PY410 / 505
Computational Physics 1

Salvatore Rappoccio

Code

- Code is in CompPhys/ReviewCpp/BasicExamples

Flow control : Iteration

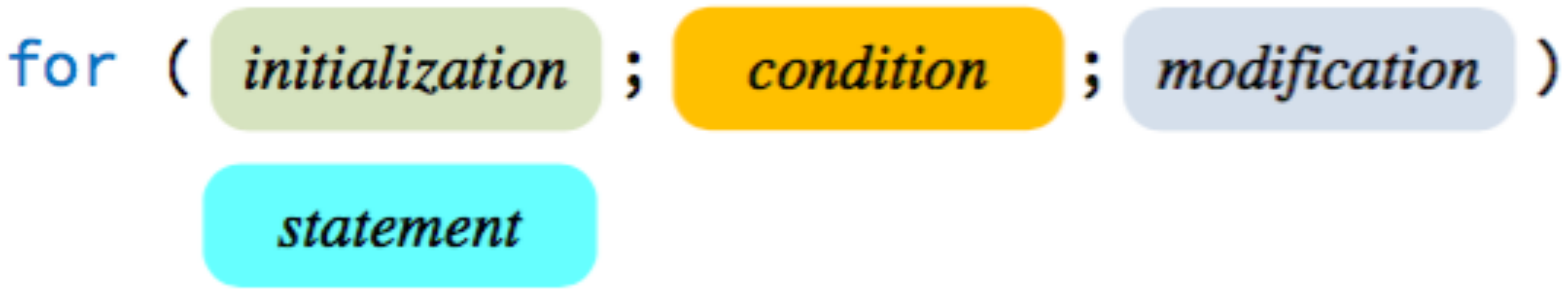
C++: Iteration

- It’s a royal pain to count. Humans suck at it.
- Computers are really, really fantastic at it, though.
- Similarly, computers are great at doing the same thing over and over (and over and over andoverandoverandover)

- This is referred to as “iteration”. C++ options:
  - “while” loop
  - “do while” loop
  - “for” loop
  - “goto” statements (never use them)

C++: Iteration

- Most commonly used is probably “for” loops:



```
for ( initialization ; condition ; modification )
      statement
```

- Initializes with “initialization”
- Executes “statement” until “condition” is met
- After each iteration, “modification” is performed

C++: Iteration

- Example: “forloop.cc”:

```cpp
#include <iostream>
int main(void){

  for ( unsigned int i = 0; i < 10; ++i) {
    std::cout << i << ", ";
  }
  std::cout << std::endl;
  return 0;
}
```

- compile and execute, and you get:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 
```

C++: Iteration

- Can also nest them: “forloop_nested.cc”:

```
#include <iostream>
int main(void){

  for ( unsigned int i = 0; i < 10; ++i) {
    for ( unsigned int j = i; j < 10; ++j ){
      std::cout << "(" << j << ", " << i << "), ";
    }
    std::cout << std::endl;
  }
  return 0;
}
```

- Compile and run, what do you get?

C++: Iteration

- Related concepts:
  - “continue” : automatically continue to the next iteration, don’t execute the rest
  - “break”: get out of the loop right away
  - Useful for termination abnormally and for error checking

- Things to be careful about:
  - Infinite loops : you didn’t give a correct termination condition
  - Incorrect initialization : your initialization was incomplete

C++: Iteration

- Similar to "for" loops are "do, while" and "while" loops

```
do
      statement
while ( condition );
```

```
while ( condition )
      statement
```

- Very similar, except the "do, while" loop ALWAYS executes the "statement" at least once, whereas "while" will only do it if the "condition" is met

- Use "break" and "continue" to get out, OR adjust the variables in "condition"

C++: Iteration

- “while” and “for” loops can be made semantically identical (while syntactically different)

```
for ( initialization ; condition ; modification )
    statement
```

=

```
initialization
while ( condition ) {
    statement
    modification
}
```

C++: Iteration

- Example: “whileloop.cc”

```
#include <iostream>
int main(void){
  int i = 0;
  while( i < 5 ) {
    std::cout << i << ", ";
    ++i;
  }
  std::cout << std::endl;
  return 0;
}
```

C++: Iteration

- How about using the “break” statement? “whileloop_break.cc"

