3 Batches of Data and the STL

For additional information on the topics covered in this lecture you can consult (Weiss 2004, Section 2.3) (parameter passing), (Prata 2012, Chapter 6) (simple file I/O), (Prata 2012, Chapter 17) (details on streams, buffers, and I/O), as well as (Prata 2012, Chapter 16) (STL, `vector` class).

3.1 Lvalues and Rvalues

The names "Lvalue" and "Rvalue" are derived from the assignment statement

65

```
v = e;
```

Although the assignment looks symmetrical, it is not. When it is executed, the right side, `e`, must yield a ***value*** and the left side, `v`, must yield a ***memory address*** in which the value of `e` can be stored.

> ***Anything which can appear on the left of = (that is, anything that can represent an address) is called an Lvalue.***

> ***Anything which can appear on the right of = (that is, anything that can yield a value) is called an Rvalue.***

"Lvalue" and "Rvalue" are often written without capitals, as "lvalue" and "rvalue". We use initial capitals in these notes for clarity and emphasis.

All Lvalues are Rvalues because, having obtained an address, we can find the Rvalue stored at that address. The operation of obtaining an Rvalue from an Lvalue is called ***dereferencing***. But there are Rvalues that are not Lvalues. Lvalues include: simple variables (`x`); array components (`a[i]`); fields of objects (`o.f`); and a few other more exotic things. Rvalues that are not Lvalues include literals (for example, `67`, `"this is a string"`, `true`) and expressions.

dereferencing

This is (roughly) the original definition for Lvalues and Rvalues, but the introduction of `const` complicated things a little, since we can now have `const` variables that have an address, but cannot be used on the left-hand side of an assignment. To be more precise, we have to distinguish between ***non-modifiable Lvalues*** (such as `const` variables) and ***modifiable Lvalues***.

non-modifiable vs. modifiable Lvalues

3.2 Functions

free vs. member functions

In C++, we can define functions that are not associated with any class. These functions are sometimes called "free functions" to distinguish them from ***member functions***, which are associated with a class. (In other object-oriented languages, member functions are often called "methods".)

C++ provides various ways of passing arguments to functions. The same rules apply to both free functions and member functions.

Terminology. In the function definition

```
double sqr(double x) { return x * x; }
```

the list `(double x)` is a ***parameter list*** and `x` is a ***parameter***. When we call the function, as in

```
cout << sqr(2.71828) << endl;
```

the expression 2.71828 is an ***argument*** that is ***passed*** to the function.

parameter vs. argument

Some authors say "formal parameter" instead of "parameter" and "actual parameter" instead of "argument". We will use the shorter (and more correct) terms ***parameter*** and ***argument***. ("Parameter" can be roughly translated from Greek as "unknown quantity". When we write `sqr` above, we don't know the value of `x`, so it is reasonable to call `x` a parameter.)

> ***Functions have parameters. When a function is called, arguments are passed to it.***

Passing Arguments. There are a number of ways of passing arguments in C++. Three of them suffice for most applications. The program in Figure 20 on the next page includes examples of these three. When run, it produces the following output:[11]

66
67
68

```
Pass by value: 65
Pass by reference: 66
Pass by constant reference: 55
```

Pass by value: the parameter is unqualified. For example, `double x`. When the function is called, the run-time system makes a copy of the argument and passes the copy to the function. The function can change the value of its parameter but these changes have no effect in the calling environment. This explains the output "`65`": the function increments its parameter `k`, but the argument `pbv` remains unchanged.

```
#include <iostream>

using namespace std;

void passByValue(int k)
{
    ++k;
    cout << k;
}

void passByReference(int & k)
{
    ++k;
    cout << k;
}

void passByConstantReference(const int & k)
{
//  ++k;                                Not allowed!
    cout << k;
}

int main()
{
    cout << "Pass by value: ";
    int pbv = 5;
    passByValue(pbv);
    cout << pbv;

    cout << endl << "Pass by reference: ";
    int pbr = 5;
    passByReference(pbr);
    cout << pbr;
//  passByReference(5);                 Not allowed!

    cout << endl << "Pass by constant reference: ";
    int pbcr = 5;
    passByConstantReference(pbcr);
    cout << pbcr << endl;
}
```

