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 and over and over and over and over)

• 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:
  - 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”:

```cpp
#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

- 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)
C++: Iteration

- Example: “whileloop.cc”

```cpp
#include <iostream>
int main(void){
    int i = 0;
    while( i < 5 ) {
        std::cout << i << " , ";
        ++i;
    }
    std::cout << std::endl;
    return 0;
}
```
• How about using the “break” statement? “whileloop_break.cc"

```cpp
#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;
}
```
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
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"

```cpp
#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
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)


### Trigonometric functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cos</code></td>
<td>Compute cosine (function)</td>
</tr>
<tr>
<td><code>sin</code></td>
<td>Compute sine (function)</td>
</tr>
<tr>
<td><code>tan</code></td>
<td>Compute tangent (function)</td>
</tr>
<tr>
<td><code>acos</code></td>
<td>Compute arc cosine (function)</td>
</tr>
<tr>
<td><code>asin</code></td>
<td>Compute arc sine (function)</td>
</tr>
<tr>
<td><code>atan</code></td>
<td>Compute arc tangent (function)</td>
</tr>
<tr>
<td><code>atan2</code></td>
<td>Compute arc tangent with two parameters (function)</td>
</tr>
</tbody>
</table>

### Hyperbolic functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cosh</code></td>
<td>Compute hyperbolic cosine (function)</td>
</tr>
<tr>
<td><code>sinh</code></td>
<td>Compute hyperbolic sine (function)</td>
</tr>
<tr>
<td><code>tanh</code></td>
<td>Compute hyperbolic tangent (function)</td>
</tr>
<tr>
<td><code>acosh</code></td>
<td>Compute area hyperbolic cosine (function)</td>
</tr>
<tr>
<td><code>asinh</code></td>
<td>Compute area hyperbolic sine (function)</td>
</tr>
<tr>
<td><code>atanh</code></td>
<td>Compute area hyperbolic tangent (function)</td>
</tr>
</tbody>
</table>

### Exponential and logarithmic functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exp</code></td>
<td>Compute exponential function (function)</td>
</tr>
<tr>
<td><code>frexp</code></td>
<td>Get significand and exponent (function)</td>
</tr>
<tr>
<td><code>ldexp</code></td>
<td>Generate value from significand and exponent (function)</td>
</tr>
<tr>
<td><code>log</code></td>
<td>Compute natural logarithm (function)</td>
</tr>
<tr>
<td><code>log10</code></td>
<td>Compute common logarithm (function)</td>
</tr>
<tr>
<td><code>modf</code></td>
<td>Break into fractional and integral parts (function)</td>
</tr>
<tr>
<td><code>exp2</code></td>
<td>Compute binary exponential function (function)</td>
</tr>
<tr>
<td><code>expm1</code></td>
<td>Compute exponential minus one (function)</td>
</tr>
<tr>
<td><code>ilogb</code></td>
<td>Integer binary logarithm (function)</td>
</tr>
<tr>
<td><code>log1p</code></td>
<td>Compute logarithm plus one (function)</td>
</tr>
<tr>
<td><code>log2</code></td>
<td>Compute binary logarithm (function)</td>
</tr>
<tr>
<td><code>logb</code></td>
<td>Compute floating-point base logarithm (function)</td>
</tr>
<tr>
<td><code>scalbn</code></td>
<td>Scale significand using floating-point base exponent (function)</td>
</tr>
<tr>
<td><code>scalbln</code></td>
<td>Scale significand using floating-point base exponent (long) (function)</td>
</tr>
</tbody>
</table>

### Power functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pow</code></td>
<td>Raise to power (function)</td>
</tr>
<tr>
<td><code>sqrt</code></td>
<td>Compute square root (function)</td>
</tr>
<tr>
<td><code>cbrt</code></td>
<td>Compute cubic root (function)</td>
</tr>
<tr>
<td><code>hypot</code></td>
<td>Compute hypotenuse (function)</td>
</tr>
</tbody>
</table>
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" $x^2$

```cpp
#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)”
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**:  
```c++
float xsquared( float );
```

**Definition**:  
```c++
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 :)

21
C++: Functions

• Can also specify a DEFAULT value for function inputs:

```cpp
#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? “funcsScope.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”:

```cpp
#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”

```cpp
#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…
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:

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

This is NOT A FUNCTION. This is a TEMPLATE for a function.
```cpp
template< class T >
T squared(T x) { return x*x; }
```

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