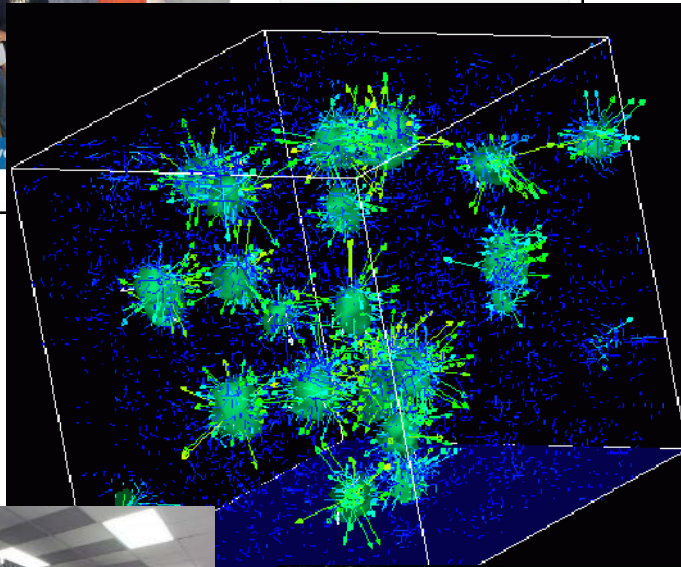




# Survey of Scientific Computing (SciComp 301)

Dave Biersach  
Brookhaven National  
Laboratory  
[dbiersach@bnl.gov](mailto:dbiersach@bnl.gov)



```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Windows.Forms;
9
10 namespace SimpleEvents
11 {
12     public partial class Form1 : Form
13     {
14         Person person = new Person();
15
16         public Form1()
17         {
18             InitializeComponent();
19             person.FirstName = "Christian";
20             person.LastName = "Pano";
21         }
22
23         private void button1_Click(object sender, EventArgs e)
24         {
25             person.MainColor = textBox1.Text;
26         }
27     }
28 }
```

