Math and Random Numbers

2025 Winter APS105: Computer Fundamentals
Jon Eyolfson

Lecture 5
1.1.1

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

C Also Comes with A Math Library

For more math more advanced than arithmetic we need to use this library

To add all the function prototypes from the library to our code we need to add

```
  #include <math.h>
```

on a new line, before creating the `main` function

A function prototype tells you the inputs and output of a function
(a function declaration is almost the same, don't worry about the difference)

A Function Prototype Tells You the Input and Output Types

All function prototypes have the form:

```
  <output_type> <function_name>(<input_types>)
```

`void` is a special type that means “nothing”

There can be multiple `<input_types>` separated by commas

- `<input_types>` can be `void` meaning there are no inputs

Some Examples of Function Prototypes

```
/* This means there's a function called "foo" that takes a single double as
   input and outputs a double. */
double foo(double);
```

Some Examples of Function Prototypes

```
/* This means there's a function called "foo" that takes a single double as
   input and outputs a double. */
double foo(double);

/* This means there's a function called "foo" that takes two doubles as
   inputs and outputs a double. */
double foo(double, double);
```

Some Examples of Function Prototypes

```
/* This means there's a function called "foo" that takes a single double as
   input and outputs a double. */
double foo(double);

/* This means there's a function called "foo" that takes two doubles as
   inputs and outputs a double. */
double foo(double, double);

/* This means there's a function called "foo" that takes a single double as
   input and has no output. */
void foo(double);
```

Some Examples of Function Prototypes

```c
/* This means there's a function called "foo" that takes a single double as
   input and outputs a double. */
double foo(double);

/* This means there's a function called "foo" that takes two doubles as
   inputs and outputs a double. */
double foo(double, double);

/* This means there's a function called "foo" that takes a single double as
   input and has no output. */
void foo(double);

/* This means there's a function called "foo" that has no inputs and outputs a
   double. */
double foo(void);
```

Function Prototypes Can Optionally Name Arguments

For example, you could write:

```
double foo(double x, double y);
```

The names are just to help you understand which argument is which
- The order matters!

When you use the functions, your variable names do not have to match
- You could also just use values (the compiler calls them literals)

There is a C Preprocessor That Runs Before Compiling

This is not testable, but will help you understand what's happening

Anything you write that starts with a `#` is called a preprocessor directive
(in other words it tells the preprocessor what to do)

The `#include` directive will replace itself with the contents of the specified file

Header Files Contain Function Prototypes of Functions

By convention any file that ends with `.h` we call a header file

A library will come with one or more header files
- The header files contain function prototypes for all usable functions

Let’s see what we can use in the math library...

There’s a Function to Compute the Square Root

In math, we’d write $\sqrt{x}$, in C we need to use:

```c
/* Computes the square root of x. */
double sqrt(double x);
```

Example:

```
  sqrt(4.0) ⟶ 2.0
```

Here’s an Example of Using Square Root

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    double y = 4.0;
    double y_sqrt = sqrt(y);
    printf("sqrt(%.1lf) = %.1lf\n", y, y_sqrt);
    return EXIT_SUCCESS;
}
```

We Could Create Our Own Header File

Create a file called `sqrt.h` with the following content:

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

We can then use it in our program:

```c
/* Same as before */
#include "sqrt.h"

int main(void) {
  /* Same as before */
}
```

Before Compilation, the Preprocessor Does Replacements

In this case, the result after the replacements is:

```c
/* Same as before */
double sqrt(double x);

int main(void) {
  /* Same as before */
}
```

Note that `<file>` tells the compiler to look for the file in library directories and `"file"` means also look for the file in the same directory as the code

There’s a Function for Exponents

In math, we’d write x^y, in C we need to use:

```
/* Computes x to the power of y. */
double pow(double x, double y);
```

Example:

```
  pow(2.0, 8.0) → 256.0
```

There are Trigonometric Functions

```
/* Computes sin using radians. */
double sin(double radians);
```

The `math.h` header also includes the value of π
- To use it, you type `M_PI`

The preprocessor replaces `M_PI` with the value of π
- The value used is 21 digits of π

There are Trigonometric Functions

