# CS3520 Programming in C++
# Maps and Sets

Ken Baclawski
Fall 2016

# Outline

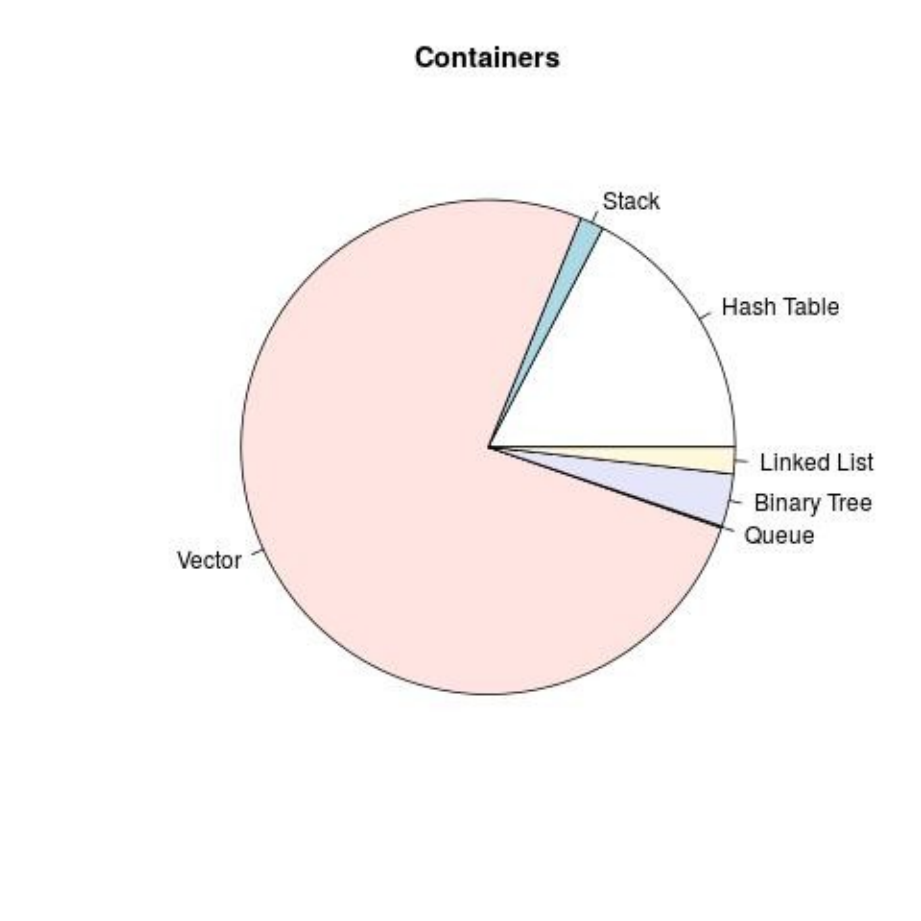
- Overview of Containers
- Associative Structures
  - Maps
  - Sets
- Random Number Generation