**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

$$\text{Card \#} = \text{Suit} * 13 + \text{Rank}$$



Suit = **0**

Rank = **0**

$$0 * 13 + 0 = \underline{0}$$



Suit = **2**

Rank = **10**

$$2 * 13 + 10 = \underline{36}$$



Suit = **3**

Rank = **12**

$$3 * 13 + 12 = \underline{51}$$

# 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

$$\text{Suit} = (\text{Card \#}) / 13$$

$$\text{Rank} = (\text{Card \#}) \% 13$$

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

$$\text{Suit} = (\text{Card \#}) / 13$$

$$\text{Rank} = (\text{Card \#}) \% 13$$

Card # = **11**

Suit =  $11 / 13 =$ **0**

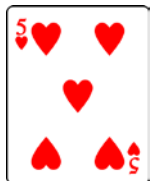
Rank =  $11 \% 13 =$ **11**



Card # = **29**

Suit =  $29 / 13 =$ **2**

Rank =  $29 \% 13 =$ **3**



Card # = **48**

Suit =  $48 / 13 =$ **3**

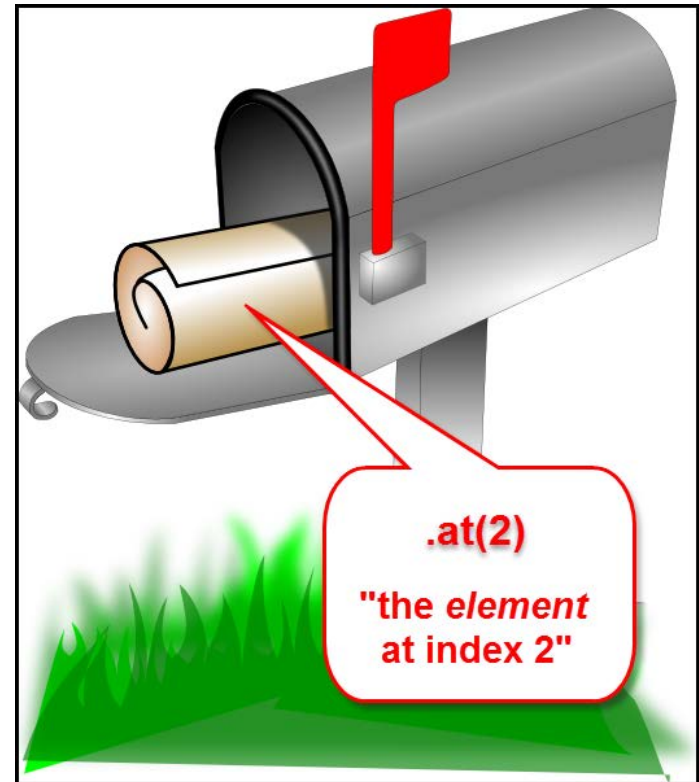
Rank =  $48 \% 13 =$ **9**



# Vectors

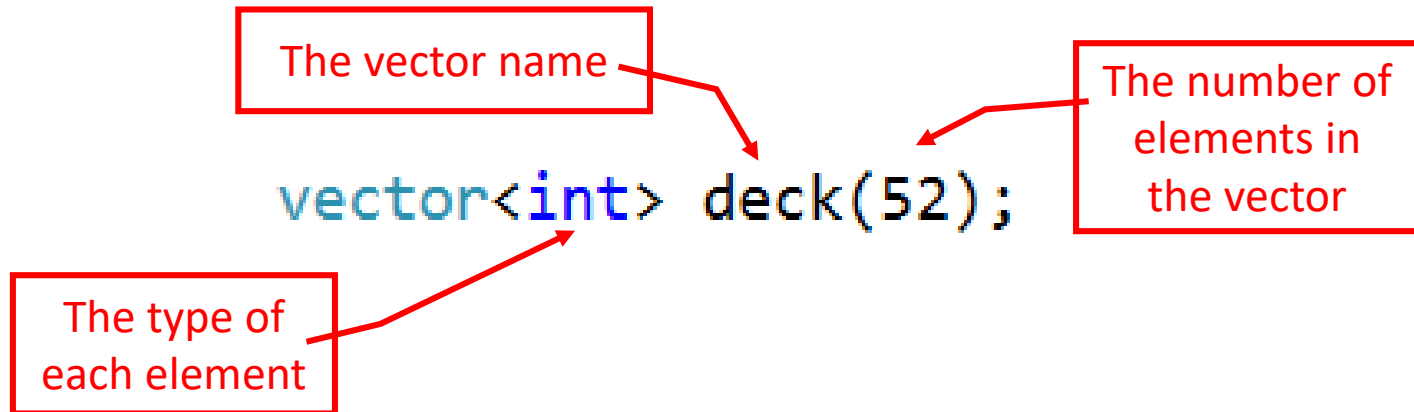
- A vector is a set of *elements* that share *a common type*
  - Example: a vector of **int**, where every element is an integer
  - Example: an vector of **bool**, where every element is **true** or **false**
  - All elements in an array have the exact same data type
- Individual elements in a vector 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 has a fence 100m long
- He wants to divide it into 100 equal parts
- How many fence posts does he 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 error

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.

# Edit Lab 1 – List Cards

```
ListCards.cpp
1  // ListCards.cpp
2
3  #include "stdafx.h"
4
5  using namespace std;
6
7  void InitDeck(vector<int>& deck)
8  {
9      for (size_t i{}; i < deck.size(); ++i)
10         deck.at(i) = i;
11 }
12
13 void DisplayCards(vector<int>& deck)
14 {
15     const vector<string> suit{ "Clubs", "Diamonds",
16                               "Hearts", "Spades" };
17
18     const vector<string> rank{ "Deuce", "Three", "Four",
19                               "Five", "Six", "Seven",
20                               "Eight", "Nine", "Ten",
21                               "Jack", "Queen", "King",
22                               "Ace" };
23
24     for (size_t i{}; i < deck.size(); ++i) {
25         int card = deck.at(i);
26         cout << "Card in position " << i
27              << " is the " << rank.at(0)
28              << " of " << suit.at(0) << endl;
29     }
30 }
31
32 int main()
33 {
34     vector<int> deck(52);
35
36     InitDeck(deck);
37
38     DisplayCards(deck);
39
40     return 0;
41 }
42
```

