Review Sheet 2b

CS 70: Data Structures and Program Development

Thursday, January 30, 2020

Today's Learning Targets

1. I understand how to break C++ into code files and header files.
2. I can write C++ classes (as header and code files).

Terminology: Java → C++

- "superclass" → "base class"
- "subclass" → "derived class"
- "field" → "data member"
- "method" → "member function"

Data Members

```
size_t spots_;
size_t age_;
```

This is what defines what a Cow object looks like in memory.

size_t

- Unsigned integer type.
- Need `#include<cstddef>` to use it.
- `typedef size_t = ...` system dependent

Member Functions

Declare them in the .hpp:

```
void moo(size_t numMoos);
```

Implement in the .cpp:

```
void Cow::moo(size_t numMoos) {
```

Call them with dot(.):

```
bessie.moo()
```

Scope resolution operator ::

In the implementation file, need to say which class's method we are implementing.

We might have a cow and a sheep that both eat differently

```
void cow::eat(){
    cout << "eating corn" << endl;
}

void sheep::eat(){
    cout << "eating grass" << endl;
}
```

Constructors

- Default
  - Parameterless constructor: `Cow()`
  - Used for default initialization (e.g. `Cow bessie;`)
  - Every class has one by default (does nothing)
- Parameterized
  - Constructor with parameters: `Cow(size_t numSpots, size_t age)`
  - Must be invoked explicitly (e.g. `Cow bessie{numSpots, age};`
- Delete
  - Used to disable the ability to call a function
  - Most useful for implicitly/automatically defined functions
  - e.g. `Cow() = delete;` ensures that there is no default constructor

Member initialization lists

```
Cow::Cow(size_t numSpots, size_t age)
    : spots_{numSpots}, age_{age}
{
    cout << "Made a cow with " << spots_ <<
                 " spots!" << endl;
}
```

Instantiating

```
Cow bessie{3,12}
```

- Use curly braces.
- This is modern style (different than Java and Python!).
- We will grade you on this in in CS70.

Semicolon at the end!

```
class className{
    //code
}; // this semicolon is important
```

If you forget this semicolon, you could get "fun" errors.

Header Files

- .hpp file contains the class *declaration*
  - What it contains, what it can do
- A file that `#include`s the .hpp can use the class
  - The compiler knows about the class
- .cpp file contains the member function *definitions*
  - The instructions for each function
- .cpp file is compiled to .o and linked into the final executable
  - Now the executable has the instructions
- `#include "cow.hpp"` in both `main.cpp` and `cow.cpp`. Why?

The (C/C++) Preprocessor

```
#include <iostream>
#define C_STYLE_CONSTANT 42

int main()
{
  std::cout << C_STYLE_CONSTANT << "\n";
  #ifdef WINDOWS
   // ...code specific to Windows
  #else
   // ...alternate code for a Unix-based OS
  #endif
}
```

Processes your code BEFORE compiling.

Confusingly, this tool is called cpp even though it has nothing to do with C++.

Include Guards

```
#ifndef COW_HPP_INCLUDED
#define COW_HPP_INCLUDED

// more includes

class Cow{
// data members and member functions
};

#endif // ifndef COW_HPP_INCLUDED
```

You are not allowed to declare something more than once!

Preprocessor trick that prevents code from being “copied” twice.

Separate Compilation

- compile `cow.cpp`
- compile `main.cpp`
- DO NOT compile `cow.hpp`
- link `cow.o` and `main.o`

Convert `Point.java` to C++

~~~
```java
public class Point {
    private int x_ = 0;
    private int y_ = 0;

    public Point(int x, int y) {
        x_ = x;
        y_ = y;
    }

    public void move(int deltaX, int deltaY) {
        x_ += deltaX;
        y_ += deltaY;
    }

    public int getX() {
        return x_;
    }
}
```
~~~

`point.hpp`

```
#ifndef POINT_HPP_    // C++ #include guard.
#define POINT_HPP_ 1

class Point {
 public:
    Point(int x, int y);
    void move(int delta_x, int delta_y);
    int getX() const;

 private:
    int x_;
    int y_;
};

#endif
```

`point.cpp`

```
#include "point.hpp"

Point::Point(int x, int y) {
    x_ = x;    // Correct, but not
    y_ = y;    //   preferred C++ way
}

void Point::move(int deltaX, int deltaY) {
    x_ += deltaX;
    y_ += deltaY;
}

int Point::getX() const {
    return x_;
}
```

What’s on the stack at the return?

~~~
```c++
#include "point.hpp"

int main()
{
Point p1{30,40};    // "new" syntax
Point p2(50,60);    // "old" syntax

Point p3{p2};       // "Copy" constructor
Point p4(p2);       // "Copy" constructor
Point p5 = p2;      // "Copy" constructor (!)

p2.move(5, -5);

return 0;
}
```
~~~