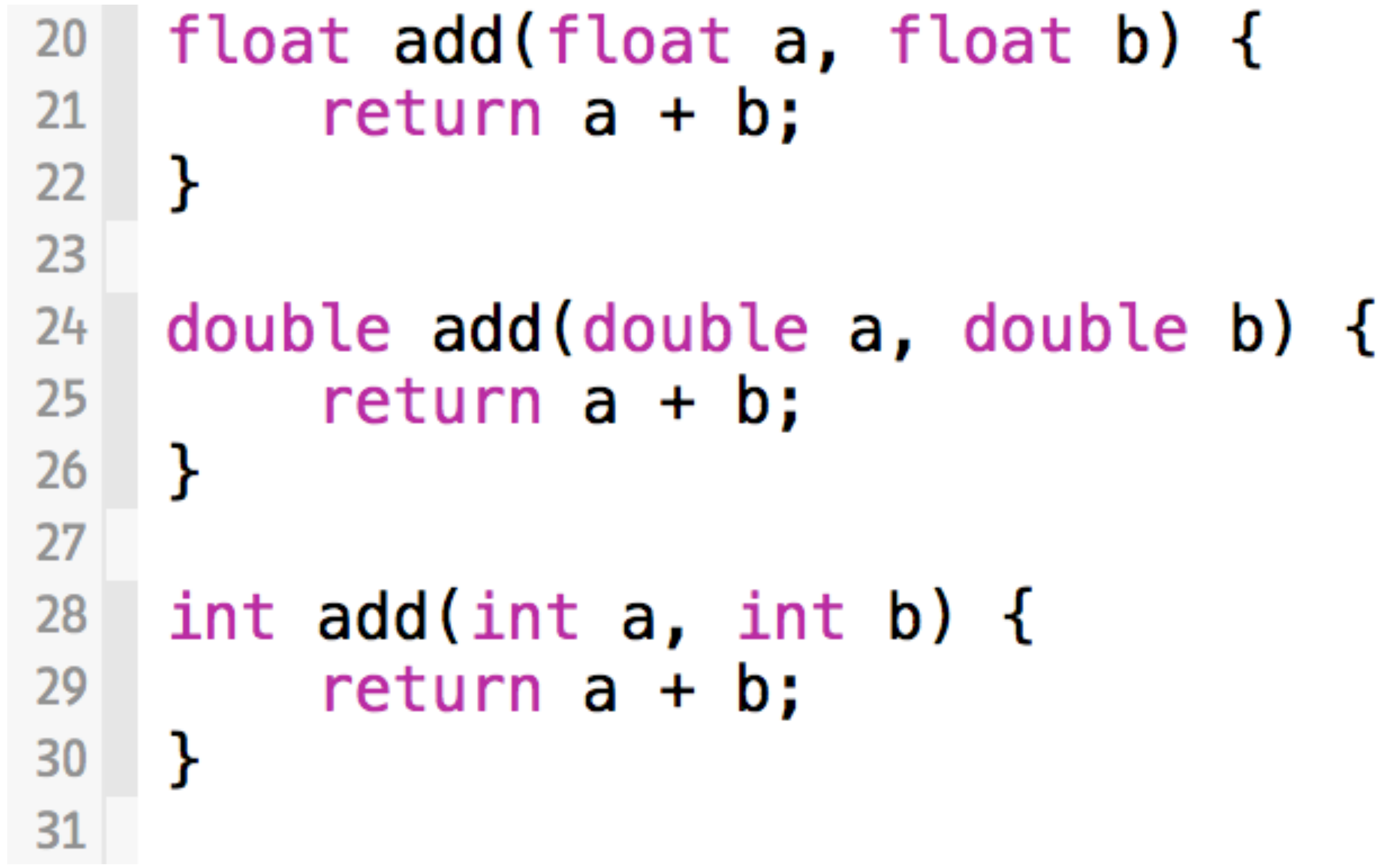
C++ Templates and STL Library

copied and modified in sequence from
http://cs.brown.edu/~jak/proglang/cpp/stltut/tut.html

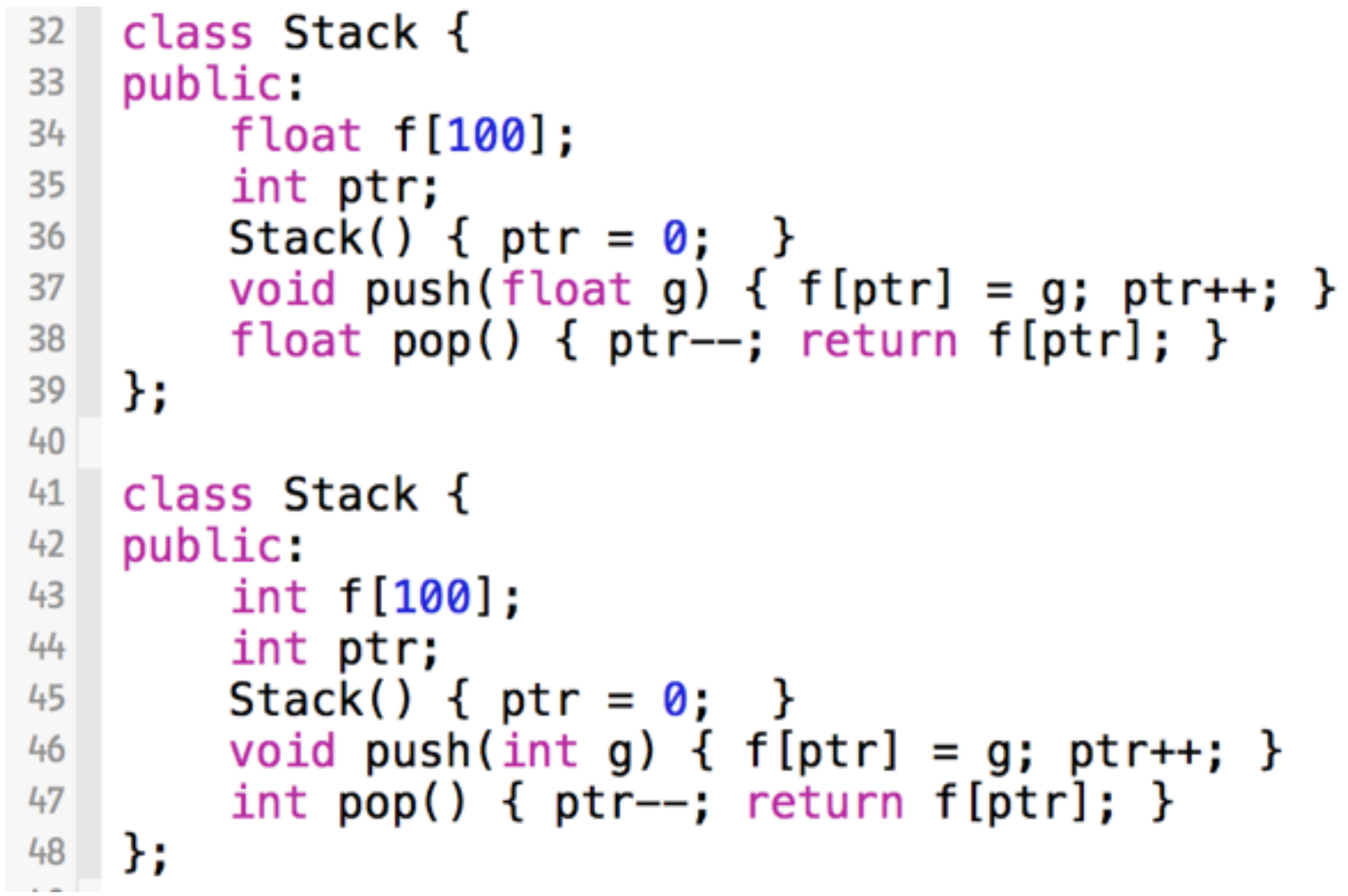
Templates — or why do we need them



```
float add(float a, float b) {
        return a + b;
}

double add(double a, double b) {
        return a + b;
}

int add(int a, int b) {
        return a + b;
}

```

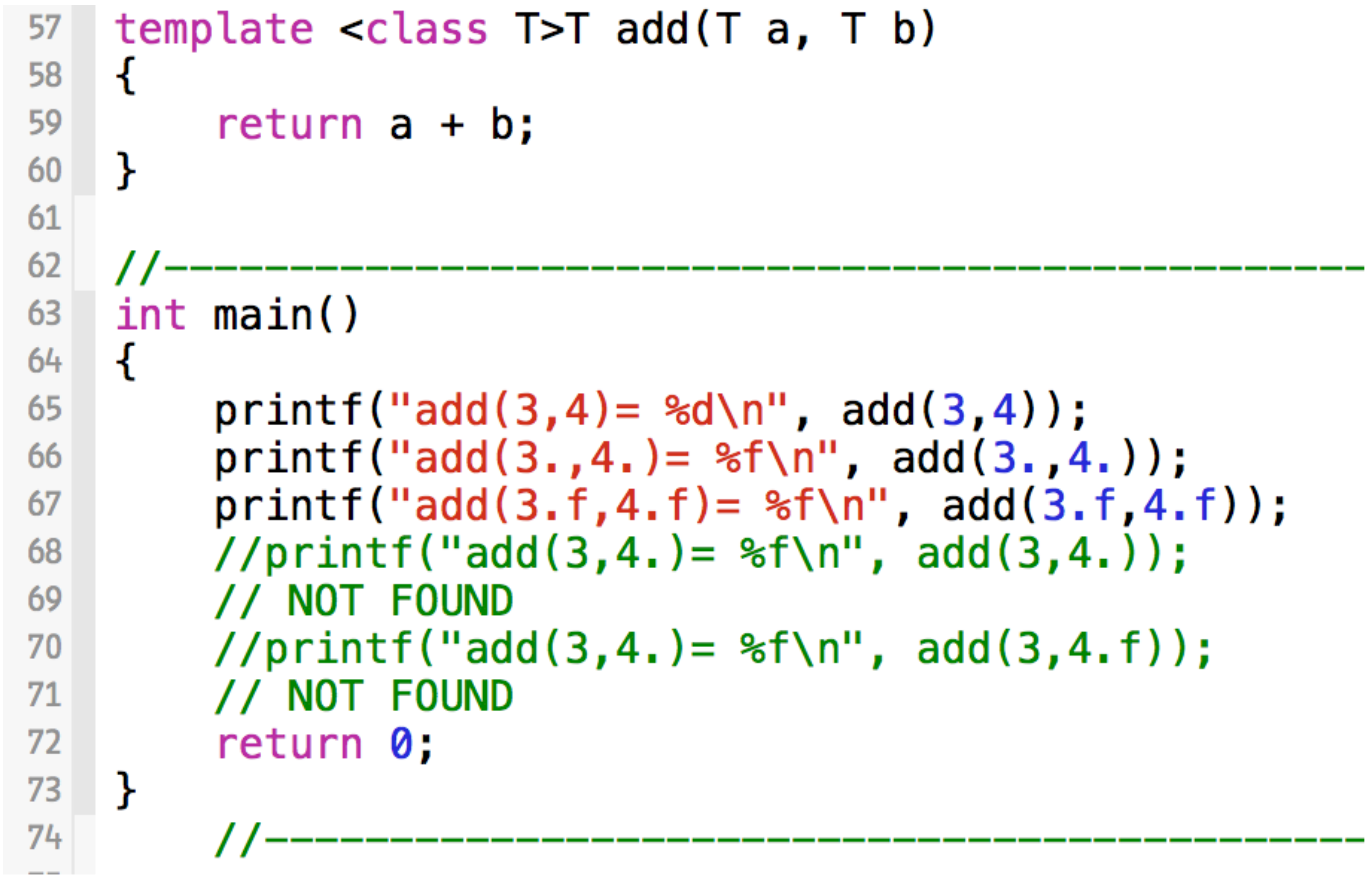
- Every time I want to add two variables of a new type, I must write a new **add** function
- This gets complicated very fast