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



```
24     for (size_t i{}; i < deck.size(); ++i) {
25         int card = deck.at(i);
26         cout << "Card in position " << i
27              << " is the " << rank.at(card % 13)
28              << " of " << suit.at(card / 13) << endl;
29     }
30 }
```

```
list-cards
File Edit View Terminal Tabs Help
Card in position 0 is the Deuce of Clubs
Card in position 1 is the Three of Clubs
Card in position 2 is the Four of Clubs
Card in position 3 is the Five of Clubs
Card in position 4 is the Six of Clubs
Card in position 5 is the Seven of Clubs
Card in position 6 is the Eight of Clubs
Card in position 7 is the Nine of Clubs
Card in position 8 is the Ten of Clubs
Card in position 9 is the Jack of Clubs
Card in position 10 is the Queen of Clubs
Card in position 11 is the King of Clubs
Card in position 12 is the Ace of Clubs
Card in position 13 is the Deuce of Diamonds
Card in position 14 is the Three of Diamonds
Card in position 15 is the Four of Diamonds
Card in position 16 is the Five of Diamonds
Card in position 17 is the Six of Diamonds
Card in position 18 is the Seven of Diamonds
Card in position 19 is the Eight of Diamonds
Card in position 20 is the Nine of Diamonds
Card in position 21 is the Ten of Diamonds
Card in position 22 is the Jack of Diamonds
Card in position 23 is the Queen of Diamonds
Card in position 24 is the King of Diamonds
Card in position 25 is the Ace of Diamonds
Card in position 26 is the Deuce of Hearts
Card in position 27 is the Three of Hearts
Card in position 28 is the Four of Hearts
Card in position 29 is the Five of Hearts
```

## 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

We will all use 2016 to verify  
your code with mine

Generator  
initialized to seed

```
seed_seq seed{ 2016 };  
default_random_engine generator{ seed };  
uniform_int_distribution<int> distribution(0, 51);
```

Emits random  
integers using a  
**uniform** distribution

Sets low &  
high range,  
both inclusive

## Open Lab 2 – Bogus Card Dealer

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

```
17 void DealCards(vector<int>& deck)
18 {
19     for (auto& card : deck)
20         card = distribution(generator);
21 }
```

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

```
17 void DealCards(vector<int>& deck)
18 {
19     for (auto& card : deck)
20         card = distribution(generator);
21 }
```

Run Lab 2



```
dealer-bogus
File Edit View Terminal Tabs Help
Card in position 0 is the Eight of Hearts
Card in position 1 is the Jack of Hearts
Card in position 2 is the Jack of Spades
Card in position 3 is the Nine of Spades
Card in position 4 is the Ace of Clubs
Card in position 5 is the Nine of Diamonds
Card in position 6 is the Seven of Hearts
Card in position 7 is the Five of Spades
Card in position 8 is the Six of Spades
Card in position 9 is the Jack of Hearts
Card in position 10 is the Deuce of Spades
Card in position 11 is the Three of Clubs
Card in position 12 is the Deuce of Clubs
Card in position 13 is the Three of Clubs
Card in position 14 is the King of Clubs
Card in position 15 is the Deuce of Clubs
Card in position 16 is the Four of Hearts
Card in position 17 is the Deuce of Hearts
Card in position 18 is the Five of Diamonds
Card in position 19 is the Deuce of Diamonds
Card in position 20 is the Six of Diamonds
Card in position 21 is the Six of Diamonds
Card in position 22 is the Six of Diamonds
Card in position 23 is the Nine of Spades
Card in position 24 is the Deuce of Diamonds
Card in position 25 is the Queen of Spades
Card in position 26 is the Jack of Hearts
Card in position 27 is the Deuce of Hearts
Card in position 28 is the Four of Diamonds
Card in position 29 is the Jack of Spades
Card in position 30 is the Three of Diamonds
Card in position 31 is the Three of Spades
```

