# Random Real Numbers

http://people.sc.fsu.edu/~jburkardt/isc/week06_lecture_11.pdf

.....

ISC3313:
Introduction to Scientific Computing with C++
Summer Semester 2011

.....

John Burkardt
Department of Scientific Computing
Florida State University


Last Modified: 29 July 2011

# Functions

- **Introduction**
- Data Conversion
- Function Parameters
- Random Real Numbers
- Approximating an Area
- Lab Exercise #6

# INTRO: Schedule

Read:

- Today's class covers information in Sections 3.9 (data conversion), 5.4, 5.5, 5.6, 5.8

Next Class:

- Random Integers

Assignment:

- Today: in-class lab exercise #6
- Thursday: Programming Assignment #4 is due.

# INTRO: Overview

We start today by trying to understand the rules about data types, how C++ tries to be flexible when looking at the numbers and variables we use, and how, if necessary, we can insist that certain data be treated as a different (more accurate) data type temporarily.

I will remind you about our discussion of user functions, and how they are declared "locally", or can be made available to all parts of the program using "file scope". We will also consider functions with no input, or no output.

# INTRO: Overview

Then I will talk a little about the C++ **rand()** function, which produces random integers, and how we can produce a corresponding random real number between 0 and 1.

We will then look at how random real numbers can be used for sampling. Sampling can be used to estimate the value of an integral, or, more interestingly, the area inside a 2D curve.

Our classroom exercise will require you to estimate the area of a somewhat complicated crescent shape.

# Functions

- Introduction
- **Data Conversion**
- Function Parameters
- Random Real Numbers
- Approximating an Area
- Lab Exercise #6

# CONVERSION: Changing Data Types

We have seen that C++ has a separate type for integers, two types for real numbers, and a type for characters. We will soon see types for logical variables and for text strings.

It makes sense to declare a variable's type, so that C++ can warn us if we accidentally try to take the square root of the letter **'B'**.

We know that C++ built in libraries, such as the math functions in <**cmath**>, provide a declaration that indicates what kind of data it expects as input, and what it will produce as output:

```
double sqrt ( double x );
```

To what extent must we follow these rules? How much leeway does C++ give us to bend them? What do we do if the type of our data interferes with the computation we want to carry out?

# CONVERSION: Changing Data Types

Let's concentrate on the differences between **int**, **float** and **double** data types. And let's start with the question of assignments in which a variable is given a literal value.

We know that a literal with a decimal point, such as 2.0 or -17.45, is some kind of real value. Other numeric literals, such as 1 or -88, are integers. Does that mean we have to always spell out exactly the kind of literal we are using in an assignment?

Fortunately, C++ is very sensible about this issue. In an assignment statement, it compares the type of the numeric value on the right hand side with the type of the destination on the left-hand side, and makes the necessary adjustments.

# DATA TYPES: Conversion

```
int i;
float x;
double d;

i = 1;          <-- int variable gets integer value..,.OK!
x = 2.0;        <-- float variable gets real value...OK!
x = 2;          <-- float variable gets integer value
x = i + 7;      <-- float variable gets int + integer value
i = 3.7;        <-- 3.7 is truncated to 3
i = 3.7 + 5.6;  <-- What will be stored in i?
i = x + 2.1     <-- int variable gets float + real value
d = 10;         <-- double variable set to integer
```

# DATA TYPES: Conversion

Assignment statements have the form

```
lvalue = rvalue;
```

where **rvalue** is a literal, a variable, or a formula and **lvalue** is a variable that will hold the result.

To carry out the assignment, C++ must first determine the value of the right hand side.

Since the right hand side is not guaranteed to be a variable whose type would be a clue, C++ must "guess" whether to use integer, float, or double arithmetic. It does this based on the type of the arguments that it is combining. But if it is combining data of two different types, it moves to the more precise arithmetic, that is, it promotes the calculation to the higher precision. It is just trying to make sure it gets you an answer as accurate as you want.

# DATA TYPES: Conversion

Once the right hand side has a value, C++ then looks at the left hand side. If the left hand side does not have the same type as the result, C++ adjusts the result to "fit" into the left hand side. It truncates a **float** result that must be store in an **int**; it drops the extra decimal places in a **double** that has to fit into a **float**. It adds blank decimal places when an **int** value goes into a **float** or **double**, and so on.