more examples

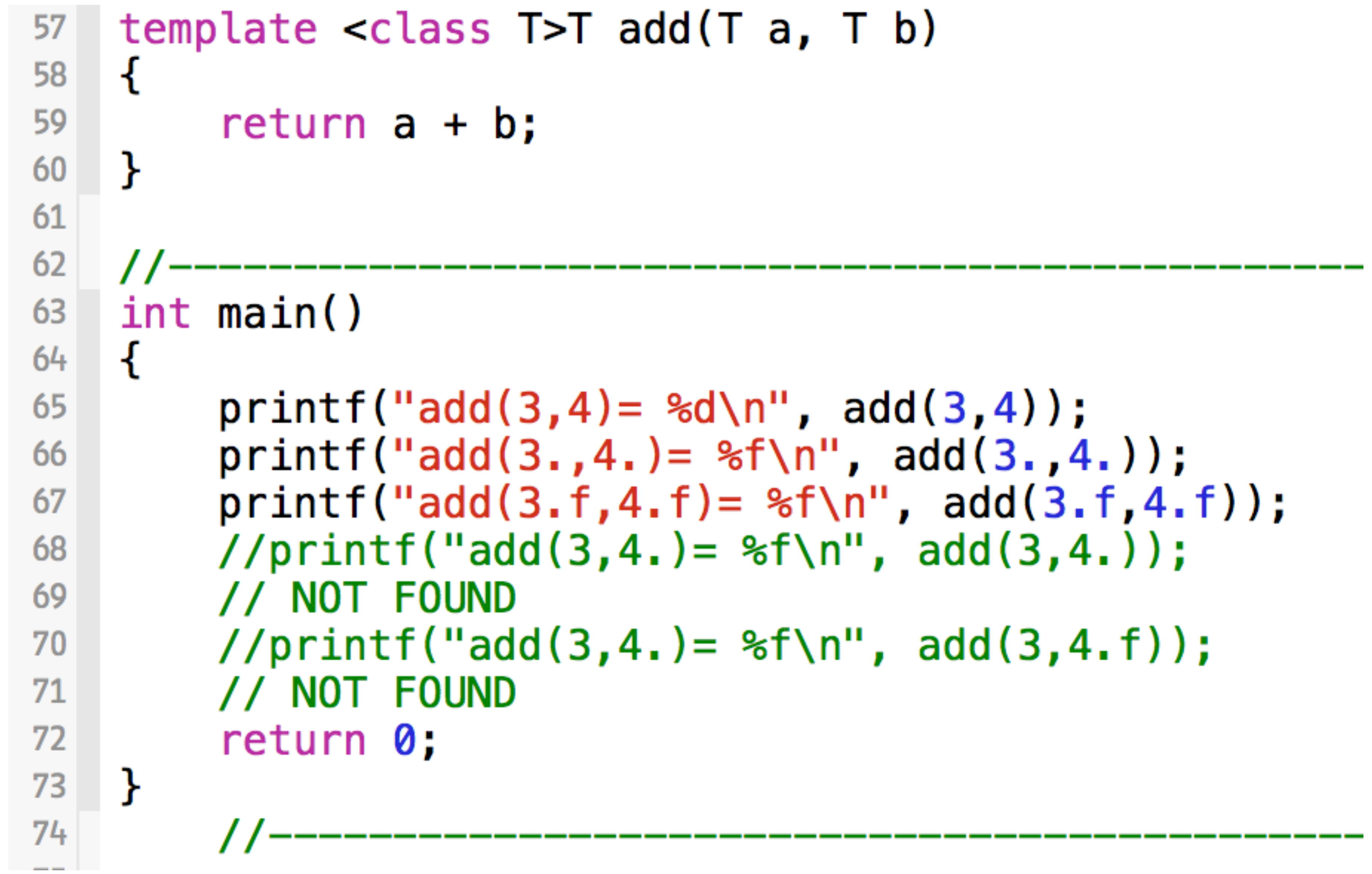


```
class Stack {
public:
    float f[100];
    int ptr;
    Stack() { ptr = 0;  }
    void push(float g) { f[ptr] = g; ptr++; }
    float pop() { ptr--; return f[ptr]; }
};

class Stack {
public:
    int f[100];
    int ptr;
    Stack() { ptr = 0;  }
    void push(int g) { f[ptr] = g; ptr++; }
    int pop() { ptr--; return f[ptr]; }
};
```

- One can have stacks of things
- These things can be ints, floats, doubles, strings, or other stacks.
- How to handle all different types at the same time?
- Use templates!

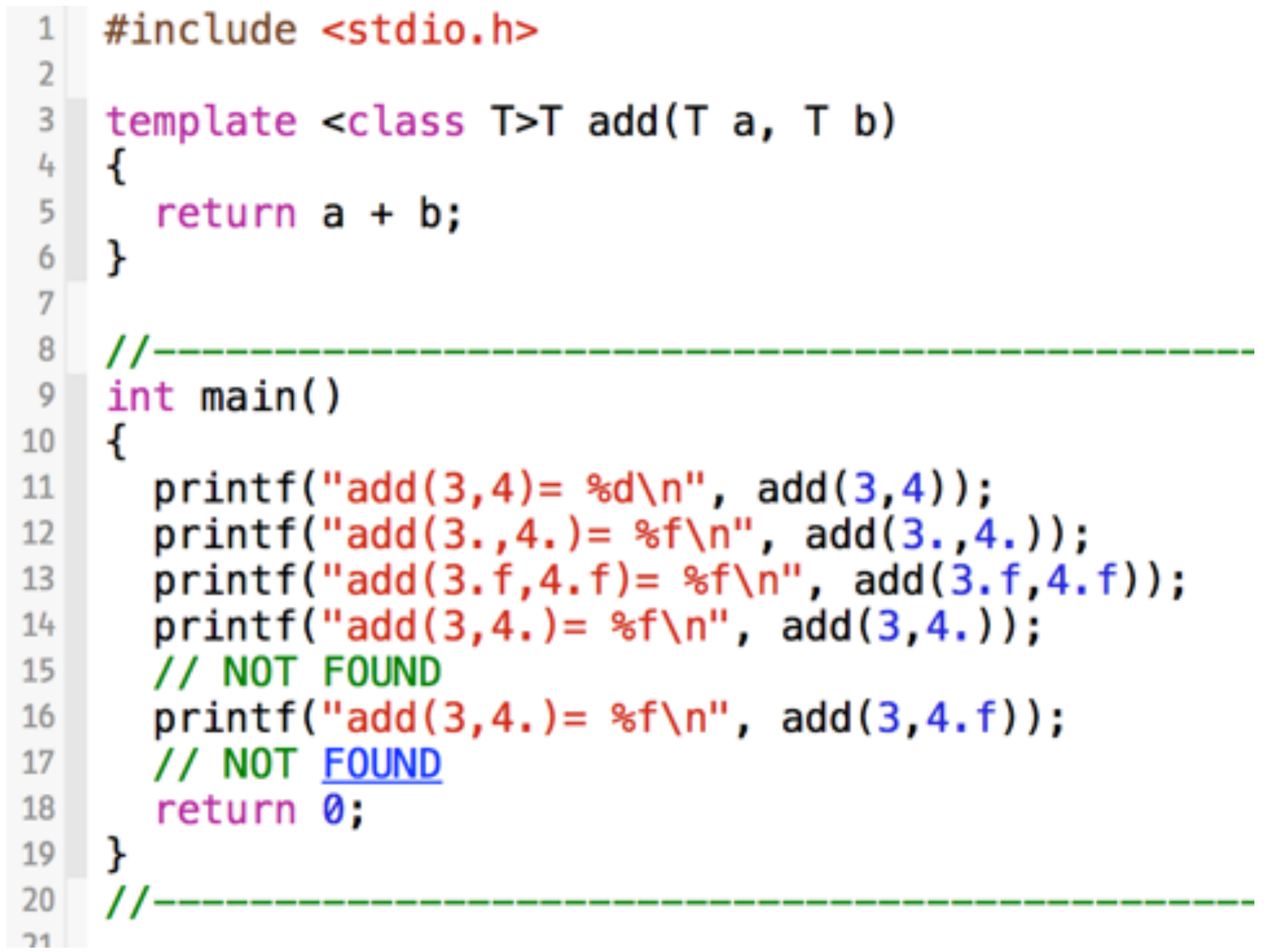


```cpp
template <class T>T add(T a, T b)
{
    return a + b;
}

//----------------------------------------------------------
int main()
{
    printf("add(3,4)= %d\n", add(3,4));
    printf("add(3.,4.)= %f\n", add(3.,4.));
    printf("add(3.f,4.f)= %f\n", add(3.f,4.f));
    //printf("add(3,4.)= %f\n", add(3,4.));
    // NOT FOUND
    //printf("add(3,4.)= %f\n", add(3,4.f));
    // NOT FOUND
    return 0;
}
    //------------------------------------------------------
```



```cpp
template <class T>T add(T a, T b)
{
    return a + b;
}

//-----------------------------------------
int main()
{
    printf("add(3,4)= %d\n", add(3,4));
    printf("add(3.,4.)= %f\n", add(3.,4.));
    printf("add(3.f,4.f)= %f\n", add(3.f,4.f));
    //printf("add(3,4.)= %f\n", add(3,4.));
    // NOT FOUND
    //printf("add(3,4.)= %f\n", add(3,4.f));
    // NOT FOUND
    return 0;
}
    //-----------------------------------
```

```
nagal:programs>et1
add(3,4)= 7
add(3.,4.)= 7.000000
add(3.f,4.f)= 7.000000
```



```
 1  #include <stdio.h>
 2
 3  template <class T>T add(T a, T b)
 4  {
 5    return a + b;
 6  }
 7
 8  //-----------------------------------------
 9  int main()
10  {
11    printf("add(3,4)= %d\n", add(3,4));
12    printf("add(3.,4.)= %f\n", add(3.,4.));
13    printf("add(3.f,4.f)= %f\n", add(3.f,4.f));
14    printf("add(3,4.)= %f\n", add(3,4.));
15    // NOT FOUND
16    printf("add(3,4.)= %f\n", add(3,4.f));
17    // NOT FOUND
18    return 0;
19  }
20  //-----------------------------------------
```

