C++
I/O

Michael Burrell

January 8, 2020

Readings for this set of slides

- Chapter 2.3
- Chapter 4.1
- Chapter 4.10

Includes and namespaces

- We're not reading to talk about `#include` and namespaces in detail yet
  - `#include` is coming up very soon
  - Namespaces we'll talk about a few weeks later
- We should at least understand them enough to get an idea of how "Hello world" works

No standard prelude

- Unlike many modern languages, C and C++ do not have a "standard prelude"
- When the compiler is run, it has *no* knowledge of *any* symbols at all
  - *Symbol* in this context refers to type definitions, classes, methods, objects, variables or constants
- This is in contrast to modern languages like Java
  - In Java, every symbol in the `java.lang` package is part of the "standard prelude"
  - It is implicitly imported into every Java file
- Note that *keywords* like `int` are not symbols

Declaring foreign symbols

- In C++, it is possible to declare foreign symbols (symbols defined in some other library, like the standard library) manually
- We will see how to do this in coming weeks
- It can get quite tedious to do this manually for every symbol we want to use
- The standard C++ library gives us *header files* (we can call them *include* files for now)
  - These files contain a list of some of the symbols that are defined in the standard library

Two different types of includes

- In lab 1, we saw two different types of includes
  - #include "stdafx.h" used quotation marks
  - #include ¡iostream¿ used angle brackets
- Quotation-mark header files are header files that are defined within your project (not part of the standard library)
- Angle-bracket header files are header files that are external to your project (standard library or some 3rd party library)

Our two header files from lab 1

stdafx.h — honestly, this header file is annoying and doesn't really do anything. It is a requirement for some Visual Studio projects (depending on your version of Visual Studio and how the project was created). Nobody other than Visual Studio has any concept of this header file

iostream — part of standard C++. Defines all symbols needed for "I/O streams"

- We use I/O streams for console input/output, so this header file is very useful!
- It defines `cout`, `endl`, `cin`, and more

Namespaces

- For those of you who have taken CPSC 1181, namespaces are similar in concept to packages
  - They are not *exactly* the same, however
- Namespaces/packages both allow us to organize symbols together
  - They also prevent symbols from conflicting with one another that have the same name
  - E.g., `java.util.Date` is distinct from `java.sql.Date`

Java time

Here's how most of you were probably taught to write Java that involves symbols from other packages.

```
1  import java.util.Scanner;
2
3  public class StupidExercise {
4      public static void main(String[] args) {
5          Scanner s = new Scanner(System.in);
6          System.out.print("Enter␣your␣name:␣");
7          String name = s.nextLine();
8          System.out.println("Sup,␣" + name + "?");
9      }
10 }
```

Java time

Here's a totally equivalent way of doing the same thing:

```
1  public class StupidExercise {
2      public static void main(String[] args) {
3          java.util.Scanner s = new
               java.util.Scanner(System.in);
4          System.out.print("Enter␣your␣name:␣");
5          String name = s.nextLine();
6          System.out.println("Sup,␣" + name + "?");
7      }
8  }
```

You are *never* required to write an `import` in Java!!

Java time

Here's a totally equivalent way of doing the same thing:

```
1 public class StupidExercise {
2     public static void main(String[] args) {
3         java.util.Scanner s = new
              java.util.Scanner(System.in);
4         System.out.print("Enter␣your␣name:␣");
5         String name = s.nextLine();
6         System.out.println("Sup,␣" + name + "?");
7     }
8 }
```

However, if you don't give an `import`, symbols from outside `java.lang` must always be "fully qualified", prefixed with the package name.

