CS 216

Lecture 4

February 7th, 2013

Administrivia

PA1 Part 2 posted today, more info in class next week.

PA1 Part 2 Questions

Every class with scalar member variables must have a constructor

(e.g., every class in the hierarchy)

The (non-constructor) methods don’t need to do anything, just exist.

You don’t need a main function (or a makefile)

```
g++ -c *.cpp
```

Review

Virtual methods:
Call the method in the object pointed to/referenced.

Non-virtual methods: Call the method in the class of the calling expression

Virtual: what matters is the type of the object

Non-virtual: what matters is the type of the calling expression

It is a reasonable decision to make *everything* virtual.

Compiling individual files:

```
g++ -c Item.cpp
```

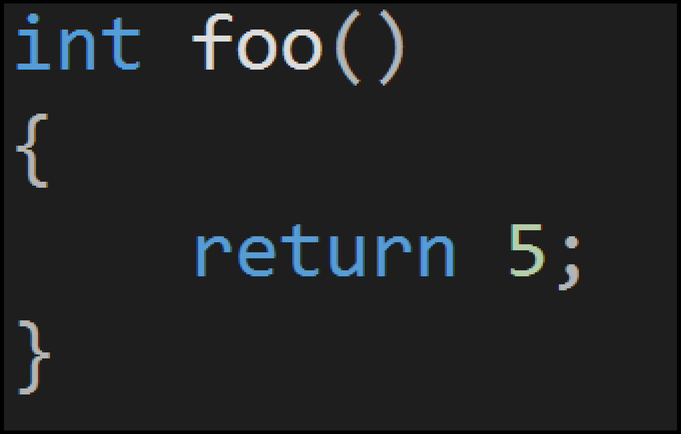
This gives us an object file – the compiled code, but it is not linked (and therefore can’t be run)

Function pointers

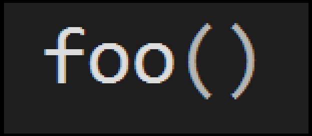
But first…
Everything is an expression.

Everything is an expression.

Expressions have types and values.



```
int foo()
{
    return 5;
}
```



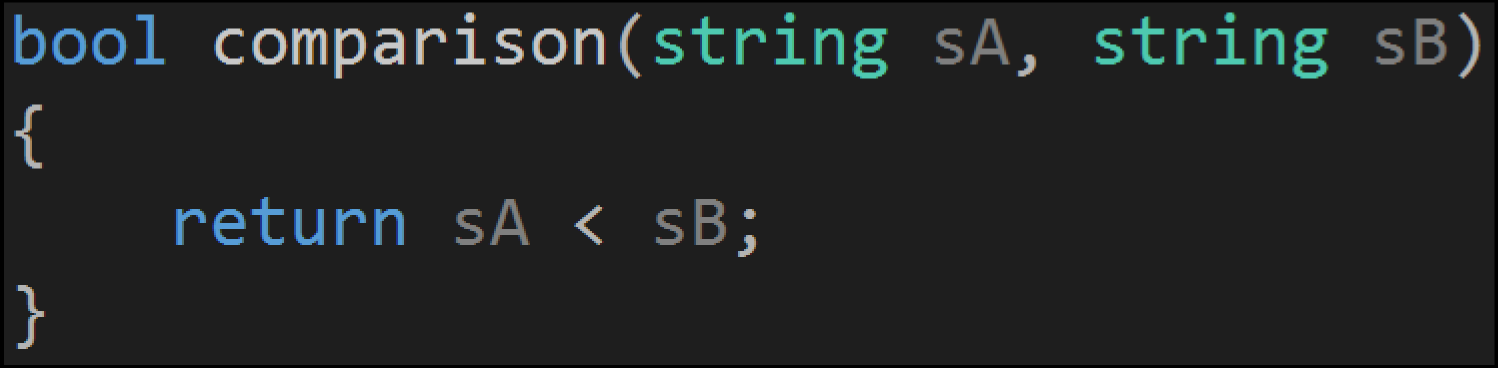
```
foo()
```

```
foo
```

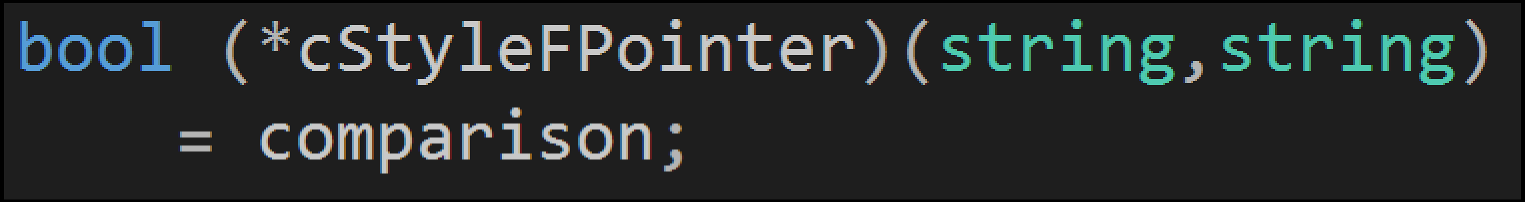
Variables, too, have types and values.