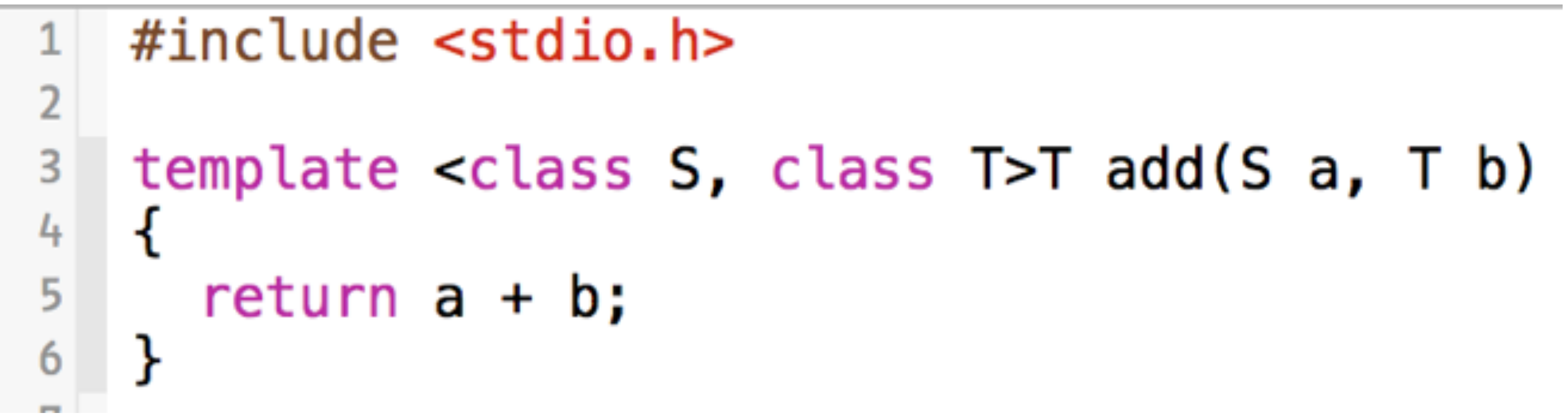
```
nagal:programs>g++ example_T1.cpp -oet1
example_T1.cpp:14:29: error: no matching function for call to 'add'
  printf("add(3,4.)= %f\n", add(3,4.));
                            ^~~
example_T1.cpp:3:21: note: candidate template ignored: deduced conflicting types for
parameter 'T'
      ('int' vs. 'double')
template <class T>T add(T a, T b)
                    ^
example_T1.cpp:16:29: error: no matching function for call to 'add'
  printf("add(3,4.)= %f\n", add(3,4.f));
                            ^~~
example_T1.cpp:3:21: note: candidate template ignored: deduced conflicting types for
parameter 'T'
      ('int' vs. 'float')
template <class T>T add(T a, T b)
                    ^
2 errors generated.
```

In the previous code, the following functions were generated:

- add(int, int)
  add(float, float)
  add(double, double)
- but these were not!
  add(int, float) is not found
  add(int, double) is not found
  add(double, float) is not found

Remedy

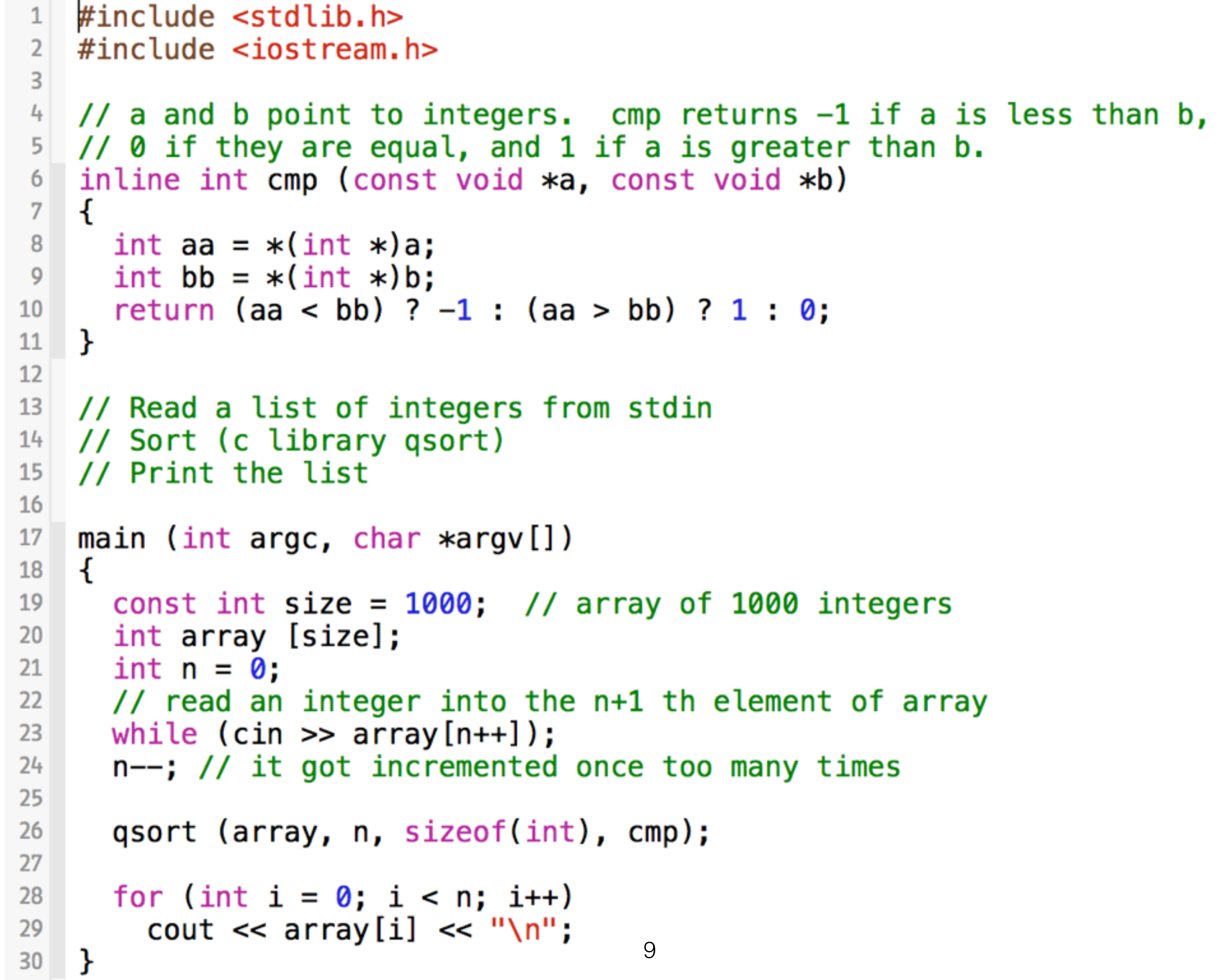
- template <class S, class T>
  T add(S a, T, b) { …. }
- What type should function return?
- Not a good function, although it will now handle add(5, 5.) which it could not previously



```
#include <stdio.h>

template <class S, class T>T add(S a, T b)
{
  return a + b;
}
```

```
nagal:programs>et1
add(3,4)= 7
add(3.,4.)= 7.000000
add(3.f,4.f)= 7.00000
add(3,4.)= 7.000000
add(3,4.)= 7.000000
```