Namespaces in C++

- Symbols in C++ work similarly
- There is a concept of a "fully qualified" name for a symbol in C++
  - It includes both the name of the namespace and the name of the symbol
  - In C++, we use `::` as a separator after namespaces, unlike `.` in Java

An example with fully qualified names in C++


```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Hello world" << std::endl;
6      return 0;
7  }
```

- Note the `#include` is necessary for bringing the `cout` and `endl` symbols into scope
- We still need to (by default) use the fully qualified name for any imported symbols

Using using


```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     cout << "Hello world" << endl;
6     return 0;
7 }
```

- C++ offers a mechanism (called `using`) which functions *sort of kind of a little bit* like an `import` in Java
- It allows us to use the bare symbol name instead of the fully qualified name

include and namespace roundup

- Always use `#include <iostream>` and know that it's necessary for doing I/O
- Use `#include "stdafx.h"` if you're using Visual Studio on the lab computers because Visual Studio is super annoying
- Use or don't use `using namespace std;`
  - Know that it allows you to skip giving the fully qualified name of some symbols
  - Later on in the course, I will care when and how you use it
  - For now, I don't care: do whatever you're comfortable with

cout

- `cout`[1] is a standard object for printing character data to standard output
- It is an instance of the `ostream` class
- We have seen that we can use the << operator to print things out with it
- Note that it makes no difference how we break up printing into separate operations

[1] "character out"

Breaking up cout


```
cout << "hi" << 3 << endl
     << 9 << "boo";
```

```
1 cout << "hi";
2 cout << 3;
3 cout << endl;
4 cout << 9;
5 cout << "boo";
```

These are equivalent.

endl

- `endl` is a "basic I/O stream"
  - It is one of the types of objects which can be printed out with an `ostream` like `cout`
- It does two things:
  1. It prints out the newline `'\n'` character
  2. It flushes the stream, just as if you did `cout.flush()`
- It is the preferred way to end lines in C++ because `'\n'` by itself is not guaranteed to flush the stream

Equivalence of methods and shift operator

- By convention, we (almost) always use << with `ostream` objects rather than calling methods directly
- If you find it clearer, you can usually use a method to do the same thing
- E.g., `cout << flush` is totally equivalent to `cout.flush()`
- We will generally follow C++ convention of using << everywhere

Formatting

- Formatting in C++ can be done with a few flags that are in the `std` namespace

  left, right, internal — indicating horizontal alignment for upcoming items to print

  dec, oct, hex — which base to use when printing integers

  scientific, fixed — which format to use when printing floating-point numbers

- We can also modify formatting with the following *methods*:

  width — sets the width of the upcoming items to print

  precision — number of digits after the decimal point, for floating-point numbers

Demo

Screwing around

Let's screw around with cout formatting.

cin

- `cin`[2] is the counterpart to `stdout`
- It can read in items from standard input in a variety of different ways
- It is an object of the `istream` class

[2] "character in"

cin


```
1 int x, z;
2 double y;
3 cin >> x >> y >> hex >> z;
```

Similarly to `cout`, we can inject things like `dec`, `oct`, `hex` to change how values are interpreted.

ignore

- Much like `Scanner` in Java, sometimes `cin`'s buffer has characters that you don't want to consider
- E.g., there might be a newline character hanging around that want to ignore
- `cin` has an `ignore` method that will allow you to do this

ignore

```
1 int x;
2 string z;
3 cin >> x;
4 cin.ignore(numeric_limits<streamsize>::max(), '\n');
5 cin >> z;
```

ignore has two parameters: the maximum number of characters to ignore, and the type of character to stop ignoring at.

numeric_limits

- On the previous slide, we say `numeric_limits<streamsize>::max()`
- This is using an advanced C++ feature called templating
  - We won't learn about this until later in the course
- Just take it on faith for now that `numeric_limits<streamsize>::max()` means "ignore the maximum number of characters possible"

Checking return values

- C++ *has* exceptions
- C++ programmers almost never use them
- Exceptions have a serious performance cost associated with them
- Instead of relying on exceptions, we check return values to see if something worked

Did cin work?


```
1 int x;
2 cin >> x;
3 if (cin) {
4     cout << "You␣entered␣in␣" << x;
5 } else {
6     cout << "That␣was␣not␣a␣valid␣integer";
7 }
```

We can treat `cin` as if it were a `bool` to check if it's in an *error state* or not.

Error states

- If an `istream` enters an error state (due to bad input, end of file, keyboard unplugged, etc.) it *will not work* until its error state is *cleared*
- We can clear an error state using the `clear` method

Demo

Screwing around

Let's screw around with input.

What we learned

- Includes are necessary for bringing in symbols
- `using namespace` is sometimes convenient for not giving a fully qualified name of a symbol
- Formatted output in `cout`
- Input with `cin`