Using Failed Input for Processing

- Using a `bool` variable in this way is disliked by many programmers.

  Why?

- `cin.fail` is set when >> fails.
  This allows the use of an input *itself* to be used as the test for failure.

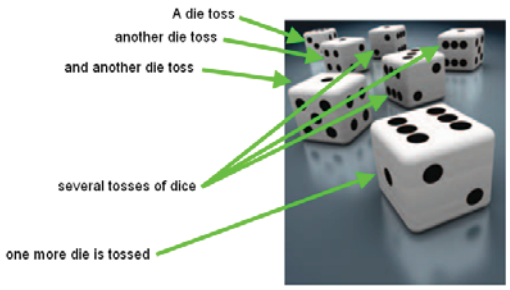
- Again note that if you intend to take more input from the keyboard, you must reset the keyboard with `cin.clear`.

Using Failed Input for Processing

Using the input attempt directly we have:

```
cout << "Enter values, Q to quit: ";
while (cin >> value)
{
   // process value here
}
cin.clear();
```


Random Numbers and Simulations (4.6)



was that an English lesson?

Simulations

A *simulation program* uses the computer to simulate an activity in the real world (or in an imaginary one).

Simulations

- Simulations are commonly used for
  - Predicting climate change
  - Analyzing traffic
  - Picking stocks
  - Many other applications in science and business

Randomness for Reality (Simulating)

- Programmers must model the "real world" at times.

EX: Consider the problem of modeling customers arriving at a store.

Do we know the rate?

Does anyone?

How about the shopkeeper!

Randomness for Reality (Simulating)

To accurately model customer traffic, you want to take that random fluctuation into account.

How?

The rand Function

The C++ library has a random number generator:

```
rand()
```


The rand Function

`rand` is defined in the `cstdlib` header

Calling `rand` yields a random integer between 0 and `RAND_MAX`

(The value of `RAND_MAX` is implementation dependent)

The rand Function

Calling `rand` again yields a different random integer

Very, very, very rarely it might be the same random integer again.

(That's OK. In the real world this happens.)

The rand Function

`rand` picks from a very long sequence of numbers that don't repeat for a long time.

But they do eventually repeat.

These sorts of "random" numbers are often called *pseudorandom numbers*.

The rand Function

`rand` uses only one pseudorandom number sequence

and it always starts from the same place.

Oh dear

The rand Function

When you run your program again on another day, the call to `rand` will start with:

*the **same** random number!*

Is it very "real world" to use the same sequence over and over?

No, but it's really nice for testing purposes.

but...

Seeding the rand Function

You can "seed" the random generator to indicate where it should start in the pseudorandom sequence

Calling `srand` sets where `rand` starts

`srand` is defined in the `cstdlib` header

Seeding the rand Function

But what value would be different every *time* you run your program?

(hint)

How about the time?

Seeding the rand Function

You can obtain the system time.

Calling `time(0)` gets the current time

Note the zero. It is required.

`time` is defined in the `ctime` header

```
#include <ctime>
```


Seeding the rand Function

Calling `srand` sets where `rand` starts.

Calling `time(0)` gets the current time.

So, to set up for "really, really random" random numbers on each program run:

```
srand(time(0)); // seed rand()
```

sets the seed to the current time

(Well, as "really random" as we can hope for.)

Modeling Using the rand Function

EX:

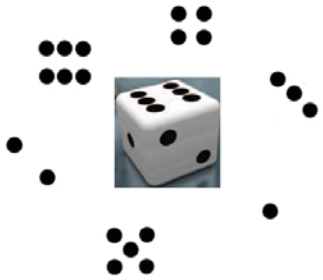
Let's model a pair of dice,

Modeling Using the `rand` Function

one die at a time.

Modeling Using the `rand` Function

What are the numbers on one die?

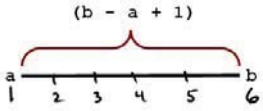


Modeling Using the `rand` Function

What are the bounds of the range of numbers on one die?
1 and 6 (inclusive)

We want a value randomly between those endpoints (inclusively)

Modeling Using the `rand` Function

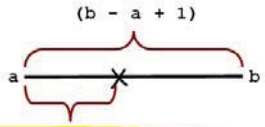


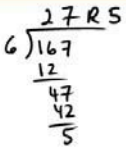
Given two endpoints,
a and b, recall there are

`(b - a + 1)`

values between a and b,
(including the bounds themselves).