# Overview of Containers

# Classification of Containers

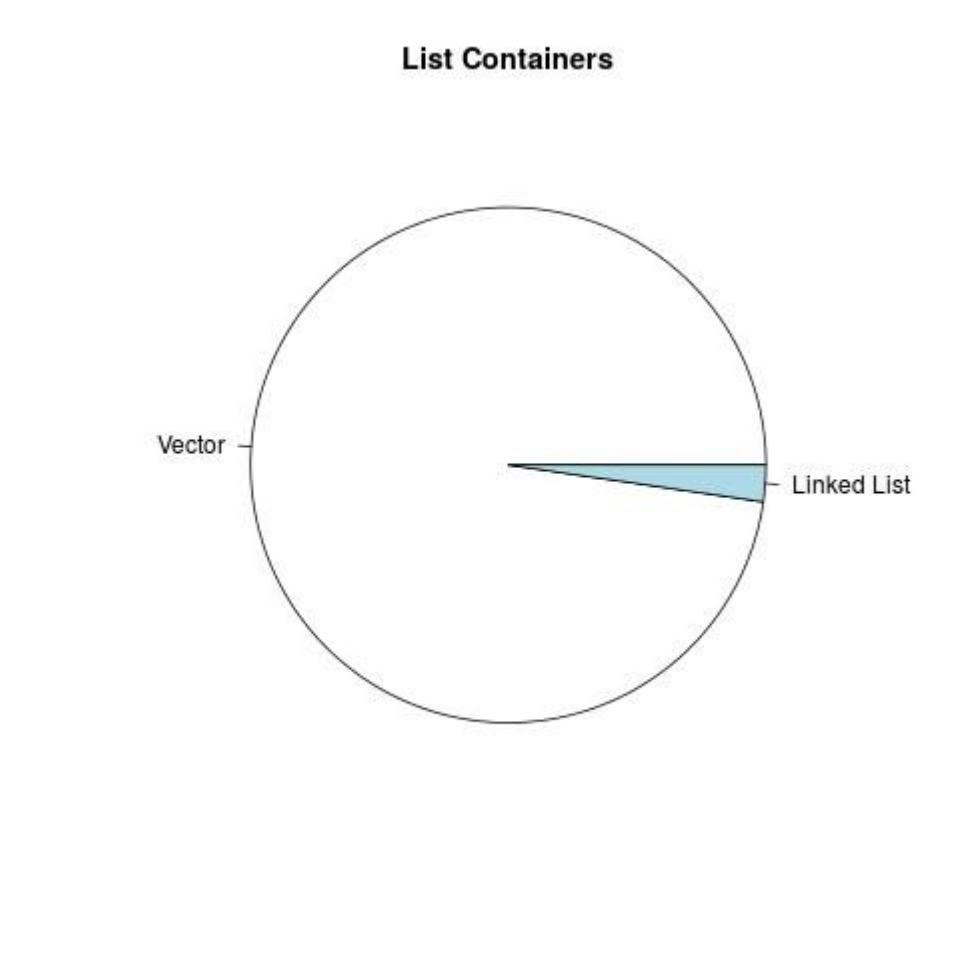
- List Containers
  - Array list
  - Linked list
    - Singly linked
    - Doubly linked
  - Queue
    - FIFO queue
    - LIFO queue (stack)
    - Double ended queue (deque)
    - Priority queue (heap)
- Associative Containers
  - Hash or Tree
  - Map or Set
  - Single or Multiple
- There are 8 cases in all

# Statistics for Containers




- This pie chart was based on some open-source code from a variety of projects
  - Around 1M lines of code
- The vector container dominates, followed by hash tables and binary trees.
- Linked lists and stacks are seldom used
- Queues are rarely used

