CS3520 Programming in C++
Vectors

Ken Baclawski
Fall 2016

Outline

- Word median program
  - Vector template
  - Index notation
- Median program
  - Sorting
  - Unsigned integers
  - Setting precision
- Stacker program
  - Stack template
  - Proving termination
  - Conversions
- Requirements

Word Median Program

Specification

- Read words from the standard input
- Compute the median (middle) word(s).
  - No words: Undefined
  - Odd number of words: Exactly one median word
  - Even number of words: Exactly two median words
- If no words are entered, then print “No words were entered.” Status code is 1.
- Otherwise, print the median word(s) in order, separated by a space if there are two of them. Status code is 0.

Includes

- This program requires two new libraries:
  - The vector library is for lists of objects
  - The algorithm library has many algorithms for searching, sorting, copying, max/min, etc.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

The `vector` template

- Vector is a *template*
  - It is a list of objects of some type
  - The type must be specified in angle brackets
  - Default initialization is to an empty list
- Similar to the Java ArrayList generic class
- The `push_back` method adds a new object to the end of the list
  - The list is dynamically expanded
  - This is similar to a stream but adds a specific type of object

```
int main() {

  // Read all input words into a vector.

  string word;
  vector<string> words;
  while (cin >> word) {
    words.push_back(word);
  }

...
```

Sorting a container of objects

- The vector template is one of many *container* templates
  - The various containers share common code
  - The containers use the same method names when possible
- The `empty` method checks for an empty container.
- The `sort` function sorts a subsequence of a container
  - The subsequence is specified with two *iterators*
  - Iterators are fundamental to the C++ standard library

```
...

// Check for no words at all.

if (words.empty()) {
  cout << "No words were entered."
       << endl;

  // Status code is 1.

  return 1;
}

// Sort the set of words.

sort(words.begin(), words.end());

...
```

Even or Odd

- The modulo (`%`) operator computes the remainder after division
  - It allows distinguishing an even integer from an odd integer
- The division operator (`/`) for integers truncates any remainder
- The `auto` keyword specifies that the type of a variable is to be inferred from the type of the initial value
  - Type inference is a new feature of C++11
  - It avoids the need for dealing with elaborate type definitions

```
...

if ((words.size() % 2) == 0) {

  // Even number of words

  auto middle = words.size() / 2;
  cout << words.at(middle - 1) << " "
       << words.at(middle) << endl;
} else {

  // Odd number of words

  auto middle = (words.size() - 1) / 2;
  cout << words.at(middle) << endl;
}

// Status code is 0.

return 0;

...
```

Indexing

- One can use either the `at` method or `[ ]` notation to obtain one object in a container
  - The `at` method is safe
- This is similar to strings
  - In fact, a string may be regarded as being a vector of bytes
  - However, a string cannot be dynamically expanded

```
...

if ((words.size() % 2) == 0) {

  // Even number of words

  auto middle = words.size() / 2;
  cout << words.at(middle - 1) << " "
       << words.at(middle) << endl;
} else {

  // Odd number of words

  auto middle = (words.size() - 1) / 2;
  cout << words.at(middle) << endl;
}

// Status code is 0.

return 0;

...
```

Median of Numbers

Specification

- Read numbers from the standard input
- Compute the median.
  - No numbers: Undefined
  - Odd number of numbers: Exactly one median number
  - Even number of numbers: Average of two median numbers
- If no numbers are entered, then print “No numbers were entered.” Status code is 1.
- Otherwise, print the median with three-place accuracy. Status code is 0.

Reading numbers

- Similar to reading words but now reading numbers
- If a non-number is encountered in the input stream, then the input stream state is changed to “failed”
  - Additional input is not possible
  - Testing such a stream in a conditional returns `false`

```
int main() {

  // Read all input numbers
  // into a vector.

  double number;
  vector<double> numbers;
  while (cin >> number) {
    numbers.push_back(number);
  }
```

- A `double` should almost always be used instead of `float`
  - More accurate than `float`
  - Faster than `float`
  - Uses more memory

Sorting a container of numbers

- Check for empty list as in the median word program
- Sorting is also done the same way
- The same code is used for sorting both strings and numbers
  - The only difference is the comparison function
  - One can specify a different comparison function for sorting

```
...

if (numbers.empty()) {
  cout << "No numbers were entered."
       << endl;

  // Status code is 1.

  return 1;
}

// Sort the set of numbers.

sort(numbers.begin(), numbers.end());

...
```

Compute median

- Once again use type inference
- This time use the fact that integer division truncates the remainder
- The ? and : form a *ternary* operator (i.e., three arguments)

```
...

auto middle = numbers.size() / 2;
double median =
  ((numbers.size() % 2) == 0)?
  0.5 * (numbers.at(middle - 1) +
         numbers.at(middle)) :
  numbers.at(middle);

...
```

`?:` is the same as `if/else` except that the second and third arguments are expressions, not statements