STL



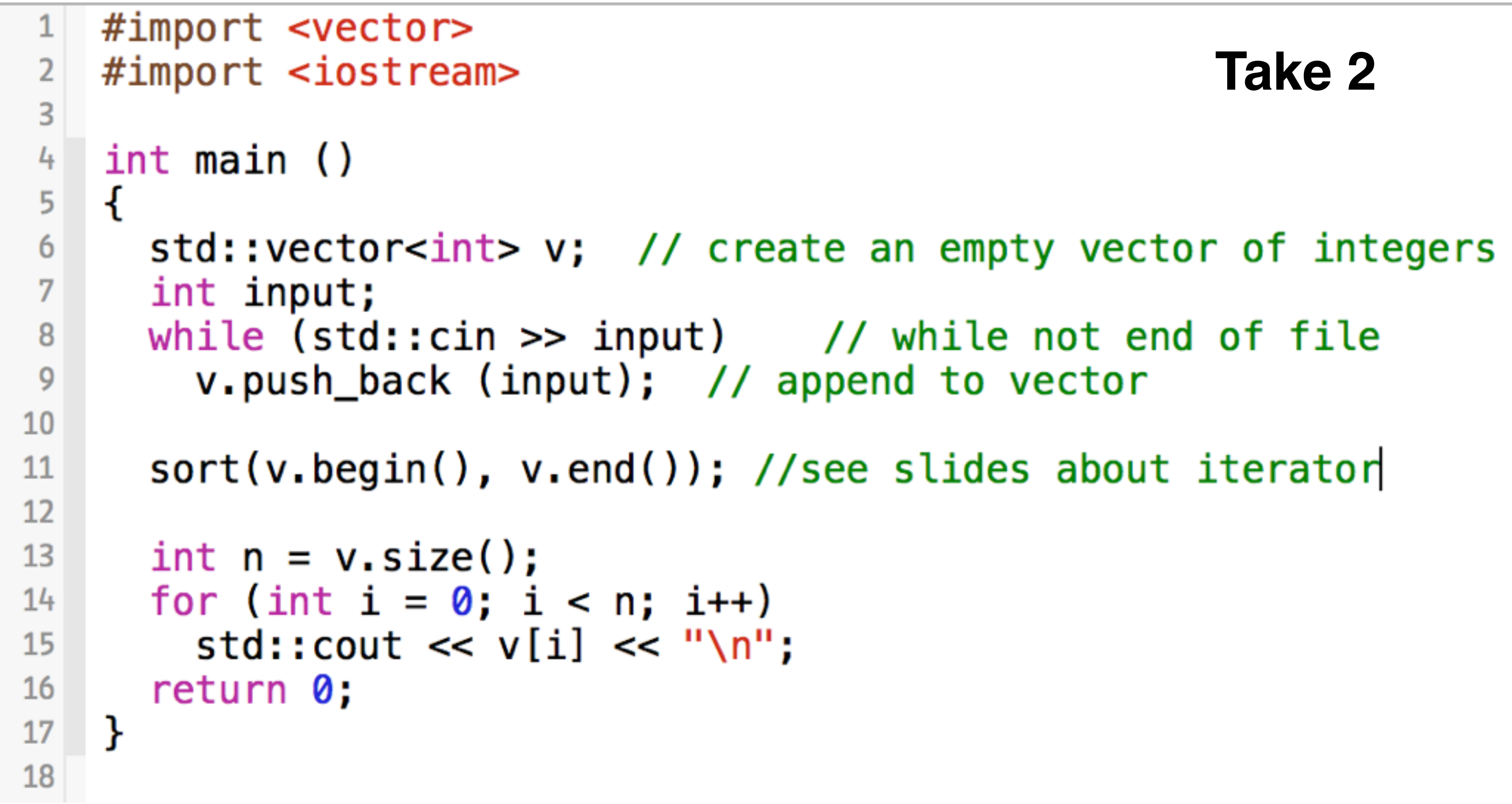
```
#include <stdlib.h>
#include <iostream.h>

// a and b point to integers.  cmp returns -1 if a is less than b,
// 0 if they are equal, and 1 if a is greater than b.
inline int cmp (const void *a, const void *b)
{
  int aa = *(int *)a;
  int bb = *(int *)b;
  return (aa < bb) ? -1 : (aa > bb) ? 1 : 0;
}

// Read a list of integers from stdin
// Sort (c library qsort)
// Print the list

main (int argc, char *argv[])
{
  const int size = 1000;  // array of 1000 integers
  int array [size];
  int n = 0;
  // read an integer into the n+1 th element of array
  while (cin >> array[n++]);
  n--; // it got incremented once too many times

  qsort (array, n, sizeof(int), cmp);

  for (int i = 0; i < n; i++)
    cout << array[i] << "\n";
}
```

Version 2: containers, iterators, algorithms

STL provides a number of container types, representing objects that contain other objects. One of these containers is a class called **vector** that behaves like an **array**, but can grow itself as necessary. One of the operations on vector is **push_back**, which pushes an element onto the end of the vector (growing it by one).
A vector contains a block of *contiguous* initialized elements -- if element index **k** has been initialized, then so have all the ones with indices **less than k**.
A vector can be presized, supplying the size at construction, and you can ask a vector how many elements it has with **size**. This is the *logical* number of elements -- the number of elements up to the highest-indexed one you have used. There is a also a notion of **capacity** -- the number of elements the vector can hold before reallocating.

Let's read the elements and push them onto the end of a vector. This removes the arbitrary limit on the number of elements that can be read. Also, instead of using **qsort**, we will use the STL **sort** routine, one of the many **algorithms** provided by STL. The sort routine is generic, in that it will work on many different types of containers. The way this is done is by having algorithms deal not with containers directly, but with ***iterators***.

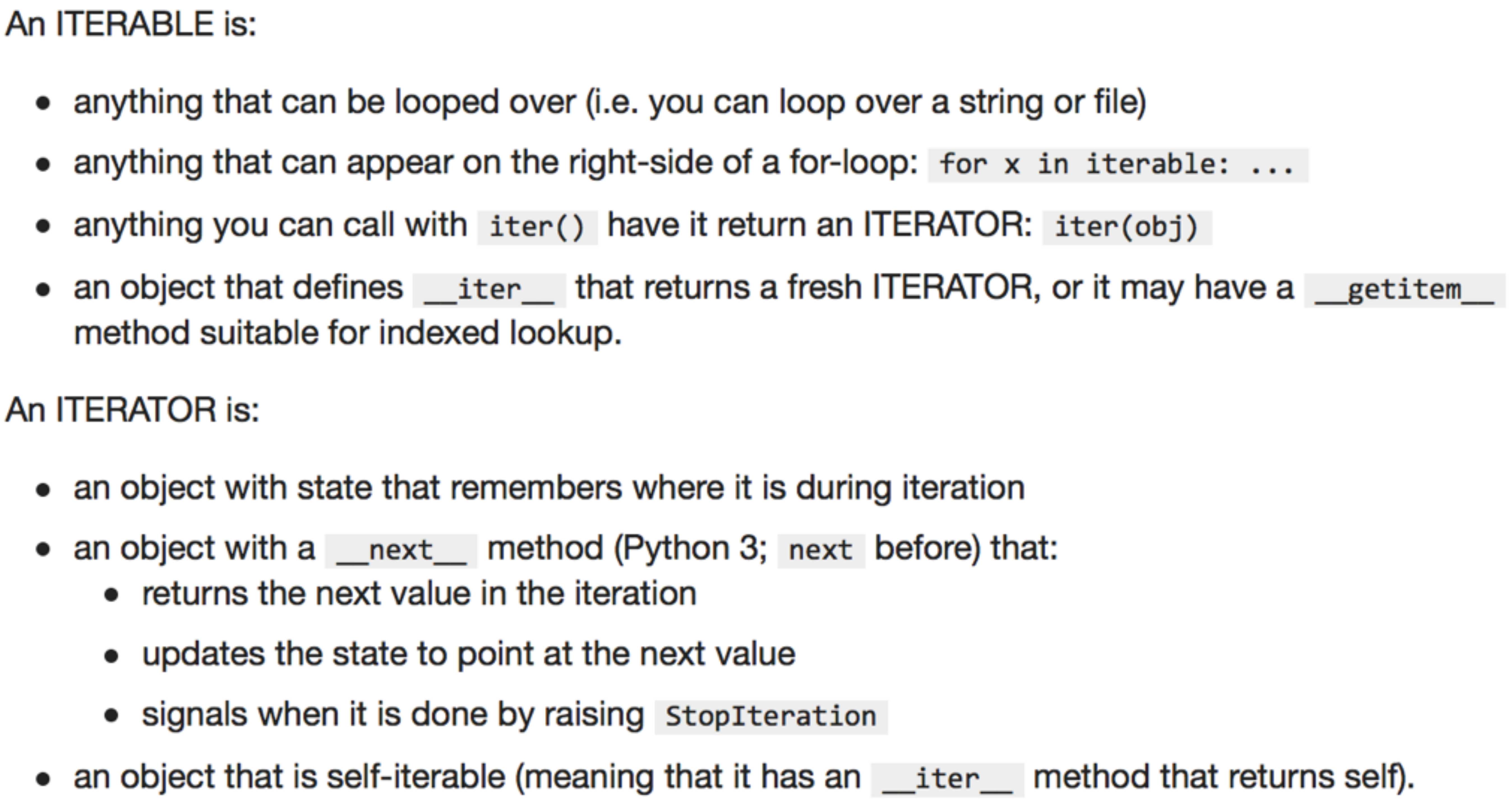
Take 2



```
#import <vector>
#import <iostream>

int main ()
{
  std::vector<int> v;  // create an empty vector of integers
  int input;
  while (std::cin >> input)    // while not end of file
    v.push_back (input);  // append to vector

  sort(v.begin(), v.end()); //see slides about iterator

  int n = v.size();
  for (int i = 0; i < n; i++)
    std::cout << v[i] << "\n";
  return 0;
}
```

Iterators

Remember in Python:



An ITERABLE is:

- anything that can be looped over (i.e. you can loop over a string or file)
- anything that can appear on the right-side of a for-loop: `for x in iterable: ...`
- anything you can call with `iter()` have it return an ITERATOR: `iter(obj)`
- an object that defines `__iter__` that returns a fresh ITERATOR, or it may have a `__getitem__` method suitable for indexed lookup.

An ITERATOR is:

- an object with state that remembers where it is during iteration
- an object with a `__next__` method (Python 3; `next` before) that:
  - returns the next value in the iteration
  - updates the state to point at the next value
  - signals when it is done by raising `StopIteration`
- an object that is self-iterable (meaning that it has an `__iter__` method that returns self).

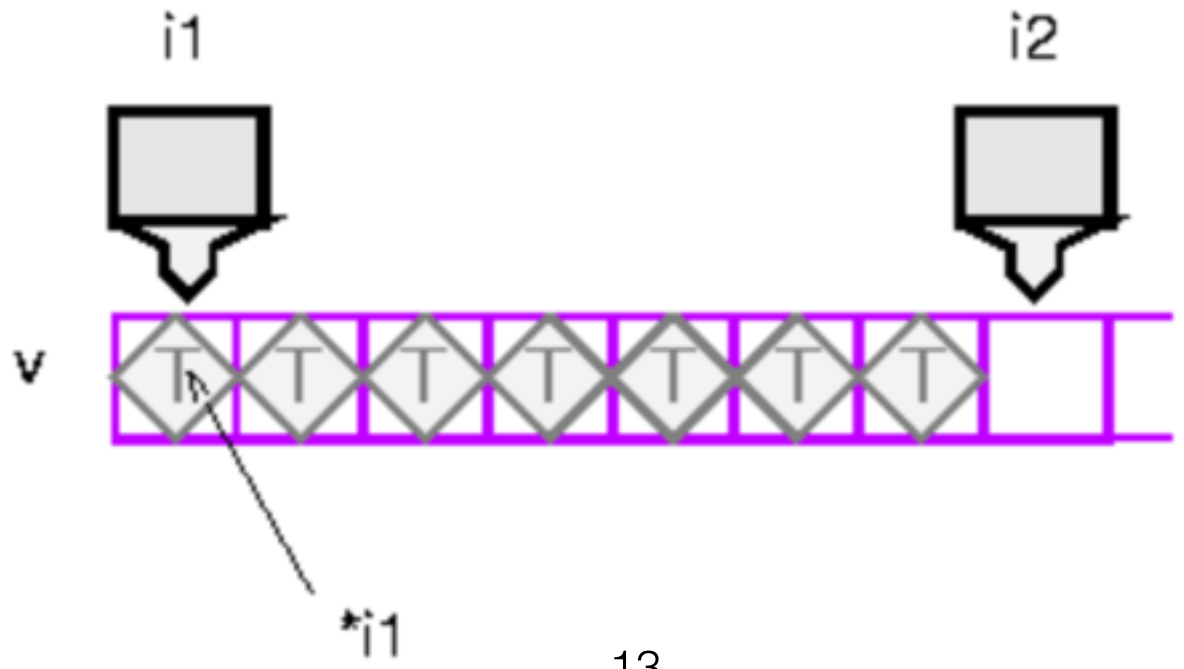
Iterators provide a way of specifying a position in a container.