Figure 20: Passing arguments in C++

reference type

Pass by reference: the parameter is qualified by `&`, which is read as "reference to". When the function is called, the run-time system passes a ***reference*** to the argument to the function. Pass by reference is usually implemented by passing an address. When the parameter is used in the function body, it is effectively a synonym for the argument. Any change made to the parameter is also made to the argument. This explains the output "`66`": the function increments its parameter `k`, and the argument `pbr` is also incremented.

An argument to be passed by reference must be a Lvalue. The call `passByReference(5)` does not compile, because 5 is not an Lvalue.

`const &`

Pass by constant reference: the parameter is qualified by `const &`, which is read "const reference to". The run-time system passes an address, but the compiler ensures that this address is used only as an Rvalue. Changing the value of the parameter in the function body is not allowed. Note that this will work for both `const` arguments (non-modifiable Lvalues) and non-`const` arguments (modifiable Lvalues) – another good reason for using `const` wherever possible.

69 Figure 21 compares ways of passing arguments in C++ and Java.

Method: pass by . . .	C++	Java
value	`T x`	primitive types (e.g., `int`)
reference	`T & x`	objects (e.g., `Integer`)
`const` reference	`const T & x`	no equivalent

Figure 21: Passing arguments in C++ and Java

70 Use the following rules when deciding how to pass an argument:

1. ***Pass small arguments by value.***

   Addresses are used for passing by reference. An address is 4 bytes or, on modern 64bit machines, 8 bytes. If an argument is smaller, or not much bigger, than an address, it is usually passed by value. In practice, this means that the standard types (`bool`, `char`, `short`, `int`, `float`, `double`, etc.) are usually passed by value, and user-defined types (that is, instances of classes) are usually passed by reference.

2. ***Pass large objects by constant reference.***

   "Constant reference" is usually considered to be the "default" mode for C++; Default is in quotes because constant reference is not the compiler's default: it must be selected explicitly by the user. It should be used for large objects, including instances of user-defined classes, except when there is a good reason not to use it.

[11] These modes are sometimes referred to as "call by value", "call by reference", etc. They mean the same thing but "pass" seems more precise than "call".

3. ***Pass by reference only when caller needs changes.***

   Pass by reference should be used ***only*** when changes made by the function must be passed back to the caller. The usual way of returning results is to use a `return` statement. However, `return` can return only one value and it is sometimes necessary to return more than one value. In these and similar situations, reference parameters may be used to return several items of information.

Note that C++'s constant reference is significantly different from Java's `final` modifier. In Java, even when you declare a parameter as final:

C++ `const &` vs. Java `final`

```
void foo( final Student s )     // Java final is not const!
```

you can still call member functions of `s` within `foo`, such as `s.fail()`, which change the state of `s`.

3.3 Working with files

One of the nice features of C++ streams is that they make most kinds of input and output look the same to the programmer. The program in Figure 22 on the next page asks the user for a file name, reads a list of numbers from the file, and displays the mean.

71 72

Points of note include:

- The directive "`#include <fstream>`" is required for any program that uses input or output files.
- The type `ifstream` is the type of input streams. The variable `fin` is an instance of this type.
- The constructor for `ifstream` needs a file name. Curiously, it cannot accept the file name as a `string`; instead, it requires a C style string, of type `const char*`. Consequently, we have to use the conversion function `c_str` to convert the `string` to a `const char*`. Calling the constructor with a file name has the effect of opening the file.

  stream file names must be C style strings, not `string` objects!
- After the file has been opened, we use `fin` to read from the file in exactly the same way that we use `cin` to read from the keyboard.
- Recall that `fin >> observation` returns the updated input stream as its value. There are two possibilities:

  1. A value has been read and the stream is in a "good" state. In this case, the value of `fin >> observation` will be considered `true`.
  2. A problem was encountered and the stream is in a "bad" state. In this case, the value of `fin >> observation` will be considered `false`.

