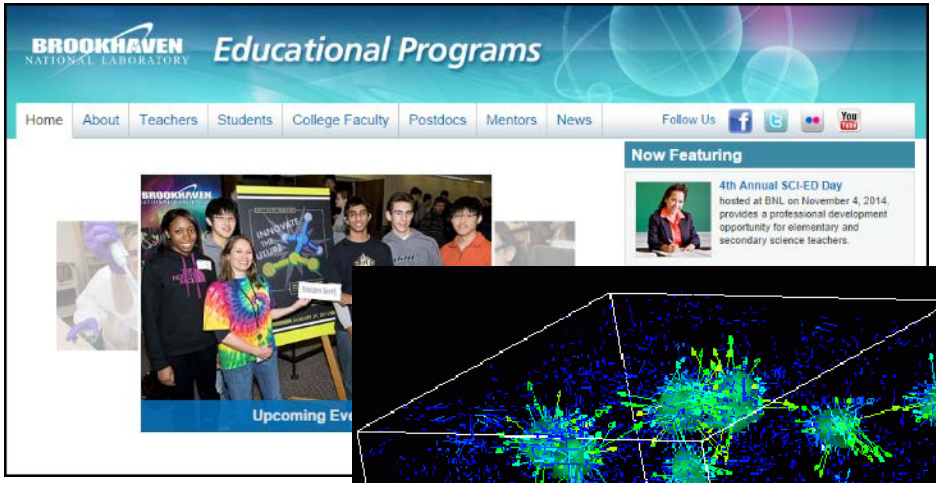
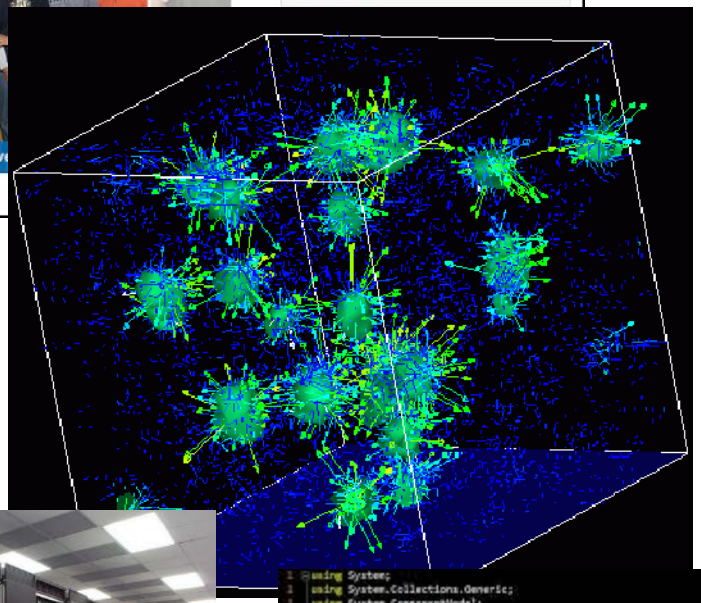
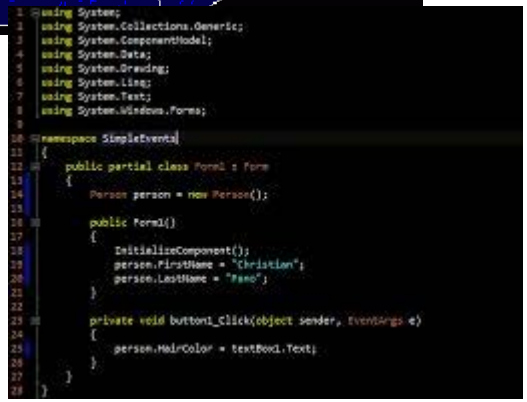






Survey of Scientific Computing (SciComp 301)

Dave Biersach
Brookhaven National Laboratory
dbiersach@bnl.gov

Session 04
Vectors,
Random Numbers,
Timing

Session Goals

- Write code to correctly & efficiently deal a deck of cards
  - **Encode** and **decode** two independent concepts into a single integer using the **/** and **%** (modulus) operators
  - Declare and define a **vector** of a given data type
  - Access elements of a vector using the **.at()** method
- Use the high precision **clock()** function to
  - … carefully measure the elapsed run time of algorithms
  - … to systematically improve code performance
  - … because **in SciComp speed & accuracy are paramount!**

Encoding (Representation)

- Cards in a deck are numbered **0 – 51**

Card Suit	
0	**Clubs**
1	**Diamonds**
2	**Hearts**
3	**Spades**

Card Rank	
0	**Deuce**
1	**Three**
2	**Four**
3	**Five**
4	**Six**
5	**Seven**
6	**Eight**
7	**Nine**
8	**Ten**
9	**Jack**
10	**Queen**
11	**King**
12	**Ace**

How can we convert *to* & *from* a **card #** and a specific **suit** and **rank**?

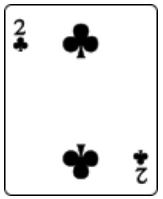
Encoding (Representation)

- Cards in a deck are numbered **0 – 51**

Card Suit	
0	**Clubs**
1	**Diamonds**
2	**Hearts**
3	**Spades**

Card Rank	
0	**Deuce**
1	**Three**
2	**Four**
3	**Five**
4	**Six**
5	**Seven**
6	**Eight**
7	**Nine**
8	**Ten**
9	**Jack**
10	**Queen**
11	**King**
12	**Ace**

Card # = Suit * 13 + Rank



Suit = **0**
Rank = **0**
0 * 13 + 0 = **0**

Suit = **2**
Rank = **10**
2 * 13 + 10 = **36**

Suit = **3**
Rank = **12**
3 * 13 + 12 = **51**

Decoding (Representation)

- Cards in a deck are numbered **0 – 51**

Card Suit	
0	**Clubs**
1	**Diamonds**
2	**Hearts**
3	**Spades**

Card Rank	
0	**Deuce**
1	**Three**
2	**Four**
3	**Five**
4	**Six**
5	**Seven**
6	**Eight**
7	**Nine**
8	**Ten**
9	**Jack**
10	**Queen**
11	**King**
12	**Ace**

Suit = (Card #) / 13
Rank = (Card #) % 13

With integers,
/ returns a whole number
39 / 7 = 5

% is the modulus
(remainder)
39 % 7 = 4

Decoding (Representation)

- Cards in a deck are numbered **0 – 51**

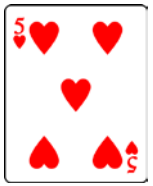
Card Suit	
0	**Clubs**
1	**Diamonds**
2	**Hearts**
3	**Spades**

Card Rank	
0	**Deuce**
1	**Three**
2	**Four**
3	**Five**
4	**Six**
5	**Seven**
6	**Eight**
7	**Nine**
8	**Ten**
9	**Jack**
10	**Queen**
11	**King**
12	**Ace**