## Check Lab 2 - Bogus Card Dealer

# 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?



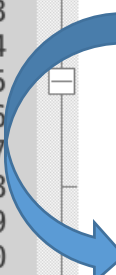
# Instrumenting Your Code

- Instrumenting code is the process of taking accurate **timings** of the runtime performance of key algorithms within the program
- C++ provides a **clock()** function that captures the current system time which is accurate to the nearest millisecond ( $1/1000^{\text{th}}$  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

```
47  int main()
48  {
49      vector<int> deck(52);
50
51      const int maxDeal{ 10000 };
52
53      clock_t startTime{ clock() };
54
55      for (int deal{}; deal < maxDeal; ++deal) {
56          InitDeck(deck);
57          DealCards(deck);
58      }
59
60      clock_t stopTime{ clock() };
61
62      DisplayCards(deck);
63
64      double totalTime{ ((double)(stopTime - startTime)
65                        / CLOCKS_PER_SEC) * 1000 };
66
67      cout.imbue(std::locale(""));
68      cout << "Total deals: " << maxDeal << endl;
69      cout << "Total run time (ms): " << totalTime << endl;
70
71      return 0;
72  }
```



## 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  that has not yet been dealt. We need a vector to record the fact that card # has been dealt

```
17 void DealCards(vector<int>& deck)
18 {
19     vector<bool> alreadyDealt(52, false);
20     for (auto& card : deck) {
21         card = distribution(generator);
22         while (alreadyDealt.at(card))
23             card = distribution(generator);
24         alreadyDealt.at(card) = true;
25     }
26 }
```

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

```
dealer-slow
File Edit View Terminal Tabs Help
Card in position 21 is the King of Spades
Card in position 22 is the Five of Diamonds
Card in position 23 is the Five of Hearts
Card in position 24 is the Three of Diamonds
Card in position 25 is the Three of Hearts
Card in position 26 is the Nine of Clubs
Card in position 27 is the Jack of Hearts
Card in position 28 is the King of Hearts
Card in position 29 is the King of Clubs
Card in position 30 is the Jack of Spades
Card in position 31 is the Seven of Diamonds
Card in position 32 is the Deuce of Clubs
Card in position 33 is the Ace of Spades
Card in position 34 is the Four of Clubs
Card in position 35 is the Four of Diamonds
Card in position 36 is the Eight of Hearts
Card in position 37 is the Deuce of Spades
Card in position 38 is the Ten of Clubs
Card in position 39 is the Deuce of Hearts
Card in position 40 is the Seven of Clubs
Card in position 41 is the Jack of Diamonds
Card in position 42 is the Nine of Diamonds
Card in position 43 is the Five of Clubs
Card in position 44 is the Four of Hearts
Card in position 45 is the Deuce of Diamonds
Card in position 46 is the Ten of Diamonds
Card in position 47 is the Jack of Clubs
Card in position 48 is the Ten of Hearts
Card in position 49 is the Nine of Spades
Card in position 50 is the Four of Spades
Card in position 51 is the Ace of Hearts
Total deals: 10,000
Total run time (ms): 337.001
```

## Check Lab 3 - Slow Card Dealer

337 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

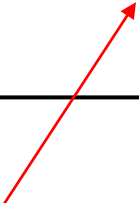
- Sadly there is an inherent inefficiency in the naïve algorithm employed in the **DealCards()** function from Lab 3
- It takes **longer and longer**, as more cards are dealt, to randomly pick (**i.e. to find**) a card that has not yet been dealt
- We need to discover an algorithm that, while ensuring 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 7<sup>th</sup> 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

```
dealer-fast
File Edit View Terminal Tabs Help
Card in position 21 is the Four of Clubs
Card in position 22 is the Four of Spades
Card in position 23 is the Queen of Hearts
Card in position 24 is the Ace of Spades
Card in position 25 is the Ten of Clubs
Card in position 26 is the Six of Spades
Card in position 27 is the Nine of Clubs
Card in position 28 is the Seven of Diamonds
Card in position 29 is the Eight of Hearts
Card in position 30 is the Jack of Hearts
Card in position 31 is the Jack of Spades
Card in position 32 is the Three of Hearts
Card in position 33 is the Nine of Spades
Card in position 34 is the Deuce of Spades
Card in position 35 is the Ten of Spades
Card in position 36 is the Queen of Diamonds
Card in position 37 is the Ace of Hearts
Card in position 38 is the Deuce of Hearts
Card in position 39 is the Six of Clubs
Card in position 40 is the Five of Hearts
Card in position 41 is the Five of Diamonds
Card in position 42 is the Deuce of Diamonds
Card in position 43 is the Seven of Clubs
Card in position 44 is the Three of Diamonds
Card in position 45 is the Eight of Spades
Card in position 46 is the Eight of Diamonds
Card in position 47 is the King of Diamonds
Card in position 48 is the Six of Hearts
Card in position 49 is the Seven of Hearts
Card in position 50 is the Six of Diamonds
Card in position 51 is the Ten of Hearts
Total deals: 10,000
Total run time (ms): 47.195
```

