# C++ and Object-Oriented Programming

Functions

# *Function Calls and Headings*

- Goals
  - Evaluate a few math functions
  - Use arguments in functions calls
  - Appreciate why programmers divide software into functions
  - Use integer objects
  - Use quotient remainder division
  - Show the general categories of errors with examples

# cmath functions

- C++ defines a large collection of standard math and trig functions such as

```
sqrt(x) // return the square root of x
fabs(x) // return the absolute value of x
```

- Functions are called by specifying the function name followed by the argument(s) in parentheses:

```
cout << sqrt(4.0) << "  " << fabs(-2.34);
```

  - Output ?___________ ?

# *The **`pow`** function*

- The `pow` function returns the first argument to the second argument's power.
- For example, `pow(2.0, 3.0)` returns 2 to the 3rd power ($2^3 = 8.0$)

```
double base, power;
base = 2.0;
power = 4.0;
cout << pow(base, power); // Output _______?
```

- The following table shows some of the other `cmath` functions.

# Some cmath functions

| *Function Call* | *Class of Argument* | *Class of Result* | *Value Returned* | *Example* | *Result* |
|---|---|---|---|---|---|
| **ceil(x)** | double | double | Smallest integer ≥ x | ceil(2.1) | 3.0 |
| **cos(x)** | double | double | Cosine of x radians | cos(1.0) | 0.54 |
| **fabs(x)** | double | double | Absolute value of x | fabs(-1.5) | 1.5 |
| **floor(x)** | double | double | Largest integer ≤ x | floor(2.9) | 2.0 |
| **pow(x,y)** | double | double | x to the yth power ($x^y$) | pow(2,4) | 16.0 |
| **sin(x)** | double | double | Sine of x radians | sin(1.0) | 0.84 |
| **sqrt(x)** | double | double | Square root of x | sqrt(4.0) | 2.0 |

# *Evaluate Some Function Calls*

- Different arguments cause different return values

```
ceil(0.1)   _______ ?    sqrt(16.0)       ______ ?
ceil(1.1)   _______ ?    sqrt(sqrt(16)) ______ ?
pow(2.0, 3)_______ ?    fabs(-1.2)       ______ ?
sqrt(4.0)   _______ ?    floor(3.99)      ______ ?
```

- To use mathematical functions, you must

```
#include <cmath>
```

Optional Demo mathfuns.cpp

# *Rounding Example*

***Problem:*** *Write a program that rounds any number to a given number of decimal places.*

- Code that will round x to n decimal places uses **`pow`** and **`floor`**:

```
x = 456.789;
n = 2;
x = x * pow(10, n);    // x __________ ?
x = x + 0.5;           // x __________ ?
x = floor(x);          // x __________ ?
x = x / pow(10, n);    // x __________ ?
```

# *Calling Documented Functions*

- C++ has many functions available
  - standard functions such as

`sqrt`  `pow`  `string::length`  `ostream::width`

  - programmer-defined functions written for specific applications
  - functions that are members of off-the-shelf classes: bankAccount::withdraw
- Functions are more easily understood when documented with comments

# *Preconditions and Postconditions*

- Preconditions and postconditions are C++ comments that represents a contract between the implementers of a function and the user (client) of that function.
  - *Preconditions*: What the function requires.
  - *Postconditions*: What the function will do if the preconditions are met.

# *Pre and Post Conditions*

- The preconditions are the circumstances that must be true in order for the function to successfully fulfill the postconditions.
- Example    Precondition abbreviates to *pre* :

```
double sqrt(double x)
//  pre: x >= 0
// post: Returns square root of x
```

  - Note: On most systems now, `sqrt(-1.0)` returns not a number `NaN` or infinity `-1.#IND`

# *Function Headings*

- A *function heading* is the complete declaration of a function without its implementation (sometimes called the function's *signature).*
- For example, this signature tells us how to call the function, but not how it works:

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

# *Function Headings*

Note: No Semicolon!

- General form of a function heading:

  *return-type function-name* **(** *parameter-list* **)**

  - the *return-type* is any C++ class e.g. `double` or `string`
  - the *function-name* is any valid identifier
  - the *parameter-list* is a set of 0 or more parameters

- General form for declaring parameters:

| *class-name identifier* | *class-name* **&** *identifier* |
|---|---|
| `double f(double x)` | `void init(grid& g)` |
| `int max(int j, int k)` | `int roots(double a, double b, double c, double& r1, double& r2)` |

# *Argument/Parameter Associations*

- Example call to f (above) shows that arguments match parameters by position

```
double f(double x, double y)
// post: return x-y;
```

```
cout << f(3.0, -5.32);
```

- The value of the first argument is copied to the first parameter, the value of the second argument to the second parameter, and so on.
  - Like these two assignments:

```
x = 3.0;
y = -5.32;
```

# Self-Help Exercise

- Write the output generated by these function calls:

```
cout << f( 1.0, 1.0) << endl;  // _____
cout << f( 1.0, 1.5) << endl;  // _____
cout << f( 3.0, 0.5) << endl;  // _____
cout << f(-1.0, 1.0) << endl;  // _____
```

- Write valid for each valid function call or explain why the code is incorrect?

```
a. f(1.0)                d. f(1, 2)

b. f("Bob", "Sue")       e. f(1, 2, 3)

c. f(-0.001, +4.5)       f. f((1, 2)
```

# *Summary*

- Documented function headings provide the following information:
  - The type (or class) of value returned by the function.
  - The function name.
  - The number of arguments to use in a call.
  - The type (class) of arguments required in the function call.
  - Pre- and post-conditions tell us what the function will do if the preconditions are met.

