Separate Compilation Model

Recall: For a **function call** to compile, either the function's definition or declaration must appear previously in the same file.

Goal: Compile only modules affected by recent changes.

Technique: Separate "implementation" from "interface". Files will #include header files for other modules.

- **implementation file (foo.cpp)**
  - Function definitions
  - Class method implementations (later in our course)

- **header file (foo.h)**
  - Function declarations
  - Class declarations (later in our course)

Example: arithmetic.cpp
arithmetic.h
triangleMath.cpp

- arithmetic.cpp may be used in MANY other modules
- arithmetic.h will be #included

Visual C++ Express

- Download Free:

http://www.microsoft.com/express/Downloads/#2010-Visual-CPP

- WARNING: Not 100% compatible with g++ compiler, which we will use for testing!

- Press F5 to save, compile and run in debugger
- Debugger is very good! (Let's try it…)

Also demonstrate SSH secure shell

For Mac: CyberDuck?

Variables: Java Review

Java gives us exactly 2 choices:

1. primitive

   ```
   int x;
   ```

2. reference to object

   ```
   Cat y;
   y = new Cat();                          // “default” Cat
   y = new Cat(“Morris”, 17.5);
   ```

C++ – 4 kinds of Variables…

1. **primitives** – essentially like Java
2. **objects**

```
Cat y;   // Instantiates a "default" Cat!
Cat z("Morris", 17.5);
```

These variables are **not** pointers and they are **not** references. They hold objects.

They cannot be **null**.

The object is destroyed automatically when the variable goes out of scope.

C++ – 4 kinds of Variables…

3. **Pointers**

- Similar to C
- Can point to a primitive, an object, or another pointer
- Can point to something on the Stack or on the Heap

```
int x = 7;
Cat y;        // Remember, y IS a Cat!

int *p = &x;
Cat *q = &y;
Cat **z = &q;
```

C++ – 4 kinds of Variables…

4. References

- The variable "refers" to something, but doesn't actually hold the value.
- Typically used for function parameters.
- Kind of like Java references, but different (see next slide…)
- Can refer to a primitive, an object, or a pointer.
- Can refer to something on the Stack or on the Heap.

```
int x = 7;
Cat y;
int *p = &x;

int &r = x;          //   r is a reference to a primitive
Cat &s = y;          //   s is a reference to an object
int *&t = p          //   t is a reference to a pointer
```

Note: The symbol "&" is NOT being used as the address operator here!

Note: The symbol "=" is NOT being used as the assignment operator here!

Reference Variables: C++ vs. Java

Ways C++ references are different:

- You use & to declare them
- Can refer to primitives and pointers (not just objects)
- Can refer to something on the heap, but USUALLY refer to something on the stack
- You MUST initialize it when declared (no null reference possible)
- Once declared, you CANNOT change it refer to a different entity. (The entity may mutate, however.)
- Assignment operator **copies data**, does NOT move the reference elsewhere

Example: references.cpp

What is the output?

```
int a = 7;
int &b = a;
b = 4;
cout << a << endl;
int c = 6;
b = c;
b++;
cout << a << b << c << endl;
```

Strings

- C-style strings (null-terminated char arrays) still available:
  `char *x = "hello";`
  Literals like `"hello"` are always C-style strings

- Much easier/safer to use C++ string class:
  `string x("hello");`

  More convenient than Java String class because operators are overloaded (==, <=, >=, +, etc. Code example later…)

Important: C++ string objects are mutable!

Memory Maps: Java vs. C++

Consider **Java** code:

```
String x("hello");     // local variable on stack,
                       // object on the heap

String y;              // local variable
y = x;                 // aliasing
y = "hello";           // new object created,
                       // reference y is moved
```

Now let's look at the exact same code in C++…

Memory Maps: C++

```
string x("hello");  // object instantiated on the stack
string y;           // empty string instantiated
                    // Remember:  "y IS the string"!
y = x;              // COPYING THE DATA!!!
                    // The existing string y is mutated
```

Important Concept: In C++ the assignment operator copies the state from one object to another!

```
y = "hello";    // Mixing types!  The assignment
                // operator has been overloaded
                // to do what we want here…
                // Again, the string y is mutated.
```

Example: stringExample.cpp

- Review of memory map issues
- Assignments
- Using square brackets []
- Comparisons (==, <, >, etc.)