## Check Lab 4 - Fast Card Dealer

48 ms

**600% *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

```
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
- 600% faster
- Discovered by a 7<sup>th</sup> grader

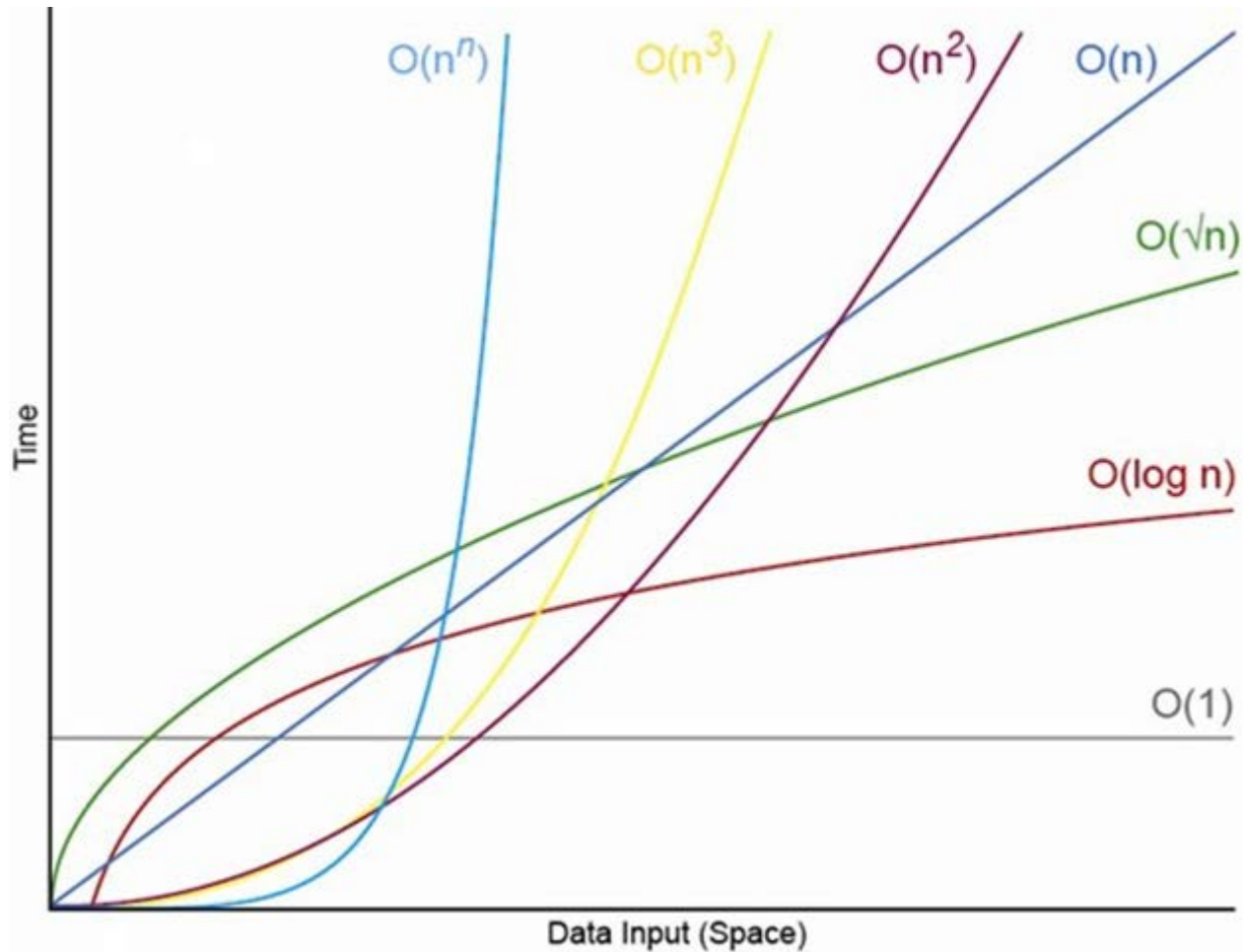
# Computing is a **New** Science

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

## 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 **system\_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

