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**

- **We've used files to store text and StringSets to store sets of strings; vectors are more general and more versatile, but are simply another way to store objects**
  - **We can use vectors to count how many times each letter of the alphabet occurs in *Hamlet* or any text file**
  - **We can use vectors to store CD tracks, strings, or any type**
- **Vectors are a class-based version of *arrays*, which in C++ are more low-level and more prone to error than are Vectors**

Vector basics

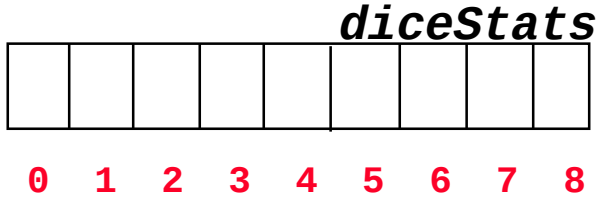
- We're using the class `tvector`, need `#include"tvector.h"`
  - Based on the standard C++ (STL) class vector, but safe
  - Safe means programming errors are caught rather than ignored: sacrifice some speed for correctness
  - In general correct is better than fast, programming plan:
    - Make it run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

```
tvector<int> ivals(10);     // store 10 ints
vals[0] = 3;
tvector<string> svals(20); // store 20 strings
svals[0] = "applesauce";
```

Tracking Dice, see *dieroll2.cpp*

```
const int DICE_SIDES = 4;
int main()
{
    int k, sum;
    Dice d(DICE_SIDES);
    tvector<int> diceStats(2*DICE_SIDES+1);
    int rollCount = PromptRange("how many rolls",1,20000);

    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\t\t# of occurrences" << endl;
    for(k=2; k <= 2*DICE_SIDES; k++)
    {   cout << k << "\t\t" << diceStats[k] << endl;
    }
    return 0;
}
```



Defining `tvector` objects

- Can specify # elements in a vector, optionally an initial value

```
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;
string filename = PromptString("enter file name: ");
ifstream input(filename.c_str());

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?

Comparing `size()` and `capacity()`

- When a vector is defined with no initial capacity, and `push_back` is used to add elements, `size()` returns the number of elements actually in the vector
  - This is the number of calls of `push_back()` if no elements are deleted
  - If elements deleted using `pop_back()`, size updated too

- The capacity of vector is accessible using `tvector::capacity()`, clients don't often need this value
  - An initial capacity can be specified using `reserve()` if client programs know the vector will resize itself often
  - The function `resize()` grows a vector, but not used in conjunction with `size()` – clients must track # objects in vector separately rather than vector tracking itself

Passing vectors as parameters

- **Vectors can be passed as parameters to functions**
  - **Pass by reference or const reference (if no changes made)**
  - **Passing by value makes a copy, requires time and space**

```
void ReadWords(istream& input, tvector<string>& v);
// post: v contains all strings in input,
//       v.size() == # of strings read and stored

void Print(const tvector<string>& v)
// pre: v.size() == # elements in v
// post: elements of v printed to cout, one per line
```

- **If `tvector::size()` is *not* used, functions often require an int parameter indicating # elements in vector**

Vectors as data members

- A tvector can be a (private) instance variable in a class
  - Constructed/initialized in class constructor
  - If size given, must be specified in initializer list

```
class WordStore
{
   public:
     WordStore();
    private:
     tvector<string> myWords;
};
WordStore::WordStore()
  : myWords(20)
{
}
```

  - What if `push_back()` used? What if `reserve()` used?

Vectors as data members (continued)

- It's not possible to specify a size in the class declaration
  - Declaration is what an object looks like, no code involved
  - Size specified in constructor, implementation .cpp file

```
class WordStore
{
   private:
     tvector<string> myWords(20);  // NOT LEGAL SYNTAX!
};
```

- If push_back is used, explicit construction not required, but ok

```
WordStore::WordStore()
  : myWords()        // default, zero-element constructor
{ }
```

  - No ( )'s for local variable: `tvector<string> words;`

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 ticks, a collection of methods for attacking a problem. … One technique will never suffice

- **In 1999 is developing a CD-based book for learning to program with Java**

Picking a word at random

- **Suppose you want to choose one of several words at random, e.g., for playing a game like Hangman**
  - **Read words into a vector, pick a random string from the vector by using a `RandGen` or `Dice` object. Drawbacks?**
  - **Read words, shuffle the words in the vector, return starting from front. Drawbacks?**
- **Steps: read words into vector, shuffle, return one-at-a-time**
  - **Alternatives: use a class, read is one method, pick at random is another method**
  - **Don't use a class, test program with all code in main, for example**

First approach, pick a word at random

```
tvector<string> words;
string w, filename = “words.txt”;
RandGen gen;
ifstream input(filename.c_str());

while (input >> w)
{   words.push_back(w);
}

for(k=0; k < words.size(); k++)
{   int index = gen.RandInt(0,words.size()-1);
    cout << words[index] << endl;
}
```

- **What could happen in the for-loop? Is this desired behavior?**

Shuffling the words (shuffle.cpp)