  In case 2, the most likely "problem" is that we have reached the end of the file.
- When the program has finished reading data from the file, it calls `fin.close()` to close the stream. This step is not strictly necessary, because the destructor, called at the end of the scope, would close the stream if it was still open. Nevertheless, it is good practice to explicitly close a stream that is no longer required by the program.

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

int main()
{
    cout << "Enter file name: ";
    string fileName;
    cin >> fileName;
    ifstream fin(fileName.c_str());

    int obsCount = 0;
    double sum = 0;
    double observation;
    while (fin >> observation)
    {
        sum += observation;
        ++obsCount;
    }
    fin.close();
    cout <<
        "The mean of " << obsCount <<
        " observations is " << sum / obsCount <<
        endl;
}
```

Figure 22: Finding the mean from a file of numbers

73

- The operations of constructing and opening a file can be separated. Instead of

```
ifstream fin(fileName.c_str());
```

open() a stream

we could have written:

```
ifstream fin;
// ....
fin.open(fileName.c_str());
```

- Whenever a program tries to open a file for input, there is a possibility that the file does not exist. As above, we can use the stream object as a boolean to check whether the stream was opened successfully:

```
ifstream fin;
fin.open(fileName.c_str());
if (!fin)
{
    cerr << "Failed to open " << fileName << endl;
    return;
}
```

- Another possibility would be to trigger an assertion failure when a file cannot be opened:

```
ifstream fin;
fin.open(fileName.c_str());
assert(fin);
```

  However, this would go against the recommendations of Section 2.7 on page 34: There are many reasons why opening a file might fail, and the failure does not imply that there is a ***logical*** fault in the program.

3.3.1 Writing to a file

Writing is very similar to reading. The class for output files is `ofstream`. We use the insert operator `<<` to write data to the file. The methods `open` and `close` work in the same way as they do for input files. Figure 23 shows a very simple program that writes random numbers to an output file. Note that you need to `#include <cstdlib>` in addition to `fstream`.

74

random numbers with `rand()`

```
int main()
{
    ofstream fout("randomnumbers.txt");
    for (int n = 0; n != 50; ++n)
        fout << rand() << '\n';
    fout.close();
    return 0;
}
```

Figure 23: Writing random numbers to a file

3.3.2 Stream States

In general, the states of an input stream are ***good***, ***bad***, and ***end-of-file***. These are not mutually exclusive. If the state is ***good***, it cannot also be either ***bad*** or ***end-of-file***. However, if the state is ***bad***, it may or may not be ***end-of-file***. The input statement `cin >> n`, where `n` is an integer, for example, will put the stream into a ***bad*** state if the next character in the stream is not a digit or "−" (the minus sign). But reading can continue after calling `cin.clear()` to clear the bad state.

This section provides a brief overview of stream states. For details, consult a reference work, such as (Langer and Kreft 2000, pages 31–35) or (Josuttis 2012, Chapter 15.4).

The state of the stream is represented by four bits; Figure 24 on the following page shows their names and meanings. Here are some examples of how the bits can get set:

75

- The program wants to read an integer and the next character in the stream is `'x'`. After the input operation, `failbit` is set and the stream position is unchanged.
- The program wants to read an integer. The only characters remaining in the file are "white space" (blanks, tabs, and newlines). After the input operation, `failbit` and `eofbit` are both set and the stream is positioned at end-of-file.

Name	Meaning
`goodbit`	The stream is in a "good" state – nothing's wrong
`eofbit`	The stream is positioned at end-of-file – no more data can be read
`failbit`	An operation failed but recovery is possible
`badbit`	The stream has "lost integrity" and cannot be used any more

Figure 24: Stream bits and their meanings

- The program wants to read an integer. The only characters remaining in the file are digits. After the input operation, `eofbit` is set and the stream is positioned at end-of-file.

- The program wants to read data from a disk file but the hardware (or operating system) reports that the disk is unreadable. After the operation, `badbit` is set and the stream cannot be used again.

76

Figure 25 on the next page shows how to test the stream bits. These functions are members of the stream classes. For example, if `fin` is an input file stream, then `fin.bad()` tests its `badbit`. There are several things to note about these functions:

- The first five return `bool` values – that is, `true` or `false`.

- `fail()` returns `true` if `failbit` is set ***or*** if `badbit` is set.

- `operator!()` is called by writing `!` before the stream name, as in