Unsigned integers

- The size of a container is actually an unsigned int
- Unsigned types are a bad idea
  - Java does not support them
  - Ironically, it was Stroustrup who recommended this
- Comparing unsigned ints with signed ints will produce unexpected results as in the code on the right.

```
unsigned int x = 0;
if (x < -1) {
  cout << "0 < -1" << endl;
}
```

- This code will compile but if you use `–Wall` it will warn you about comparing a unsigned int with a signed int.
- It is better to use unsigned ints only when necessary

Setting numeric precision for output

- The `precision` method of `ostream` sets the number of places that are printed
- It also returns the old value
- One should always save the original value and restore it after printing.
  - Other code uses the standard output

```
// Change the precision and
// print the median.

const auto originalPrecision =
  cout.precision(3);
cout << "Median is "
     << median << endl;
cout.precision(originalPrecision);
```

- One can also use a stream manipulator as in the textbook
  - This is mainly useful for setting the precision to several different values while printing

Stacker

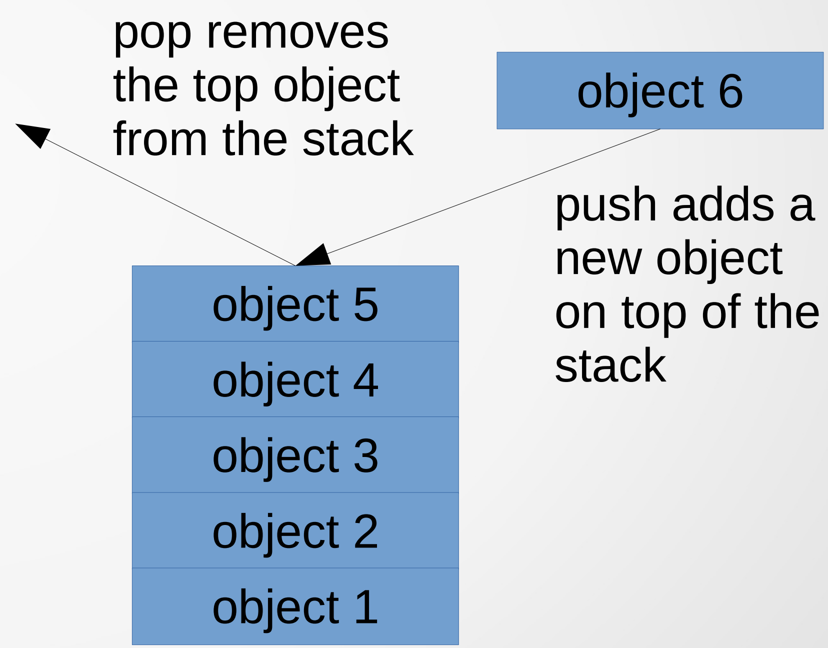
Specification

- Reads words from the standard input
  - Eliminate repeats.
- Print in reverse order
- Print on one line:
  - The total number of words
  - The average size of a word

The stack template

- This program requires a new library: the stack library
- A stack is a collection of objects stored in last-in-first-out (LIFO) order
  - Easily implemented using the vector template
  - The stack template has a different interface than the vector template
  - This is an example of the Adapter design pattern

```
#include <iostream>
#include <stack>
using namespace std;
```



Pushing on the stack

- Like vector, stack is a template
  - The type of the objects that are allowed on the stack must be specified
- The logical or ensures that the call to `top` is valid
- The short-circuit property of logical or is essential to prevent an exception being thrown
- The `push` method dynamically expands the stack

```
int main() {

  // Push the words on a stack of words.

  string word;
  stack<string> words;
  while (cin >> word) {

    // If no words are on the stack
    // or the new word is different,
    // then push the new word on the
    // stack.

    if (words.empty() ||
        words.top() != word) {
      words.push(word);
    }
  }
...
```

Popping from the stack

- The object on the top of the stack is removed using the `pop` method
- The loop invariant is:

  `totalLength` is the sum of the lengths of all words that have been popped from the stack

- When the loop terminates `words.empty()` must be true because there is no break or return in the loop.

```
...
double totalLength = 0.0;
const auto wordCount = words.size();

while (!words.empty()) {

  const string poppedWord(words.top());
  words.pop();
  totalLength += poppedWord.length();

  cout << word << endl;
}
...
```

- Therefore, when the loop terminates, `totalLength` is the sum of the lengths of all the words that were originally on the stack

Termination

- An important property of a function is that it *terminates* (i.e., eventually finishes).
- Alan Turing showed that proving termination is *undecidable*.
- It is impossible to have a function that will determine whether a general function will terminate.
  - Can one program a function `halts` which takes a function as its input such that for any function `f`, `halts(f)` is `true` if `f` eventually finishes (for any input to `f`) and is `false` if `f` does not finish for some input to `f`?
- This is known as the *halting problem*.
- The proof is simple: Apply halts to itself!
  - What made the proof difficult is that there was no formal notion of a program at the time.
  - Turing had to invent Computer Science to solve this problem!