# Statistics for List Containers

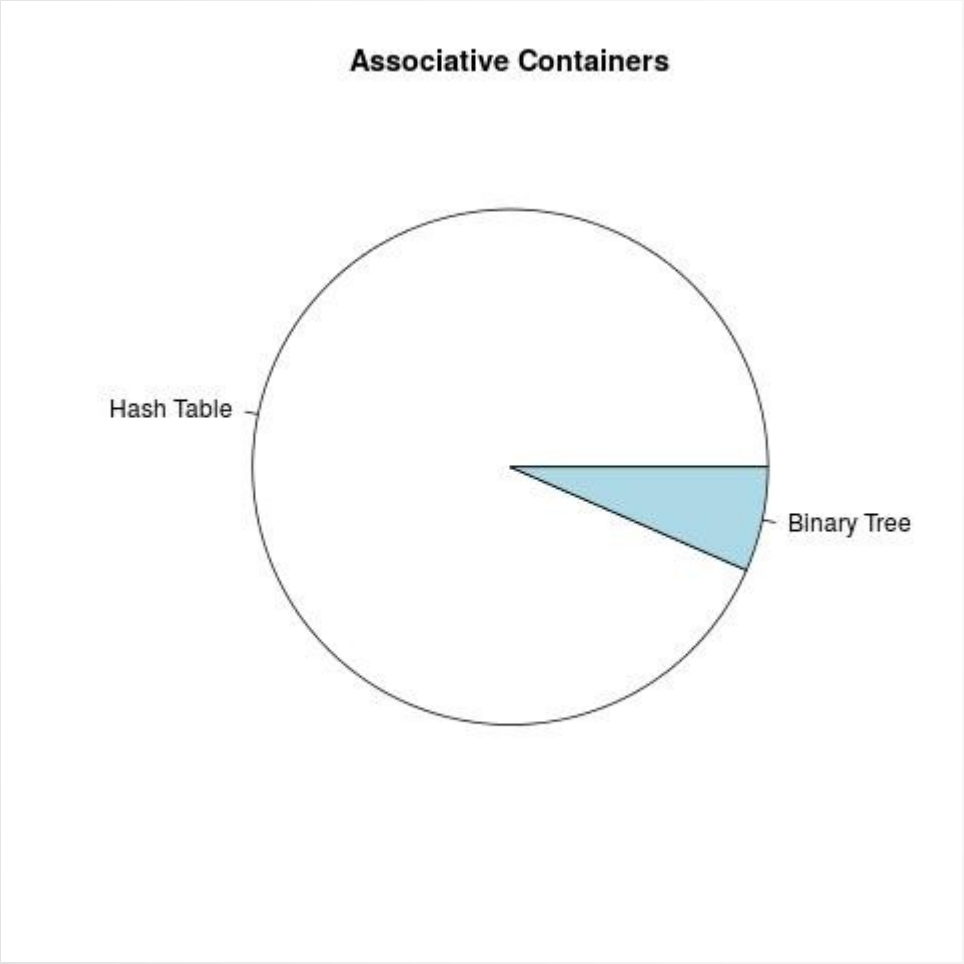



- Considering only the list containers
  - Array lists (vectors) dominate
  - Linked lists are seldom used
- Linked lists represent less than 9% of all sequential list containers in this study

# Why are linked lists unpopular?

- Substantial storage overhead compared with a vector
- Operations, such as retrieving one element and sorting, are much slower compared with a vector
- Operations, such as removing the first element, are seldom needed
  - Even when they are needed, they are seldom done in isolation
  - Removing a whole set of elements using the `remove` algorithm is faster for a vector than for a linked list
- Vectors are almost always the best choice for a list
  - However, be careful to remove elements in large batches rather than individually

# Statistics for Associative Containers




- Considering only associative containers
  - Hash tables dominate
  - Binary trees are seldom used
- Binary trees represent less than 17% of all associative containers in this study

# Hash tables vs binary trees

- Requires a hash function
- Sensitive to the details of the hash function
- Keys are in random order
- Retrieval in constant time
- Lower storage overhead
- Much more commonly used

- Requires a comparison function
- Not sensitive to the details of the comparison
- Keys are in sorted order
- Retrieval in logarithmic time
- Greater storage overhead
- Less frequently used

# Why are hash tables so popular?

- Defining a good hash function is no more difficult than defining a comparison function
  - For complex structures, hash functions are somewhat easier to define than a comparison function
- Keeping the keys in sorted order is not often needed
- Even when sorting is needed, one will usually need several sort orders, while binary trees only support one sort order
- The improvement in time and space is substantial, even for relatively small collections
- Hash tables are almost always the best choice for an associative container

# Associative Containers

# Types of Associative Container

- There are 8 main types of associative container
  - Hash or Tree
  - Map or Set
  - Single or Multiple

# Hash Table

- An array list such that each element is placed in the position given by a *hash function*.
- If there is a *collision* (i.e., two values have the same hash function), then the element is placed in another position.
  - There are several algorithms for dealing with collisions

| | |
|---|---|
| 0 | jumped |
| 1 | The |
| 2 | |
| 3 | |
| 4 | over |
| 5 | quick |
| 6 | brown |
| 7 | fox |

```
hash("The") == 1
hash("quick") == 5
hash("brown") == 6
hash("fox") == 7
hash("jumped") == 5
hash("over") == 4
```