```
#include <iostream>
int main(void){
  int i = 0;
  std::cout << "Enter a number, negative number to quit" << std::endl;
  while( std::cin >> i ) {
    if ( i < 0 ) {
      std::cout << "Negative number entered, exiting." << std::endl;
      break;
    } else {
      std::cout << "You entered i=" << i << std::endl;
    }
  }
  return 0;
}
```

C++: Iteration

- Another nice “either / or but not both” construct is the “conditional” operator “?”. Syntax is:

(*condition*) ? *expression 1* : *expression 2*

- Fast way of saying :
  - if (condition) expression 1
  - else expression 2

Scope

C++: Scope

- Now in a position to talk about “scope”
- Scope is the lifetime of a variable, denoted by curly braces “{ }”
- A variable must be unique IN THE CURRENT SCOPE, but can be duplicated in DIFFERENT scopes
- Loops have different scopes because they are separated by {}
- So what does this give you? “scope.cc"

```
#include <iostream>
int main(void){
  unsigned int i = 1000;
  for( unsigned int i = 0; i < 10; ++i ) {
    std::cout << i << std::endl;
  }
  std::cout << "Outside the loop, i = " << i << std::endl;
  return 0;
}
```

C++: Scope

- This is the first instance of something having the same name but different scope

- You can declare variables to have GLOBAL scope or LOCAL scope