This should help when considering formulas like:

```
i = 3.7 + 5.6;    <--the right hand side is 9.1, and i is 9
x = 1 / 10;       <--the right hand side is 0, and x is 0.0
y = 1 / 10.0;     <--the right hand side is 0.1, and y is
i = 1 / 10.0;     <--the right hand side is 0.1, but i is
```

# DATA TYPES: Function Calls

C++ also makes things simpler for you when you call a function. The **sqrt()** function, like all the **cmath** functions, expects a **double** value as its input. However, the following are all legal:

```
int i = 4, j;
float x = 9.0, y;
double d = 16.0, e;

j = sqrt ( x );  <-- input float, output stored in int
y = sqrt ( d );  <-- input double, output stored in float
e = sqrt ( i );  <-- input int, output stored in double
i = sqrt ( 17 ); <-- input int, output stored in int
```

C++ automatically "promotes" the input to a **double**. The computed result is also a **double**, but C++ adjusts it if necessary to fit in the variable where the result is to be stored.

# DATA TYPES: Explicit Conversion

In C++, conversion is given the unusual name of performing a cast. You might here a programmer say something like "I divided the number of days by 30 and **cast** the result to an **int**."

You can actually force a number to be converted to any type for which the conversion makes sense. You do this using the name of the desired type in parentheses:

```
int i;
float x;
i = ( int ) 3.7;   <-- convert 3.7 to an int, then assign to i.
i = ( int ) x;     <-- convert value of x to int, assign to i.
x = ( float ) 17   <-- convert 17 to a float, then assign to x.
d = ( double ) i; <-- convert value of i to double, assig  d.
```

The textbook shows a more modern way, using **static_cast**. Ignore that information for now!

# DATA TYPES: Rules For Evaluating a Fraction

Because C++ works hard to make things simple for you, you usually do not have to worry about converting your data types in a computation. One common exception, though, occurs when division is involved. As we already know, C++ looks at the types of the top and bottom of the fraction being computed. If both are integers, then it computes the result as an integer division problem (with no remainder.) It doesn't matter if the result is to be stored in a real variable.

```
int i;
float x;
x = 1 / 100;    <-- Integer arithmetic, result 0, st
i = 3.75 / 1.5; <-- Real arithmetic, result 2.5,
```

# DATA TYPES: Rules For Evaluating a Fraction

To force integers to divide like reals, we could always copy the data into real variables; a better way converts the data as we use it.

```
int m = 10, n = 25;
double d, m_double, n_double;

d = m / n;    <-- Not what we want!

m_double = m;
n_double = n;
d = m_double / n_double;  <-- Works, but requires
                              extra variables;

d = ( double ) m / ( double ) n;  <-- Short and s   et.
```

# Functions

- Introduction
- Data Conversion
- **Function Parameters**
- Random Real Numbers
- Approximating an Area
- Lab Exercise #6

# Functions

Last time, we introduced the idea of user-defined functions. We saw that the main program could call such a function, and that one user function could call another one. However, I explained that in order for a function to be called, it must be declared, that is, that its name, output type, and type of input arguments be listed.

So a typical declaration might look like

```
double area ( double width, double height )
```

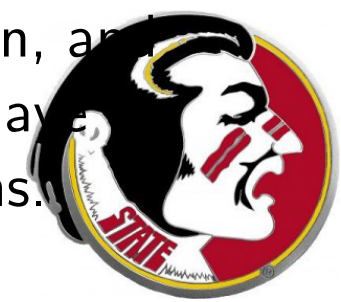
and this declaration must be inserted into any function that wants to use the **area()** function.

# FUNCTIONS: File Scope

Now I want to remind you that, in the examples I showed, this declaration occurred in the calling function, and along with the usual declarations of variables. That's fine. Such a declaration has local scope, that is, the **area()** function is now available, but only to the function within which you placed the declaration.

However, it is common practice to place such declarations at the very beginning of the file, after the usual include and using statements, but before the main function has started.

Where you place the declaration matters. If you place the declaration at the beginning, it has file scope - that is, all the functions within the file essentially have seen the declaration, and none of them have to repeat it. This can be handy if you have a function that is going to be used in many different functions.