```
if (!fin)
    // failbit is set or badbit is set
else
    // file is OK
```

- `operator void*()` is called by writing the stream name in a context where an expression is expected. For example:

```
if (fin)
    // file is OK
else
    // failbit is set or badbit is set
```

- `operator void*()` is also called when we write, for example

```
if (fin >> data)
```

Output streams use the same state bits, but it is not often necessary to use them. An output stream is always positioned at end-of-file, ready for the next write. Output operations fail only when something unusual happens, such as a disk filling up with data.

Function	Value/Effect
`bool good()`	None of the error flags are set
`bool eof()`	`eofbit` is set
`bool fail()`	`failbit` is set or `badbit` is set
`bool bad()`	`badbit` is set
`bool operator!()`	`failbit` is set or `badbit` is set
`operator void*()`	Null pointer if `fail()` and non-null pointer otherwise
`void clear()`	Set `goodbit` and clear the error bits

Figure 25: Reading and setting the stream bits

3.3.3 Summary of file streams

The stream class hierarchy is quite large. For practical purposes, there are four useful kinds of stream: 77

`ifstream`: input file stream

`ofstream`: output file stream

`istringstream`: input string stream

`ostringstream`: output string stream

We will discuss string streams later. For each of these kinds of stream, there is another with "`w`" in front (for example, `wifstream`). These are streams of "wide" (16–bit or Unicode) characters.[12] Unicode

File streams are opened by providing a file or path name. The name can be passed to the constructor or to the `open` method. When a file is no longer needed, the `close` method should be called to close it.

The extract operator `>>` reads data from an input stream. The insert operator `<<` writes data to an output stream. The right operand of an extractor must be an Lvalue. The right operand of an inserter is an Rvalue.

The right operand of an extractor or inserter may also be a ***manipulator***. Manipulators may extract or insert data, but they are usually used to control the state of the stream. We have already seen `endl`, which writes a new line and then flushes the buffer of an output stream. stream manipulators

Although `endl` is defined in `iostream`, most other manipulators are not. They are defined in `iomanip` and so we have to write `#include <iomanip>` in order to use them. Here is a selection of commonly used manipulators for output streams:

`left`: Start left-justifying output data (appropriate for strings) 78

`right`: Start right-justifying output data (appropriate for numbers)

`setprecision(`n`)`: Put n digits after the decimal point for `float` and `double` data

`setw(`n`)`: Write the next field using at least n character positions (only applies to the next field, not subsequent fields) 79

[12] C++11 added two more types for dealing with Unicode, `char16_t` and `char32_t`.

`fixed`: Use fixed-point format for `float` and `double` data (for example, `3.1415926535`)

`scientific`: Use "scientific" format for `float` and `double` data (for example, `1.3e12`)

For example, the program shown is Figure 26 displays:

```
Alice          41    1169699.780
Boris        6500    3014133.560
Ching        1478    7915503.960
Daoust       4464    1605672.250
```

3.4 End-of-file Handling

Figure 27 on the next page shows a program that reads a list of numbers, computes their mean, and displays it. This is what the console window looks like after running this program:

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <cstdlib>

using namespace std;

int main()
{
    vector<string> names;
    names.push_back("Alice");
    names.push_back("Boris");
    names.push_back("Ching");
    names.push_back("Daoust");

    for ( vector<string>::const_iterator it = names.begin();
          it != names.end();
          ++it)
    {
        cout << left << setw(8) << *it;
        cout << right << setw(10) << rand() % 10000;
        cout << fixed << setprecision(3) << setw(15) <<
                rand() * (rand() / 100.0);
        cout << endl;
    }
    return 0;
}
```