Suit = (Card #) / 13
Rank = (Card #) % 13

Card # = **11**
Suit = 11 / 13 = **0**
Rank = 11 % 13 = **11**

Card # = **29**
Suit = 29 / 13 = **2**
Rank = 29 % 13 = **3**

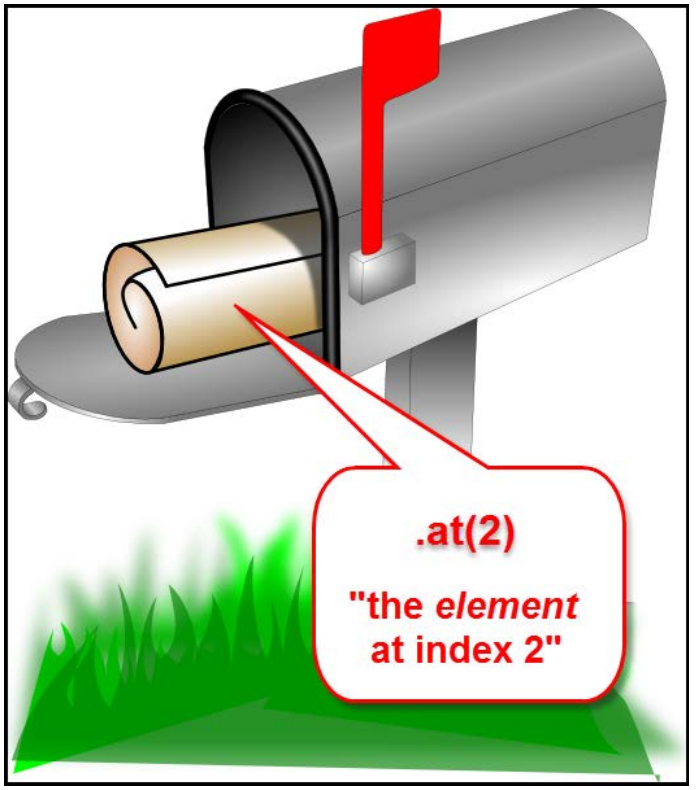


Card # = **48**
Suit = 48 / 13 = **3**
Rank = 48 % 13 = **9**

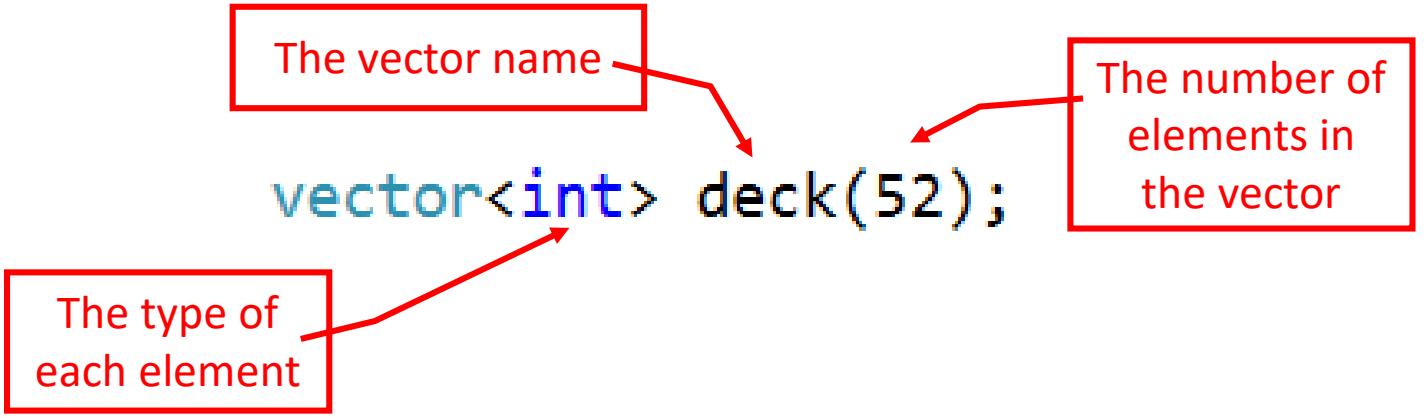
Vectors

- A vector is a set of *elements* that share *a common type*
  - Example: a vector of **int**, where every element is an integer
  - Example: an vector of **bool**, where every element is **true** or **false**
  - All elements in an array have the exact same data type
- Individual elements in a vector are accessed by using their **index number**
  - Every element has a unique index number
  - No two elements share the exact same index number
  - **The first element has an index = 0**

Vectors



Vectors



- Every element in a vector can be accessed by providing an **index** number within the **.at()** method - the **first element has index 0 (zero)!!**
  - Example: **`deck.at(0)`** is the first element in that 1D array
  - Example: **`deck.at(5)`** is the sixth element in that 1D array
- Arrays are zero index based, so the highest index (last addressable element) is **`deck.size() – 1`**

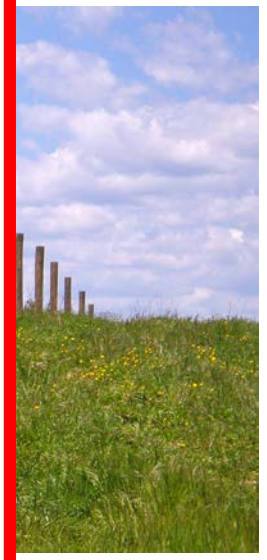
The Bane of All Programmers

- A farmer
  100m lon
- He wants
  each fenc
- How mar
  need?

This problem is why we all agree to always use ZERO as the *first* index value in an array.

Remember…

ZERO is a THING!



Off-by-one err

From Wikipedia, the free encyclopedia

An **off-by-one error** (**OBOE**), also commonly known as an **OBOB** (**off-by-one bug**), is a logic error involving the discrete equivalent of a boundary condition. It often occurs in computer programming when an iterative loop iterates one time too many or too few. This problem could arise when a programmer makes mistakes such as using "is less than or equal to" where "is less than" should have been used in a comparison or fails to take into account that a sequence starts at zero rather than one (as with array indices in many languages). This can also occur in a mathematical context.

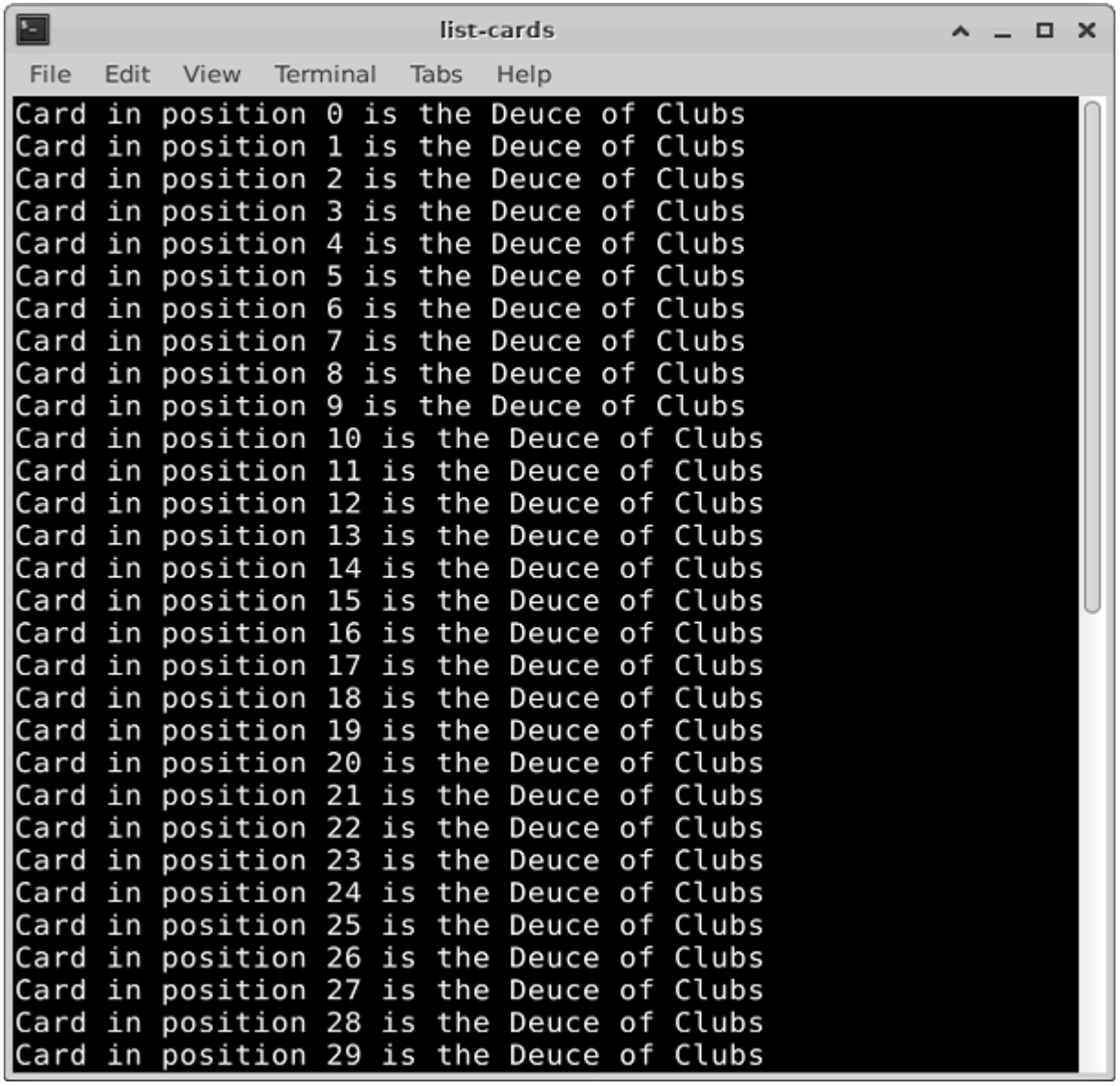
main.cpp

```
1  #include "stdafx.h"
2
3  using namespace std;
4
5  void InitDeck(vector<int>& deck)
6  {
7      for (size_t i{}; i < deck.size(); ++i)
8          deck.at(i) = i;
9  }
10
11 void DisplayCards(vector<int>& deck)
12 {
13     const vector<string> suit{ "Clubs", "Diamonds",
14                                "Hearts", "Spades" };
15
16     const vector<string> rank{ "Deuce", "Three", "Four",
17                                "Five", "Six", "Seven",
18                                "Eight", "Nine", "Ten",
19                                "Jack", "Queen", "King",
20                                "Ace" };
21
22     for (size_t i{}; i < deck.size(); ++i)
23     {
24         int card = deck.at(i);
25         cout << "Card in position " << i
26              << " is the " << rank.at(0)
27              << " of " << suit.at(0) << endl;
28     }
29 }
30
31 int main()
32 {
33     vector<int> deck(52);
34
35     InitDeck(deck);
36
37     DisplayCards(deck);
38
39     return 0;
40 }
41
```