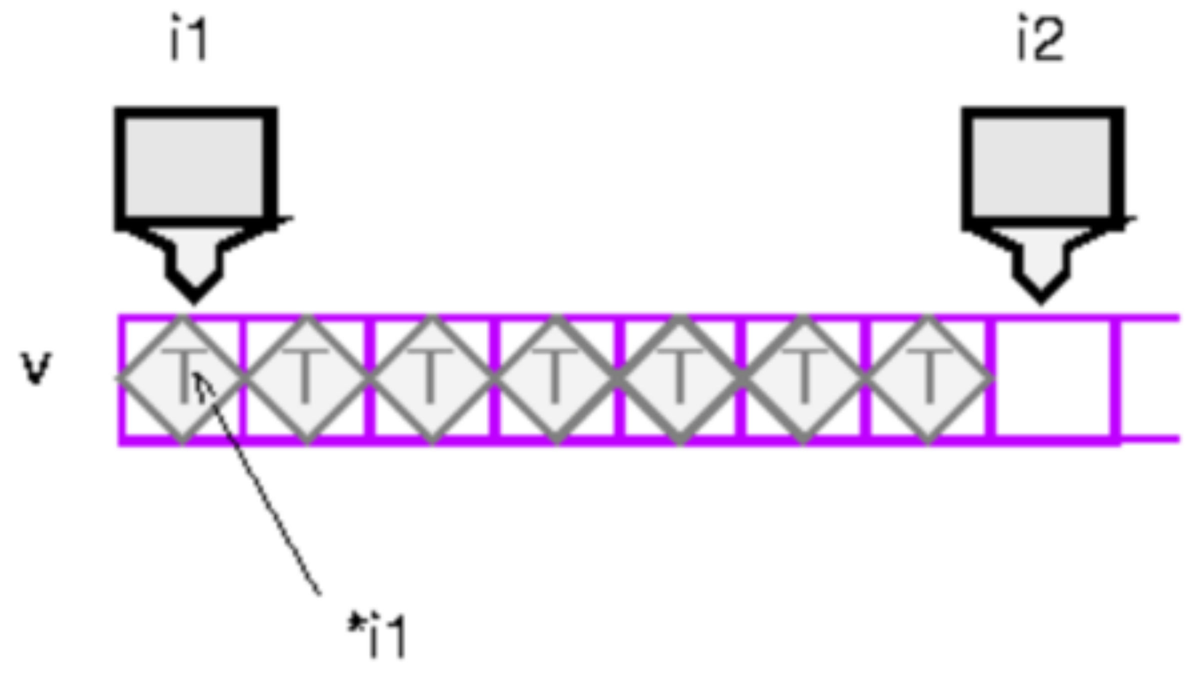
An iterator can be incremented or dereferenced, and two iterators can be compared.

There is a special iterator value called "past-the-end".
You can ask a vector for an iterator that points to the first element with the message begin. You can get a past-the-end iterator with the message end. The code

```
vector<int> v;
// add some integers to v
vector::iterator i1 = v.begin();
vector::iterator i2 = v.end();
```

will create two iterators like this:



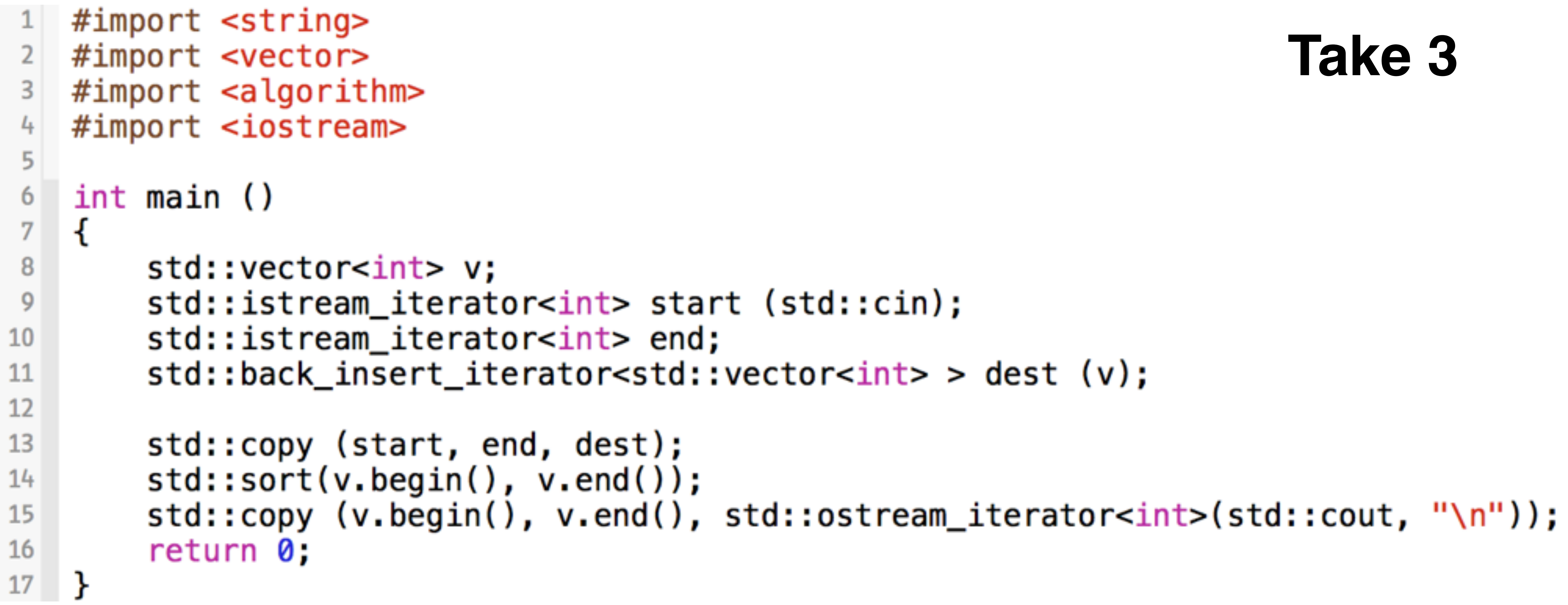


Operations like **sort** take two iterators to specify the source range. To get the source elements, they increment and dereference the first iterator until it is equal to the second iterator. Note that this is a semi-open range: it includes the start but not the end.

Two vector iterators compare equal if they refer to the same element of the same vector.

Incidentally, this is much faster than **qsort**; at least a factor of 20 on the examples I tried. This is presumably due to the fact that comparisons are done inline.

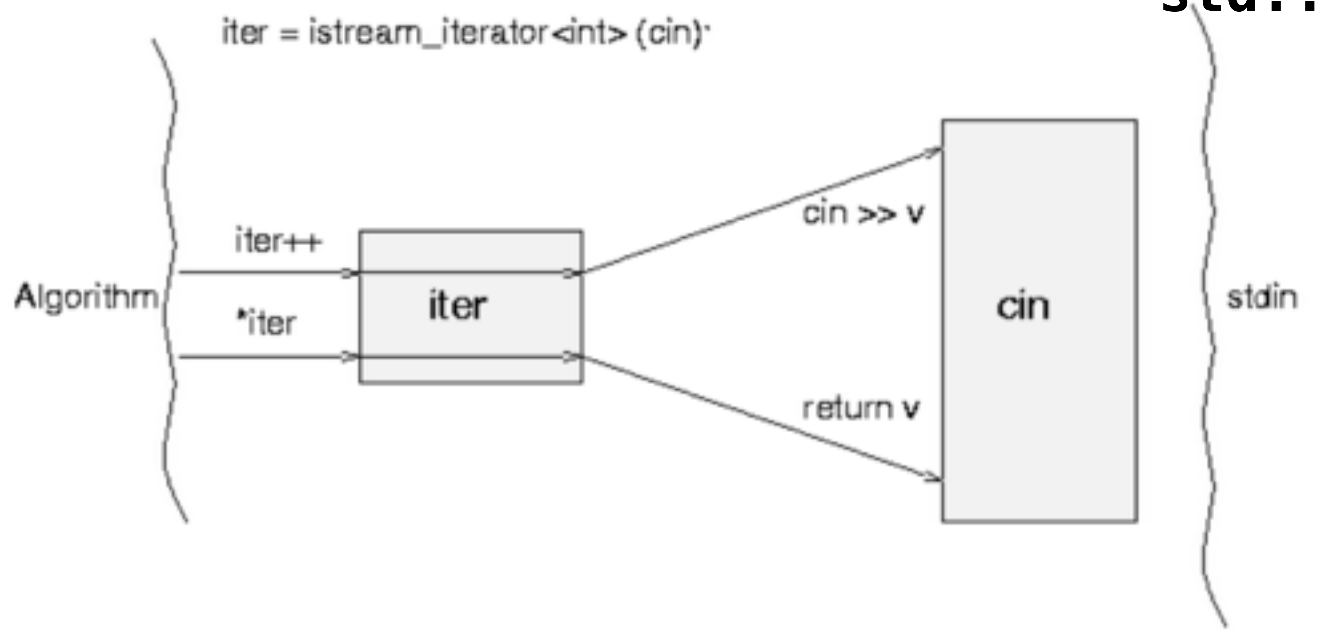
Take 3



```cpp
#import <string>
#import <vector>
#import <algorithm>
#import <iostream>

int main ()
{
    std::vector<int> v;
    std::istream_iterator<int> start (std::cin);
    std::istream_iterator<int> end;
    std::back_insert_iterator<std::vector<int> > dest (v);

    std::copy (start, end, dest);
    std::sort(v.begin(), v.end());
    std::copy (v.begin(), v.end(), std::ostream_iterator<int>(std::cout, "\n"));
    return 0;
}
```

In addition to iterating through containers, iterators can iterate over **streams**, either to read elements or to write them.
An **input stream like cin** has the right *functionality* for an input iterator: it provides access to a sequence of elements. The trouble is, it has the wrong *interface* for an iterator: operations that use iterators expect to be able to increment them and dereference them.
STL provides **adaptors**, types that transform the interface of other types. This is very much how electrical adaptors work. One very useful adaptor is istream_iterator. This is a template type (of course!); you parameterize it by the type of object you want to read from the **stream**. In this case we want **integers**, so we would use an **istream_iterator<int>**. Istream iterators are initialized by giving them a stream, and thereafter, dereferencing the iterator reads an element from the stream, and incrementing the iterator has no effect. An istream iterator that is created with the default constructor has the past-the-end value, as does an iterator whose stream has reached the end of file.

```
std::istream_iterator<int> start (std::cin);
```