Figure 26: Output manipulators

```
Enter observations, terminate with ^D:
2.6 9.32 5.67 2.1 6.78 5.1 ^D
The mean of 6 observations is 5.26167
```

```
int main()
{
    cout << "Enter observations, terminate with ^D:" << endl;
    int obsCount = 0;
    double sum = 0;
    double observation;
    while (cin >> observation)
    {
        sum += observation;
        ++obsCount;
    }
    cout <<
        "The mean of " << obsCount <<
        " observations is " << sum / obsCount <<
        endl;
    return 0;
}
```

Figure 27: Finding the mean

The character `^D` (CTRL–D on the keyboard) is used to indicate the end of an input stream reading from the keyboard (i.e., `cin`), for Unix/Linux/MacOS – on Windows, end-of-file is indicated with `^Z`.[13]

CTRL–D vs. CTRL–Z for end-of-file

The loop in this program, as shown in Figure 22 on page 48, is controlled by

```
while (cin >> observation)
```

The condition `cin >> observation` raises two questions: First, why does it work? Second, why do other, seemingly more natural constructions, ***not*** work?

Here is why it works:

1. The expression `cin >> observation` returns an updated value of `cin`. Note that this is ***not*** a boolean value (`true` or `false`), but rather of type `istream`. Consequently,

   ```
   if (cin >> observation) ....
   ```

   is equivalent to

   ```
   cin >> observation;
   if (cin) ....
   ```

[13] Note that not all C++ implementations handle EOF from keyboard input in a consistent way, so you might get different results when experimenting with this.

2. When `cin` appears in a condition context, the compiler attempts to convert it into something that can be considered boolean. By means of a technical trick, this boolean value is `true` if the stream is in a "good" state and `false` if the stream is in a "bad" state.

3. If the stream is in a good state, then the operation `cin >> observation` must have succeeded and a value for `observation` was read successfully.

4. If the stream is in a bad state, the operation failed and the value of `observation` is undefined.

5. Looking at the way in which `while (cin >> observation)` is used in the program above, we see that everything works out nicely: the program ***either*** successfully reads a value and processes it ***or*** does not manage to read a value and terminates the loop.

The "technical trick" mentioned in step 2, in case you are interested, is as follows. Since the compiler cannot interpret an instance of `istream` (the class of `cin`) as a boolean, it looks for a conversion that would enable it to do so. Class `istream` provides a conversion from `istream` to `void*`, the type of pointers that point to nothing in particular. If the stream is in a good state, the result of this conversion is a non-null pointer (probably, but not necessarily, the address of the stream object), which is considered `true`. If the stream is in a bad state, the conversion yields a null pointer, which is zero, and is therefore considered `false`.

There are several reasons why a stream can get into a bad state. The most likely reason, and the one we expect here, is that the program has reached the end of the stream (indicated, in this example, by the `^D` key). Another reason is that a disk has failed, although obviously this does not apply to `cin`.

```
void v1()
{
   vector<int> v;
   int k;
   while (cin >> k)
   {
      v.push_back(k);
   }
   for (vector<int>::const_iterator it = v.begin(); it != v.end(); ++it)
      cout << *it << ' ';
   cout << endl;
}
```

Figure 28: Testing end-of-file

The second question we have to answer is: why do other approaches not work? Problems arise because the normal input mode skips blanks to find the next datum. We will use the program in Figure 28 to explore the potential problems: This program works correctly whether or not there is white space between the last datum and the end of the file. In this example and subsequent examples, end-of-file is indicated by `^D`. Note the blank after `4` in the second example.

```
1 2 3 4^D
1 2 3 4
```

```
1 2 3 4 ^D
1 2 3 4
```

Class `stream` provides a function `eof` ("end-of-file") that yields `false` except at the end of the stream, where it returns `true`. A function that returns a boolean value is called a ***predicate***; `eof` is therefore a predicate. We can test for end-of-file *before* attempting to read the input: `eof()` 84

```
while (!cin.eof())
{
   cin >> k;
   v.push_back(k);
}
```

This is what happens:

```
1 2 3 4^D
1 2 3 4

1 2 3 4 ^D
1 2 3 4 4
```

If there is no white space after the last datum, this code works correctly. If there is a blank, the following sequence of events occurs:

- The stream reads the last datum, `4`, correctly.
- Since the next character is a blank, `cin.eof()` is `false`.
- The stream reads the blank, finds no integer, and puts the stream into a "bad" state. The value of `k` is not changed.
- The value in `k`, which is still `4`, is stored in the vector *again*.

