I/O

C++ intro

Michael Burrell

March 5, 2024

Readings for this set of slides

Chapter 1 — 1.2

Chapter 8 — 8.1

Includes and namespaces

- We're not ready to talk about `#include` and namespaces in detail yet
  - `#include` is coming up very soon
  - Namespaces will be explained even sooner
- 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 Python
  - In Python, many symbols (e.g., `print`, `input`) are part of the "standard prelude"
  - They are implicitly imported into every Python file
- Note that *keywords* like `int` are not symbols and are known to the compiler innately

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

- There are two different types of includes in C++
  - E.g., `#include "point.h"` with quotation marks
  - E.g., `#include <iostream>` with 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 header file from the first week

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

- Namespaces in C++ has some similarity to modules in Python
- They are a way to organize symbols (classes, variables)
- E.g., `cout` is a symbol which exists within the `std` namespace
- `std::cout` is the *fully-qualified name* of that symbol
- The `::` is used as a separator between namespace name and symbol name

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 brings in all symbols from a namespace
- It allows us to use the bare symbol name instead of the fully qualified name

Using using


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

- This is also legal (and common) in C++
- `using` follows the same scoping rules as variables
- In C++, we generally try to restrict `using` to the smallest scope reasonable

include and namespace roundup

- Always use `#include <iostream>` and know that it's necessary for doing I/O
- Use or don't use `using namespace std;`
  - Know that it allows you to skip giving the fully qualified name of some symbols
  - Try to use it only when necessary, and in a small scope
  - Later on in the course, I will care more when and how you use it

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 common 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 item to print

  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

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 >> z;
```

ignore

- Much like `input` in Python, 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(SSIZE_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, I used `SSIZE_MAX` as the maximum number of characters to ignore
- We want to say "ignore *any* number of characters up to the next newline", which has to be stated as "ignore the maximum number of characters up to the next newline"
- `SSIZE_MAX` will *probably* be correct, but in production code, you should use `numeric_limits<streamsize>::max()` instead
  - This is using an advanced C++ feature called templating
  - In this course, you are permitted to use `SSIZE_MAX` instead

Checking return values

- C++ *has* exceptions
- C++ programmers almost never use them
- Exceptions have a 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      cin.clear();
8      cin.ignore(SSIZE_MAX, '\n');
9  }
```

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

Clearing the buffer

- The `clear` method does *not* clear the buffer
- The `clear` method does NOT clear the buffer
- **The `clear` method does NOT clear the buffer**
- The `clear` method only transitions `cin` from an error state to a normal state
- Use the `ignore` method after the `clear` method to clear out the input buffer

Repeatedly reading in

Class exercise

Let's write a C++ program which repeatedly asks the user for input until the user enters in a number.

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`
- Use `if (cin)` to check if input was read successfully, then `clear+ignore` if it wasn't