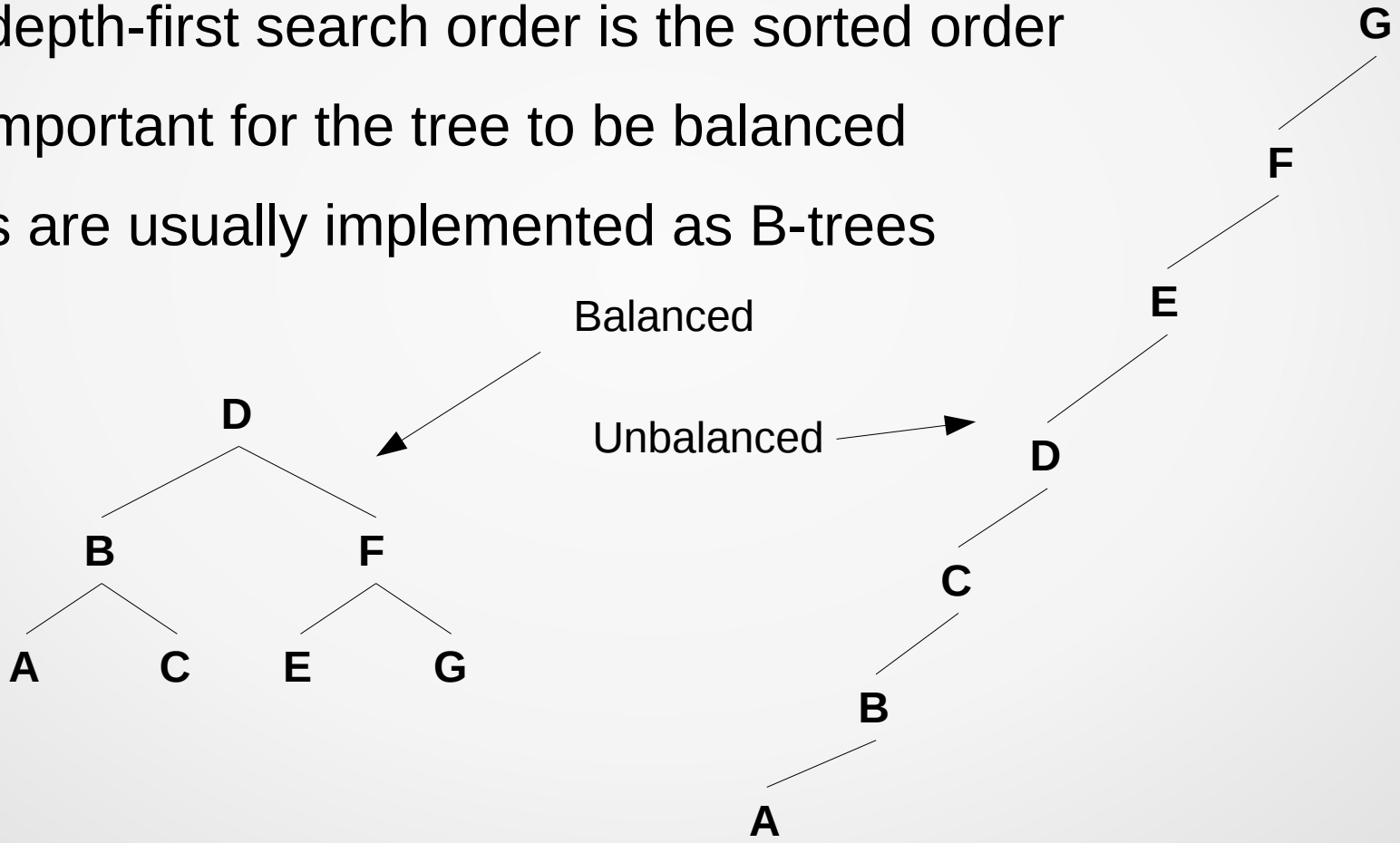
# Hash Table

- A hash table requires
  - A hash function
  - An equality operator
  - The two must be compatible:

    if a == b then hash(a) == hash(b)
- Standard objects already have these functions
  - The most popular key type is string
- The hash function for a class is defined by combining the hash functions of its data members

# Binary Tree

- The elements are the nodes of a binary tree
- The depth-first search order is the sorted order
- It is important for the tree to be balanced
- Trees are usually implemented as B-trees



# Binary Tree

- Requires a comparison function
  - Not all classes have a reasonable comparison function
  - An equality operator is not required
- Standard classes have comparison operators
  - The most popular key type is string
- Comparison operators are usually lexicographic

# Constructing Maps

- Each key maps to a value
- For example map<string, string> maps a string to a string
  - This could be used to map email addresses to names

    “k.baclawski@neu.edu” → “Ken Baclawski”

    “adaimi.r@husky.neu.edu” → “Rene Adaimi”
- Here is the code in C++:

```
unordered_map<string, string> email2name({
  { "k.baclawski@neu.edu", "Ken Baclawski" },
  { "adaimi.r@husky.neu.edu", "Rene Adaimi" }});
```

# Using Maps

- Accessing an element using a key
  - Using brackets (index notation) inserts a new mapping if the key was not mapped
  - Using the at method throws an exception if the key was not mapped
- Inserting a new mapping
  - The `insert` and `emplace` methods only insert if there is no existing mapping
  - Using index notation overrides any existing mapping
  - The `at` method throws an exception if the key was not already mapped
- Delete a mapping with the `erase` method.

```
cout << email2name["k.baclawski@neu.edu"] << endl;
cout << email2name.at("adaimi.r@husky.neu.edu") << endl;
email2name.insert({"kenb@ccs.neu.edu", "Kenneth Baclawski"});
email2name.emplace("kenb@ccs.neu.edu", "Kenneth Baclawski");
email2name["kenb@ccs.neu.edu"] = "Ken Baclawski";
email2name.at("kenb@ccs.neu.edu") = "Kenneth Baclawski";
email2name.erase("kenb@ccs.neu.edu");
```

# Multimaps

- Same as maps but a key can map to several values (which can be the same value)
- The index operator (brackets) and `at` method are not defined
- The `equal_range` method gets the values for a key

# Sets and Multisets

- Essentially the same as a map or multimap with each key equal to its value
- For a set one can insert, remove and test whether a key is in the set using `insert`, `erase` and `count`.
- The same methods are used for multisets.

