint onto the end of vy (or pop one off)
vy[3].push_back(66);                   // You can push an int onto the end of one of the component Vints
vy[4].pop_back();                      // (or pop one off) - the component Vints do not all have to be the same length
```

Arrays

C-style arrays are less flexible than C++ vectors, but they are still occasionally useful. An array can be declared as follows:

```
int a[6];
```

creates an array of six integers, all uninitialized (like ordinary `int` variables).

- Arrays are not objects in the C++ sense, so they do not have member functions. If `a` is an array, you cannot have things like `a.size()` or `a.push_back(99)`
- The size of an array is calculated by the compiler and cannot be set or changed at runtime.

Note one consequence of the second point – the size must be known at compile time. People are often tempted to write something like this:

```
void proc(int len)
{       int a[len];             // Can't be done!
        ...
}
```

The array is local to the procedure and the programmer is trying to set the length of the array with a parameter. But this value is only passed to the procedure at run time. The compiler cannot set aside the right amount of storage for this array at compile time. This cannot be done.

A curious feature of arrays in C (and therefore in C++) is that the name of an array is really the name of a pointer to the first element of the array. One consequence of this is that, when you pass an array as parameter, you can define it like this:

```
int somefunc(int a[])
```

or like this:

```
int somefunc(int* a)
```

The two are equivalent. Then, inside the function, you could refer to a[0], a[5] etc, or, if you preferred, to *a, *(a+5) etc.

(In fact, since a[5] is equivalent to *(a+5), which is equivalent to *(5+a), you could actually write 5[a], which looks very weird.)

Since what is passed is the address of the first element, it follows that arrays are always passed by reference (even though there is no &).

Another consequence of this is that the function knows only what type of thing the array consists of and where (in memory) it begins; it does not know how long it is. So, when you pass an array to a function, you need also to pass its length.

When we passed vectors as parameters, the procedures or functions that used them typically contained lines like this:

```
int func(const vector<int>& v)
{       for (int i = 0; i < v.size(); i++)
                ...
}
```

But we can't do that with an array since there is no `a.size()` function. So, if we want to pass an array as a parameter, we also have to pass the length as a separate parameter, like this:

```
int func(int a[], int alen)
```

We have to pass the array length, as there is no other way for the function to know how long it is. (You can put a number inside the square brackets if you like, for example `int func(int a[6], int alen)`, but the compiler ignores it.)

Since arrays are always passed by reference (even though you don't include an &), a procedure or function that takes an array as a parameter can potentially make changes to the array. Sometimes you don't want this to happen and you can prevent it by adding a `const` to the parameter, like this:

```
int func(const int a[], int alen)
```

Now the function should treat `a` as though it were composed of `const int`. (At the least you should get a warning from the compiler if the function contains code that might change a value in `a`.)

sizeof

There is a `sizeof` operator in C and C++ (yes, it's an operator, not a function, despite its name). `sizeof some_variable` gives you the size of `some_variable` *in bytes* (a reminder that C is a relatively low-level language). For example, if `d` is a `double`, `sizeof d` gives 8 on most current machines. `sizeof` can also take a type as its argument, in which case you put the argument in parentheses, as in, for example `sizeof (double)`.

When applied to an array `sizeof` gives you the size of the array in bytes. So, if `ax` was an array of six integers, `sizeof ax` would give 24 (four bytes per int, times six). It is therefore possible to calculate the number of elements in an array `ax` (giving us something like the `size` function for vectors) with the rather cumbersome formula `sizeof ax / sizeof ax[0]`.

Can we not use this to calculate the number of elements in an array that has been passed to a function, thus avoiding the necessity of passing the length as a separate parameter? Sadly no. Recall that, in a function that begins `int func(int ax[])`, what gets passed to the function is actually a pointer to the first element, and, if you get `sizeof ax` inside the function, you will get the size of a pointer – usually four bytes.

Initialising arrays (and, indirectly, vectors)

One advantage of arrays over vectors is that you can initialize them with a set of different values as follows:

```
int a[6] = {2, 5, -9, 14};
```

which produces an array of 6 elements as follows:

2	5	-9	14	0	0

The last two elements are initialized to zero as a by-product of the initialization. (If you explicitly initialize any elements, even just the first, the rest get initialized to zero.) The argument in square brackets is optional, and if omitted, the array is populated only with the elements specified in curly brackets. In this example, if you left out the 6, you'd get an array of 4 elements.

This feature of arrays provides a back-door route to initializing vectors. One of the constructors in the vector class takes two pointers (memory addresses) as its parameters. The first of these is to the element we want first in our vector, the second is to the element *after* the last one that we want. Since `a, a+3` and so on are pointer values, we can use an array to initialize a vector as follows:

```
int a[6] = {2, 5, -9, 14};
vector<int> v(a, a + 6);
```

This code will populate the vector with the first 6 elements of the array. Note that the second argument of the initialization (`a + 6`) is the address of a (here non-existent) element **after** the end of the array. As another example, the following code:

```
int a[] = {1, 5, 10, 50, 100, 500, 1000};
vector <int> v (a + 1, a + 5)
```

populates the vector with the values 5, 10, 50 and 100.

C-strings

Just as C++ strings are like *vectors* of `char`, C-strings are *arrays* of `char`. For example:

```
char cs[4] = "C++"; // four characters to allow for the null byte at the end
```

C-strings always occupy one byte more than their apparent length because a C-string always ends with a null byte. (As a literal, a null byte is written as `'\0'`.) All the functions in C that manipulate strings rely on the presence of this null byte.

If you are providing a string literal as initialization, you need not specify how long the array is to be - you can leave it to the compiler to work it out from the literal:

```
char csx[] = "mary smith";
```

The compiler determines how long the array has to be to hold the string and the null byte.

C-strings, like other arrays, are always passed by reference. However, we do not need also to pass the length in the case of C-strings because we can find where it ends by looking for the null byte. Consider the following procedure that receives an array containing a C-string and converts all the spaces it contains into asterisks:

```
void stars (char cs[])
{  for (int i = 0; cs[i] != '\0'; i++)
      if (cs[i] == ' ')
            cs[i] = '*';
}
```

or, thinking of the parameter explicitly as a pointer (which is what it is):

```
void stars (char* cs)
{  for (char* p = cs; *p != '\0'; p++)
      if (*p == ' ')
            *p = '*';
}
```

Converting strings to C-strings

C-strings are not equivalent to C++ strings. For example:

```
char ch[] = "E"; // an array of two bytes, an E and a \0
string s = "E";  // a vector-like container of character data, containing an E
```

In some C++ compilers, certain functions cannot handle C++ strings as arguments. In particular the `open()` function for an `ifstream` on many compilers takes a C-string as its argument (it can also take a string literal, but that is because a string literal is a C-string). If, for example, you ask a user for the name of a file to be opened and store the filename as a string, you must convert it before passing it to the `open()` function as follows:

```
string filename;
cin >> filename;
infile.open(filename.c_str());
```

Similarly, the function `atoi()` (from the `cstdlib` library) only takes C-strings. This function is used to convert a string (holding the character representation of an integer) into an integer. For example:

```
string s = "1234";
int n = atoi(s.c_str());
cout << "string " << s << " is integer " << n << endl;
```

Or we can use an istringstream for the same purpose:

```
string s = "1234";
int    n;
istringstream iss(s);
iss >> n;
```

Input and output of characters

get() and put()

If we want to analyse files on a character-by-character basis, we use the `input_stream.get(char)` function, to read the next character from the specified input stream. Similarly you place char data into the output stream using the `output_stream.put(char)` function. The following code reads in character data from the input stream and places a copy of it in the output stream:

```
int main ( )
{  char ch;
   while (cin.get(ch))
      cout.put(ch);
}
```

ignore()

If for any reason you need to ignore a character in the input stream, for example because you know it will send the stream into a fail state, you can use the `input_stream.ignore()` function. If you need to ignore several characters, you can pass arguments to `ignore()` to specify a character to act as a delimiter (ignore all characters up to and including the delimiter) and the maximum number of characters to ignore. For example:

```
//assume cin contains abcde$fgh
cin.ignore(10, '$');
string s;
cin >> s;  // s is now "fgh": characters up to and including $ ignored

// again assume cin contains abcde$fgh
cin.ignore(4, '$');
string s;
cin >> s;  // s is now "e$fgh": maximum of 4 characters ignored
```

You know that you can effectively skip the rest of a line with a `getline(instream, junk);` You could also do it with `instream.ignore(INT_MAX,'\n');`

peek()

`input_stream.peek()` returns the next character in the buffer without `get`ting it, so you can have a look at the next character that's coming up before you have committed yourself to reading it. This could be useful if you wanted to avoid sending a stream into a fail state, e.g. by reading `char` data into an `int` variable.

unget()

`input_stream.unget()` replaces the last character you obtained (with a `get()`) and pushes it back into the input stream, so that it will be the next character to be read. You can only `unget()` one character. Suppose you had two procedures - A and B - taking it in turns to read from the same input stream. A reads a series of characters until it hits one that marks the start of B's section, whereupon A hands over to B, and vice-versa. When A stops, perhaps it has already read the character that marks the start of B's section, in which case it might `unget` that character before handing over to B.

Precision of doubles (round-off errors)

Are computers good at arithmetic? The following code seems to suggest not:

```
#include <iostream>
using namespace std;

int main( )
{  double d = 7;
   for (int i = 0; i < 10 ; i++)
      d -= 0.7;
   if (d == 0.0) cout << "Yes";
   else cout << "No";
}
```

The representation of doubles is necessarily approximate. Consider the decimal representation of one third - 0.3333333... No matter how many 3's you stick on the end, you haven't quite got it exactly. Computer representation of doubles is subject to the same sort of problem. In this example, after subtracting 0.7 from 7.0 ten times, we ought to be at 0.0, but, because of these approximations, a tiny amount of error creeps in so that the final value of `d` is not exactly zero.

In computations involving huge numbers of calculations, typical of many applications in science and engineering, these cumulative errors can rend the final result wildly inaccurate and so completely useless. Consequently great effort is devoted in such applications to devising algorithms that minimise these errors, and the results are not presented as being completely accurate, but rather as accurate to within certain limits.

An important lesson to take from this is that a test for absolute equality with a double is dangerous.

If you wanted to fix the example program so that it output "Yes" as it ought to, you would decide what level of accuracy you were prepared to accept and write a `near_enough` function that took two doubles and returned **true** if they were within your accepted distance of each other. For example:

```
bool near_enough(double x, double y)
{       const double ACCEPT = 0.000001;
        return x <= y ? x >= y - ACCEPT : y >= x - ACCEPT;
}
```

Notes on R. Mitton's lectures by S.P . Connolly, edited by R. Mitton, 2000