# FUNCTIONS: File Scope Declaration for Entire File

```
# include <cstdlib>
using namespace std;
double area ( double width, double height );  <-- Declaration of area() for entire file

int main ( )                                  <-- main can use area()
{
...
}
double area ( double width, double height )   <-- The text of area()
{
...
}
int rectangle_count ( int a, int b, double c ) <-- rectangle_count can use area()
{
  double intersect ( double tolerance );       <-- Only rectangle_count can use intersect.
...
}
double intersect ( double tolerance )          <-- intersect can use area()
{
...
}
```

# FUNCTIONS: No Input?

Another thing we should think about is how to declare a function if it doesn't have any input, or doesn't have any output. First of all, is such a thing even possible?

Here are examples of useful functions with no input:

- **main()**, all our main functions have (so far) had no input!
- **clock ( )** returns number of clock ticks since program start (#include <ctime>);
- **pi()**, a user function which would return the value of $\pi$;
- **timestamp()**, a user function which prints the current date.

Such functions are declared in the usual way, but the input parentheses are empty.

- **int clock ( );**
- **double pi();**
- **void timestamp();**

# FUNCTIONS: No Output?

Many functions might take input, but don't return an output value. In such a case, the data type of the function must be listed as **void**. Within the function, the return statement should not have a value associated with it, that is, it should just read "**return;**".

Here is an example of a void function. It prints the ratio of its input, but has no value to return.

```
void print_ratio ( double top, double bot )
{
  if ( bot == 0.0 )
  {
    cout << "Bottom is 0, fraction is illegal.\n";
    return;
  }
  cout << "Ratio is " << top / bot << "\n";
  return;
}
```

# FUNCTIONS: Promotion of Input Arguments

Another thing to keep in mind is that C++ tries to make your life easier by recognizing when your input is not exactly of the right type, but could easily be adjusted to make sense.

We mentioned that the declaration of the **sqrt()** function assumes that its input is a **double**. However, C++ will allow you to pass in a **float** or an **int**, and it will do the right thing. By the way, this is *not* true when using the earlier language C!

This means that, although we have declared the **area()** function as:

```
double area ( double width, double height )
```

it is perfectly legal to write **area ( 4, 3 )**, for instance, for wh the answer 12.0 will be returned.

# Functions

- Introduction
- Data Conversion
- Function Parameters
- **Random Real Numbers**
- Approximating an Area
- Lab Exercise #6

# RANDOM: A Review

In order to show you examples with coin tosses, I have already had to sneak in some random number calculations without explaining much of what was going on.

You should have noticed that I called an **srand()** function once to set up or scramble the random number generator, and that I then called the **rand()** function repeatedly.

Note that both **srand()** and **rand()** are declared by the statement **#include** <**cstdlib**>, the "C++ standard library".

I told you that we could use the output of the **rand()** function to simulate a coin toss, based on whether it was even or odd. Today we will learn more about random numbers, and how to use the C++ functions to generate random samples in a variety of cases.

# RANDOM: Why Random Numbers?

The C++ function **rand()** returns a "random" **int** between 0 and **RAND_MAX**, which is typically the very large number 2,147,483,647.

We must figure out how to take random integers and use them to introduce randomness into a calculation. What we will do depends on whether our problem is *discrete* (only a few possibilities to choose from) or *continuous* (a smooth range of possibilities).

Thus, the kind of random number we actually need is either:

- a random integer between 1 and **n**;
- a random real number between **a** and **b**;

Today, we will concentrate on determining a random real value.

# RANDOM: A Random Real Number

Our first attempt doesn't seem to work:

```
value = rand ( ) / RAND_MAX;   <-- Oops!  Wrong!
```

This formula is 0.0 every time (with one exception, which is...?)

I need to convert the numbers on top and bottom into **double**s.

The way to make a copy of an **int** that works like a real number is to use the **( double )** operator. You must use parentheses around the word **double** for this to work!

```
value = ( double ) rand ( )
      / ( double ) RAND_MAX;
```

We can store the final result in a **float** or a **double**.

# RANDOM: A Random Real Number

So let's write a function **random_double.cpp** that computes a random double:

```
double random_double ( )
{
  double value;
  value = ( double ) rand ( ) / ( double ) RAND_MAX;
  return value;
}
```

This function can be used whenever we need a random **float** or **double** in the range $0.0 \leq x \leq 1.0$.