Modeling Using the `rand` Function





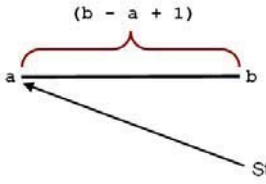
`rand() % (b - a + 1)`

167 % 6 = 5

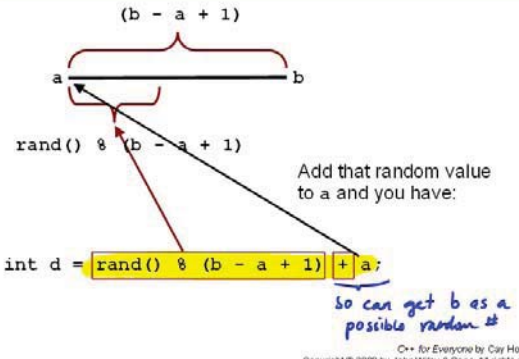
↑ remainder operator

Obtain a random value
between 0 and `b - a`
by using the `rand()` function

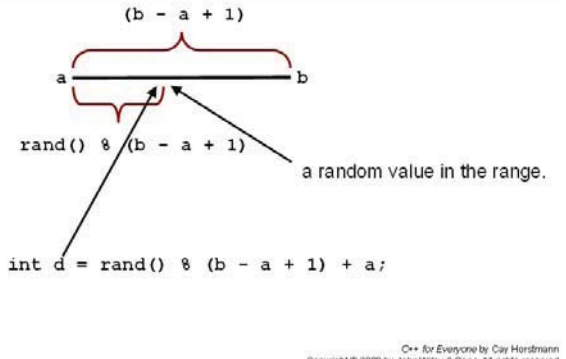
Modeling Using the `rand` Function



Modeling Using the rand Function



Modeling Using the rand Function



Modeling Using the rand Function

int d = rand() % (6-1+1) + 1
b-a+1 +a

= rand % 6 + 1

Using 1 and 6 as the bounds
and
modeling for two dice,
running for 10 tries,
we have:

Modeling Using the rand Function

ch04/dice.cpp

```
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
   srand(time(0)); ← set seed
   int i;
   for (i = 1; i <= 10; i++)
   {
      int d1 = rand() % 6 + 1;
      int d2 = rand() % 6 + 1;
      cout << d1 << " " << d2 << endl;
   }
   cout << endl;
   return 0;
}
```

One of many different Program Runs:

```
5 1
2 1
1 2
5 1
1 2
6 4
4 4
6 1
6 3
5 2
```


The Monte Carlo Method

EX:

The premier gaming *"table d'darts"*
at one of the less well known casinos in Monte Carlo,
somewhat close but not quite next door to Le Grand Casino.

The Monte Carlo Method

The Monte Carlo method is a method for finding approximate solutions to problems that cannot be precisely solved.

Here is an example: compute π

This is difficult.

The Monte Carlo Method

While we are in this fine casino,
we should at least play one game at the "*table d'darts*"

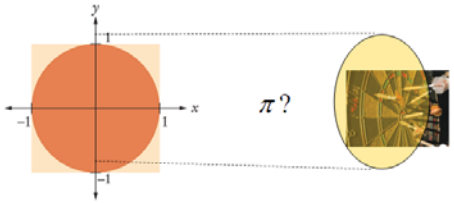
The Monte Carlo Method

THAT'S IT!

By shooting darts (and a little math)
we can obtain an approximation for π.

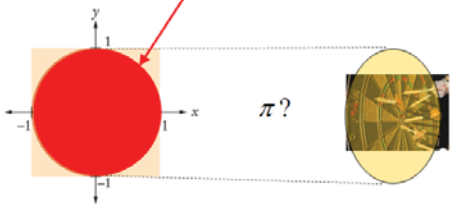
The Monte Carlo Method

Consider placing the round dartboard
inside an exactly fitting square



The Monte Carlo Method

As we toss darts at the target,
if we are able to just *hit* the target – at all – it's a hit.

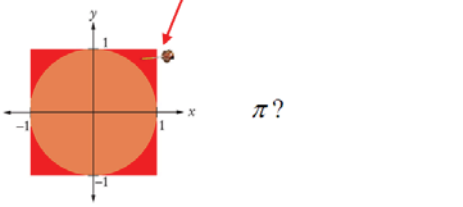


(no wonder this is such a patheric casino)

The Monte Carlo Method

and a *miss* is a miss.