# *int Arithmetic*

- int object are similar to double, except they store whole numbers (integers) only

```
int anInt;
anInt = 123;          // anInt _________ ?
anInt = 1.99;         // anInt _________ ?
```

- Division with integer constants and int objects is also a different operation.

```
anInt = 9 / 2;        // anInt _________ ?
anInt = anInt / 5; // anInt _________ ?
```

# *The integer % operation*

- C++ has the % operator that returns the remainder in integer division

```
anInt = 9 % 2;          // anInt ________ ?
anInt = 101 % 2;        // anInt ________ ?
anInt = 5 % 11;         // anInt ________ ?
anInt = 361 % 60;       // anInt ________ ?

int quarter;
quarter = 79 % 50 / 25;  // quarter ___ ?
quarter = 57 % 50 / 25;  // quarter ___ ?
```

# Integer quotient/remainder

- Some formulas using quotient and remainder

```
int total, hours, minutes, seconds;
total = 3726; // Write formulas to compute:

hours   = _______________________________ ;

minutes = ________________________________ ;

seconds = ________________________________ ;
```

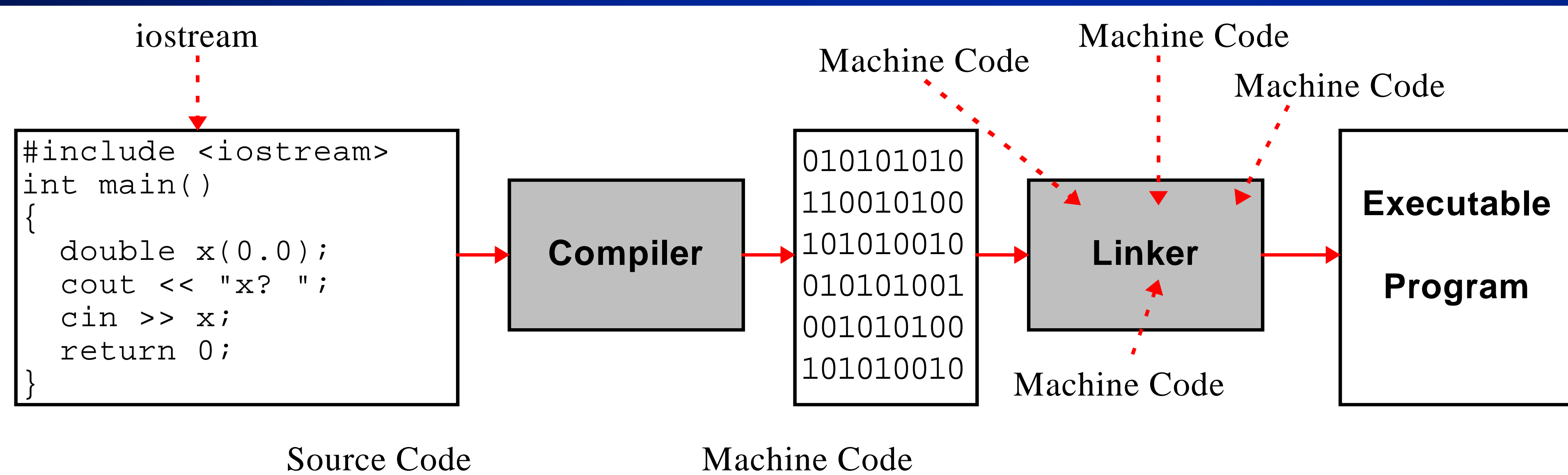
- Also note: For int / float, first promote the integer operand to it's floating point equivalent
- Example

```
5 / 2.0 = _____ ?
```

# *Implementation Errors*

- Categories of errors and warnings are detected during program implementation
  1. Compiletime errors
  2. Warnings
  3. Linktime errors
  4. Runtime errors  (e.g., user inputs wrong type)
  5. Intent errors (Logic errors)
- You've probably experienced many errors, especially those detected at compiletime

# *Compiling and Linking to Create an Executable Program*



# *Errors Detected at Compiletime*

- Generated while the compiler is processing the source code.
- Compiler reports violations of syntax rules. For example given these general forms:
  - object-declaration

    *class-name identifier-list ;*
  - identifier-list:

    *identifier*

    -or-

    *identifier-1, identifier-2, identifier-3, ..., identifier-n*

# *Pretend you are the compiler*

- What should the compiler do when it is processing this source code?

```
int student Number;
// ...

cin >> student;
```

# *Warnings generated by the compiler*

- The compiler generates warnings when it discovers something that is legal, but potentially problematic
- Example:

```
double x, y, z ;
y = 2 * x ;
// Warning: x used before being intialized
// Warning: z declared but never used
```

# *Linktime Errors*

- Errors that occur while trying to put together (link) an executable program
- For example, we must always have function main (Main or MAIN won't do).

Optional Demo: linkerr.cpp

# *Runtime Errors*

- Errors that occur at runtime, that is, while the program is running.
- Examples: Invalid numeric input, division by 0, picking up a cookie that is not there.
  - Some systems return infinity or **NaN** (Not A Number) or **-1.#IND**
  - Some terminate the program prematurely

Optional Demo: runerr.cpp

# *Intent Errors*

- When the program does what you typed, not what you intended.
- Imagine this code

```
cout << "Enter sum: ";
cin >> n;
cout << "  Enter n: ";
cin >> sum;
average = sum / n;
```

Whose responsibility is it to catch this error __ ?

# *When the software doesn't match the specification*

- If none of the preceding errors occur, the program may still not be right.
- The working program may not match the specification because either
  - The programmers did not match, or understand the problem statement
  - The problem statement may have been incorrect
  - Someone may have changed the problem statement