Syntax and Semantics

CS 1044

Structure and Meaning

- The **validity** of a computer program is defined by two things:
  - **Syntax** – Can the compiler make sense of what you wrote? Is it **structured correctly**?
  - **Semantics** – Does it generate the results you expected? Does it **mean** what you think it does?

Syntax

- **Syntax** determines whether a piece of text is a **valid C++ program**
- Similar to English syntax:
  - "The dog went up the hill." **Syntactically correct**
  - "Hill dog; up. the went the" **Syntactically incorrect**

Syntax

- Just like a natural language, C++ has a **grammar** that you must abide by
  - For example, executable statements must end with a semicolon
- Compiling code that does not match the grammar results in **syntax errors**

Semantics

- **Semantics** are the **meaning** or **behavior** of a program
- A program, like an English sentence, can be **syntactically correct** but have a **different meaning** than you might intend

Panda:
Eats shoots and leaves

Semantics

- **Semantics** are the **meaning** or **behavior** of a program
- A program, like an English sentence, can be **syntactically correct** but have a **different meaning** than you might intend

Panda:
Eats, shoots and leaves

Syntax vs. Semantic Errors

- **Syntax** errors are typically **easy** to find
  1. Try to compile your code
  2. Did it yell at you? Then you've got syntax errors
- **Semantic** errors are **much harder** to detect
  - If your program produces incorrect results, you might have to dig to find exactly where the error occurred
  - Semantic errors also called **logical errors**

Basic Program Structure

- `#include` statements to import features we want to use in our programs
- Other “bookkeeping” directives (`using namespace`...)
- **Declarations** of data types and functions that you have written in your program
- **Definitions** of functions declared above

Declaration vs. Definition

- **Declaration** – a “hint” to the compiler to say “this is what something looks like; it will be defined later”
- **Definition** – the actual code associated with something, like a function
- Functions must be declared before they’re used, but not necessarily defined before then
- This is more important later on

Namespaces

- **Namespaces** are like “folders” that contain data types and functions
- Intended to reduce the chance of programmers writing code that used the **same names** to mean **different things**
- Most built-in C++ features live in the `std` namespace

Namespaces

- You “drill down” into namespaces with the `::` (double-colon) symbol
  - `std::cout << “Hello world” << std::endl;`
- This gets tedious after a while!
- Write “using namespace...” as a shortcut

  -
    ```
    using namespace std;
    cout << “Hello world” << endl;
    ```

Putting it all together

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```