Position Number	0	1	2	3	4
Card Number	0	1	2	3	4

Open Lab 1

- Write a program to display on screen the rank and suit of all 52 cards in an **sorted** deck



Run Lab 1
List Cards

main.cpp

```
1   #include "stdafx.h"
2
3   using namespace std;
4
5   void InitDeck(vector<int>& deck)
6   {
7       for (size_t i{}; i < deck.size(); ++i)
8           deck.at(i) = i;
9   }
10
11  void DisplayCards(vector<int>& deck)
12  {
13      const vector<string> suit{ "Clubs", "Diamonds",
14                                  "Hearts", "Spades" };
15
16      const vector<string> rank{ "Deuce", "Three", "Four",
17                                  "Five", "Six", "Seven",
18                                  "Eight", "Nine", "Ten",
19                                  "Jack", "Queen", "King",
20                                  "Ace" };
21
22      for (size_t i{}; i < deck.size(); ++i)
23      {
24          int card = deck.at(i);
25          cout << "Card in position " << i
26               << " is the " << rank.at(0)
27               << " of " << suit.at(0) << endl;
28      }
29  }
30
31  int main()
32  {
33      vector<int> deck(52);
34
35      InitDeck(deck);
36
37      DisplayCards(deck);
38
39      return 0;
40  }
41
```

Position Number	0	1	2	3	4
Card Number	0	1	2	3	4

Edit Lab 1

- Write a program to display on screen the rank and suit of all 52 cards in an **sorted** deck
- You must fix the **cout**

main.cpp

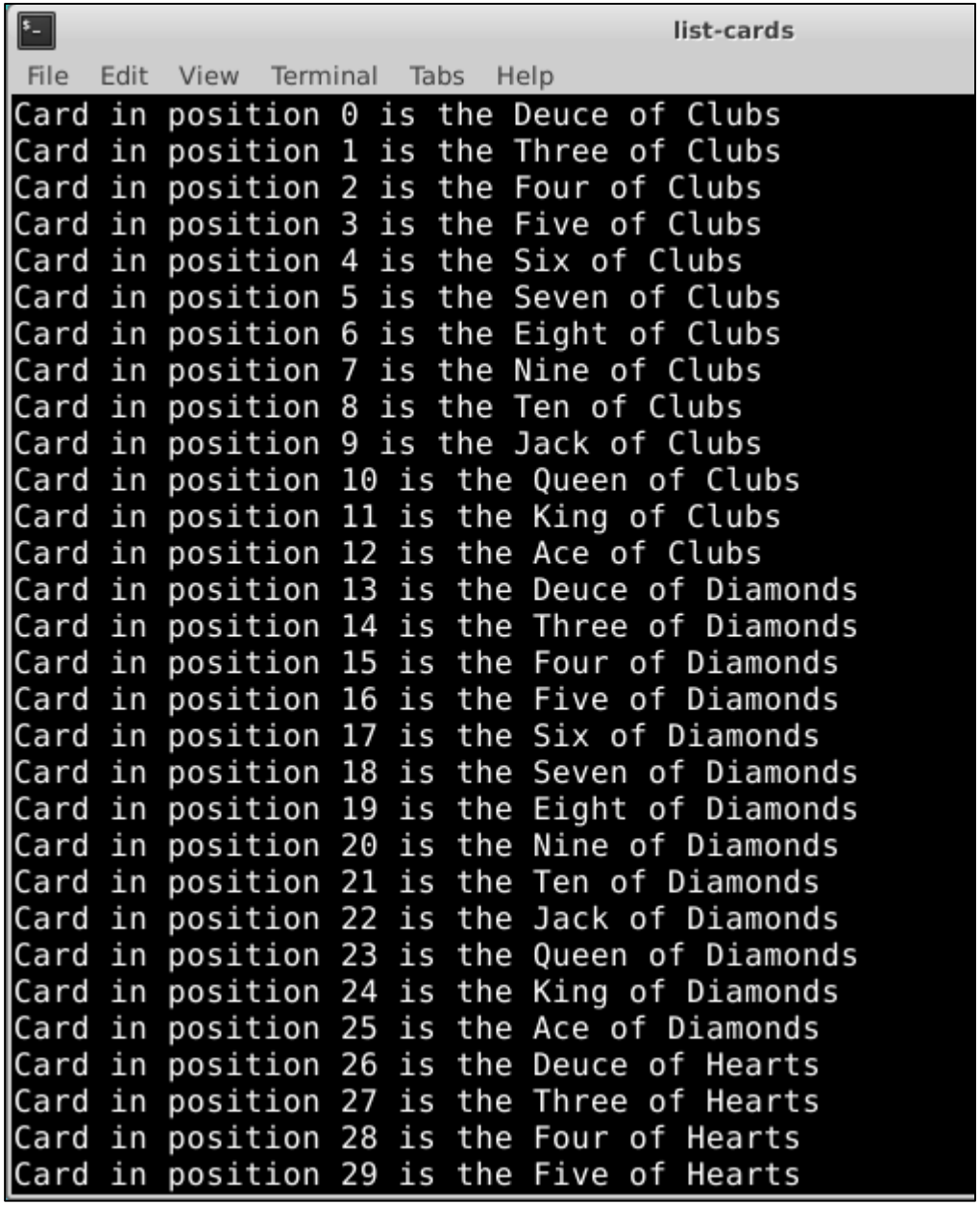
```
1   #include "stdafx.h"
2
3   using namespace std;
4
5   void InitDeck(vector<int>& deck)
6   {
7       for (size_t i{}; i < deck.size(); ++i)
8           deck.at(i) = i;
9   }
10
11  void DisplayCards(vector<int>& deck)
12  {
13      const vector<string> suit{ "Clubs", "Diamonds",
14                                 "Hearts", "Spades" };
15
16      const vector<string> rank{ "Deuce", "Three", "Four",
17                                 "Five", "Six", "Seven",
18                                 "Eight", "Nine", "Ten",
19                                 "Jack", "Queen", "King",
20                                 "Ace" };
21
22      for (size_t i{}; i < deck.size(); ++i)
23      {
24          int card = deck.at(i);
25          cout << "Card in position " << i
26               << " is the " << rank.at(0)
27               << " of " << suit.at(0) << endl;
28      }
29  }
30
31  int main()
32  {
33      vector<int> deck(52);
34
35      InitDeck(deck);
36
37      DisplayCards(deck);
38
39      return 0;
40  }
41
```

Position Number	0	1	2	3	4
Card Number	0	1	2	3	4

Edit Lab 1

- Write a program to display on screen the rank and suit of all 52 cards in an **sorted** deck
- You must fix the **cout**

```
22      for (size_t i{}; i < deck.size(); ++i)
23      {
24          int card = deck.at(i);
25          cout << "Card in position " << i
26               << " is the " << rank.at(card % 13)
27               << " of " << suit.at(card / 13) << endl;
28      }
29  }
```