Another possibility is to test for end-of-file *after* reading: 85

```
while (true)
{
   cin >> k;
   if (cin.eof())
      break;
   v.push_back(k);
}
```

This code gives the following results:

```
1 2 3 4^D
1 2 3

1 2 3 4 ^D
1 2 3 4
```

If there is no blank after `4` then, after this datum has been read, `cin.eof()` is `true`. The loop terminates and `4` is not stored in the vector. When there is a blank, the program works correctly.

3.5 Storing Data: STL Containers

For a few applications, such as computing an average, it is sufficient to read values; we do not have to store them. For most applications, it is useful or necessary to store values as we read
86 them. The program in Figure 29 shows one way of doing this.

The new feature in this program is

```
vector<double> observations;
```

STL vector<> This declaration introduces `observations` as an instance of the class `vector<double>`. The ***template class*** `vector` is one of the ***containers*** provided by the Standard Template Library (STL). A template class is ***generic*** – it stands for many possible classes – and must be instantiated by providing a type. In this case, the type is `double`, giving us a `vector` of `double`s. When we use a vector, we must also write

```
#include <vector>
```

```
int main()
{
    cout << "Enter file name: ";
    string fileName;
    cin >> fileName;
    ifstream fin(fileName.c_str());

    vector<double> observations;
    double obs;

    while (fin >> obs)
    {
        observations.push_back(obs);
    }
    fin.close();

    for ( vector<double>::size_type i = 0; i != observations.size(); ++i )
        cout << observations[i] << '\n';
}
```

Figure 29: Storing values in a vector

The relation between a generic class and an instantiated class is analogous to the relation between
87 a function and a function application:

	Generic	Application
Function	log	$\log x$
Class	`vector`	`vector<double>`

There are various things we can do with a vector. The operation v.`push_back`(x) inserts the value x at the back end of the vector v. We do not have to specify the initial size of the vector, and we do not have to worry about how much stuff we put into it; storage is allocated automatically as the vector grows to the required size.

v.push_back(x)

The argument for `push_back` is passed ***by value***. This means that the vector gets its own copy of the argument, which is usually what we want.

The method `size` returns the number of elements in the vector. As with `string`, the type of the size is not `int` but `vector<double>::size_type`. This is the type that we use for the control variable `i` in the final `for` loop of the program.

A vector can be subscripted, like an array. If v is a vector, $v[i]$ is the i^{th} element. In the program, `observations[i]` gives the `i`th element of the vector of observations.

Vector indexes are not checked!

This means that, when we write `observations[20]`, for example, there is no check that this element exists.[14] If it doesn't exist, the result will be garbage. Even worse, if we use this expression on the left of an assignment, we can write into any part of memory!

`<sermon>` This is an astonishing oversight. Here is Tony Hoare (1981):

> *"...we asked our customers whether they wished us to provide an option to switch off these [array index] checks in the interests of efficiency on production runs. Unanimously, they urged us not to – they already know how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long ago been against the law."*

In this example, the language was Algol 60 and the computer was an Elliott 503. The 503 weighed several tons, had 8K words of memory, and needed 7 μsec to add two numbers.[15]

Another 30 years have passed, and it is more than 40 years since Hoare's customers wanted subscript checking. Today's computers are around 100,000 times faster than the 503, and programmers are still concerned about the time taken to check subscripts. A high proportion of viruses, worms, phishers, and other kinds of malicious software exploits precisely this loophole. `</sermon>`

Fortunately, there are safer ways of accessing the elements of a vector than using subscripting. One way is to use the function `at`. The call `array.at(i)` has the same effect as `array[i]` but checks that the index is within range and throws an exception otherwise.

array.at(i) provides index checking

[14] Unlike in Java, which would throw an exception in this case.