```
primalrace.cpp ✕
1 // primalrace.cpp
2
3 #include "stdafx.h"
4
5 using namespace std;
6 using namespace chrono;
7
8 seed_seq seed{ 2016 };
9 default_random_engine generator{ seed };
10 uniform_int_distribution<int> distribution(100000, 999999);
11
12 int CountPrimes(unique_ptr<vector<int>> const &samples)
13 {
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33 int main()
34 {
35     const auto samples{ make_unique<vector<int>>(100000) };
36
37     for (auto &sample : *samples)
38         sample = distribution(generator);
39
40     cout.imbue(std::locale(""));
41     cout << "Counting primes in vector of "
42          << samples->size() << " random integers..."
43          << endl;
44
45     auto startTime = system_clock::now();
46     int numPrimes = CountPrimes(samples);
47
48     auto stopTime = system_clock::now();
49
50     auto totalTime = duration_cast<milliseconds>(stopTime - startTime);
51
52     cout << "Number of Primes: " << numPrimes << endl;
53     cout << "Total run time (ms): " << totalTime.count() << endl;
54
55     return 0;
56 }
57
58
```

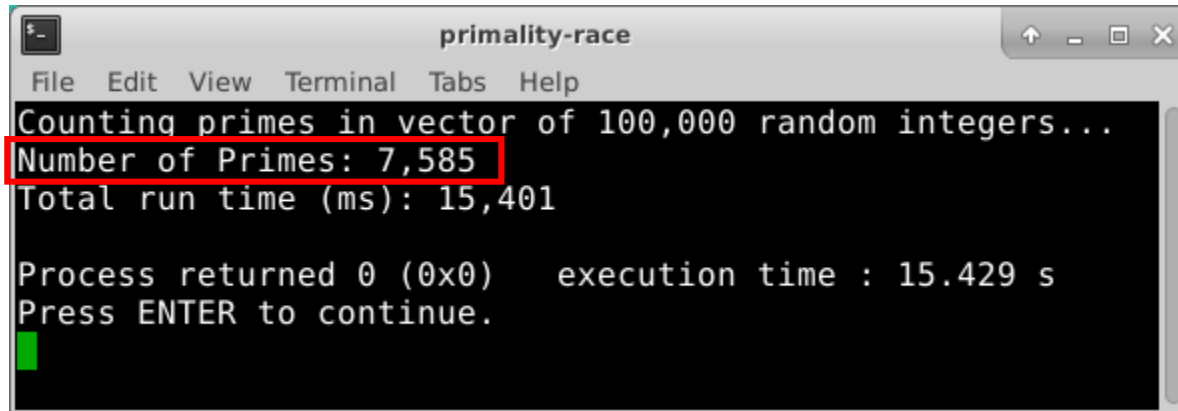
## View Lab 5 - Primality Race

```
12 int CountPrimes(unique_ptr<vector<int>> const &samples)
13 {
14     int numPrimes{};
15     for (const auto &sample : *samples) {
16         if (sample % 2 != 0) {
17             bool isPrime = true;
18             int n{ 2 };
19             while (n < sample) {
20                 if (sample % n == 0) {
21                     isPrime = false;
22                     break;
23                 }
24                 n++;
25             }
26             if (isPrime)
27                 numPrimes++;
28         }
29     }
30     return numPrimes;
31 }
32
```

Can we further optimize the **CountPrimes()** function?

Run Lab 5

# Check Lab 5 - Primality Race



```
primality-race
File Edit View Terminal Tabs Help
Counting primes in vector of 100,000 random integers...
Number of Primes: 7,585
Total run time (ms): 15,401

