What does this do?

// n is the length of the array
void whatDoesThisDo (int x[], int n) {
    for (auto pass=1; pass < n; pass++) {
        for (auto i=0; i < n-pass; i++) {
            if (x[i] > x[i+1]) {
                auto temp = x[i];
                x[i] = x[i+1];
                x[i] = temp;
            }
        }
    }
}
What does this do?

// n is the length of the array
void whatDoesThisDo (int x[], int n) {
    for (auto pass=1; pass < n; pass++) {
        for (auto i=0; i < n-pass; i++) {
            if (x[i] > x[i+1]) {
                auto temp = x[i];
                x[i] = x[i+1];
                x[i+1] = temp;
            }
        }
    }
}

Supposing that we would like a general solution to sorting n things in ascending order, in what two ways is the above solution very specific?
“The STL is based on generalization.

Arrays are generalized into containers and parameterized on the types of objects they contain.

Functions are generalized into algorithms and parameterized on the types of iterators they use.

Pointers are generalized into iterators and parameterized on the type of objects they point to.”

— Scott Meyer, Effective STL
1 Preliminaries
   • Templates
   • Nested Type Definitions

2 Containers

3 Iterators

4 The vector Container
   • vector versus Array
   • Constructing a vector
   • Inserting and Deleting Elements
   • vector Traversal

5 Algorithms
   • Ranges
   • Algorithms, Containers and Iterators
   • Using Algorithms
template <typename T> // template parameter list
t T bigger(const T& first, const T& second)
{
    if (first > second)
        return first;
    else
        return second;
}

int main()
{
    auto e = 5.62, f = 3.48;
    cout << bigger(e, f) << endl;
    auto g = 'A', h = 'a';
    cout << bigger(g, h) << endl;
    // ...

Provide *synonyms* or aliases for primitive or user-defined types.

```cpp
using Wages = double;

auto hourly = Wages{100};
auto daily = Wages{800};
```

Improve readability of template declarations:

```cpp
using ComplexNumber = complex<float>;
auto a = ComplexNumber{1,-1};

using MarksContainer = vector<int>;
auto elen3009 = MarksContainer{};
```
```cpp
class Person {
public:
    using Age = float; // nested type definition

    Person(const string& name, Age age):
        name_(name),
        age_(age)
    {}

    string name() const { return name_; }
    Age age() { return age_; }
    void age(Age age) { age_ = age; }
private:
    string name_;
    Age age_;
};
```
Accessing a nested type:

```cpp
auto jeanne = Person{"Jeanne", 25};

// auto deduces Person::Age
auto age_in_years = jeanne.age();

cout << age_in_years << endl;
```

Nested types give us the ability to hide type information. See:
- `string::size_type` in Lab 1, Screen class
- `iterator types` for containers
1 Preliminaries
   - Templates
   - Nested Type Definitions

2 Containers

3 Iterators

4 The vector Container
   - vector versus Array
   - Constructing a vector
   - Inserting and Deleting Elements
   - vector Traversal

5 Algorithms
   - Ranges
   - Algorithms, Containers and Iterators
   - Using Algorithms
Sequence Containers - hold an ordered collection of elements of a single type
- **vector** - contiguous memory
- **deque** - contiguous memory
- **list, slist** - non-contiguous memory

Associative Containers - support efficient query as to the presence and retrieval of an element
- **map** - key/value pair
- **set** - a single key value
**vector** — insertion at the back is efficient. Expensive to insert or delete an item anywhere else. Random access to elements.

**deque** — efficient insertions and deletions at the *front* and back.

**list** — efficient insertions and deletions anywhere. Sequential access to elements.
Preliminaries
- Templates
- Nested Type Definitions

Containers

Iterators

The vector Container
- vector versus Array
- Constructing a vector
- Inserting and Deleting Elements
- vector Traversal

Algorithms
- Ranges
- Algorithms, Containers and Iterators
- Using Algorithms
Iterators

Provide a common way of sequentially accessing each element within a container. Every standard container defines two associated iterator types:

1. `container-type::iterator`
2. `container-type::const_iterator`

```cpp
vector<string> vec;
// obtain an iterator, iter is of type: vector<string>::iterator
auto iter = begin(vec);
```

Iterator is a type nested inside the vector class, an iterator instance is retrieved with the `begin` function.

All containers provide iterators which minimally support:

- `++iter`
- `*iter`
- `iter != iter2`
- `iter == iter2`
4 The vector Container
- vector versus Array
- Constructing a vector
- Inserting and Deleting Elements
- vector Traversal

5 Algorithms
- Ranges
- Algorithms, Containers and Iterators
- Using Algorithms
vector is Modelled on an Array

But there are significant differences:

- Array size is constant and known at compile-time
  ```cpp
  const int size = 100;
  int integers[size]; // array of 100 integers
  ```
- Built-in array type is not a fully-fledged abstraction
  - No self-knowledge eg: size, boundaries
  - Assignment operator (\=) not defined
- Function return type cannot be an array

vector:

- Self-growing
- Knows its own size, how its elements can be accessed, etc
- Can be assigned and returned from functions

Prefer vector to arrays

If you must use an array, use the STL::array
Creating and Initialising vectors

#include <vector>

// empty vector - typical use
auto integers = vector<int>{};

// using an initializer list
auto first_five_integers = vector<int>{1, 2, 3, 4, 5};

// a vector of vectors
auto vector2D = vector<vector<int>>{};

See the reference sheet for other ways of creating vectors
At the back of the vector

```cpp
auto text_word = "s;  
while (cin >> text_word) {  
    svec.push_back(text_word); // append element  
}  
```

This is the most **efficient** form of insertion for a vector.