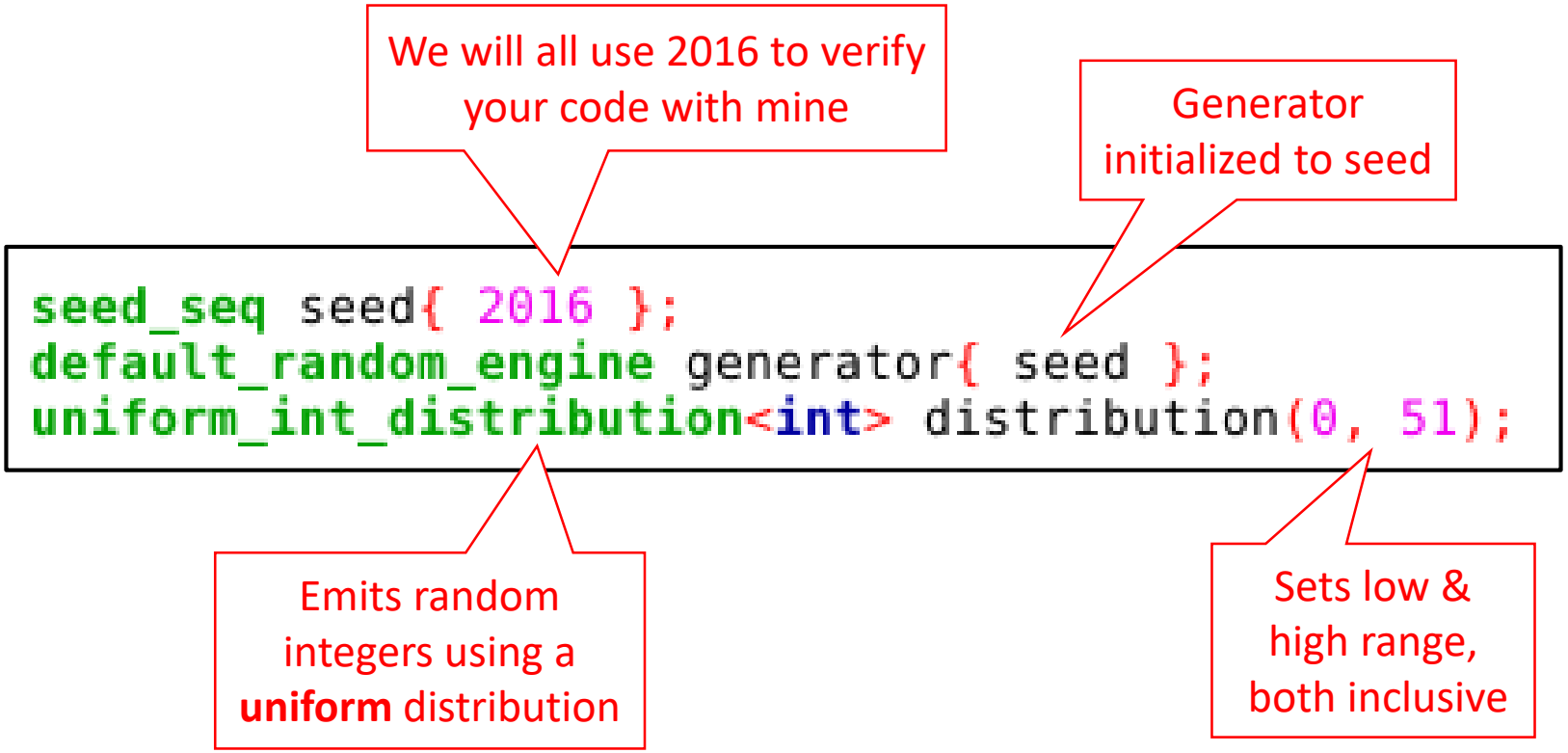


Check Lab 1 - List Cards

“Random” Numbers

- C++ has built-in support to generate random numbers
  - Step #1: Create a **seed**, based upon a **fixed** number
  - Step #2: Create a random **generator** using that seed
  - Step #3: Create a **distribution** based upon the desired range
  - Step #4: Generate random numbers by passing the generator into the distribution
- By using the same seed value, you will get the same sequence of “random” numbers ***every*** run of your program
  - ***All of our computers*** will return the exact same sequence if we all initialize our PRNG with the same seed value!
  - This is a pseudo-random number generator (PRNG)

“Random” Numbers



Open Lab 2 – Bogus Card Dealer

- Write a program to *initialize* then randomly **deal** and display a deck of cards

```
16  void DealCards(vector<int>& deck)
17  {
18      for (auto& card : deck)
19          card = distribution(generator);
20  }
```

This is a **ranged based** for() loop.
It is a simpler way to enumerate an entire vector to set the value of each element

Each time this **distribution()** function is called, it returns a new uniformly distributed random integer between 0 and 51

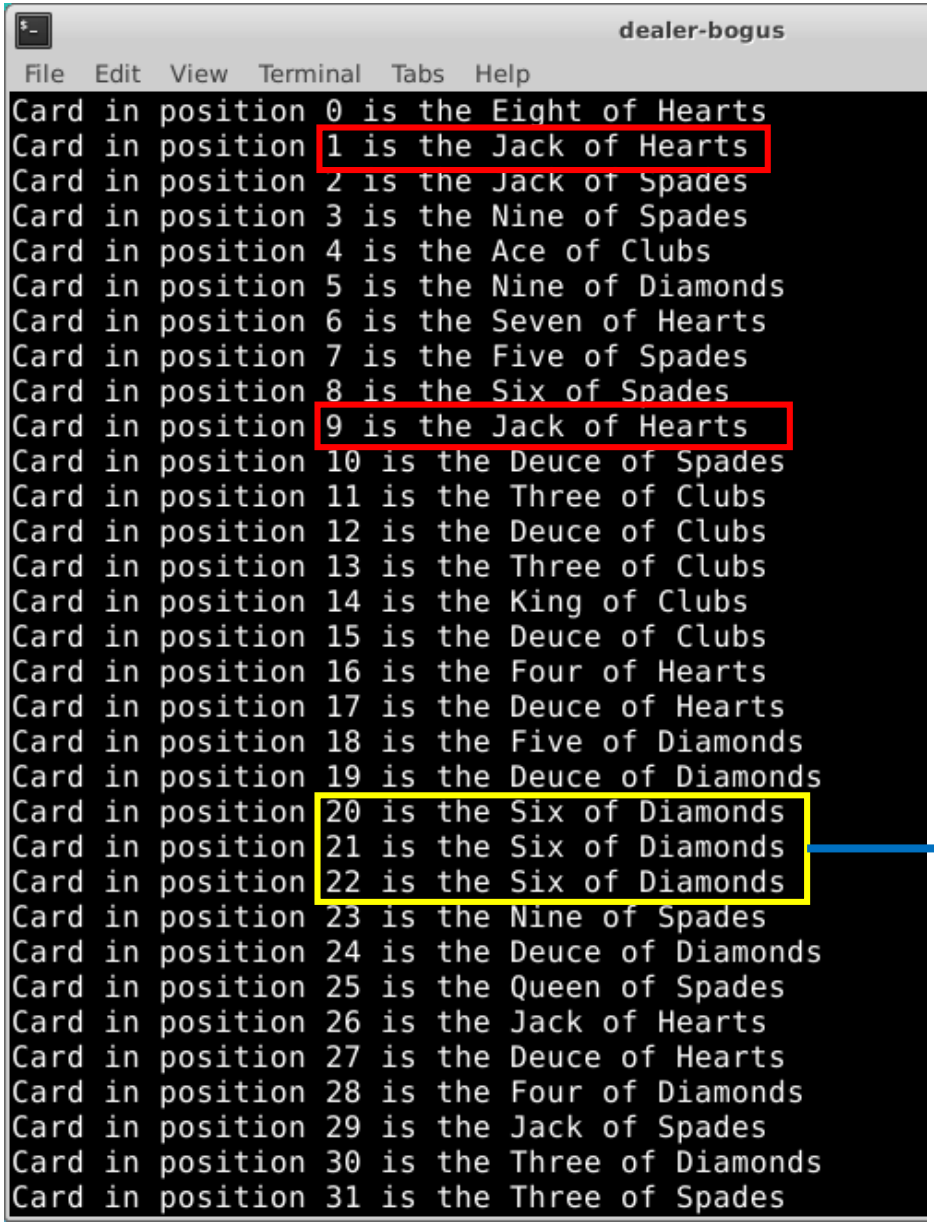
Run Lab 2 – Bogus Card Dealer

- Write a program to *initialize* then randomly **deal** and display a deck of cards

```
16  void DealCards(vector<int>& deck)
17  {
18      for (auto& card : deck)
19          card = distribution(generator);
20  }
```

Position Number	0	1	2	3	4
Card Number	13	36	29	52	2

Run Lab 2



Check Lab 2 - Bogus Card Dealer

Position Number	19	20	21	22	23
Card Number	13	17	17	17	46

Random… but no repeats?