```c
/* Computes sin using radians. */
double sin(double radians);
```

The `math.h` header also includes the value of π

- To use it, you type `M_PI`

The preprocessor replaces `M_PI` with the value of π

- The value used is 21 digits of π

Example:

`sin(M_PI/2.0)` $\longrightarrow$ `1.0`

There are Logarithm Functions Too

In math, we'd write $\ln x$ or $\log_e x$, in C we need to use:

```
/* Computes the natural logarithm (to base e) of x. */
double log(double x);
```

We can also use the value of e using `M_E`

In math, we'd write $\log x$ or $\log_{10} x$, in C we need to use:

```
/* Computes the common logarithm (to base 10) of x. */
double log10(double x);
```

We Can Still Get the Remainder Using `double` Values

We can't use `%` when either operand is a `double` we have to use a function

```c
/* Computes the remainder of x divided by y. */
double fmod(double x, double y);
```

We Can Still Get the Remainder Using `double` Values

We can't use `%` when either operand is a `double` we have to use a function

```
/* Computes the remainder of x divided by y. */
double fmod(double x, double y);
```

Example:

```
  fmod(3.5, 2.0) → 1.5
```

We Can Get the Minimum and Maximum of Two Values

```
/* Outputs the smaller of the two values. */
double fmin(double a, double b);

/* Outputs the larger of the two values. */
double fmax(double a, double b);
```

This may not be useful if both arguments are literal values (literals)
- However, it may be useful if at least one of the arguments is a variable

We Can Get the Minimum and Maximum of Two Values

```
/* Outputs the smaller of the two values. */
double fmin(double a, double b);

/* Outputs the larger of the two values. */
double fmax(double a, double b);
```

This may not be useful if both arguments are literal values (literals)
  However, it may be useful if at least one of the arguments is a variable

Examples:

```
fmin(2.0, 3.5) → 2.0
fmin(3.5, 2.0) → 2.0
fmax(2.0, 3.5) → 3.5
fmax(3.5, 2.0) → 3.5
```

There’s Multiple Ways to Round to the Nearest Integer

```c
/* Computes the nearest integer, rounding away from zero in halfway cases. */
double round(double x);

/* Computes the nearest integer, rounding towards even in halfway cases(*).
   *: By default, there are ways to change this. */
double rint(double x);
```

There’s Multiple Ways to Round to the Nearest Integer

```
/* Computes the nearest integer, rounding away from zero in halfway cases. */
double round(double x);

/* Computes the nearest integer, rounding towards even in halfway cases(*).
   *: By default, there are ways to change this. */
double rint(double x);
```

Examples:

```
  round(1.5) → 2.0
  round(2.5) → 3.0
  rint(1.5) → 2.0
  rint(2.5) → 2.0
```

Using a `double` Format Specifier Does Not Change the Value

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    double a = 2.5;
    printf("%.0lf\n", a); /* prints "2" */
    printf("%.1lf\n", a); /* prints "2.5" */
    return EXIT_SUCCESS;
}
```

Note that for `"%.0lf`, the rounding used is `rint`,
if there are more digits, e.g. `"%.1lf` C rounds towards 0

We Can Move the Decimal Point For Rounding

```
printf("%.1lf\n", 2.55);
printf("%.1lf\n", 2.65);
```

$\rightarrow$ prints 2.5

$\rightarrow$ prints 2.6

We Can Move the Decimal Point For Rounding

`printf("%.1lf\n", 2.55);` → prints `2.5`
`printf("%.1lf\n", 2.65);` → prints `2.6`

The rounding rules only apply to the nearest integer,
so we move the decimal point by multiplying by `10.0`

```
  2.55 * 10.0 → 25.5
```

The digit we want to round should be the first digit to the left of the decimal
  After rounding, divide by the same number for the original decimal point