# RANDOM: "Stretching" a Random Real Number

Of course, now we wonder what to do if, instead, we want random reals between values **a** and **b**.

If **r** is a number between 0 and 1, then we can compute a corresponding number **s** between **a** and **b** by this formula:

$$s = a + (b - a) * r;$$

Can you see why this formula will work? It's a linear function, so we only have to check that it's right twice, and we've got it. So what happens when **r** is 0? When **r** is 1?

# RANDOM: A Function for Reals between A and B

The function **random_double_ab.cpp** produces a random real in the range **a** to **b**.

```
double random_double_ab ( double a, double b )
{
  double r;
  double value;

  r = ( double ) rand ( ) / ( double ) RAND_MAX;

  value = a + ( b - a ) * r;

  return value;
}
```

# Functions

- Introduction
- Function Parameters
- Integer Arithmetic
- Random Real Numbers
- **Approximating an Area**
- Lab Exercise #6

# AREA: Integral Approximation by Averaging

Remember one of our early homework problems, in which we approximated an integral by choosing $n$ equally spaced points $x_i$ over the interval $[a, b]$, evaluating the function $f(x)$, and writing

$$\int_a^b f(x)\,dx \approx dx * \sum_{i=1}^{n} f(x_i)$$

Since $dx = \frac{(b-a)}{n}$ we can write this as

$$\begin{aligned}\int_a^b f(x)\,dx &\approx (b-a) * \frac{1}{n}\sum_{i=1}^{n} f(x_i) \\ &= (b-a) * \text{average}\{f(x_1), f(x_2), ..., f(x_n)\}\end{aligned}$$

which gives us a different view of approximating an integral.

# AREA: Integral Approximation by Averaging

In other words, instead of worrying about carefully spacing my points through the interval, it is enough to have them randomly scattered.

$$\int_a^b f(x)\,dx \approx (b-a) * \text{average}\{f \text{ at } n \text{ random points}\}$$

And what's better is that if I decide my current estimate with $n$ points isn't good enough, I can compute more points, and update my average, rather than throwing away the old results.

# AREA: Integral Approximation by Averaging

**integral.cpp** uses sampling for Homework Program #1:

```
beginning stuff
int main ( )
{
  int i, n = 10000;
  double a = - 4.0, b = 5.0, sum = 0.0, x;
  double f ( double x );                          <-- function to integrate
  double random_double_ab ( double a, double b ); <-- our random real function

  for ( i = 1; i <= n; i++ )
  {
    x = random_double_ab ( a, b );
    sum = sum + f ( x );
  }
  cout << "Integral estimate using " << n << " points is "
       << ( b - a ) * sum / ( double ) n << "\n";
  return 0;
}
double f ( double x )
{
  double value;
  value = x * x + 2 * x - 3;
  return value;
}
double random_double_ab ( double a, double b )
{
  double r;
  double value;
  r = ( double ) rand ( ) / ( double ) RAND_MAX;
  value = a + ( b - a ) * r;
  return value;
}
```

# AREA: Approximating an Area

Here is a more challenging problem which suggests how powerful random sampling can be, especially if we don't have any exact method for getting an answer.

Suppose I have a two dimensional region $R$, and I have some rule that lets me know whether any given point $(x, y)$ is inside or outside the region. What is the area of the region?

Seems somewhat impossible to answer.

Suppose I surround the region with a rectangular box $B$ defined by $a \leq x \leq b$, $c \leq y \leq d$. Then, for one thing, I know the area of the region can't be greater than the area of the box, that is,

$$\text{Area}(R) \leq \text{Area}(B) = (b - a) * (d - c)$$

# AREA: Approximating an Area

Now suppose I can compute random points $(x, y)$ in B. I can do this in C++ by

```
x = random_double_ab ( a, b );
y = random_double_ab ( c, d );
```

I was told I have a formula that lets me know if (x,y) is actually inside $R$. Let's say the formula is $4x^2 + 9y^2 \leq 36$. Then I can count the sample points inside the region by