[15] "Some Old Computers," http://members.iinet.net.au/~daveb/history.html

Iterators are another alternative to subscripts and, in many cases, a better one. An iterator is an object that keeps track of the objects in a container and provides facilities for accessing them. The type of the iterator that we need is

```
vector<double>::const_iterator
```

Two of its values are `observations.begin()`, which refers to the first element of the vector `observations`, and `observations.end()`, which refers to the first element ***not*** in the vector – that is, one *past* the end. Iterators have increment (`++`) and decrement (`--`) operators. Iterators also have pointer-like behaviour: dereferencing an iterator yields an element of the container.

Putting all these things together, we can write the following loop to access the elements in the
88 vector `observations`:

```
for (  vector<double>::const_iterator i = observations.begin();
       i != observations.end();
       ++i )
    cout << *i << '\n';
```

This code has two significant advantages over the original version:

- Using the iterator and, specifically, the function `end`, ensures that we access exactly the elements that are stored.
- This code can be used with other kinds of container.

For example, if we replace each occurrence of `vector` by `list`:

```
list<double> observations;
....
for (  list<double>::const_iterator i = observations.begin();
   i != observations.end();
   ++i )
cout << *i << '\n';
```

the program compiles and runs with exactly the same effect.

89 However, if we replace `!=` in the loop condition by `<`, the compiler complains – see Figure 30. The problem is that list iterators, unlike vector iterators, provide equality (`==` and `!=`) but not ordering (`<`, etc.).

```
error: no match for 'operator<' in
    'i < observations.std::vector<_Tp, _Alloc>::end
    [with _Tp = double, _Alloc = std::allocator<double>]()'
```

Figure 30: Complaints from the compiler

In this case, the compiler diagnostic starts with `no match for 'operator<'` and, since introducing `<` was the only change we made to the program, it is not hard to figure out that it is

this operator that caused the problem. Unfortunately, the STL can produce even worse error messages that can be very hard to interpret.[16]

Iterator loops. C++11 introduced a new kind of loop, the range-based `for` loop, which works similar to the *for-each* loop added in Java 5:

```
for ( double i: observations )
    cout << i << '\n';
```

Note that you will need a compiler supporting C++11; in `gcc` it was added in version 4.6. 90

Sorting the vector of observations is very easy. Only one extra line of code is required: 91

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

However, `sort` is not a part of `vector`; it is one of the algorithms provided by the STL. Consequently, we also need the directive

```
#include <algorithm>
```

3.6 Summary of the Standard Template Library (STL)

The STL provides containers, iterators, algorithms, function objects, and adaptors:

Containers are data structures used to store collections of data of a particular type. The operations available for a container, and the efficiency of the operations, depend on the underlying data structure. For example, `vector` provides array-like behaviour: elements can be accessed randomly, but inserting or deleting elements may be expensive. In contrast, a `list` can be accessed sequentially but not randomly, and provides efficient insertion and deletion.

The containers also include `set` for unordered data and `map` for key/value pairs without duplicates. The containers `multiset` and `multimap` are similar but allow duplicates.

Iterators provide access to the elements stored in containers. They are used to ***traverse*** containers (i.e., visit each element in turn) and to specify ***ranges*** (i.e., groups of consecutive elements in a container).

Algorithms provide standard operations on containers. There are algorithms for finding, searching, copying, swapping, sorting, and many other applications.

Function objects are objects that behave as functions. Function objects are needed in the STL because the compiler can sometimes select an appropriate object in a context where it could not select an appropriate function. However, there are also other uses of function objects.

[16] Some programmers write Perl scripts to parse compiler diagnostics and pick out the key parts.

Adaptors allow interface modification and increase the flexibility of the STL. Suppose we want a stack. There are several ways to implement a stack: we could use a vector, or a list, or some other type. The STL might provide a class for each combination (`StackVector`, `StackList`) and perhaps `Stack` as a default.