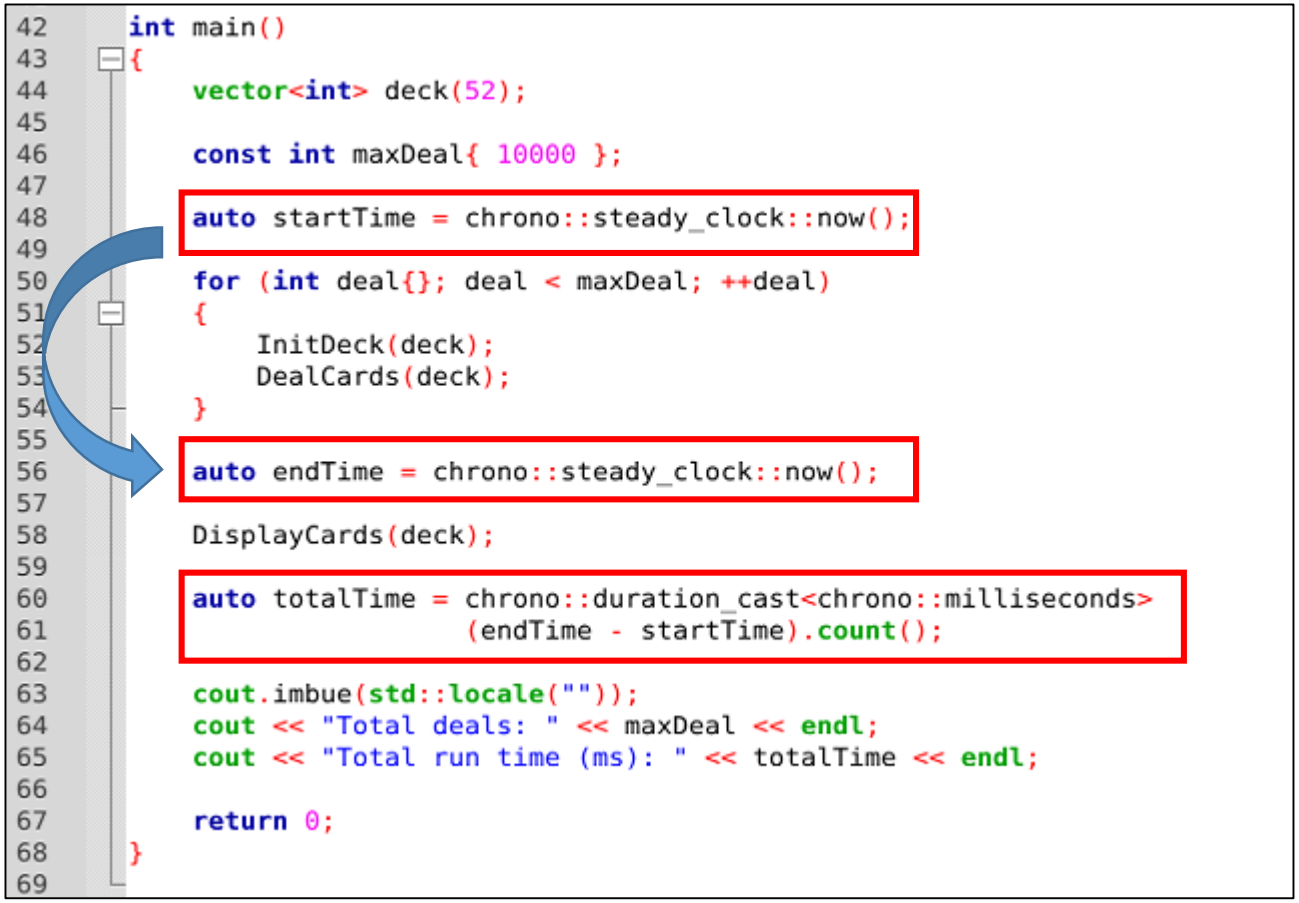
- How can we get a set of random numbers where no number is repeated until ***all*** numbers are picked at least once?
- Can we flag that a particular card # has already been dealt, and therefore not deal that card again?

Instrumenting Your Code

- Instrumenting code is the process of taking accurate **timings** of the runtime performance of key algorithms within the program
- C++ provides a **steady_clock** object that can measure time to the nearest millisecond (1/1000th of a second) which is sufficient in most situations
- We bracket the code under analysis by measuring the clock immediately ***before*** the start and again ***after*** the end of the algorithm to calculate the **elapsed** time
- Careful tracking of code timings will provide objective empirical evidence if changes to algorithms and/or data structures are indeed making the program more efficient

Open Lab 3 – Slow Card Dealer

Write code to **correctly** deal **10,000** decks



```
42  int main()
43  {
44      vector<int> deck(52);
45
46      const int maxDeal{ 10000 };
47
48      auto startTime = chrono::steady_clock::now();
49
50      for (int deal{}; deal < maxDeal; ++deal)
51      {
52          InitDeck(deck);
53          DealCards(deck);
54      }
55
56      auto endTime = chrono::steady_clock::now();
57
58      DisplayCards(deck);
59
60      auto totalTime = chrono::duration_cast<chrono::milliseconds>
61                       (endTime - startTime).count();
62
63      cout.imbue(std::locale(""));
64      cout << "Total deals: " << maxDeal << endl;
65      cout << "Total run time (ms): " << totalTime << endl;
66
67      return 0;
68  }
69
```

Edit Lab 3 – Slow Card Dealer

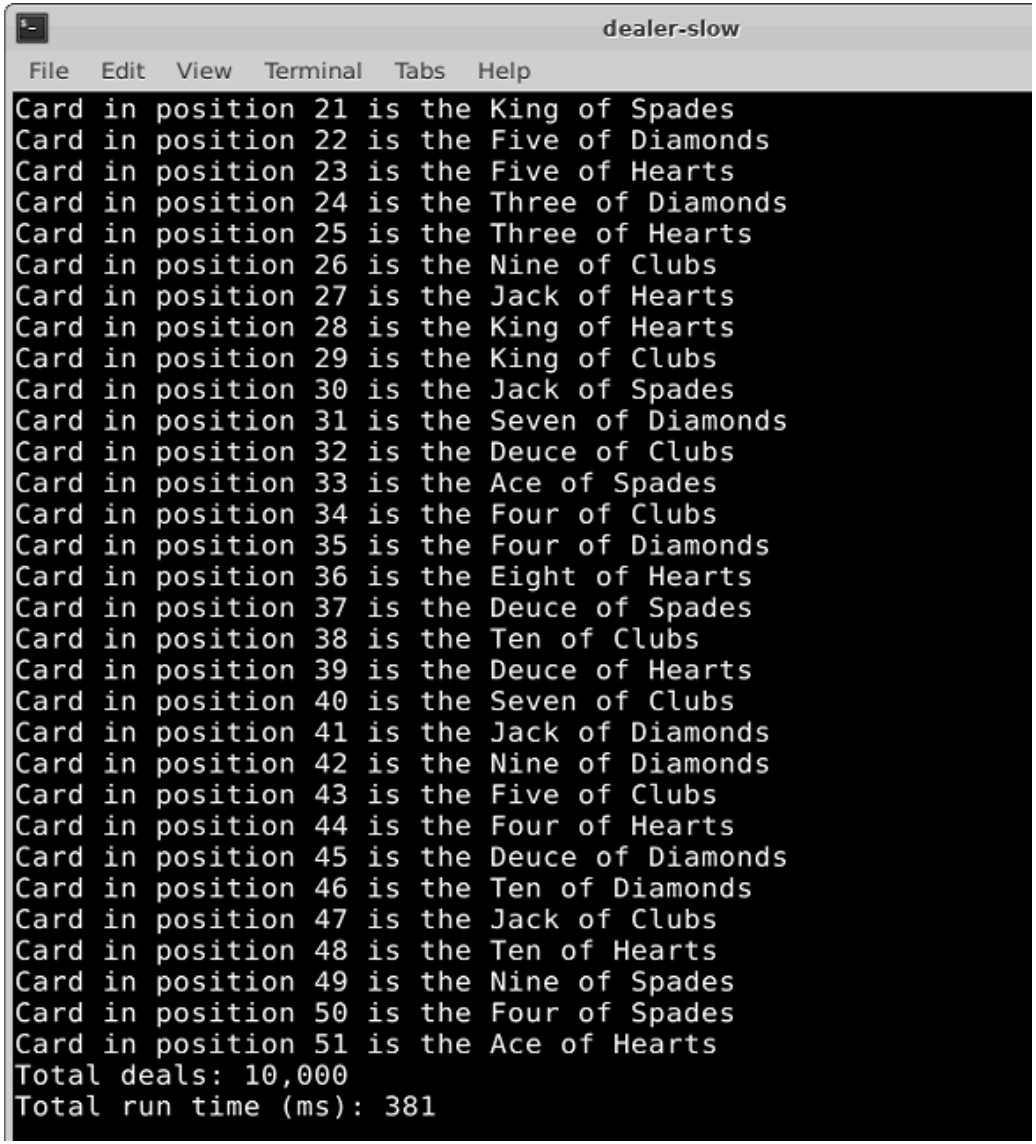
- We need a ***helper*** vector to store a **true** or **false** flag to record if a random "trial" card # has already been dealt
- Keeping [illegible] hat has not yet b[illegible] r to record the fact that card # has been dealt