In order to read the elements into the vector from standard input, we will use the STL `copy` algorithm; this takes three iterators. The first two specify the source range, and the third specifies the destination.
The names can get pretty messy, so make good use of typedefs. Iterators are actually parameterized on two types; the second is a distance type, which I believe is really of use only on operating systems with multiple memory models. Here is a typedef for an iterator that will read from a stream of integers:

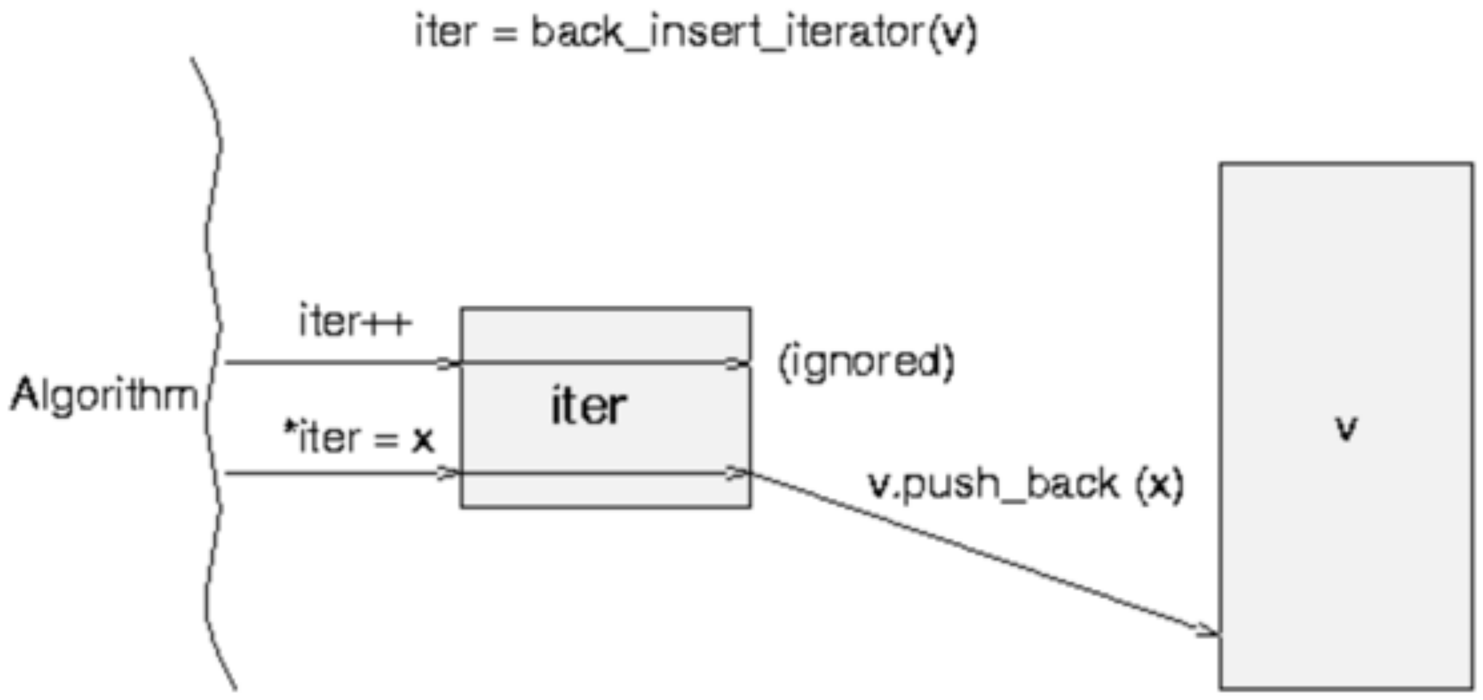
```
typedef istream_iterator<int> istream_iterator_int;
```

So to copy from standard input into a vector, we can do this:

```
 copy (istream_iterator_int (cin), istream_iterator_int (),v.begin());
```

The first iterator will be incremented and read from until it is equal to the second iterator. The second iterator is just created with the default constructor; this gives it the past-the-end value. The first iterator will also have this value when the end of the stream is reached. Therefore the range specified by these two iterators is from the current position in the input stream to the end of the stream.
There is a bit of a problem with the third iterator, though: if v does not have enough space to hold all the elements, the iterator will run off the end, and we will dereference a past-the-end iterator (which will cause a segfault).

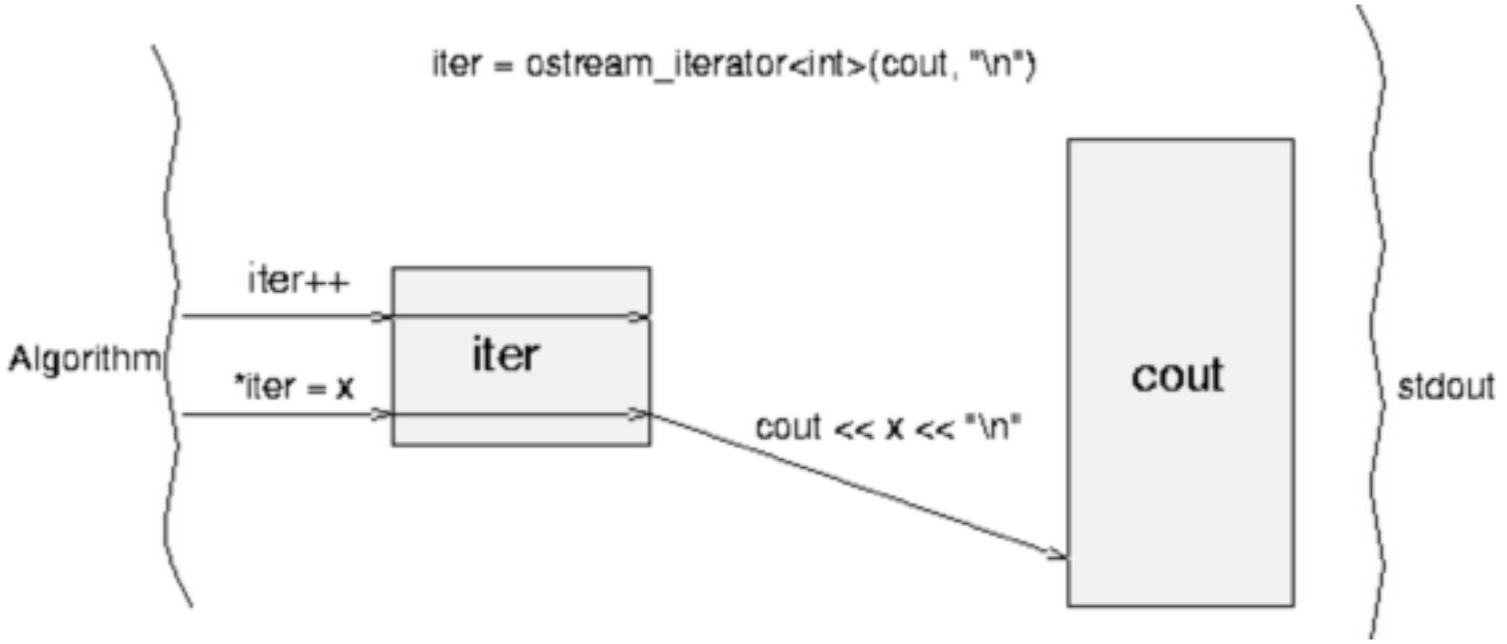
What we really want is an iterator that will do insertion rather than overwriting. There is an adaptor to do this, called back_insert_iterator. This type is parameterized by the container type you want to insert into.



So input is done like this:

```
typedef istream_iterator<int> istream_iterator_int;

vector<int> v;
istream_iterator_int start (cin);
istream_iterator_int end;
back_insert_iterator<vector<int> > dest (v);

copy (start, end, dest);
```

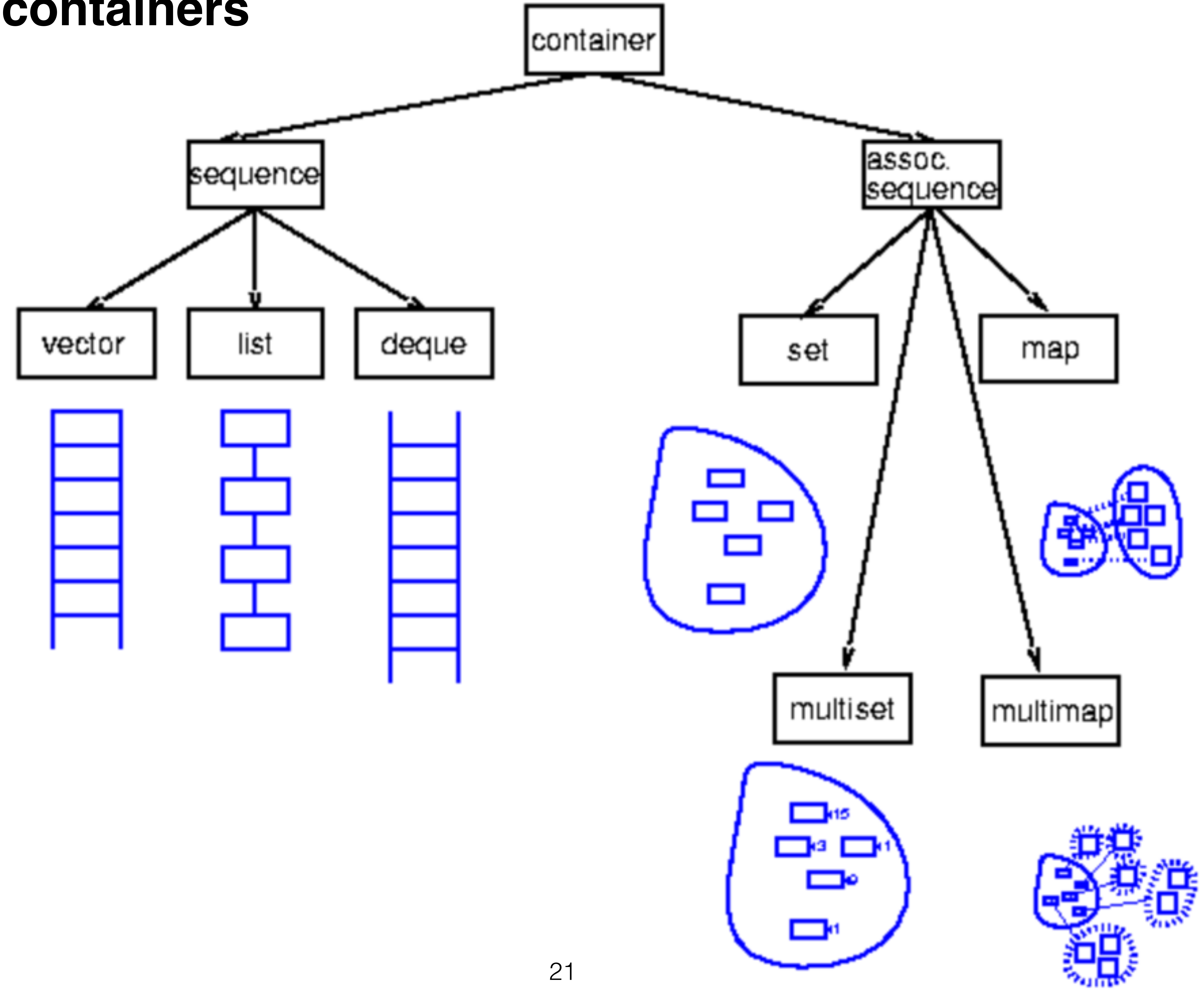
`ostream_iterator` is another adaptor; it provides **output iterator functionality**: assigning to the dereferenced iterator will write the data out. The `ostream_iterator` constructor takes two arguments: the stream to use and the separator. It prints the separator between elements.



```
#import <string>
#import <vector>
#import <algorithm>
#import <iostream>

int main ()
{
    std::vector<int> v;
    std::istream_iterator<int> start (std::cin);
    std::istream_iterator<int> end;
    std::back_insert_iterator<std::vector<int> > dest (v);

    std::copy (start, end, dest);
    std::sort(v.begin(), v.end());
    std::copy (v.begin(), v.end(), std::ostream_iterator<int>(std::cout, "\n"));
    return 0;
}
```

