Vectors

- Vectors are homogeneous collections with random access
  - Store the same type/class of object, e.g., int, string, ...
  - The 1000th object in a vector can be accessed just as quickly as the 2nd object

- Strings are collections of characters supporting operations:
  - length(), substr(), find(), concatenation with +, creating strings, ...

- Vectors are collections of int, double, Date, ... supporting
  - Creation, addition of elements, accessing elements
  - size(), indexing with [], construction, ...

Strings are collections of characters supporting operations:
- length(), substr(), find(), concatenation with +, creating strings, ...

Vectors are collections of int, double, Date, ... supporting
- Creation, addition of elements, accessing elements
- size(), indexing with [], construction, ...

Vector basics

- We're using the class tvector, need #include "tvector.h"
  - Based on the standard C++ (STL) class vector, but safe
  - Safe means programming errors are caught rather than ignored: sacrifice some speed for correctness
  - In general correct is better than fast, programming plan:
    - Make it run
    - Make it right
    - Make it fast

- Vectors are typed, when defined must specify the type being stored, vectors are indexable, get the 1st, 3rd, or 105th element
  
```cpp
tvector<int> ivals(10);    // store 10 ints
ivals[0] = 3;
tvector<string> svals(20); // store 20 strings
svals[0] = "applesauce";
```

Tracking Dice, see dieroll2.cpp

```cpp
const int DICE_SIDES = 4;
int main()
{  
  int k, sum;
  Dice d(DICE_SIDES);
tvector<int> diceStats(2*DICE_SIDES+1);
  int rollCount = 20000;
  for(k=2; k <= 2*DICE_SIDES; k++)
  {   diceStats[k] = 0;
  }
  for(k=0; k < rollCount; k++)
  {    sum = d.Roll() + d.Roll();
    diceStats[sum]++;
  }
  cout << "Roll 1 and 2" << endl;
  for(k=2; k <= 2*DICE_SIDES; k++)
  {   cout << k << "\t" << diceStats[k] << endl;
  }
  cout << "Roll 2 and 3" << endl;
  for(k=2; k <= 2*DICE_SIDES; k++)
  {   cout << k << "\t" << diceStats[k] << endl;
  }
  return 0;
}
```

Defining tvector objects

- Can specify # elements in a vector, optionally an initial value
  
```cpp
tvector<int> values(300);   // 300 ints, values ?
tvector<int> nums(200,0);   // 200 ints, all zero
tvector<double> d(10,3.14); // 10 doubles, all pi
tvector<string> w(10,"foo"); // 10 strings, "foo"
tvector<string> words(10);  // 10 words, all ""
```

- The class tvector stores objects with a default constructor
  - Cannot define tvector<Dice> cubes(10); since Dice doesn't have default constructor
  - Standard class vector relaxes this requirement if vector uses push_back, tvector requires default constructor
**Vectors as lists**

- The “vector as counters” example constructs and initializes a vector with a specific number of elements.
- Other uses of vector require the vector to “grow” to accommodate new elements:
  - Consider reading words from *Hamlet*, storing them in a vector.
  - How big should we define vector to be initially? What are potential problems?
  - Analogy of shopping list on the refrigerator, what happens when we run out of room on the list?
- When a vector is used as a list we’ll use a different method for adding elements to the vector so that the vector can “grow”:
  - The vector grows itself, we (as client programmers) don’t.

**Reading words into a vector**

tvector<string> words;
string w;
ifstream input("c:/data/hamlet.txt");
while (input >> w){
  words.push_back(w);
}
cout << "read " << words.size() << " words" << endl;
cout << "last word read is "
  << words[words.size() - 1] << endl;

- What header files are needed? What happens with *Hamlet*?
- Where does `push_back()` put a string?

**Using `tvector::push_back`**

- The method `push_back` adds new objects to the “end” of a vector, creating new space when needed:
  - The vector must be defined initially without specifying a size.
  - Internally, the vector keeps track of its capacity, and when capacity is reached, the vector “grows”.
  - A vector grows by copying old list into a new list twice as big, then throwing out the old list.
- The capacity of a vector doubles when it’s reached: 0, 2, 4, 8, 16, 32, …
  - How much storage used/wasted when capacity is 1024?
  - Is this a problem?

**David Gries**

- Advocates formal methods as integral part of program development:
  - Formal means well-founded mathematically.
  - Loop invariants are an example of formalisms that help program development.
  - A programmer needs a bag of tricks, a collection of methods for attacking a problem. … One technique will never suffice.
- Has developed a CD-based book for learning Java with his son (uses singing!)