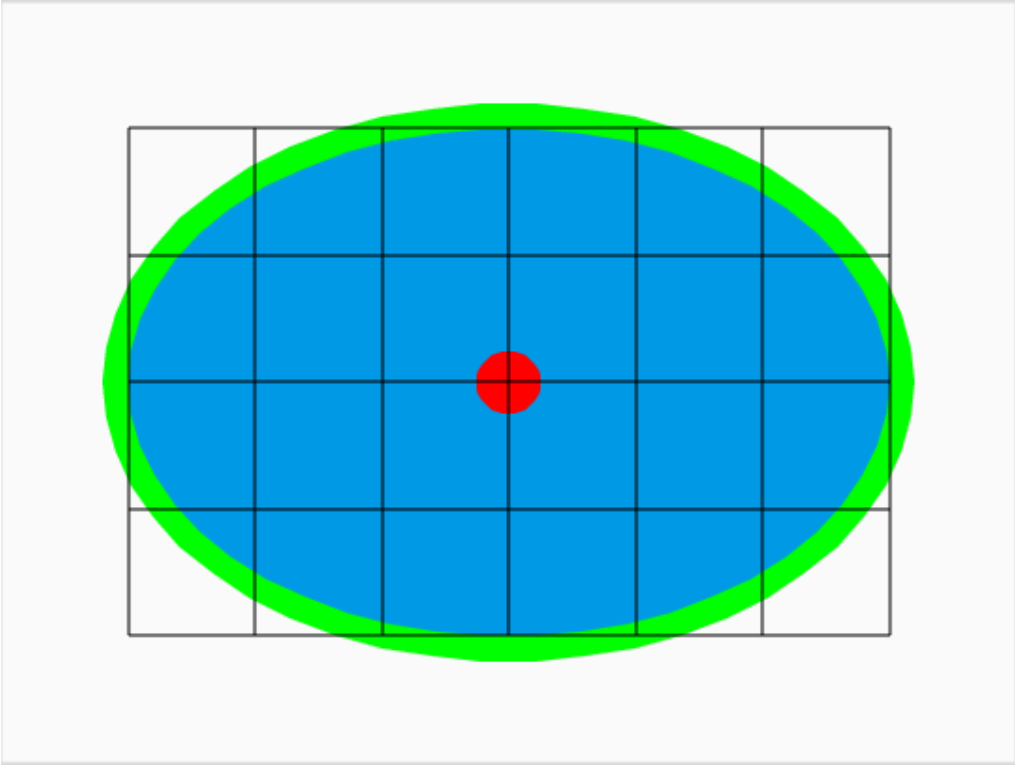
```
if ( 4*x^2 + 9*y^2 <=36 )
{
  m = m + 1;
}
```

If I used $n$ sample points total, then we can estimate

$$\text{Area}(R) \approx \frac{m}{n} * \text{Area}(B) = \frac{m}{n} * (b - a) * (d - c)$$

# AREA: A Logical Diagram of the Pool

The rectangle is $-3 \leq x \leq +3, -2 \leq y \leq +2$, with area 24. The pool is the blue ellipse.



# AREA: Estimating the Area in C++ Using Sampling

I would like volunteers to rewrite this "pseudocode" into C++! There are enough lines to be rewritten that I think everyone will get a chance.

```
beginning stuff

int main ( )
{
  declarations

  Get value of n from user

  Initialize data.

  Do this n times:
  {
    generate random x and y values;

    if ( (x,y) is inside ellipse )
    {
      update the count
    }

  }
  Estimate the area of the ellipse.

  Print estimate of area
}
What useful function should appear here?
```

