C++
Core language

Michael Burrell

January 14, 2020

Textbook readings

- Chapter 1
- Chapter 2

Goals for this set of slides

- Understand how to break up problems using `if`, `else`, `while`, `do`, and `switch`
- Understand type inference in C++

More history

- One of the earliest divergences between C and C++ is over the use of booleans
- From 1970 to 1999, C did not have *any* booleans
- In contrast, C++ had booleans right from the 1980s
- This is important because it changes how we think about conditions (e.g., `if` statements)

Booleans in C++

- The word `bool` is a keyword (reserved word) in C++
  - As are `true` and `false`
- It exists outside of the usual integer type hierarchy
  - Its representation is completely implementation-defined
  - Most commonly, it is represented as a single byte (`sizeof (bool)` is very often 1)
- However, it *is* an integer type, of sorts. . . .

Integers and booleans

- To maintain better compatibility with C (which historically didn't have a `bool` type), C++ treats `bool`s as integers

  false — is defined to be 0

  true — is defined to be 1
- An integer will be implicitly converted into a `boolean`
  - Any non-zero value will be interpreted to be `true`
- Arithmetic (-, +, --, ++, etc.) is possible on `bool`s, too, though discouraged

Idiomatic C++


```
int num_factors(unsigned int x)
{
    if (!x) {
        return 0;
    }
    int c = 1;
    for (unsigned int i = 2; i < x; i++) {
        if (x % i == 0) {
            c++;
            x /= i;
        }
    }
    return c;
}
```

Note the use of `if (!c)`

Integers as booleans

- Many C++ programmers (especially those who also use C) will idiomatically use integers as if they were booleans and vice versa
  - Also with pointers, which we'll see before long
- The behaviour that 0=false and anything-other-than-0=true is well-defined and usually a safe thing to take advantage of
- Just make sure that your code is clear and understandable

Most structures are the same

- `if`, `while`, `do` all work the same in C++ as they do in Java
- Like in Java, an `else` is possible, and `else ifs` may be chained together indefinitely
- Like in Java, curly braces are optional if there is only a single statement in the body of the control structure

Boolean operators are the same

- All of the boolean operations are the same in C++ as they are in Java
- `<`, `>`, `<=`. `>=`, `==`, `!=`, `&&`, `||`, `? :`, `!`, etc.
- Just be aware of the fact that the result of a boolean expression could be turned into an integer at any moment
  - E.g., `int x = (y < z) * 10;`

For loops

- Basic for loops (`for ( ; ; )`) are the same in C++ as they are in Java
- For-each loops (*enhanced for loops* in Java, `range-based for loops` in C++) are different though!
  - C++ does not have the concept of an `Iterable` interface like Java does
  - For-each loops in C++ are considerably more flexible and complex
  - Even with arrays, C++ for-each loops offer a lot of flexibility
  - We will look at these when we discuss pointers

Switch statements

- The basics of `switch` statements are the same between C++ and Java
- The difference is that C++ `switch` statements may *only* be used with integer constants
  - Strings may not be used
- "Integer constants" includes enumerations, which we'll discuss later in the course

Conclusion of control flow

- Use basic `if`, `for`, `while`, etc., as you would in Java
- Be aware of the fact that integers and booleans are interchangeable
- false=0, true=1, 0=false, non-zero=true

Scope

- Scope of variables works the same as in Java
  - Curly-braces demark the scope of a variable
  - Variables are deallocated when they fall out of scope
- C++ has globals (declared outside of any scope), which Java doesn't have
  - The `static` keyword can be used to turn a global variable into a variable accessible only within the current file

Type inference

- In C++11, we were introduced to *type inference*
- This was expanded in C++14
- Type inference may be used for local variables and return types (and lambdas, which we don't know about yet), but *not* function parameters
- With type inference, we declare the type of the variable to be `auto` and the compiler will infer its real type based on first-usage
  - Note this is still static typing
  - The variable still has a fixed (unchangeable) type

Example


```
auto foo(int x, double y) {
    auto z = "";
    for (auto i = 0; i < x; i++) {
        auto j = y * 2;
        z += '0' + (int)j;
    }
    return z;
}
```

C++ with a minimum of typing information given.

Guidelines for usage

- Taking advantage of type inference too much can hamper readability
- Use type inference when the name of a type will be very long or complicated
- Use type inference in a small scope
- Type inference can be used for the return types for function *definitions*, but not function *prototypes*

decltype

- `decltype` can be used in more complex situations
- `decltype` is a type specifier which can take any expression as an argument
- It evaluates to the type of that expression

decltype examples


```cpp
auto x = 2L;
auto y = 3.12'34f;
decltype(x + y) z;
cout << sizeof (decltype(z + foo(y)));
```

decltype example


```
for (auto i = decltype(n)(0); i < n; i++) {
    cout << xs[n] << endl;
}
```

This code now does not depend on the type of `n`.
If the type of `n` changes (from `int` to `long` or whatever), the types of the values used in the loop will automatically change to match it.

Conclusion

- Variables work generally as they do in Java or C
- We have seen that type inference can reduce code in some instances
- Be careful not to use type inference too often