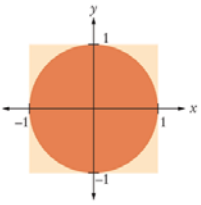
if hit outside circle



The Monte Carlo Method

The `(x,y)` coordinate of a *hit* is when $(x^2 + y^2) \leq 1$.
In code:

```
if (x * x + y * y <= 1) { hits++; }
```



$x^2+y^2=1$

The Monte Carlo Method

Our coded random shots will give a ratio of
hits/tries
that is approximately equal to the ratio of
the areas of the circle and the square:
$\pi/4$

$\frac{hits}{tries} \approx \frac{\pi}{4}$

$\frac{\pi r^2}{s^2} = \frac{\pi 1^2}{2^2} = \frac{\pi}{4}$

$\therefore \pi \approx 4. \times \frac{hits}{tries}$

The Monte Carlo Method

Multiply by 4 and we have an estimate for π!

```
π = 4 * hits/tries;
```

The longer we run our program,
the more random numbers,
the better the estimate.

The Monte Carlo Method

−1 to 1
a b

For the `x` and `y` coordinates within the circle,
we need random `x` and `y` values between `-1` and `1`.

That's a range of $(+1 - (-1)) + 1$ or `2`.
b − a

As before, we want add some random portion
of this range to the low endpoint, `-1`.

```
rand % (b-a+1) + a
```

But we will want a floating point value, not an integer.

The Monte Carlo Method

We must use `rand` with `double` values
to obtain that random portion.

integer real #

```
double r = rand() * 1.0 / RAND_MAX;
```

real!

The value `r` is a random floating-point
value between 0 and 1.

You can think of this as a percentage if you like.

(Use `1.0` to make the `/` operator not do integer division)

The Monte Carlo Method

The computation:

between 0 and 1

```
double x = -1 + 2 * r;
```

2 is the length of the range from `-1` to `1`

The Monte Carlo Method

The computation:

```
double x = -1 + 2 * r;
```

2 is the length of the range from `-1` to `1`

`r` is some random value between `0.0` and `1.0`

The Monte Carlo Method

The computation:

```
double x = -1 + 2 * r;
```

2 is the length of the range from -1 to 1

r is some random value between 0.0 and 1.0

so (2 * r) is some portion of that range

The Monte Carlo Method

The computation:

```
double x = -1 + 2 * r;
```

2 is the length of the range from -1 to 1

r is some random value between 0.0 and 1.0

so (2 * r) is some portion of that range

We will add this portion to the left hand end of the range, -1

The Monte Carlo Method

The computation:

```
double x = -1 + 2 * r;
```

2 is the length of the range from -1 to 1

r is some random value between 0.0 and 1.0

so (2 * r) is some portion of that range

Adding this portion to the left hand end of the range gives us:

x randomly within the range -1 and 1.

The Monte Carlo Method for Approximating PI

ch04/montecarlo.cpp

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <ctime>
using namespace std;
int main()
{
   const int TRIES = 10000;
   srand(time(0));
   int hits = 0;
   for (int i = 1; i <= TRIES; i++)
   {
      double r = rand() * 1.0 / RAND_MAX; // Between 0 and 1
      double x = -1 + 2 * r; // Between -1 and 1
      r = rand() * 1.0 / RAND_MAX; ← need again to get a different random r between 0 and 1
      double y = -1 + 2 * r;
      if (x * x + y * y <= 1) { hits++; }
   }
   double pi_estimate = 4.0 * hits / TRIES;
   cout << "Estimate for pi: "
      << pi_estimate << endl;
   return 0;
}
```


Chapter Summary

1. Loops execute a block of code repeatedly while a condition remains true.

2. An off-by-one error is a common error when programming loops. Think through simple test cases to avoid this type of error.

3. The `for` loop is used when a value runs from a starting point to an ending point with a constant increment or decrement.
4. The `do` loop is appropriate when the loop body must be executed at least once.
5. Nested loops are commonly used for processing tabular structures.

6. A sentinel value denotes the end of a data set, but it is not part of the data.

7. You can use a Boolean variable to control a loop. Set the variable to `true` before entering the loop, then set it to `false` to leave the loop.
8. In a simulation program, you use the computer to simulate an activity. You can introduce randomness by calling the random number generator.

Chp. 4 Homework

R 4.1 (a - c)

R 4.21

P 4.1

P 4.15

due March 24th