**area_estimate.cpp**

# Functions

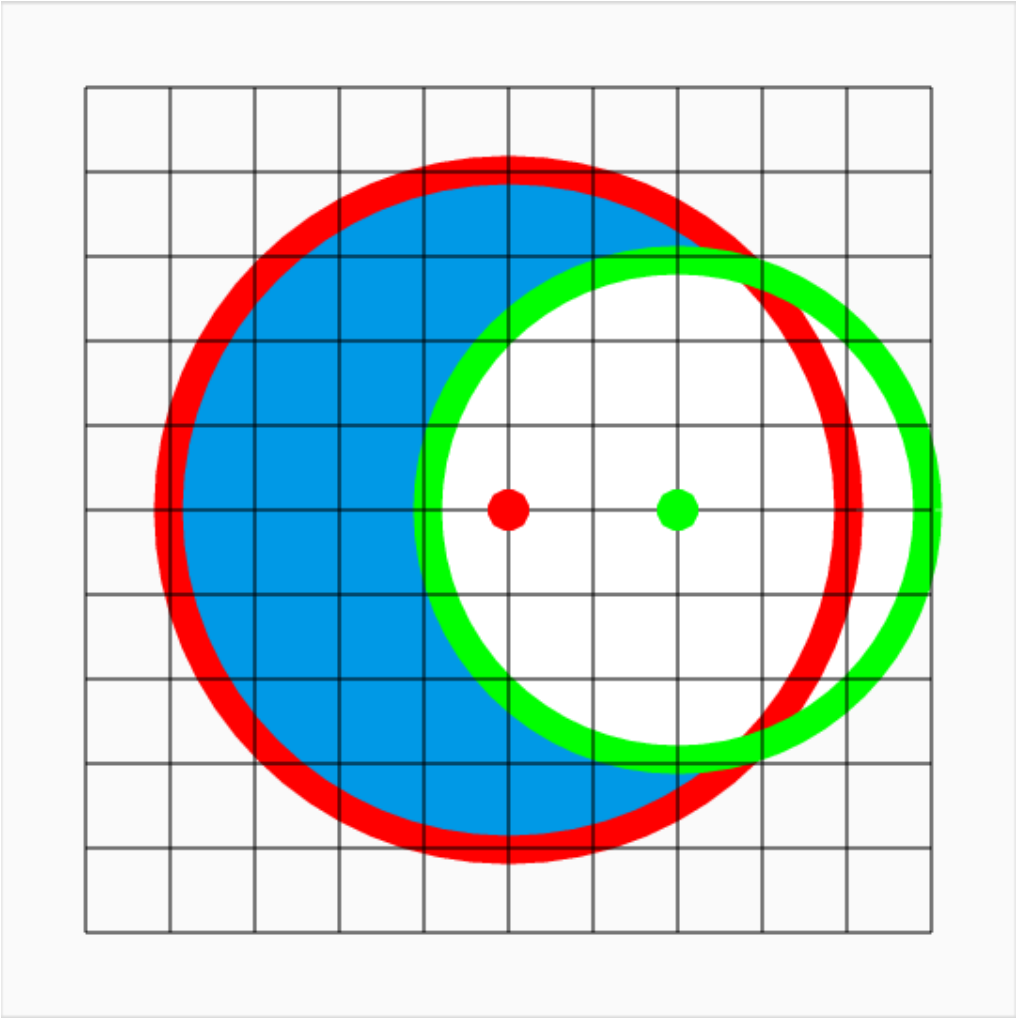
- Introduction
- Function Parameters
- Integer Arithmetic
- Random Real Numbers
- Approximating an Area
- **Lab Exercise #6**

# EXERCISE: Estimating an Area

- We are going to estimate the area of a crescent-shaped pool.
- We are going to do this by random sampling.
- We surround the pool with a rectangle whose area we know.
- We will generate random points $(x, y)$ and count the ones which fall inside the pool.
- The proportion of "hits" allows us to estimate the area.
- Each time you want a random point, you will need to call **random_double_ab()** twice, once for an $x$ and once for $y$.

# EXERCISE: A Logical Diagram of the Pool

The pool is the blue area, inside the red circle, AND not inside the green circle!



# EXERCISE: A Formula for the Pool

The red circle has center (0,0) and radius 4. A point $(x, y)$ is inside the red circle if:

$$x^2 + y^2 \leq 16$$

The green circle has center (2,0) and radius 3. A point $(x, y)$ is inside the green circle if

$$(x - 2)^2 + y^2 \leq 9$$

We are looking for points that are inside the red circle, but not in the green circle. The point (0,3), for example, is in the red circle but not in the green circle, hence it's in the pool.

# EXERCISE: Estimate the Pool Area

Write a program to estimate the area of the pool.

- the rectangle is $-5 \leq x \leq +5, -4 \leq y \leq +4$;
- determine the area **B** of this rectangular box;
- compute **n**=10,000 random points $(x, y)$ inside the box;
- Let **m** count those points which are also inside the pool;
- estimate the pool area **P** by $P \approx \frac{m}{n} * B$.

Once you have computed this value, please show your work to Detelina so you can get credit for the exercise!