Containers are objects that conceptually contain other objects. They use certain basic properties of the objects (ability to copy, etc.) but otherwise do not depend on the type of object they contain.
STL containers may contain pointers to objects, though in this case you will need to do a little extra work.
vectors, lists, deques, sets, multisets, maps, multimaps, queues, stacks, and priority queues, did I miss any? are all provided.
Perhaps more importantly, built-in containers (C arrays) and user-defined containers can also be used as STL containers; this is generally useful when applying operations to the containers, e.g., sorting a container. Using user-defined types as STL containers can be accomplished by satisfying the requirements listed in the STL container requirements definition.
If this is not feasible, you can define an adaptor class that changes the interface to satisfy the requirements.
All the types are "templated", of course, so you can have a vector of ints or Windows or a vector of vector of sets of multimaps of strings to Students. Sweat, compiler-writers, sweat!
To give you a brief idea of the containers that are available, here is the hierarchy:

STL containers



Sequences
Contiguous blocks of objects; you can insert elements at any point in the sequence, but the performance will depend on the type of sequence and where you are inserting.

Vectors
Fast insertion at end, and allow random access.

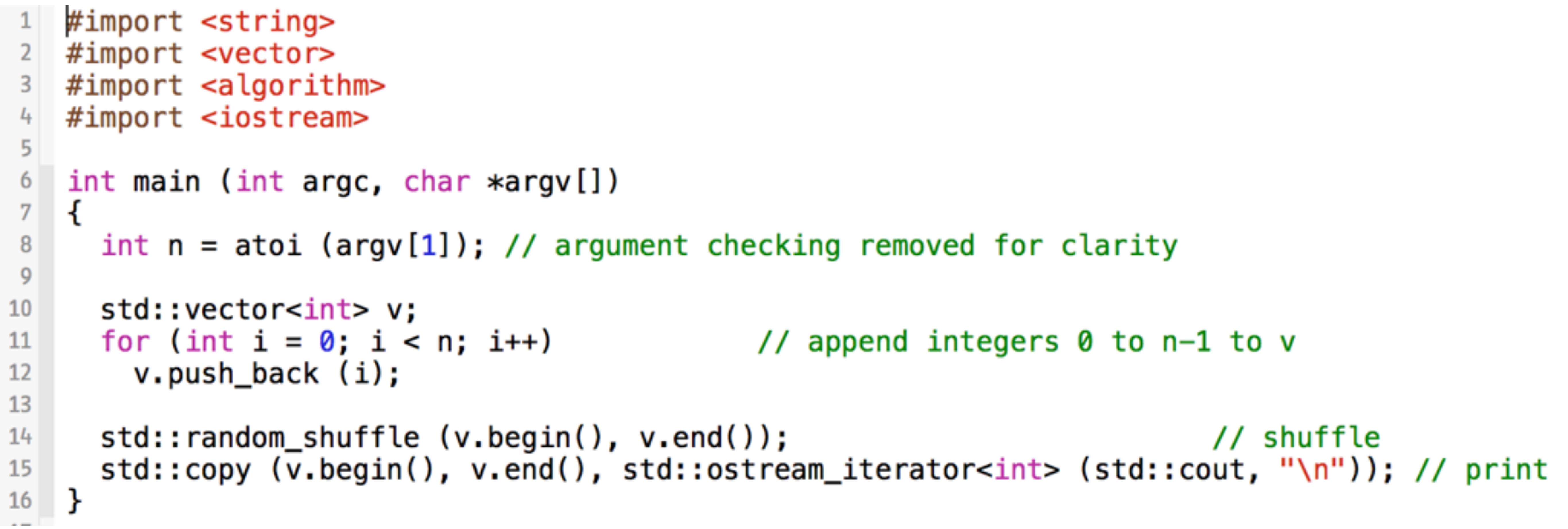
Lists
Fast insertion anywhere, but provide only sequential access.

Deques
Fast insertion at either end, and allow random access. Restricted types, such as stack and queue, are built from these using adaptors.

Stacks and queues
Provide restricted versions of these types, in which some operations are not allowed.

Examples using containers

Here is a program that generates a random permutation of the first `n` integers, where `n` is specified on the command line.



```cpp
#import <string>
#import <vector>
#import <algorithm>
#import <iostream>

int main (int argc, char *argv[])
{
  int n = atoi (argv[1]); // argument checking removed for clarity

  std::vector<int> v;
  for (int i = 0; i < n; i++)            // append integers 0 to n-1 to v
    v.push_back (i);

  std::random_shuffle (v.begin(), v.end());                        // shuffle
  std::copy (v.begin(), v.end(), std::ostream_iterator<int> (std::cout, "\n")); // print
}
```

This program creates an empty vector and fills it with the integers from 0 to `n`. It then shuffles the vector and prints it out.

Generators

Algorithms like `generate` walk through a range, calling a function object at each step, and assigning the result of the function to the current element.

For example, here is a function that always returns 0:

```
int zero() { return 0; }
```

To fill a vector with zeroes, one could use the algorithm `generate` with the function object zero:

```
vector<int> v (100);
generate (v.begin(), v.end(), zero);
```

Predicates

The second type of function object is used to test things; the parenthesis operator will be defined to return something that can be tested for truth.

`find_if` uses a function object to test each element of a range, returning an iterator pointing to the first element that satisfies the test. In this case, the function object takes an argument, the element of the range, and returns a boolean:

```
bool greaterThanZero (int i) return i > 0;
```

This could be used to move to the first strictly positive element of a range:

```
typedef vector<int>::iterator iterator;
typedef vector<int> vector;
typedef ostream_iterator<int> output;
vector v;
iterator iter = find_if (v.begin(), v.end(), greaterThanZero);
if (iter == v.end())
  cout << "no elements greater than zero" << endl;
else
  {
    cout << "elements starting from first greater than zero: ";
    copy (iter, v.end(), output (cout, " "));
  }
```

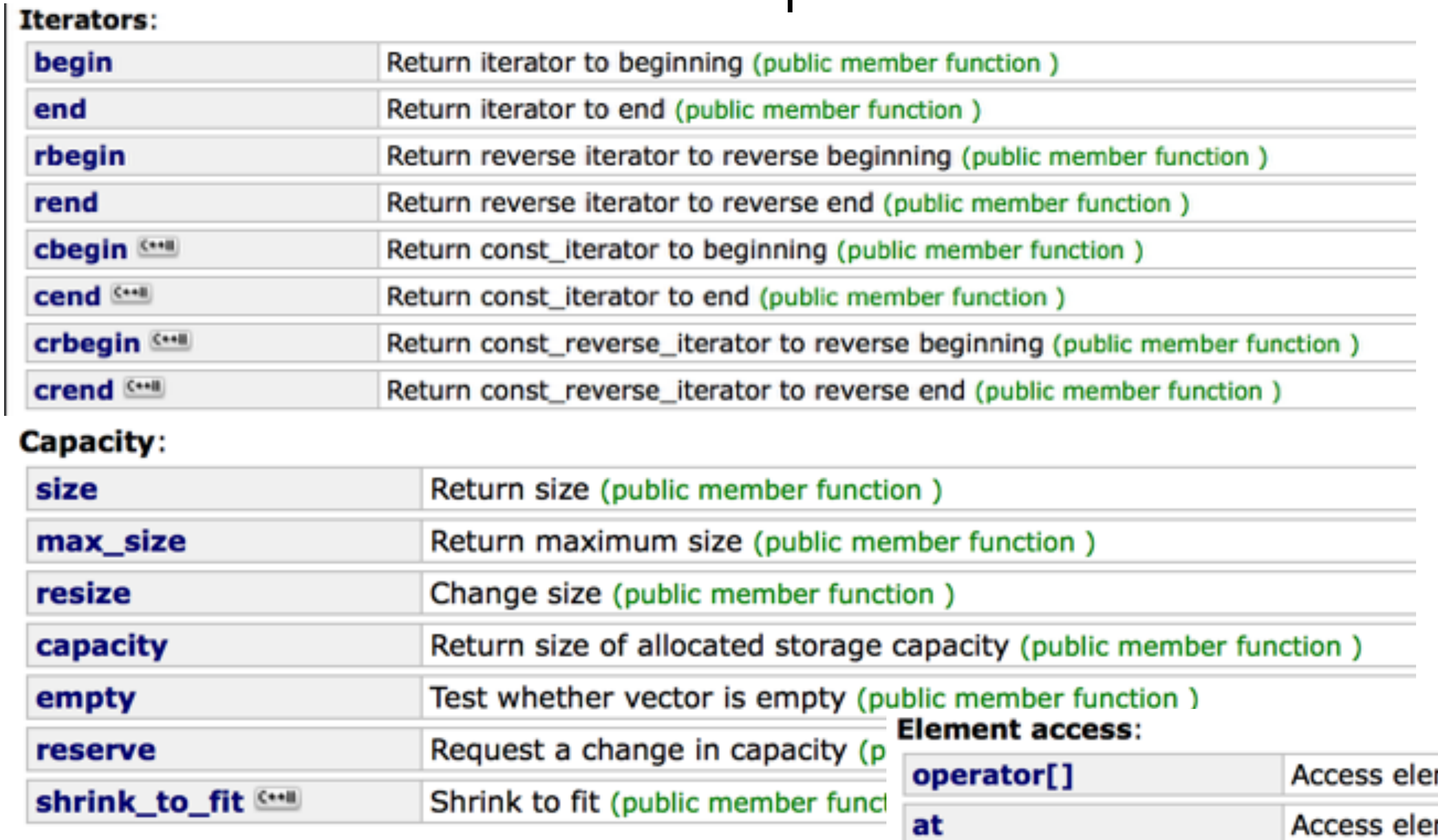
Again, it is often useful to be able to provide state in the predicate object. Here is a predicate that tests true if the element is within a specified range:

```
class InRange
{
    const T& low;
    const T& high;
  public:
    InRange (const T& l, const T& h) : low (l), high (h) { }
    bool operator()(const T& t) { return ! (t < l) && t < h; }
};
```