Run Lab 3

Make your **DealCards()** look like this

Position Number	19	20	21	22	23
Card Number	13	17	~~17~~		
	13	17	7	~~13~~	
	13	17	7	50	6

```
15  void DealCards(vector<int>& deck)
16  {
17      vector<bool> alreadyDealt(52, false);
        for (auto& card : deck)
        {
            card = distribution(generator);
            while (alreadyDealt.at(card))
                card = distribution(generator);
            alreadyDealt.at(card) = true;
        }
25  }
```



Check Lab 3
Slow Card Dealer

381 ms

Correct but inefficient…

```
void DealCards(vector<int>& deck)
{
    for (auto& card : deck)
        card = distribution(generator);
}
```

Bogus Card Dealer

```
void DealCards(vector<int>& deck)
{
    vector<bool> alreadyDealt(52, false);
    for (auto& card : deck) {
        card = distribution(generator);
        while (alreadyDealt.at(card))
            card = distribution(generator);
        alreadyDealt.at(card) = true;
    }
}
```

Slow Card Dealer

A *Faster* Card Dealer

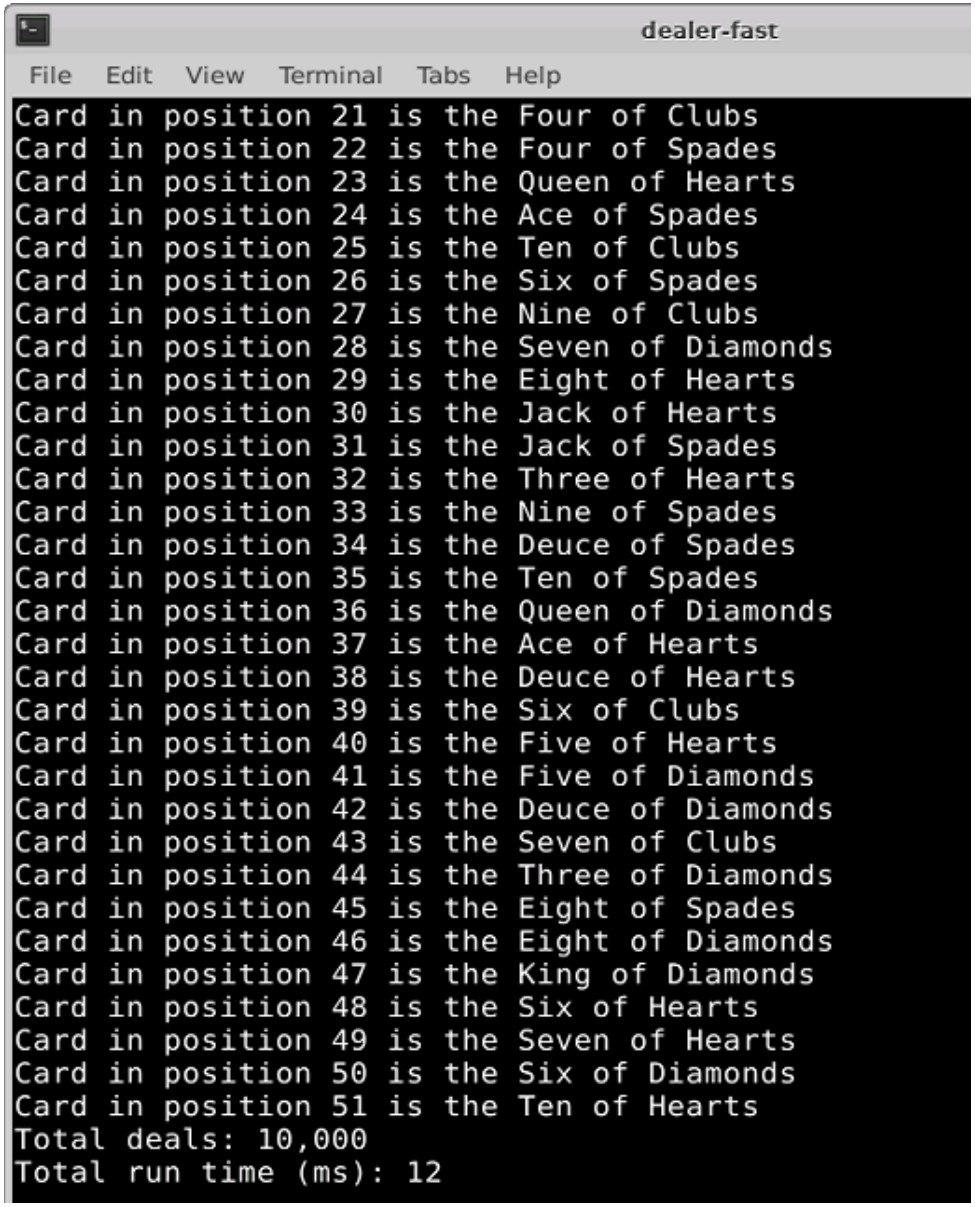
- Sadly there is an inherent inefficiency in the naïve algorithm employed in the **DealCards**() function from Lab 3
- It takes **longer and longer**, as more cards are dealt, to randomly pick (**i.e. to find**) a card that has not yet been dealt
- We need to discover an algorithm that, while ensuring every card is dealt only once, doesn't lose time at the end of the deal searching for ***that one remaining card*** that has not yet been dealt
- The improved algorithm doesn’t need an **alreadyDealt** helper **bool** vector and it **was discovered by a 7th grader!**

Run Lab 4 – Fast Card Dealer

- *No coding required – **just run the solution***
- Consider the revised function **DealCards()**

```
void DealCards(vector<int>& deck)
{
    for (auto& card : deck)
        swap(card,
             deck.at(distribution(generator)));
}
```

- C++ has a built-in function **swap**() that exchanges the values of the two parameters passed to it



Check Lab 4 - Fast Card Dealer

12 ms

~3000% *faster* than Slow Card Dealer !

Slow vs. Fast Card Dealer

```
void DealCards(vector<int>& deck)
{
    vector<bool> alreadyDealt(52, false);
    for (auto& card : deck) {
        card = distribution(generator);
        while (alreadyDealt.at(card))
            card = distribution(generator);
        alreadyDealt.at(card) = true;
    }
}
```

Slow Card Dealer

Email me with an explanation in your own words as to why this algorithm works and is so much faster!

```
void DealCards(vector<int>& deck)
{
    for (auto& card : deck)
        swap(card,
             deck.at(distribution(generator)));
}
```

Fast Card Dealer

- Fewer lines of code
- No helper vector needed
- ~3000% faster
- Discovered by a 7th grader