Process returned 0 (0x0)   execution time : 15.429 s
Press ENTER to continue.
█
```

- 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** = **system\_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](mailto:scicomp@bnl.gov)

```
primality-race
File Edit View Terminal Tabs Help
Counting primes in vector of 100,000 random integers...
Number of Primes: 7,585
Total run time (ms): 15,401

Process returned 0 (0x0)   execution time : 15.429 s
Press ENTER to continue.
```

v2

```
primality-race
File Edit View Terminal Tabs Help
Counting primes in vector of 100,000 random integers...
Number of Primes: 7,585
Total run time (ms): 7,785

Process returned 0 (0x0)   execution time : 7.811 s
Press ENTER to continue.
```

v3

```
primality-race
File Edit View Terminal Tabs Help
Counting primes in vector of 100,000 random integers...
Number of Primes: 7,585
Total run time (ms): 42

Process returned 0 (0x0)   execution time : 0.077 s
Press ENTER to continue.
```

1710% faster  
than v1

v4

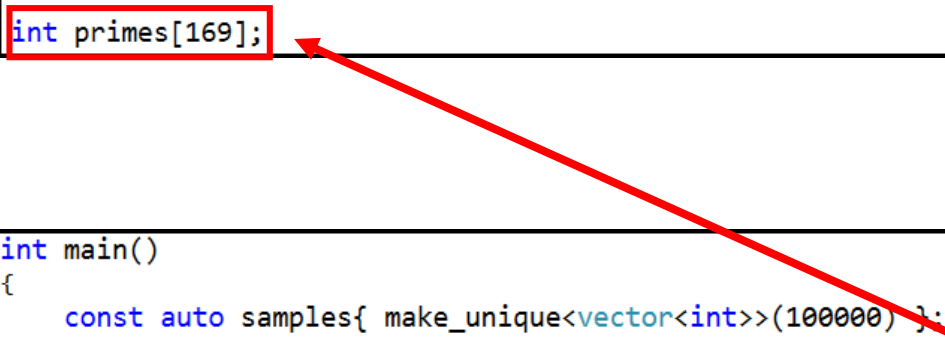
```
primality-race
File Edit View Terminal Tabs Help
Counting primes in vector of 100,000 random integers...
Number of Primes: 7,585
Total run time (ms): 9

Process returned 0 (0x0)   execution time : 0.038 s
Press ENTER to continue.
```

# Primality Race *Hints*

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

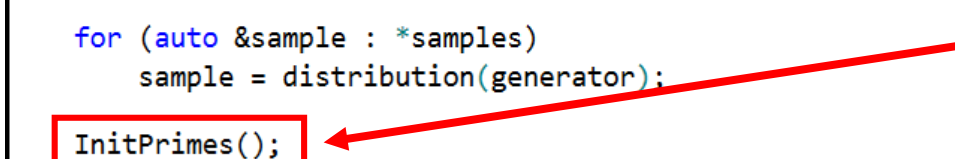
seed_seq seed{ 2016 };
default_random_engine generator{ seed };
uniform_int_distribution<int> distribution(100000, 999999);
int primes[169];
```



```
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  $p \in primes$
- 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*

```
void InitPrimes()
{
    int numPrimes{};
    int n{ 3 };
    while (numPrimes < 169) {
        if (n % 2 == 1) {
            bool isPrime = true;
            for (int p{}; p < numPrimes; ++p)
                if (n % primes[p] == 0) {
                    isPrime = false;
                    break;
                }
            if (isPrime) {
                primes[numPrimes] = n;
                numPrimes++;
            }
            n += 2;
        }
    }
}
```

- 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*

```
int CountPrimes(unique_ptr<vector<int>> const &samples)
{
    int numPrimes{};
    for (const auto &sample : *samples) {
        if (sample % 2 != 0) {
            int sqrtSample = sqrt(sample);
            bool isPrime = true;
            for (int n{}; primes[n] <= sqrtSample; ++n)
                if (sample % primes[n] == 0) {
                    isPrime = false;
                    break;
                }
            if (isPrime)
                numPrimes++;
        }
    }
    return numPrimes;
}
```

- We can immediately exclude all **even** samples from the count of primes
- We only need to trial divide primes up to  $\sqrt{\text{sample}}$
- 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