IN2029: Programming in C++

Session 2 – Sequential containers

Vahid Rafe

Department of Computer Science
City, University of London

Autumn term, 2025

This session

We'll be writing some programs that operate on batches of data, which allows us to explore

- a bit more about streams
- the standard idiom for looping to the end of an input stream
- manipulators
- vectors from the standard template library
- introduction to containers

Calculating statistics from a list of numbers

Task: read in a list of numbers and print their average.

The overall structure of our program will be:

```
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    // ... read in data ...
    // ... print results ...
    return 0;
}
```

Reading the data

The first part is to read all the numbers and record their count and sum:

```
cout << "Please enter a series of numbers\n";

// the number and total of values read
int count = 0;
double sum = 0;

// read values from standard input
double x; // a variable for reading into
while (cin >> x) {
    ++count;
    sum += x;
}
```

Library details: testing for end-of-input

We have already seen that the **>>** operator returns the input stream, in statements like

```
cin >> x >> y >> z;
```

The result of **>>** can also be used in a test, as in the common idiom for reading a series of things and testing for the end of the input:

```
while (cin >> x) {
    // .. do something with x
}
```

Testing a stream yields `true` if the last operation on the stream succeeded, and `false` if it didn't.

This idiom is much more robust than separately testing for eof.

(You can indicate end of input on the console by typing Control-Z Return on Windows, or Control-D on Unix.)

Alternative: reading from a file

If we want to read from a file instead of the standard input, we use an **ifstream** (a different kind of **istream**). At the top of the file, we need to include another header:

```
#include <fstream>
```

Then we declare a variable **in** of type **istream**, with the file name as a parameter, and then read from that instead of **cin**:

```
    // read values from a file
    ifstream in("values.txt");
    double x;
    while (in >> x) {
        ++count;
        sum += x;
    }
```

Breaking the input into words

An example reading strings:

```
#include <string>
#include <iostream>

using namespace std;

int main() {
    string s;
    while (cin >> s)
        cout << s << '\n';
    return 0;
}
```

Recall that the **>>** operator on strings reads words.

reading a line including spaces

If we want to read a string containing spaces from standard input or from a file, we use a **getline** function. At the top of the file, we need to include this header:

```
#include <iostream>
```

For reading from a file, use **getline(in,line)**, for reading from standard input use **getline(cin,line)**, you need to define **line** as a **string** and **in** should be defined as an **ifstream**:

```
    // read values from a file
    ifstream in("values.txt");
    string line;
    while (getline(in,line)) {//or you can use getline(cin,line)
        //...
    }
```

Language details: `i++` vs `++i`

The following statements all increase an `int` variable `i` by one:

```
    i = i+1;
    i += 1;
    i++;
    ++i;
```

The difference between the last two is only seen when the value of the expression is used:

```
    int i = 5;
    int j = ++i; // j is set to 6; i is now 6
    int k = i++; // k is also set to 6; i is now 7
```

- `i++` returns the value before incrementing (so the old value has to be saved somewhere, which could be expensive with some types)
- `++i` returns the value after incrementing (simpler)

Printing the results

Finally, we want to print the results:

```
    cout << count << " numbers\n";
    if (count > 0) {
        cout << "average = " << sum/count << '\n';
    }
```

By default, floating point numbers are printed with up to 5 significant figures, but we can change that:

```
        cout << "average = " << setprecision(3) <<
            sum/count << '\n';
```

Library details: manipulators

`setprecision(3)` is an example of a stream manipulator (from the **`<iomanip>`** system header), like **`flush`** or **`endl`**: a special kind of object with an overloading of the **`<<`** operator than changes the state of the stream.
This manipulator is used to adjust formatting:

```
    cout << setprecision(3);
```

doesn’t do any output, but it sets the precision for any following output.

```
    cout << setprecision(3) << x << setprecision(5) << y;
```

Other manipulators set base, paddings, etc.

Cleaning up

- We have used **`setprecision`** to set the maximum number of significant figures to what we want.
- Nothing else is happening in this program, but in general it would be polite to set the precision back to what it was before.
- We can get the current precision using **`cout.precision()`**.

This yields our final version:

```
int prec = cout.precision();
cout << "average = " << setprecision(3) <<
    sum/count << setprecision(prec) << '\n';
```

Calculating a different statistic

Task: read in a list of numbers and print their median.