Here we find, and print, all the elements of a vector that fall within a particular range:

```
typedef vector<int>::iterator iterator;
typedef vector<int> vector;
typedef ostream_iterator<int> output;

vector v (100);
generate (v.begin(), v.end(), rand);

iterator iter (v);
while (iter != v.end())
{
  iter = find (v.begin(), v.end(), InRange (0, 10000));
  cout << *iter << endl;
}
```

For example functions in VECTOR class

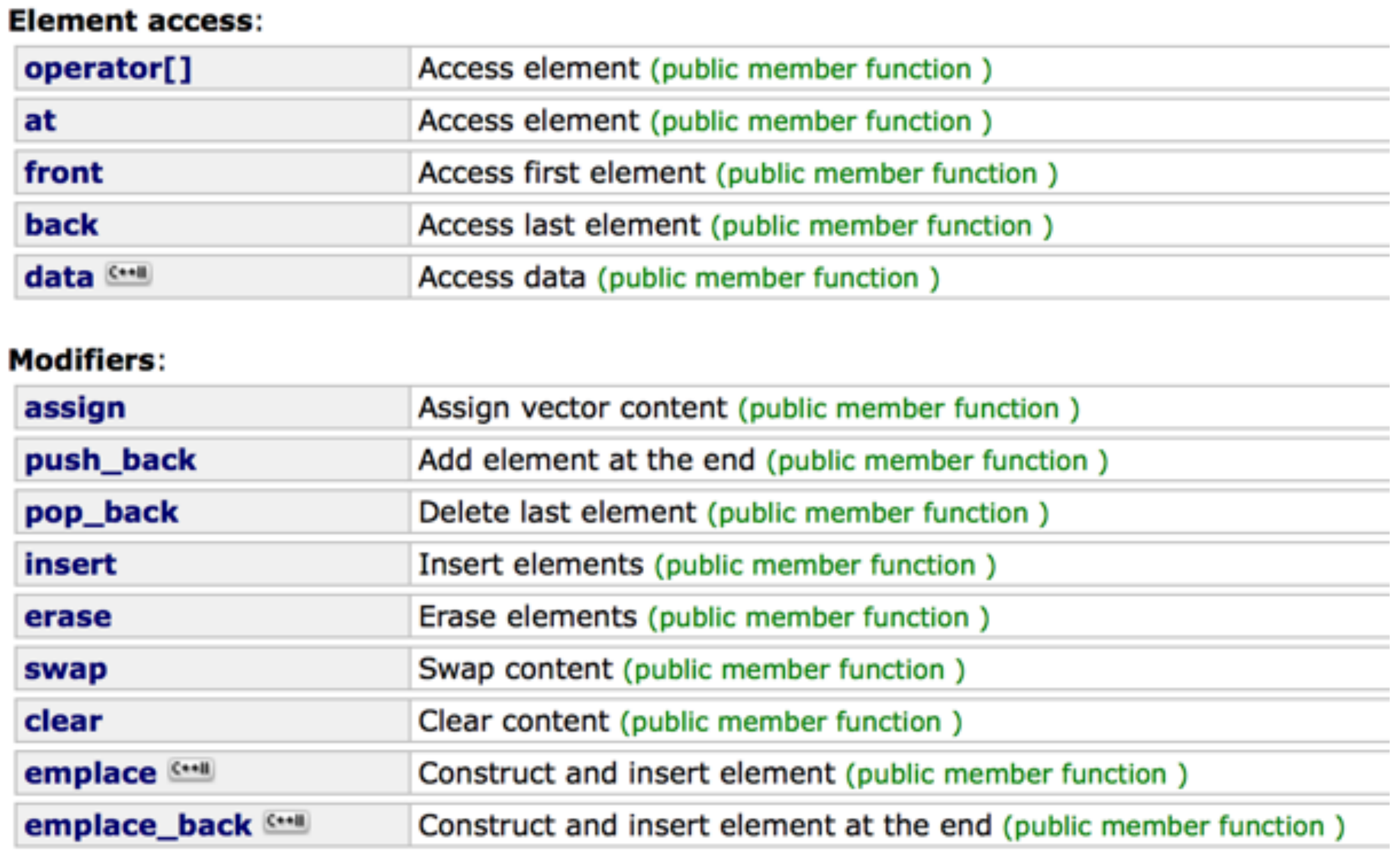


Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (p
shrink_to_fit C++11	Shrink to fit (public member funct

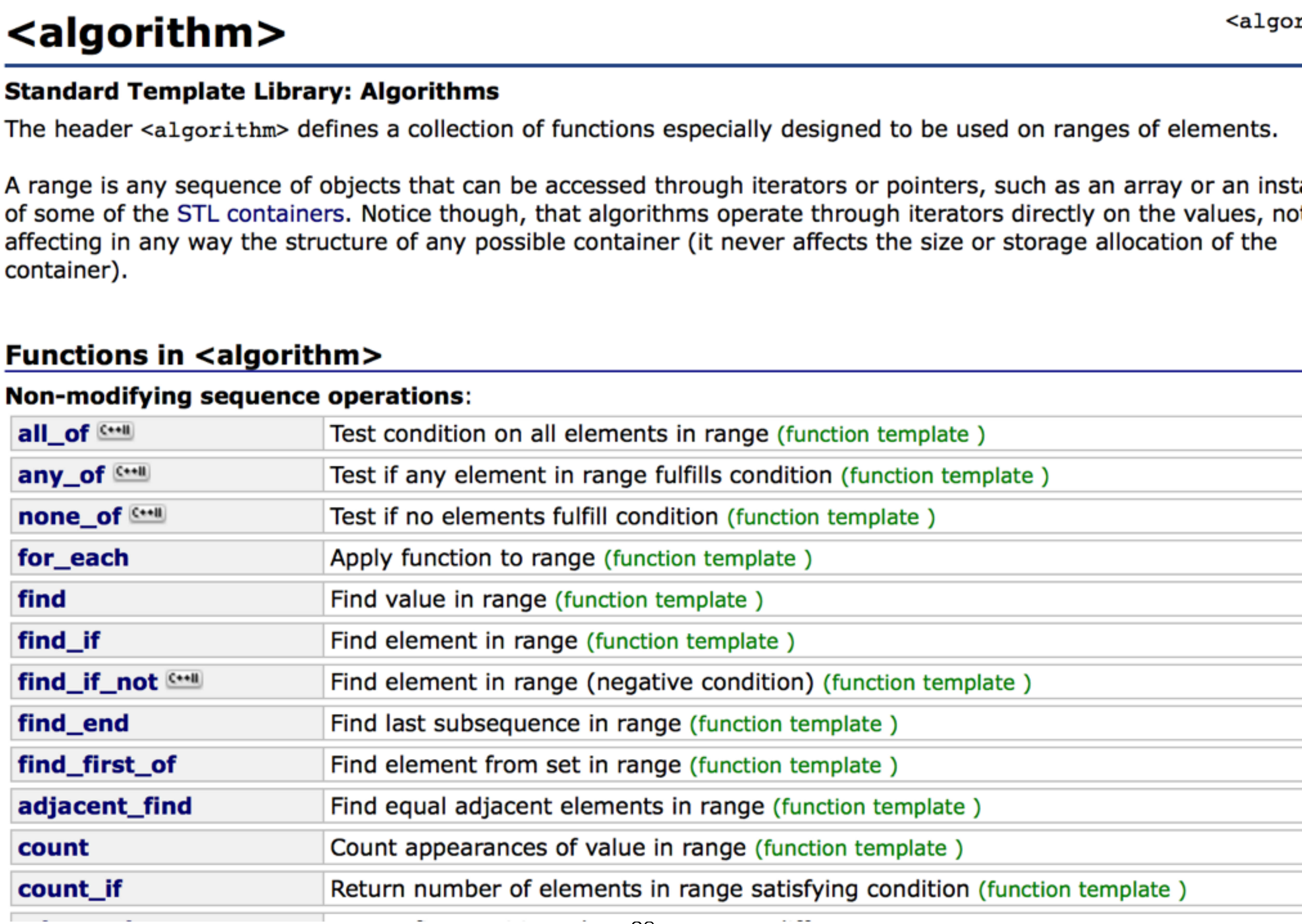


Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)



<algorithm>

<algor

Standard Template Library: Algorithms

The header `<algorithm>` defines a collection of functions especially designed to be used on ranges of elements.

A range is any sequence of objects that can be accessed through iterators or pointers, such as an array or an insta
of some of the STL containers. Notice though, that algorithms operate through iterators directly on the values, no
affecting in any way the structure of any possible container (it never affects the size or storage allocation of the
container).

Functions in <algorithm>

Non-modifying sequence operations:

all_of C++11	Test condition on all elements in range (function template)
any_of C++11	Test if any element in range fulfills condition (function template)
none_of C++11	Test if no elements fulfill condition (function template)
for_each	Apply function to range (function template)
find	Find value in range (function template)
find_if	Find element in range (function template)
find_if_not C++11	Find element in range (negative condition) (function template)
find_end	Find last subsequence in range (function template)
find_first_of	Find element from set in range (function template)
adjacent_find	Find equal adjacent elements in range (function template)
count	Count appearances of value in range (function template)
count_if	Return number of elements in range satisfying condition (function template)