At a specific position

```cpp
auto vegetable = string{"carrot"};  
// the element is inserted in front of the iterator that is passed  
svec.insert(begin(svec), vegetable);  
```
Deleting Elements

At the back of the vector (most efficient deletion)

svec.popback();

At a specific position

iter = svec.erase(iter)
// iter now points to the element just after the one that has been erased

All elements

svec.clear();

A range of elements can also be deleted/inserted
This program attempts to delete elements in the vector which are less than 5. What will the output be, and why is it not correct?

```cpp
auto random_integers = Numbers{3, 1, 5, 2, 7};

auto it = begin(random_integers);
while (it != end(random_integers))
{
    if (*it < 5)
    {
        random_integers.erase(it);
        it++;
    }
    else it++;
}
```
Give the output of the following code. What does z represent in each of the loops and what is its type?

```cpp
// using the array idiom
for (auto z = 0u; z != even_numbers.size(); ++z) {
    cout << even_numbers[z] << " ";
}

// using iterators
for (auto z = begin(even_numbers); z != end(even_numbers); ++z) {
    cout << *z << " ";
}

// using a range-based for loop - prefer where possible
for (auto z : even_numbers) {
    cout << z << " ";
}
```
Capturing Elements When Using a Range-Based For Loop

// observing primitive types - capture by value
for (auto element : even_numbers) {
    cout << element << " ";
}

// observing complex types - capture by const reference
auto names = vector<string>{{"Refilwe", "Jeff"}};
for (const auto& element : names) {
    cout << element << " ";
}

// modifying elements in place - capture by (non-const) reference
for (auto& element : names) {
    element = "Hi " + element + "!";
}
1 Preliminaries
   • Templates
   • Nested Type Definitions

2 Containers

3 Iterators

4 The vector Container
   • vector versus Array
   • Constructing a vector
   • Inserting and Deleting Elements
   • vector Traversal

5 Algorithms
   • Ranges
   • Algorithms, Containers and Iterators
   • Using Algorithms
for (int \( i = 1; i <= 10; i++ \))
    cout << "This loop does something 10 times." << endl;

for (int \( i = 0; i != 10; i++ \))
    cout << "This loop also does something 10 times." << endl;

First for loop is \textbf{symmetrical}: uses the range: \([1, 10]\)

Second for loop is \textbf{asymmetrical}: uses the range: \([0, 10]\)

\textbf{Number of elements:}

- Symmetrical range \([m, n]\) has \(n - m + 1\) repetitions/elements
- Asymmetrical range \([m, n)\) has \(n - m\) repetitions/elements
Advantages of Asymmetrical Ranges

Asymmetrical ranges are easier to deal with:

- arrays, vectors and other containers use 0-based indices
- range index is in a known state after loop finishes
- *empty* ranges are expressed as \([n, n)\) rather than \([n, n - 1]\)
begin(vec) returns an iterator pointing to the first element of a container.
end(vec) returns an iterator pointing to one past the last element of a container.
This forms an asymmetrical range: [first, last)

// asymmetrical empty ranges are treated uniformly
while (begin != end)
{
    // do something with the element begin refers to
    // .
    ++begin;
}

Container member functions and algorithms which use iterator pairs always assume
asymmetrical ranges.
Algorithms, Containers and Iterators

- **Algorithms**
  - Use
  - **Iterators**
    - Provide
  - **Containers**
Algorithms are *generic* because they depend on iterators and have no knowledge of the container which those iterators are pointing into — they are not member functions of vector, list etc.

- Searching, sorting, substitution, etc are declared in `<algorithm>`
- Numeric algorithms are declared in `<numeric>`

```cpp
#include <numeric>
// add elements
auto sum = accumulate(begin(ivec), end(ivec), 0);

#include <algorithm>
// find an element
auto result = find(begin(ivec), end(ivec), 12);
```
The transform algorithm (refer to your reference sheet)

transform(first, last, result, op)

first and last represent a pair of iterators which specify a range. result represents an iterator pointing to the start of the output range.

transform applies the function op to each of the elements in the specified range and stores the value returned by op in the range that begins at result. op can either be a function pointer or a function object. The transform function allows for the destination range to be the same as the input range to make transformations in place. transform returns an iterator pointing to the element that follows the last element written in the result sequence.
Determine the length of each string in a vector

Given:

```cpp
auto words = vector<string> {"This", "is", "a", "short", "sentence");
```

Write a program to output the length of each of the above strings. Make use of the `transform` function.

Hint: first write a function to calculate the length of a single string. The name of this function is a pointer to the function and can be passed to `transform` as the `op` argument.

Make sure that your vector for storing the results is large enough.
**Challenges**

**Lambda Functions**
Rewrite the `transform` example to use a lambda function instead of an ordinary function. Post your answer on the forum, and explain it.

**Generic Bubble Sort**
Write a generic bubble sort which works with different containers and types, and post it on the forum. Test it on a linked list of strings and a vector of floats.