We Can Move the Decimal Point For Rounding

```
printf("%.1lf\n", 2.55); → prints 2.5
printf("%.1lf\n", 2.65); → prints 2.6
```

The rounding rules only apply to the nearest integer,
so we move the decimal point by multiplying by `10.0`

```
  2.55 * 10.0 → 25.5
```

The digit we want to round should be the first digit to the left of the decimal
  After rounding, divide by the same number for the original decimal point

```
printf("%.1lf\n", rint(2.65 * 10.0) / 10.0); → prints 2.6
printf("%.1lf\n", rint(2.55 * 10.0) / 10.0); → prints 2.6
```

There’s Even More Ways to Round to the Nearest Integer

In math, we’d write $\lceil x \rceil$, in C we need to use:

```c
/* Computes the first integer larger or equal to x. */
double ceil(double x);
```

In math, we’d write $\lfloor x \rfloor$, in C we need to use:

```c
/* Computes the first integer smaller or equal to x. */
double floor(double x);
```

Examples:

```
ceil(1.1) → 2.0
ceil(2.0) → 2.0
ceil(-1.9) → -1.0
floor(1.9) → 1.0
floor(2.0) → 2.0
floor(-1.1) → -2.0
```

Truncation is Similar to Floor and Ceiling

```
/* Truncates x to an integer. */
double trunc(double x);
```

`trunc` (truncate) returns the same as
- `floor` for positive values, and
- `ceil` for negative values

Examples:

```
  trunc(1.1)  → 1.0
  trunc(2.0)  → 2.0
  trunc(-1.9)  → -1.0
  trunc(1.9)  → 1.0
  trunc(2.0)  → 2.0
  trunc(-1.1)  → -1.0
```

There's More Math Functions Available

See the entire list on Wikipedia

For this course you should only use the functions
that use `double` for all arguments and outputs

You may find `fabs` (the absolute value) useful for Lab 2

Problem: Making Change Using Nickels

Given an amount, in dollars, we want to round to the nearest nickel

We Can Create Random Numbers Too

The standard C library defines a function called `rand`

- It's function prototype is in `stdlib.h`

```
/* Returns a "random" number between 0 and RAND_MAX (inclusive). */
int rand(void);
```

We Can Create Random Numbers Too

The standard C library defines a function called `rand`
  It’s function prototype is in `stdlib.h`

```
/* Returns a "random" number between 0 and RAND_MAX (inclusive). */
int rand(void);
```

`RAND_MAX` is `2147483647` using most compilers
  Recall: this is the maximum value represented by an `int` ($2^{31} - 1$)

`rand` is a Pseudorandom Number Generator

Pseudo means not genuine

`rand` uses a formula to determine the next “random” number

  It uses a “seed” value to create a deterministic sequence of numbers
  (deterministic means it’s the same every time)

You Can Change the Seed Value for `rand` Using `srand`

```
/* Changes the seed value that `rand` uses to generate "random" numbers. */
void srand(unsigned int seed);
```

Note that the type `unsigned int` represents a whole number
- It does not have a sign bit, and still uses 4 bytes

A `unsigned int` can represent numbers from 0 to $2^{32} - 1$
- Generally it is a bad idea to mix signed and unsigned types
- (for this course we'll always use `int` if we don't need a decimal)

Use the Modulo Operator to Generate a Different Range

Recall `%` gives us the remainder after doing integer division

- `rand` returns a whole number, so the rules with negative values don't apply

`rand() % 10` results in a number between 0 and 9 (inclusive)

- If we want a number between 1 and 10 (inclusive) we can just add 1

Use the Modulo Operator to Generate a Different Range

Recall `%` gives us the remainder after doing integer division

- `rand` returns a whole number, so the rules with negative values don't apply

`rand() % 10` results in a number between 0 and 9 (inclusive)

- If we want a number between 1 and 10 (inclusive) we can just add 1

`rand() % 10 + 1` results in a number between 1 and 10 (inclusive)

We Can Generalize Generating A Random Number

If we want to generate a number between [a, b], where a $<$ b, we can use:

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