# Iteration

- One can iterate over an associative container
- The objects in a map or multimap container are pairs and the objects in a set or multiset container are elements

```
for (auto p : email2name) {
  cout << p.first << " -> " << p.second << endl;
}
```

```
for (auto email : emails) {
  cout << email << endl;
}
```

# Asymptotic Complexity

**Container Asymptotic Complexity**

| Structure | Access End | Access Any | Insert/Delete End | Insert/Delete Any | Space Overhead |
|---|---|---|---|---|---|
| **Stack** | O(1) | O(n) | O(1) | O(n) | 50% |
| **Linked List** | O(1) | O(n) | O(1) | O(n) | 300% |
| **Array List** | O(1) | O(1) | O(1) | O(n) | 50% |
| **Priority Queue** | O(log n) | O(n log n) | O(log n) | O(n log n) | 50% |
| **Binary Tree** | O(log n) | O(log n) | O(log n) | O(log n) | 300% |
| **Hash Table** | O(1) | O(1) | O(1) | O(1) | 50% |

# Asymptotic Complexity

- To compare sequential structures with associative structures, the index is regarded as a key
  - Some programming languages treat arrays as being a special case of an associative structure
- Insert/Delete time complexity includes the time to access an element in order to perform the operation
  - Some authors omit the access time
- Some structures have a special position
  - The top of a stack
  - The highest priority in a priority queue
- Space overhead is the additional space required beyond the space required to store the addresses of the objects
- Binary trees are assumed to be B-trees of order 4
  - Also called “red-black trees”

# Asymptotic Ratios

| Operation | Complexity | Comparison |
|---|---|---|
| **Access End** | O(1) | Stack = Array List < Hash Table < Linked List |
| | O(log n) | Priority Queue ≈ 1.5 Binary Tree |
| **Access Any** | O(1) | Array List < Hash Table |
| | O(n) | Stack < Linked List |
| **Insert/Delete End** | O(1) | Stack = Array List < Hash Table < Linked List |
| | O(log n) | Priority Queue ≈ 1.5 Binary Tree |
| **Insert/Delete End** | O(n) | Stack = Array List < Linked List |

# Asymptotic Ratios

- When the asymptotic complexities are the same, compare by estimating the asymptotic ratio
  - Usually qualitative comparison
- Stack is Array List with fewer operations
- Hash Table is same as Array List except for hashing the key
- Linked List operations require pointer manipulations and a memory allocation which usually take longer than obtaining the hash value of a key
- A heap (Priority Queue) requires 50% more comparisons than a Binary Tree on average
  - Experimental tests verify this to be accurate

# Conclusions

- Among sequential structures (integer key), Array List is always the best
- For a priority queue, a Binary Tree is better than a heap unless the number of elements is very large
- Among associative structures, Hash Table is better than Binary Tree with some exceptions
  - There is a unique sort order that is frequently needed
  - The Binary Tree is being used for a priority queue

# The rocket2 package

# Requirements

- A simplified form of the rocket package
  - The Rocket class only has getters
  - There is no enumeration of categories
  - No equality operators or methods
- Uses multimap rather than partition
- The HTML table eliminates adjacent duplicates

# Making an HTML table

```
string Rockets::getHtmlTable() const {

  string table = "<table border='1'>\n"
    "<tr><th>Name</th><th>Payload</th><th>Manufacturer</th></tr>";

  vector<vector<string>> matrix;
  for (const Rocket& rocket : rockets_) {
    matrix.push_back({rocket.getName(),
                      to_string(rocket.getPayload()),
                      rocket.getManufacturer()});
  }
...
```

One can construct an object with braces. In this case, the object is a vector with 3 elements. C++ uses type inference to determine that a vector is to be constructed.