  - Global: all functions and all files can see it
    - Bad! Maximally violates principle of least privilege but sometimes has a use
  - Local: only defined within { }
    - Good! Principle of least privilege satisfied

Special scopes: Functions

C++: Functions

- Now we’ve seen how to execute BLOCKS of code
- What if we want to name those blocks?
  - That’s a function

- We’ve already seen the first function (“main”)
- What about others?

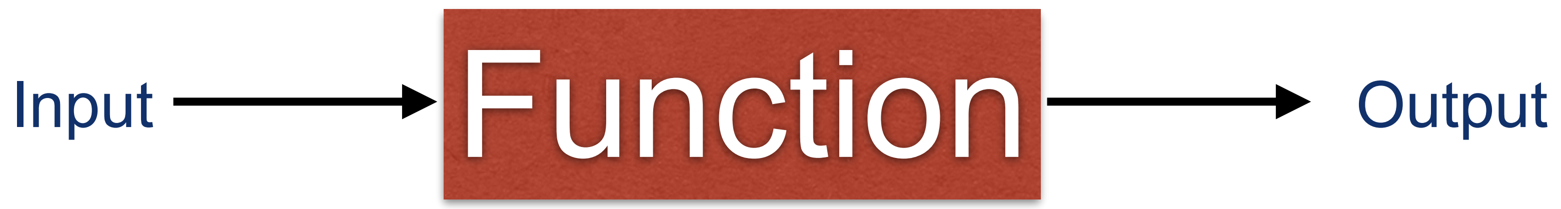
- Remember mathematical functions, like “squared”?

$$f(x) = x^2$$

- Literally: “input x, return x*x”

C++: Functions

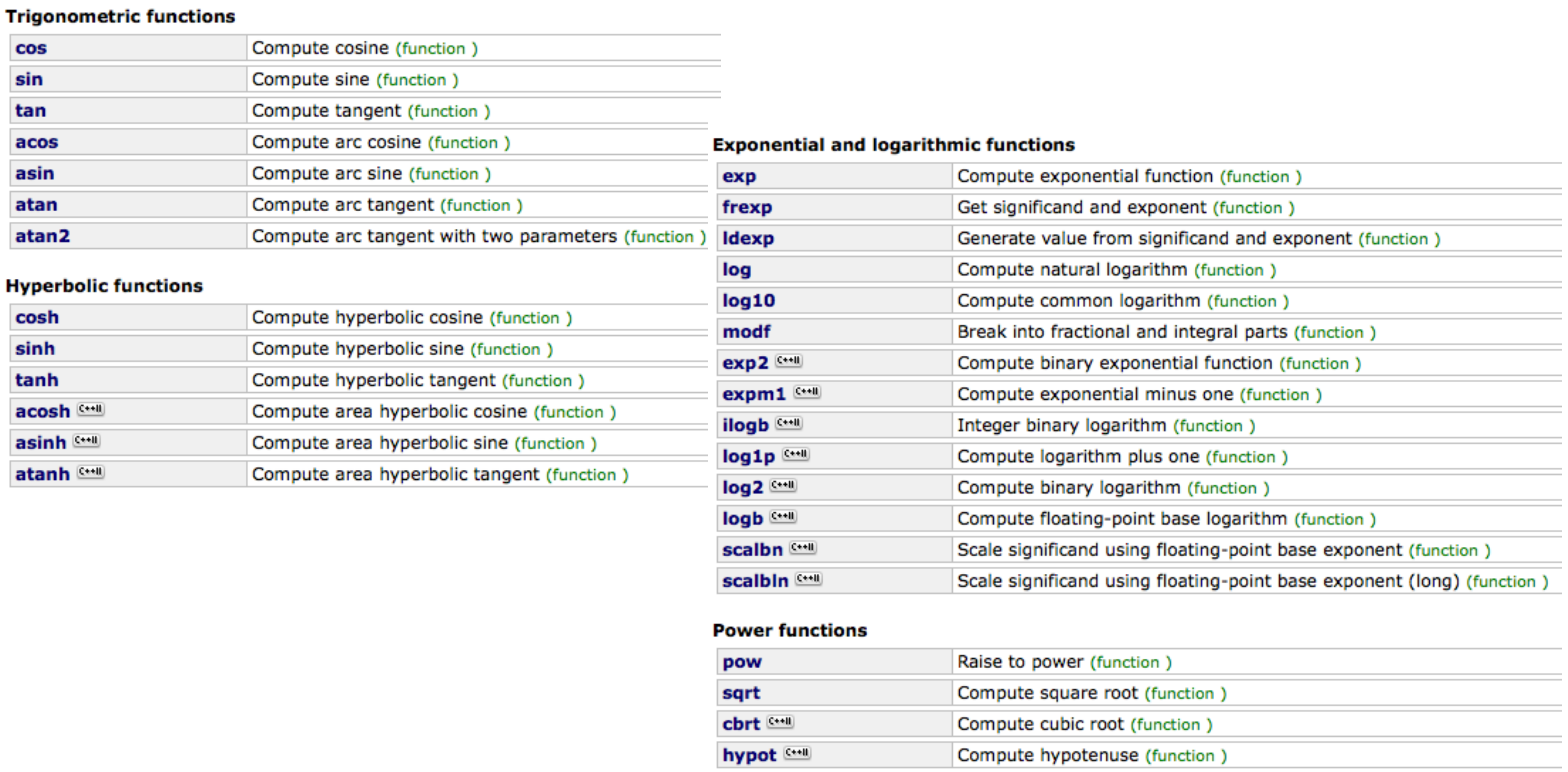
- So we can generalize:



- Take inputs, do stuff, give output

C++: Functions

- Lots of functions already defined (Example: cmath)
- http://www.cplusplus.com/reference/cmath/



Trigonometric functions

cos	Compute cosine (function)
sin	Compute sine (function)
tan	Compute tangent (function)
acos	Compute arc cosine (function)
asin	Compute arc sine (function)
atan	Compute arc tangent (function)
atan2	Compute arc tangent with two parameters (function)

Hyperbolic functions

cosh	Compute hyperbolic cosine (function)
sinh	Compute hyperbolic sine (function)
tanh	Compute hyperbolic tangent (function)