C++ lets us have variables of many, many types.

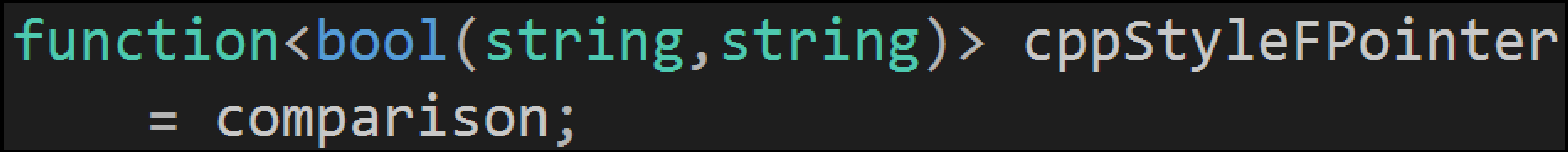
Functions can be stored in… perfectly normal variables.



```
bool comparison(string sA, string sB)
{
    return sA < sB;
}
```



```
bool (*cStyleFPointer)(string,string)
    = comparison;
```



```
function<bool(string,string)> cppStyleFPointer
    = comparison;
```

So, what exactly is a program?

Just bytes which can be loaded in the computer’s memory and executed.

When you load those bytes into memory, they then have memory addresses.

And, therefore, we can get a pointer to code in memory (a function), and execute it.

Yes, the C-style syntax for function pointers is terrible.

Really, really,
really terrible.

Really, really,
really
terrible.

Declare a variable:
return_type (*name)(arguments)

The type itself:
return_type (*)(arguments)

But once you have a variable, you can then call it as if it were a function…

```
int foo(int x, int y)
{
        return x + y;
}
```

```
int (*fptr)(int,int) = foo;

cout << fptr(15,15) << endl;
```

So this is the C-style way, what about the C++ style?

It’s actually a part of the C++11 (as in 2011) spec…

```cpp
#include <functional>
```

```
bloodroot:~/code>
bloodroot:~/code> g++ --std=c++0x x.cpp
bloodroot:~/code> _
```

```
function<int(int,int)> pFunc = foo;

cout << pFunc(5,5) << endl;
```

XML

“eXtensible Markup Language”

XML 1.0 – 1998

XML 1.1 – 2004

(1.1 is not widely used)

```xml
<?xml version="1.0" encoding="utf-8"?>
<World>
  <Item>
    <name>silver key</name>
    <properties>
      <property>metal</property>
      <property>silver</property>
    </properties>
    <weight>1</weight>
    <displayChar>)</displayChar>
    <value>10</value>
    <rarity>5</rarity>
  </Item>

  <Creature>
    <name>orc</name>
    <properties>
      <property>orcish</property>
      <property>humanoid</property>
    </properties>
    <level>2</level>
    <maxHP>15</maxHP>
    <displayChar>o</displayChar>
  </Creature>
</World>
```

Two things in an XML document

XML Header

Root Element

```
<?xml version="1.0" encoding="utf-8"?>
```

Element:
Start tag, end tag, and anything between.

Start tag:

<tag_name>

End tag:

</tag_name>

Empty element tag:

```
<tag_name />
```

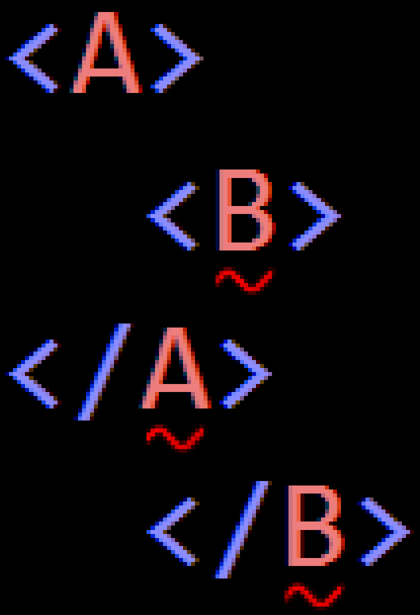
Start tags *must* be paired with end tags.

A start/end tag pair and everything it contains is an element.

Anything between the start and end tag is called content.

(An empty element tag just creates an element with no content)

Note that elements cannot overlap!



```
<A>
  <B>
</A>
  </B>
```

```
<name>orc</name>
```

Elements may contain other elements.

```xml
<properties>
  <property>orcish</property>
  <property>humanoid</property>
</properties>
```

```xml
<Creature>
  <name>orc</name>
  <properties>
    <property>orcish</property>
    <property>humanoid</property>
  </properties>
  <level>2</level>
  <maxHP>15</maxHP>
  <displayChar>o</displayChar>
</Creature>
```