# Making an HTML table

```
  for (unsigned int r = 0; r < matrix.size(); ++r) {
    table += "<tr>";
    for (unsigned int f = 0; f < matrix.at(r).size(); ++f) {
      if (r == 0 || matrix.at(r).at(f) != matrix.at(r-1).at(f)) {
        int n = 1;
        for (unsigned int k = r+1; k < matrix.size(); ++k) {
          if (matrix.at(k).at(f) == matrix.at(r).at(f)) {
            ++n;
          } else {
            break;
          }
        }
        table += "<td rowspan='" + to_string(n) + "'>" +
                 matrix.at(r).at(f) + "</td>";
      }
    }
    table += "</tr>\n";
  }
  table += "</table>\n";
  return table;
}
```

Loop over rows and then over the columns. An entry is added to the string only if it is the top one with that value in the column.

# The Category HTML Table

```
string Rockets::getCategorizedHtmlTable() {

  unordered_multimap<string, Rocket> category2rocket;
  for (Rocket rocket : rockets_) {
    category2rocket.insert({rocket.getCategory(), rocket});
  }

  string table = "<table border='1'>\n"
    "<tr><th>Category</th><th>Name</th><th>Payload</th>"
    "<th>Manufacturer</th></tr>\n";
...
```

The multimap is constructed and then the rockets are inserted by category. This loop accomplishes the same functionality as the partition function in the rocket package.

# The Category HTML Table

```
  // The outer loop is by category
  for (const string& category :
       { "Small", "Medium", "Heavy", "SuperHeavy" }) {
    // The inner loop is over rockets in one category
    auto pair = category2rocket.equal_range(category);
    for (auto iter = pair.first; iter != pair.second; ++iter) {
      // Only the first row shows the category name
      if (iter == pair.first) {
        table += "<tr><td rowspan='" +
          to_string(category2rocket.count(category)) +
          "'>" + category + "</td>";
      }
      // Show the rest of the fields in every row
      table += "<td>" + iter->second.getName() + "</td><td>" +
               to_string(iter->second.getPayload()) +
               "</td><td>" + iter->second.getManufacturer() +
               "</td></tr>\n";
    }
  }
  // The table end tag
  table += "</table>\n";
  return table;
}
```

Loop over vector defined by a list of strings in braces

Get all of the rockets in a category and loop over them

# Random Number Generation

# The random library

- The source of randomness is a sequence of bits:

  100000110100110...
- Two choices for the source:
  - true randomness
  - pseudo randomness
- True randomness
  - Not always available
  - Limited number of bits when available
  - Not reproducible (of course)

# Pseudo randomness

- Sequence of random bits starting from a seed
  - Very efficient
  - Unlimited number of bits
  - Reproducible given the seed
  - Useful for testing purposes
- While not truly random, it is very difficult to distinguish from true randomness in practice
  - Be sure to use one of the more recent random number generators

# Generating random numbers

- Requires two choices
  - The engine that generates the source of randomness
  - The probability distribution
- Engines generate unsigned integers
- A probability distribution object modifies the integers produced by an engine to generate a number with a specified probability distribution

# Seeds

- The `seed_seq` class constructs a seed for an engine
- One can use either a list of integers or a range defined by iterators

```
seed_seq seed1 = { 234, 567 };
seed_seq seed2(s.begin(), s.end());
```

# Engines

- Multiplicative congruential engines
  - Very simple and fast
  - Very little storage is required
  - Not useful for large amounts of randomness
  - This is usually used by default
- Subtract with carry engines
  - Fast
  - Relatively large storage requirements
  - Useful for moderately large amounts of randomness
  - Theoretical properties are unknown
- Mersenne twister engines
  - Slowest
  - Large storage requirements
  - Useful for large amounts of randomness
  - Good theoretical properties
- True random engine

# Probability Distributions

- uniform_int_distribution
- uniform_real_distribution
- bernoulli_distribution
- binomial_distribution
- geometric_distribution
- negative_binomial_distribution
- poisson_distribution
- exponential_distribution
- gamma_distribution
- weibull_distribution
- extreme_value_distribution
- normal_distribution
- lognormal_distribution
- chi_squared_distribution
- cauchy_distribution
- fisher_f_distribution
- student_t_distribution

# Generating random numbers

Suppose we want to generate a normally distributed number with mean 100 and standard deviation 10.

1. Choose a seed: `seed_seq seed = { 234, 567 };`
2. Choose an engine: `mt19937 engine(seed);`
3. Choose a distribution:

```
normal_distribution<double> nd(100.0, 10.0);
```

4. Generate the number: `double x = nd(engine);`

# Next Class

- Threads
- Review for MidTerm Exam