acosh C++11	Compute area hyperbolic cosine (function)
asinh C++11	Compute area hyperbolic sine (function)
atanh C++11	Compute area hyperbolic tangent (function)

Exponential and logarithmic functions

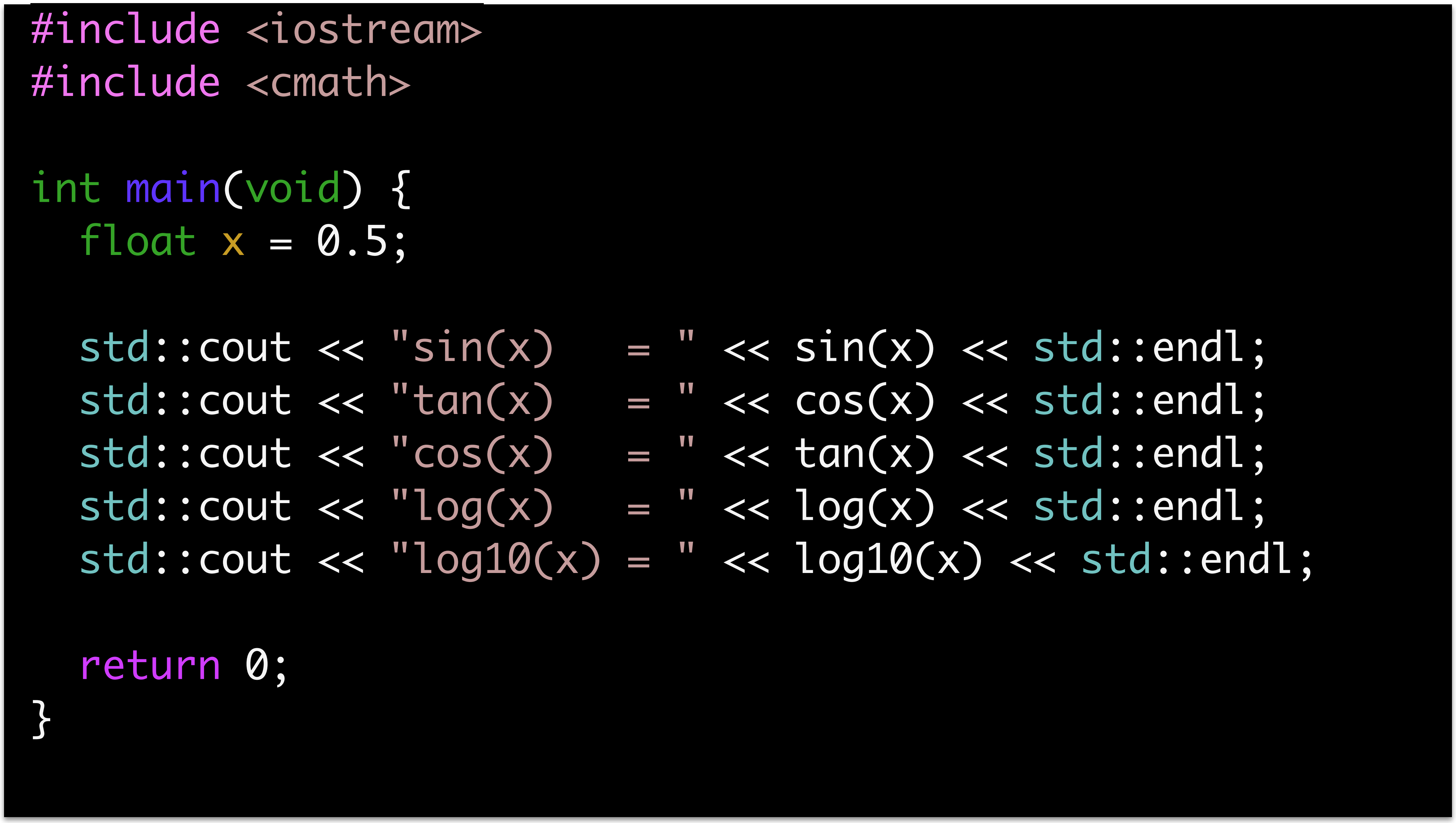
exp	Compute exponential function (function)
frexp	Get significand and exponent (function)
ldexp	Generate value from significand and exponent (function)
log	Compute natural logarithm (function)
log10	Compute common logarithm (function)
modf	Break into fractional and integral parts (function)
exp2 C++11	Compute binary exponential function (function)
expm1 C++11	Compute exponential minus one (function)
ilogb C++11	Integer binary logarithm (function)
log1p C++11	Compute logarithm plus one (function)
log2 C++11	Compute binary logarithm (function)
logb C++11	Compute floating-point base logarithm (function)
scalbn C++11	Scale significand using floating-point base exponent (function)
scalbln C++11	Scale significand using floating-point base exponent (long) (function)

Power functions

pow	Raise to power (function)
sqrt	Compute square root (function)
cbrt C++11	Compute cubic root (function)
hypot C++11	Compute hypotenuse (function)

C++: Functions

- Syntax is completely intuitive, so try “mathexamples.cc”
- Intuitive so I won’t belabor:



```cpp
#include <iostream>
#include <cmath>

int main(void) {
  float x = 0.5;

  std::cout << "sin(x)   = " << sin(x) << std::endl;
  std::cout << "tan(x)   = " << cos(x) << std::endl;
  std::cout << "cos(x)   = " << tan(x) << std::endl;
  std::cout << "log(x)   = " << log(x) << std::endl;
  std::cout << "log10(x) = " << log10(x) << std::endl;

  return 0;
}
```

C++: Functions

- Writing your own function:

```
output type    function_name ( arguments ) {

    Function's body

}
```

C++: Functions

- Example: “xsquared.cc” x^2

```
#include <iostream>

float xsquared( float x ){ return x*x; }

int main(void) {
  float x = 5.0;
  std::cout << xsquared(x) << std::endl;

  return 0;
}
```

Function must be declared ahead of time

Then you call it with parentheses: “bla(x)”

C++: Functions

- In C++, you must DECLARE a function ahead of time
- However, you can DEFINE it whenever you want
  - Declare: Shows the types.
  - Define: the actual code of the function

- declaration:

```
float xsquared( float );
```

- definition:

```
float xsquared( float x ){ return x*x; }
```

- Can be the same, but need not be
  - For complicated functions, usually don’t define them ahead of time, just declare them

C++: Functions

- Return values:
  - Can only return ONE VALUE
  - Python can do many, but not C++

- Important programming practice: returning a number “by value” as in a function makes THREE COPIES of the return type
  - Fine for built-in types
  - Terrible, horrible, no good, bad for big classes
  - C++0x and later have “move” semantics (more on that later) that makes 1.5 copies instead of three :)

C++: Functions

- Can also specify a DEFAULT value for function inputs:

```
#include <iostream>

int squared( int i = 0 ) { return i*i;}

int main(void)
{

  std::cout << squared() << std::endl;   // Returns 0
  std::cout << squared(2) << std::endl;  // Returns 4
  return 0;
}
```

C++: Functions

- What about SCOPE of variables? “funcscope.cc”
  - Global scope: variable available to ALL functions
  - Local scope: variable available to THIS function only
  - static: variable available to THIS function, but value is kept after scope ends (useful for counting)

```cpp
#include <iostream>
unsigned int i = 1000;
int duh( void ) {
  static unsigned int count = 0;
  unsigned int i = 2;
  std::cout << "for the " << count << "th time, i = " << i << std::endl;
  ++count;
  return i;
}

