

Electronic copies of handouts for L02 can be found at

<http://www.enel.ucalgary.ca/People/Norman/engg233-L02-fall2004/>

**Reading and writing a matrix.** It's not very exciting just to read a matrix and display it, but doing that is a good practice step before going on to more complicated programming problems. (That's also true for programming with other complicated collections of data.)

Here's a sample dialogue, with user input in a *slanted bold font* ...

```
How many rows for your matrix?
2
How many columns for your matrix?
4
Enter integers for row 0 separated by spaces:
10 17 -23 100
Enter integers for row 1 separated by spaces:
99 0 14 -42

Here is what you entered ...
      10      17      -23      100
      99       0       14      -42
```

The code for a program that can do the above job will be developed on the blackboard in the lecture, but for reference here is a complete listing ...

```
#include <iostream>
#include <iomanip>           // Make setw available.
#include <vector>
using namespace std;

// WARNING: This program does NOT check to see whether its input is valid!

int main()
{
    vector<vector<int> > mat;    // empty vector of vectors of ints
    int nrow, ncol;
    cout << "How many rows for your matrix?" << endl;
    cin >> nrow;
    cout << "How many columns for your matrix?" << endl;
    cin >> ncol;

    mat.resize(nrow); // Now we have a vector of nrow empty vectors of ints.

    // Resize each vector<int> to have room for ncol ints.
    for (int r = 0; r < nrow; r++)
        mat.at(r).resize(ncol);

    // Get input from user one row at a time.
    for (int r = 0; r < nrow; r++) {
        cout << "Enter integers for row " << r << " separated by spaces:" << endl;
        for (int c = 0; c < ncol; c++)
            cin >> mat.at(r).at(c);
    }

    // Display matrix, using 12 characters per element.
    cout << endl << "Here is what you entered ... " << endl;
    for (int r = 0; r < nrow; r++) {
        for (int c = 0; c < ncol; c++)
            cout << setw(12) << mat.at(r).at(c);
        cout << endl;
    }
}
```

```
    return 0;
}
```

**A note about function parameters and vector types.** The following was written about simple vector types such as `vector<int>`, but it's also relevant when passing `string`'s, vectors of vectors, and any other potentially large containers of data to functions ...

It is a bad idea to pass a vector to a function as a value argument, because this entails *copying all the elements in the vector*, which could waste a significant amount of time and computer memory. For this reason C++ programmers tend to pass vectors by reference, even when the function receiving the vector does not modify the vector in any way.

When programming with `vector` types you will frequently see and create functions that have parameter types such as `const vector<int>&`, which in English is pronounced “reference to const vector of ints”. It's important to understand exactly what this means.

It's easiest to explain the concept with an example. Suppose a function `foo` is available and has the following prototype:

```
void foo(const vector<int>& par1, vector<int>& par2);
```

What this means is that `foo` has to treat the vector accessed through `par1` as `const`: `foo` is allowed to get information from that vector (such as its size or any of its element values) but it is not allowed to do anything that would modify the vector (for example, resizing it or making assignments to its elements). On the other hand, `foo` is free to modify the vector it accesses through `par2`.

Let's consider code in some other function that tries to call `foo` a few times. First, three vectors are created:

```
vector<int> a(2);           // Make vector of 2 ints.
a.at(0) = 17;
a.at(1) = 42;
vector<int> b;             // Make vector of 0 ints.
const vector<int> c = a;   // Make c as a const copy of a;
```

Next, two calls to `foo` are made:

```
foo(c, a);                // Exact type match.
foo(a, b);                // foo will treat a as const.
```

Both of the above calls are OK. The second one is OK even though `a` is not a `const` vector—`foo` will treat it as if it were `const`. But the following call generates a compiler error:

```
foo(a, c);                // Can't treat c as non-const.
```

By the way, the error message from `g++` for the above line is

```
error: invalid initialization of reference of type
      'std::vector<int, std::allocator<int> >&' from expression of type 'const
      std::vector<int, std::allocator<int> >'
```

In reading such messages it's helpful to know that

```
std::vector<int, std::allocator<int> >
```

is the name the compiler uses for the type `vector<int>`. (Unfortunately, an explanation of *why* there is more than one name for `vector<int>` would be long and complicated.)