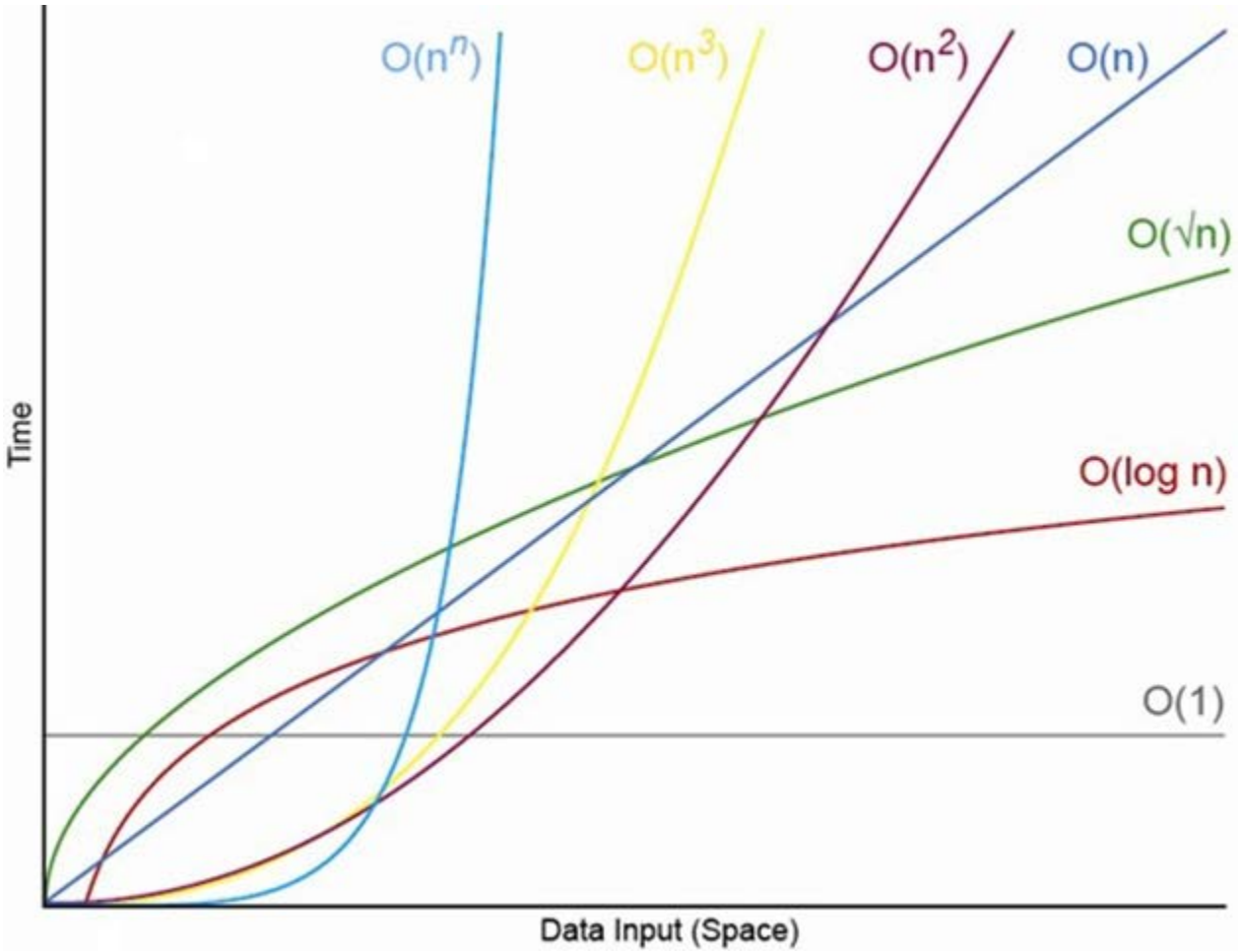
Computing is a **New** Science

- The best algorithms are the ones that leave you scratching your head thinking "**…that was so obvious – why didn't I think of that?**"
  - They are often **the shortest** algorithms in terms of source code length
  - They are also normally **the fastest** algorithms to execute
- Even young students can get a flash of inspiration and see something new – **there is always a better way!**

Algorithmic Efficiency

- Scientific computing often involves analyzing large data sets or running large-scale simulations
- It is very important to have code that runs as fast as possible while returning the correct results
- We measure algorithm efficiency by estimating the impact on the **total run time** as the **size of the input data increases**
- We are only interested in the principal term which describes the overall "**order**" of the algorithm, and are not necessarily concerned about estimating the exact run time
- The order of an algorithm is expressed in "**Big O**" notation
- The optimal algorithms have the smallest possible order

Algorithmic Efficiency



Algorithmic Efficiency

Notation	Name	Examples
$O(1)$	constant	Determining if a number is even or odd; Using a constant-size lookup table; Using a suitable hash function for looking up an item.
$O(\log n)$	logarithmic	Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a Binomial heap.
$O(n)$	linear	Finding an item in an unsorted list or a malformed tree (worst case) or in an unsorted array; Adding two *n*-bit integers by ripple carry.
$O(n \log n)$	linearithmic, loglinear, or quasilinear	Performing a Fast Fourier transform; heapsort, quicksort (best and average case), or merge sort
$O(n^2)$	quadratic	Multiplying two *n*-digit numbers by a simple algorithm; bubble sort (worst case or naive implementation), Shell sort, quicksort (worst case), selection sort or insertion sort
$O(c^n),\ c > 1$	exponential	Finding the (exact) solution to the travelling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search

Open Lab 5 - Primality Race

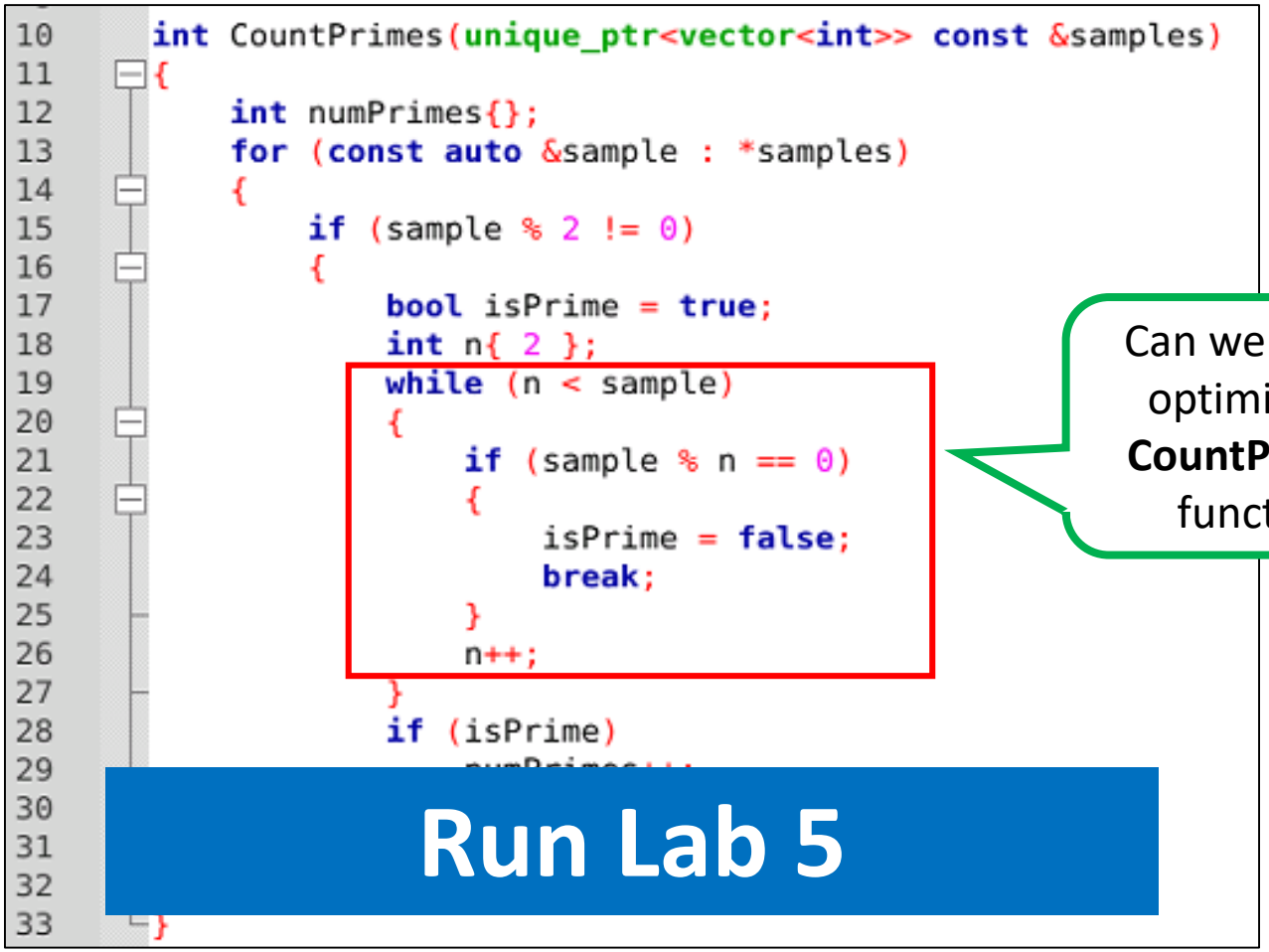
- Write a program to generate a vector of **100,000** random **samples** (all between 100,000 and 999,999 inclusive)
- Count the number of **prime** numbers within that vector
- Use the **steady_clock** object to instrument your code and measure the total run time to the nearest millisecond
- The student who writes the code that calculates **the correct prime count** with the **shortest running time** wins!