Proving Termination

- Although one cannot automate the proof of termination, it is usually easy to prove it.
- For this loop, the size of the stack decreases by 1 for each iteration, because the `pop` method removes one object from the stack and no object is pushed on the stack.
- Therefore, the stack will eventually become empty.

```
while (!words.empty()) {

  const string poppedWord(words.top());
  words.pop();
  totalLength += poppedWord.length();

  cout << word << endl;
}
```

- In practice, one proves termination by finding something that decreases for each iteration.
- You will not have to give formal proofs of termination in this course, but you should make sure that your functions terminate.

Conversions and Division by Zero

- The `to_string` function constructs a string showing the value of a variable
- For example, if `wordCount` is `5`, then `to_string(5)` is the string `"5"`.
- Notice that the program never checks whether `wordCount` is `0`. Is this a problem?

```
const string conclusion
  to_string(wordCount) + " " +
  to_string(totalLength / wordCount);

cout << conclusion << endl;
```

- Not in this case because `totalLength` is `double` not integer.
  - The division by `wordCount` is the floating-point division.
  - `wordCount` is converted to `double` before dividing.
  - Floating-point division by `0` produces a special double value called “Not a Number” or “`nan`”.

Requirements

Use Cases

- Use cases are a powerful and popular technique for specifying the functional requirements of a system to be developed
- The next team project deliverable consists of the functional requirements expressed as use cases
- A *scenario* is a narrative or story of an interaction that a system has with its environment
- A *use case* is a collection of scenarios
  - Abstraction similar to that of a class, which is a collection of objects

Use Case Specification

- The core of the specification of a use case is a list of actions or event steps that define one kind of interaction between an actor and a system
- An *actor* is a role played by a person or another system
- A use case does **not** describe:
  - how the interactions are implemented by the system
  - how actors interact with each other
  - the nonfunctional requirements of the system

Example Use Case

Use Case: Update Existing Document

Exposition: An editor updates an existing document submits the modified document

Step-by-step Description:

1. Include: Query Document
2. [Editor] The editor requests that the document be checked out.
3. [System] The system provides a template for submitting the modified document.
4. [Editor] Exception: Cancel Document Update
5. [Editor] The editor fills in the template and submits it to the system.
6. [System] The system sends a confirmation to the editor.

Discussion of the example use case

- The use case *name* is used for identifying the use case
- The *exposition* documents the purpose of the use case
- The step-by-step description defines the interaction of the use case
  - This is the *flow* of the use case
- A use case can *include* another use case
- A use case can invoke an *exception* use case when necessary
  - The UML link for an exception is «extends»
- A use case can have multiple *alternative* flows (not shown in the example)

The size of a use case

- For the team project, the size of a use case is in *points*.
- Count 1 point for each step in the use case.
- Count 1 point for each include of another use case.
- Count 1 point for the use case as a whole.
- The example use case counts 7 points.
- In addition, count 1 point for each actor.
- The total number of points for your team project must be at least 10 times the number of project members.

Use Case Step

- Defines an elementary interaction event
- Initiated by an actor or by the system
- Performs an action which is one of these:
  - An actor performs an action with the system
  - The system performs an action that either modifies the state of the system or affects an actor
- The action *must* be described by an active verb
- An actor never performs an action with another actor
  - A use case only describes interactions between the actors and the system

Alternative flow versus exception

- An alternative flow is not another use case
- An alternative flow does not have a precondition or postcondition
- Upon completion of an alternative flow, the use case postcondition will be true
- An alternative flow does not return to the use case that invoked the alternative flow

- An exception is another use case
- An exception has its own precondition and postcondition
- A use case that invokes an exception may not satisfy the use case postcondition
- An exception does not return to the use case that invoked the exception

Examples of common errors

- Using a passive verb in a step
  - “The update is confirmed”
  - Does not specify which actor confirms the update
- Interactions between actors
  - “The editor asks the administrator for permission to update a document”
  - Does not involve the system, so it is irrelevant
- Implementation details
  - “The user interface subsystem calls the storage subsystem to store the document”
  - Never mention subsystems, modules, classes, etc.
  - Use cases define requirements not design or implementation details

Common Trap

- By far the most common error is to regard use case descriptions as a kind of programming language
- It is tempting for a programmer to use familiar constructs such as constructs similar to if and while statements in a use case description.
- In other words, the use case developer is thinking in terms of implementing the system rather than specifying its requirements.
- Unfortunately, many textbook authors and the Wikipedia article on use cases have fallen into this trap.

Sample Requirements Document

- The Open Ontology Repository
  - If the link does not work, then type in this URL:

    http://www.ccs.neu.edu/home/kenb/ontologies/oor-usecase.xml
- Complete requirements document for the OOR
  - Uses CSS for viewing the requirements

    http://www.ccs.neu.edu/home/kenb/ontologies/auxfiles/styleowl.xsl
- The XML format is available at:

  http://www.ccs.neu.edu/home/kenb/ontologies/oor-usecase.owl

Next Class

- The topic of the next class is classes (sorry about the pun)