```
tvector<string> words;
string w, filename = "words.txt";
RandGen gen;
ifstream input(filename.c_str());
while (input >> w)
{   words.push_back(w);
}
// note: loop goes to one less than vector size
for(k=0; k < words.size()-1; k++)
{   int index = gen.RandInt(k,words.size()-1);
    string temp = words[k];
    words[k] = words[index];
    words[index] = temp;
}
// Print all elements of vector here
```

- Key ideas: swapping elements, choosing element "at random"
  - All arrangements/permuations equally likely

Why this is a good shuffling technique

- **Suppose you have a CD with 5 tracks, or a vector of 5 words**
  - **The first track stays where it is one-fifth of the time, that's good, since 1/5 of all permutations have track one first**
  - **If the first track is swapped out (4/5 of the time) it will then end up in the second position with probability 1/4, that's 4/5 x 1/4 = 1/5 of the time, which is what we want**
  - **Also note five choices for first entry, # arrangements is `5x4x3x2x1 = 5!` Which is what we want.**
- **One alternative, make 5 passes, with each pass choose any of the five tracks/words for each position**
  - **Number of arrangements is `5x5x5x5x5 > 5!`, not desired, there must be some "repeat" arrangements**

`tvector` details, optimizations

- Space/storage is "wasted" if we use push_back and a vector resizes itself many times
  - Don't need to worry about this in most cases, storage lost is in many ways minimal (not too bad in any case)
- We can reserve storage so that vector doesn't waste space, fills reserved spaces allocated

```
tvector<int> iv;
iv.reserve(1024);  // room to grow up to 1,024
int x = iv.size(); // stores zero in x
```

- Can also use `resize()`, this "grows" vector and changes size

```
iv.resize(512);    //size changes, maybe capacity
```

Vector idioms: insertion and deletion

- It's easy to insert at the end of a vector, use `push_back()`
  - We may want to keep the vector sorted, then we can't just add to the end
  - Why might we keep a vector sorted?

- If we need to delete an element from a vector, how can we "close-up" the hole created by the deletion?
  - Store the last element in the deleted spot, decrease size
  - Shift all elements left by one index, decrease size

- In both cases we decrease size, this is done using `pop_back()`
  - Analagous to `push_back()`, changes size, not capacity

Insert into sorted vector

```
void insert(tvector<string>& a, const string& s)
// pre: a[0] <= … <= a[a.size()-1], a is sorted
// post: s inserted into a, a still sorted
{
   int count = a.size(); // size before insertion
   a.push_back(s);       // increase size
   int loc = count;      // insert here?

   // invariant: for k in [loc+1..count], s < a[k]

   while (0 <= loc && s < a[loc-1])
   {  a[loc] = a[loc-1];
      loc--;
   }
   a[loc] = s;
}
```

- What if s belongs last? Or first? Or in the middle?

What about deletion?

```
void remove(tvector<string>& a, int pos)
// post: original a[pos] removed, size decreased
{
  int lastIndex = a.size()-1;
  a[pos] = a[lastIndex];
  a.pop_back();
}
```

- How do we find index of item to be deleted?
- What about if vector is sorted, what changes?
- What's the purpose of the pop_back( ) call?

Deletion from sorted vector

Deletion from sorted vector

```
void remove(tvector<string>& a, int pos)
// pre: a is sorted
// post: original a[pos] removed, a sorted
{
  int lastIndex = a.size()-1;
  int k;
  for(k=pos; k < lastIndex; k++)
  {
    a[k] = a[k+1];
  }
  a.pop_back();
}
```

- What happens if we start at lastIndex and shift right?
- Does `pop_back()` remove an element?

Searching a vector

- We can search for one occurrence, return true/false or index
  - Sequential search, every element examined
  - Are there alternatives? Are there reasons to explore these?

- We can search for number of occurrences, count “the” in a vector of words, count jazz CDs in a CD collection
  - Search entire vector, increment a counter
  - Similar to one occurrence search, differences?

- We can search for many occurrences, but return occurrences rather than count
  - Find jazz CDs, return a vector of CDs

Counting search

```
void count(tvector<string>& a, const string& s)
// pre: number of elements in a is a.size()
// post: returns # occurrences of s in a
{
  int count = 0;
  int k;
  for(k=0; k < a.size(); k++)
  {   if (a[k] == s)
      {
         count++;
      }
  }
  return count;
}
```

- How does this change for true/false single occurrence search?

Collecting search

```
void collect(tvector<string>& a, const string& s,
             tvector<string>& matches)
// pre: number of elements in a is a.size()
// post: matches contains all elements of a with
//       same first letter as s
{
  int k;
  matches.clear();   // size is zero, capacity?
  for(k=0; k < a.size(); k++)
  {  if (a[k].substr(1,0) == s.substr(1,0))
     {  matches.push_back(a[k]);
     }
  }
}
```

- What does `clear()` do, similar to `resize(0)`?

Algorithms for searching

- If we do lots of searching, we can do better than sequential search aka linear search where we look at all vector elements
  - Why might we want to do better?
  - Analogy to "guess a number" between 1 and 100, with response of high, low, or correct

- In guess-a-number, how many guesses needed to guess a number between 1 and 1,000? Why?
  - How do you reason about this?
  - Start from similar, but smaller/simpler example
  - What about looking up word in dictionary, number in phone book given a name?
  - What about looking up name for given number?

Binary search

- If a vector is sorted we can use the sorted property to eliminate half the vector elements with one comparison using <
  - What number do we guess first in 1..100 game?
  - What page do we turn to first in the dictionary?

- Idea of creating program to do binary search
  - Consider range of entries search key could be in, eliminate half the the entries if the middle element isn't the key
  - How do we know when we're done?
  - Is this harder to get right than sequential search?

Binary search code, is it correct?

```
int bsearch(const tvector<string>& list, const string& key)
// pre: list.size() == # elements in list, list is sorted
// post: returns index of key in list, -1 if key not found
{
    int low = 0;                        // leftmost possible entry
    int high = list.size()-1;           // rightmost possible entry
    int mid;                            // middle of current range
    while (low <= high)
    {   mid = (low + high)/2;
        if (list[mid] == key)            // found key, exit search
        {   return mid;
        }
        else if (list[mid] < key)        // key in upper half
        {   low = mid + 1;
        }
        else                             // key in lower half
        {   high = mid - 1;
        }
    }
    return -1;                           // not in list
}
```