The median of a collection of numbers is the “middle” value when they are arranged in order:

$$1\ 3\ 3\ 7\ 10\ \boxed{11}\ 11\ 13\ 14\ 15\ 15$$

However, the input data may be in any order.

- Unlike computing the average, to compute the median we will need to store all the numbers until the end of the program. We shall use a `vector` to do this.
- Then we need to arrange the values in order. We shall use the library function `sort`.
- Then the median will be the middle value in the vector.

Outline

The overall structure of our program will be:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    // ... read and store the data ...
    // ... sort the data ...
    // ... print the middle value ...
    return 0;
}
```

Vectors

```
#include <vector>
```

C++ has arrays, but we'll use vectors instead (a container like `ArrayList` in Java, except that a variable of **`vector`** type holds an object, not a reference):

```
    vector<int> vi;  // empty vector of ints
    vector<string> vs;  // empty vector of strings
```

Vectors also be extended:

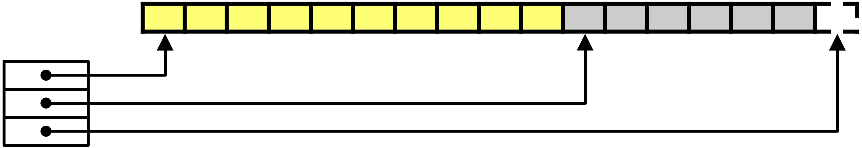
```
    vs.push_back(s);
```

The current length of **`vs`** is **`vs.size()`**
Vectors can be accessed just like arrays (indices 0 . . . **`size()-1`**):

```
    vi[1] = x;
    vi[2] = vi[1] + 3;
```

Implementation of vectors

Internally, vectors are implemented as extensible arrays (see session 3 of Data Structures and Algorithms). One possible implementation is:



The precise internals do not concern us here.

We need only use the provided operations, confident that they are *very* carefully engineered.

Reading the data into a vector

We start by reading all the numbers and storing them in a vector:

```
cout << "Please enter a series of numbers\n";

// read numbers from the standard input
// and store them in a vector
vector<double> v;
double x;
while (cin >> x)
    v.push_back(x);
```

We don't need a separate variable to count them: we can use `v.size()`.

Finding the median: outline

- Only a non-empty vector can have a median.
- First, we need to sort the vector.

```
// compute and output results
unsigned n = v.size();
cout << n << " numbers\n";
if (n > 0) {
    // sort the whole vector
    sort(v.begin(), v.end());

    // ... find the middle value
}
```

Language details: unsigned types

C++ has signed and unsigned integral types of various sizes:

Signed	?	Unsigned
`signed char`	`char`	`unsigned char`
`short`		`unsigned short`
`int`		`unsigned int` (or `unsigned`)
`long`		`unsigned long`
`long long`		`unsigned long long` (in C++11)

- Unlike in Java, the sizes are not defined by the standard (but they are non-decreasing).
- `char` may be either a signed or unsigned type, whichever is more efficient on this architecture.
- Unsigned types cannot be negative: if `i` is of unsigned type, `i < 0` can never be `true`.

Unsigned types: caution

- Unsigned integers will silently underflow:

```
unsigned i = 0;
i--;
```

will not fail – it will set `i` to a very large positive number.

- If an operation involves both a signed and unsigned type, it will silently convert the signed type to unsigned first, so in

```
int i = -5;
unsigned j = 1;
if (i < j)
```

the last test will fail, because `-5` will be silently converted to a very large positive number.

The type of `size()`

- Containers cannot have negative size.
- The return type of the `size()` member function is an unsigned type, but *which* unsigned type is implementation dependent.
- The portable name of its type is `vector<double>::size_type`.
- Here `::` selects a static attribute of the type `vector<double>`. (This is a different use of `::` from namespace qualification, as in `std::vector`.)
- We can use this as the type of the variable `n`:

```
vector<double>::size_type n = v.size();
```

Library details: `sort`, `begin`, `end`