In fact, the STL separates abstract data types (such as `stack`) and representations (such as `vector` and `list`) and provides adaptors to fit them together. Thus we have:

92

`stack<T>:` stack of objects of type `T` with default implementation

`stack<T, vector<T> >:` stack of objects of type `T` with vector implementation

`stack<T, list<T> >:` stack of objects of type `T` with list implementation

Assuming that the STL provides M container classes and N algorithms, it is tempting to assume that there are $M \times N$ ways of using it, because we should be able to apply each algorithm to each container. However, this is not in fact how the STL works. Instead:

- A container/algorithm combination works only if the algorithm is appropriate for the data structure used by the container.
- If the combination does work, its performance is guaranteed in terms of a complexity class, e.g., $\mathcal{O}(N)$.

STL guarantees performance as complexity class

We have already seen an example of this in Section 3.5 on page 56. Suppose that `i` and `j` are iterators for a container and that `*i` and `*j` are the corresponding elements. We would expect `==` and `!=` to be defined for the iterators. It is also reasonable to expect `++`, because iterators are supposed to provide a way of stepping through the container. But what does `i < j` mean? Presumably, something like "`*i` appears before `*j` in the container". This is easy to evaluate if the container is a `vector`, because vectors are indexed by integers (or perhaps pointers), which can be compared. But evaluating `i < j` for a linked list is inefficient, because it requires traversing the list. This is why the iterator for a vector provides `<` but the iterator for a list does not.

It is important to check that the algorithm you want to use works with the container that you are using. The penalty for not checking is weird error messages. For example,

93

```
void main()
{
    std::vector<int> v;
    std::stable_sort(v.begin(), v.end());
}
```

compiles correctly, but

```
void main()
{
    std::list<int> v;
    std::stable_sort(v.begin(), v.end());
}
```

94

produces the message

```
stl_algo.h: In function ‘void __merge_sort_loop<_List_iterator
  <int,int &,int *>, int *, int>(_List_iterator<int,int &,int *>,
  _List_iterator<int,int &,int *>, int *, int)’:
stl_algo.h:1448:   instantiated from ‘__merge_sort_with_buffer
  <_List_iterator<int,int &,int *>, int *, int>(
   _List_iterator<int,int &,int *>, _List_iterator<int,int &,int *>, int *, int *)’
stl_algo.h:1485:   instantiated from ‘__stable_sort_adaptive<
  _List_iterator<int,int &,int *>, int *, int>(_List_iterator
  <int,int &,int *>, _List_iterator<int,int &,int *>, int *, int)’
stl_algo.h:1524:   instantiated from here
stl_algo.h:1377: no match for ‘_List_iterator<int,int &,int *> & -
  _List_iterator<int,int &,int *> &’
```

Why doesn't the STL generate more useful diagnostics? The reason is that it is based on templates. The compiler first expands all template applications and then tries to compile the resulting code. If the code contains errors, the compiler cannot trace back to the origin of those errors, saying perhaps "`list` does not provide `stable_sort`", but can only report problems with the code that it has.

Various objects and values associated with containers have types. These types may depend on the type of the elements for the container. For example, the type of an iterator for `vector<int>` may not be the same as the type of an iterator for `vector<double>`. Consequently, the container classes must provide the types we need. In fact, we have already seen expressions such as `vector<double>::const_iterator`, which is the type of `const` iterators for a vector of `doubles`.

Using `typedef`. These type names can get quite long. It is common practice to use `typedef` directives to abbreviate them. A `typedef` has the form 95

`typedef` ⟨*type expression*⟩ ⟨*identifier*⟩

and defines ⟨*identifier*⟩ to be a synonym for ⟨*type expression*⟩. For example, after

```
typedef vector<double>::const_iterator vci;
```

we can write `vci` instead of the longer expression.

References

Josuttis, N. M. (2012). *The C++ Standard Library: A Tutorial and Reference* (2nd ed.). Addison-Wesley. http://www.cppstdlib.com/.

Langer, A. and K. Kreft (2000). *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley.

Prata, S. (2012). *C++ Primer Plus* (6th ed.). Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.