View Lab 5
Primality Race

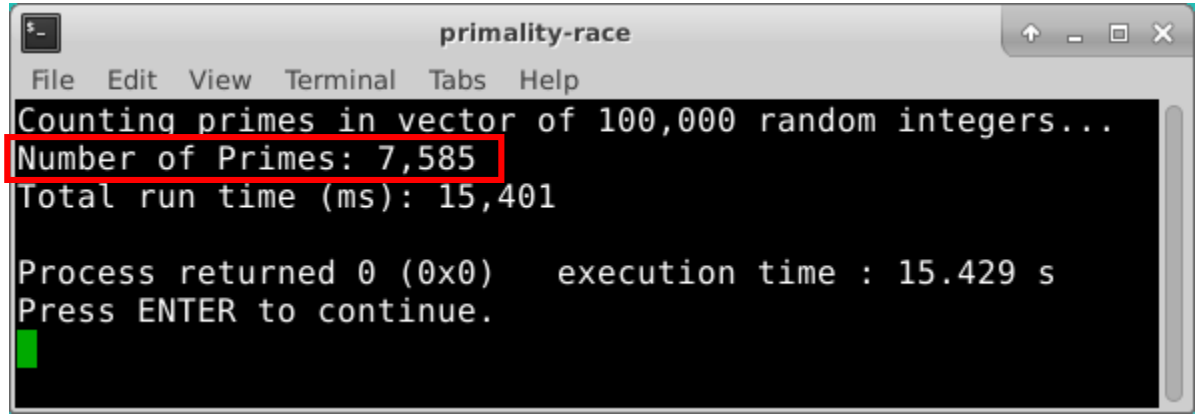
main.cpp

```
1   #include "stdafx.h"
2
3   using namespace std;
4   using namespace chrono;
5
6   seed_seq seed{ 2016 };
7   default_random_engine generator{ seed };
8   uniform_int_distribution<int> distribution(100000, 999999);
9
10  int CountPrimes(unique_ptr<vector<int>> const &samples)
11  {
34
35  int main()
36  {
37      const auto samples{ make_unique<vector<int>>(100000) };
38
39      for (auto &sample : *samples)
40          sample = distribution(generator);
41
42      cout.imbue(std::locale(""));
43      cout << "Counting primes in vector of "
44           << samples->size() << " random integers..."
45           << endl;
46
47      auto startTime = steady_clock::now();
48
49      int numPrimes = CountPrimes(samples);
50
51      auto endTime = steady_clock::now();
52
53      auto totalTime = duration_cast<milliseconds>
54                       (endTime - startTime).count();
55
56      cout << "Number of Primes: " << numPrimes << endl;
57      cout << "Total run time (ms): " << totalTime << endl;
58
59      return 0;
60  }
61
```

View Lab 5 - Primality Race

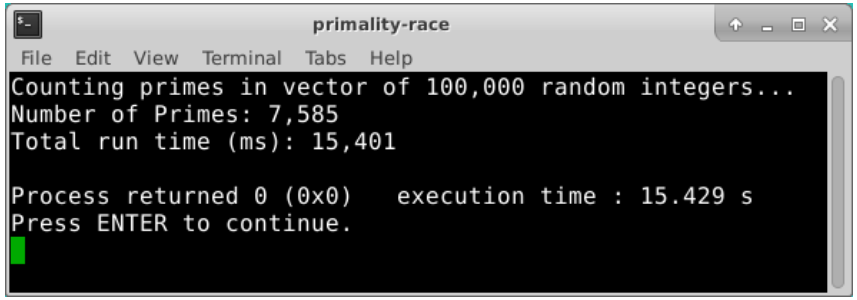


Check Lab 5 - Primality Race



- How much can you **decrease the run time?**
- You are free to create and initialize any additional helper data structures (vectors, Boolean flags, etc.) before the first call to set **startTime = steady_clock::now()**
- You can execute a **"Release build"** if you wish
- Each new version of your code should identify **7,585** primes

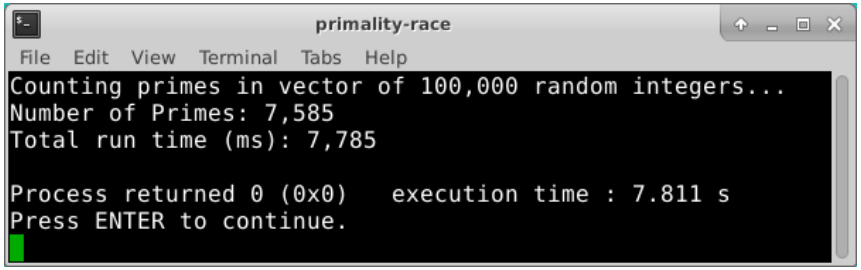
Primality Race



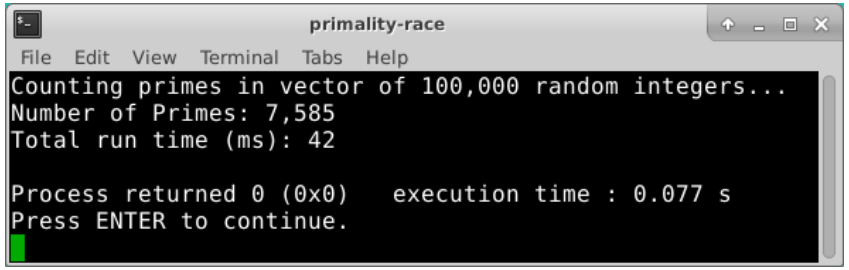
Can your code beat my *best* time?

Email your code to
scicomp@bnl.gov

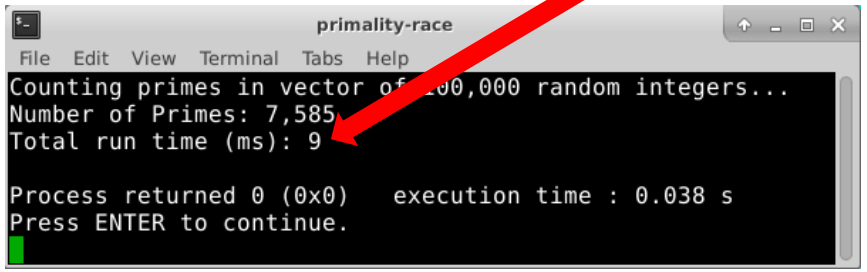
v2



v3



v4



~ 17,000 %
faster than v1

Primality Race *Hints*

```
void InitPrimes();
int CountPrimes(unique_ptr<vector<int>> const &samples);

seed_seq seed{ 2016 };
default_random_engine generator{ seed };
uniform_int_distribution<int> distribution(100000, 999999);

int primes[169];
```

```
int main()
{
    const auto samples{ make_unique<vector<int>>(100000) };

    for (auto &sample : *samples)
        sample = distribution(generator);

    InitPrimes();
```

- Avoid trial dividing every number from $2 < sample$
- The **Fundamental Theorem of Arithmetic** says all integers are composed of primes to various powers
- We only need to test if $sample \bmod p = 0$, $\forall\ p\ where\ \boldsymbol{p \in primes}$
- We **precompute** an array of primes $< \sqrt{999{,}999}$
- We will then trial divide every sample **using only this array of primes**

Primality Race *Hints*

```
12  void InitPrimes()
13  {
14      int numPrimes{};
15      int n{ 3 };
16      while (numPrimes < 169)
17      {
18          if (n % 2 == 1)
19          {
20              bool isPrime = true;
21              for (int p{}; p < numPrimes; ++p)
22                  if (n % primes[p] == 0)
23                  {
24                      isPrime = false;
25                      break;
26                  }
27              if (isPrime)
28              {
29                  primes[numPrimes] = n;
30                  numPrimes++;
31              }
32          }
33          n += 2;
34      }
35  }
```

- The prime array will only store **odd** primes starting with 3, as code in **main()** will rule out *even* samples
- Riemann's prime counting function (based upon his famous Zeta function) shows $\pi(999) = 168$
- To build the prime array, we trial divide every odd **n** by the ***existing*** numbers in the prime array
- If no **witness** is found showing **n** to be **composite**, then **n** is added to the **end** of the global prime array

Primality Race *Hints*

```
37  int CountPrimes(unique_ptr<vector<int>> const &samples)
38  {
39      int numPrimes{};
40      for (const auto &sample : *samples)
41      {
42          if (sample % 2 != 0)
43          {
44              int sqrtSample = sqrt(sample);
45              bool isPrime = true;
46              for (int n{}; primes[n] <= sqrtSample; ++n)
47                  if (sample % primes[n] == 0)
48                  {
49                      isPrime = false;
50                      break;
51                  }
52              if (isPrime)
53                  numPrimes++;
54          }
55      }
56      return numPrimes;
57  }
```

- We can immediately exclude all **even** samples from the count of primes
- We only need to trial divide primes up to $\sqrt{sample}$
- Switching to a **release** build enables further compiler optimizations
- The time penalty for building the initial prime array **is more than offset** by the speedup when testing the 100,000 samples

Now you know…

- A **vector** is a set where every element has the same type
- Every vector element has a unique index value
- The first addressable vector element has an index **= 0**
- We access the element in a vector using the **.at()** method
- The **.size()** method returns the number of elements in the vector
- The **.push_back()** method adds elements to the end of a vector
- The **modulus** (remainder) operator enables clever encoding to save space and time