```
sort(v.begin(), v.end());
```

- To sort a vector, we use the **`sort`** function, declared in the **`<algorithm>`** system header.
- Instead of a container, **`sort`** takes two positions or iterators (which we'll explore in session 4).
- These positions should be in the same container, with the first before the second.

- The vector class has member functions **`begin()`** and **`end()`**, yielding positions as the start and end of the vector.
- So the above statement sorts the whole vector – a common idiom, but using iterators is more general.

Where is the median?

There are two cases:

- odd number of elements, e.g. 9:

	0	1	2	3	**4**	5	6	7	8
`v`									

  middle element is cell 4, *i.e.* **`v[v.size()/2]`**

- even number of elements, e.g. 8:

	0	1	2	**3**	**4**	5	6	7
`v`								

  In this case we average the two middle elements (cells 3 and 4):

```
(v[v.size()/2 - 1] + v[v.size()/2])/2
```

Computing the median

We use this plan to compute the median of the sorted array:

```
// find the middle value
vector<double>::size_type middle = n/2;
double median;
if (n%2 == 1) // size is odd
    median = v[middle];
else // size is even
    median = (v[middle-1] + v[middle])/2;
cout << "median = " << median << '\n';
```

and our program is complete.

Type aliases

In C++11, we can declare a new name for a type: A `typedef` declaration allows us to introduce a new name for a type:

```
using vec_size = vector<double>::size_type;
```

This defines a new type name `vec_size` that is equivalent to the longer name. One use is to avoid repeating a long type name:

```
vec_size n = v.size();
// ...
    vec_size middle = n/2;
```

Older versions of C++ used the `typedef` declaration, with exactly the same effect:

```
typedef vector<double>::size_type vec_size;
```

C++11 shortcut: `auto`

In C++, we can just use `auto` instead of the type, which tells the compiler that the variable has the type of the initializing expression:

```
    auto n = v.size();
    // ...
    auto middle = n/2;
```

Here `n` has type `vector<double>::size_type`, and `middle` also has an unsigned type.

- A variable declared as `auto` must be initialized.
- The variable still has a type, which determines how it can be used, and what operations on the variable mean, it's just implicit.

Prefer `auto` to explicit type declarations (if there is an initializing expression with the desired type).

Vectors: further points

- A vector variable contains a whole vector:

```
vector<int> v1 = v; // copy the vector
sort(v.begin(), v.end());
```

  results in **v** being sorted, but **v1** still containing a copy of the original unsorted **v**.
- When indexing **v[i]**, the index **i** is not checked: if it is out of range, the program may crash or continue with corrupted data.
- Other vector member functions:

back()	returns the last element of the vector
pop_back()	removes the last element of the vector

Another container: **deque**

Deques (double-ended queues) can be created in a similar way:

```
deque<int> d; // an empty deque
```

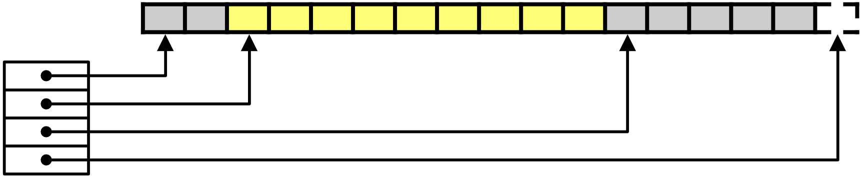
Deques support indexing with `[]`, and these member functions:

size()	the number of elements in the deque
push_back(x)	add **x** to the back of the deque
back()	returns the last element of the deque
pop_back()	removes the last element of the deque
push_front(x)	add **x** to the front of the deque
front()	returns the first element of the deque
pop_front()	removes the first element of the deque

There are common names with **vector**, but no inheritance.

Implementation of deques

Deques are also implemented as extensible arrays, but unlike vectors their front end can also move. One possible implementation is:



Once again, we just use the provided operations, relying on the carefully tuned implementation.

Another container: `stack`

Stacks can be created in a similar way:

```
stack<int> s; // an empty stack
```

Stacks do not support indexing with `[]`, but support these member functions:

size() the number of elements in the stack

push(x) add `x` to the top of the stack

top() returns the last element (on top) of the stack

pop() removes the last element of the stack

Queues are very similar to stacks: (size(), push(x), front() and pop())

```
queue<int> q; // an empty queue
```

Next week

- Functions in C++ allow us to structure and reuse code.
- Passing parameters by value (like in Java) involves copying, which can be expensive as in C++ (unlike in Java) variables contain whole objects.
- Passing parameters by reference avoids copying, and is heavily used in C++.
- It is good practice to use `const` qualifiers to declare that you're not changing something.