int main(void){
  for ( unsigned int i = 10; i < 20; ++i ) {
    std::cout << "i = " << i << ", duh() = " << duh() << ", global i = " << ::i << std::endl;
  }
  return 0;
}
```

C++: Functions

- Can call functions within functions

- Can call YOUR OWN function within functions (recursion)

- Example: Fibonacci sequence “fibo.cc”:

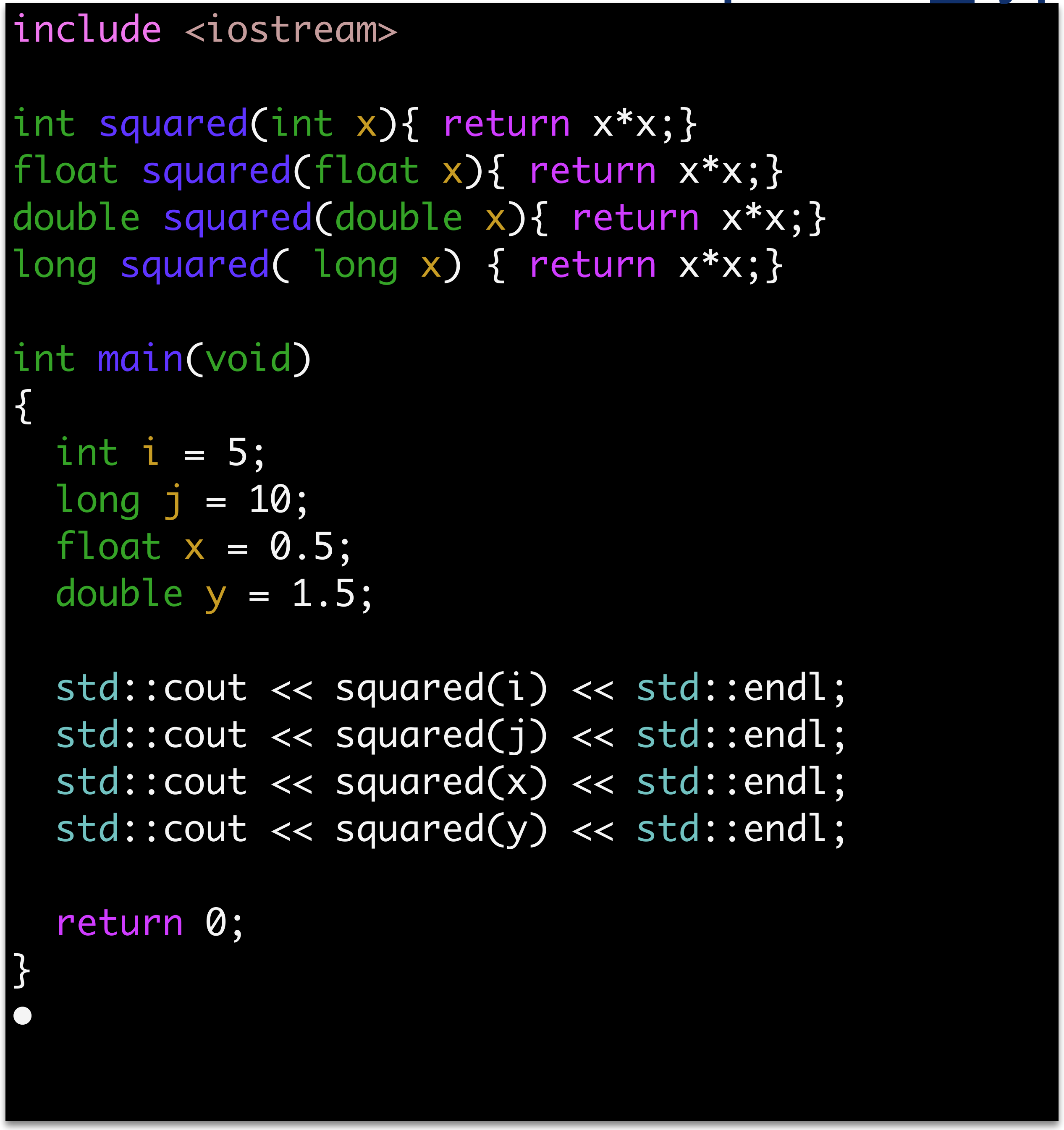
```
#include <iostream>

int fibonacci(int n) {
  if (n <= 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fibonacci(n - 2) + fibonacci(n - 1);
}

int main(void)
{
  for ( unsigned int i = 0; i < 10; ++i ) {
    std::cout << fibonacci(i) << ", ";
  }
  std::cout << std::endl;

  return 0;
}
```

C++: Functions

- C++ has a nice feature in OVERLOADING functions
  - Example: if you want x^2, what do you need?
    - Input as int
    - Input as float
    - Input as double
    - Input as unsigned int
    - Input as short
    - Input as unsigned long
    - …
- But you probably want them all to be called the same thing (xsquared)
- You can define multiple functions with different ARGUMENT TYPES
  - Caveat: Cannot differ only by return type

C++: Functions

- Looks like this: “xsquared_types.cc”



```
include <iostream>

int squared(int x){ return x*x;}
float squared(float x){ return x*x;}
double squared(double x){ return x*x;}
long squared( long x) { return x*x;}

int main(void)
{
  int i = 5;
  long j = 10;
  float x = 0.5;
  double y = 1.5;

  std::cout << squared(i) << std::endl;
  std::cout << squared(j) << std::endl;
  std::cout << squared(x) << std::endl;
  std::cout << squared(y) << std::endl;

  return 0;
}
```

C++: Functions

- Isn’t it annoying to write that over and over? And if I try a new type, I have to recompile? What a pain.

- If only there were some way to fix this…

Function templates

C++: Function Templates

- Do I have a DEAL for YOU!
- You can create a “function template” instead of a function

- This tells you HOW to create a function if you are GIVEN the types

- Syntax is a bit weird:

```
template< class T>
T squared(T x){ return x*x;}
```

This is NOT A FUNCTION.
This is a TEMPLATE for a function.

C++: Function Template

Cookie template

Cookie

Function template

```
template< class T>
T squared(T x){ return x*x;}
```

```
 squared<int>( 2 )
```

Function

C++: Function Templates

- Functions : Compiled, exist in memory
- Function templates: NOT compiled, must be given a type
- EACH type gets a SEPARATE function in memory, on demand
- More on templates later