SI 413, Unit 9: Control

Daniel S. Roche (roche@usna.edu)

Fall 2018

In our final unit, we will look at the big picture. Within a program, the "big picture" means how the program flows from line to line during its execution. We'll also see some of the coolest and most advanced features of modern languages.

We've talked about many different aspects of computer programs in some detail - functions, scopes, variables, types. But how does a running program move between these different elements? What features control the path of a program's execution through the code?

These features are what constitutes the *control flow* in a programming language, and different languages support different methods of control flow.

The concept of control flow in very early computer languages, and in modern assembly languages, is termed "unstructured" programming and consists only of sequential execution and branches (i.e., goto statements).

More modern programming languages feature the control structures that you have come to know and love: *selection* (if/else and switch statements), *iteration* (while, for, and for/each loops), and *recursion* (function calls).

1 Unstructured programming

The most basic control flow method is so ingrained in our understanding of how programs work that you might not even notice it: *Sequencing* is the process by which control flows continually to the next instruction. This comes from the earliest programs, which is still seen in assembly languages today with the *program counter* (address of the next instruction) that automatically gets incremented by 1 at each execution step.

In most programming languages, sequencing happens automatically without any special instructions. However, some functional languages in particular require special constructs to indicate that control flow should be sequential in some part. In Scheme, for example, recall the begin statement that did exactly this.

Besides sequencing, the other primary component of unstructured programming is *branching*, or GOTO statements. These again come from machine code instructions to either jump directly to another place in the program, or conditionally jump, based on the value of some certain register.

GOTOs have been largely superseded today by their structured-programming variants, However, the statement is still supported by C++ and may even be useful at times. In class, we looked at the example problem of printing the contents of a vector, separated by commas. The standard approach would be to have an extra test and print the first or last element outside of the main loop, like:

```
vector<int> v;
// ... v gets filled ...
if (v.size() > 0) cout << v[0];
int i = 1
while (i < v.size()) {
  cout << ", " << v[i];
  ++i;
}
```

This works, but has a stylistic shortcoming: the code to print out vector elements is now duplicated on two separate lines. If we wanted to, for example, change this code to print to a file instead of cout, one of the two required changes might be missed.

To avoid that problem, we could also solve the problem like this:

```
vector<int> v;
// ... v gets filled ...
int i = 0
while (i < v.size()) {
  if (i > 0) cout << ", ";
  cout << v[i];
  ++i;
}
```

the disadvantage here is that an extra check for whether i is greater than 0 is required at every loop iteration. There are a few other ways to solve this problem, but here is an alternative using a goto:

```
vector<int> v;
// ... v gets filled ...
int i = 0
if (i < v.size()) goto middle;
while (i < v.size()) {
  cout << ", ";
middle:
  cout << v[i];
  ++i;
}
```

Observe that the goto jumps into the middle of the loop. This solution seems to have neither shortcoming of previous approaches, although it may look somewhat awkward if you're not used to seeing GOTO statements in C++!

The point here is that GOTOs can be useful sometimes. However, they can also be very easily misused, and this is why they have somewhat fallen out of favor with many computer scientists. In class, we saw a big example of "spaghetti code", so named because all the GOTO links in the code form a confusing mess that is very difficult to decipher, understand, or debug. The basic problem is that code laced with numerous GOTOs can be very hard to maintain and to understand, making such programs unusable for large-scale projects.

2 Basic structured programming

In 1968, Edsger Dijkstra published a ground-breaking article titled "GOTO Statement Considered Harmful". You can actually read the entire article at this link. This short "letter to the editor" launched a whole change in the programming landscape, ushering in a new era of structured programming (as opposed to the unstructured GOTOs) that persists today. Dijkstra's main complaint was the proliferation of unorganized, unmaintainable spaghetti code like we mentioned above. So what's the alternative?

Structured programming consists of selection, iteration, and function calls (supporting recursion). I think you already know quite a bit about these, so I'm not going to spend much more time here. You should be aware that *all* of these are converted by compilers into GOTOs and branches in machine code, so the basic speed of structured programs doesn't necessarily have to suffer, although they will have advantages like proper scoping that can lead to greater code maintainability and reuse. Hooray!

The switch statement is an example of a structured programming construct that, at least in C/C++, is just a dressed-up GOTO statement. For example, the print-with-commas code above can be re-written as:

```
vector<int> v;
```

```
// ... v gets filled ...
int i = 0
switch(v.empty()) {
  while (i < v.size()) {
    cout << ", ";
  case false::
    cout << v[i];
    ++i;
  }
  case true:
}
```

Check to make sure you understand how this works for an empty vector, a one-element vector, or a three-element vector. Pretty cool, huh? This is sometimes called a "dirty switch" because of the way the cases are interleaved with another control structure (in this case, a while loop). But I like it anyway!

3 Advanced programming structures

If statements, loops, and functions were already well-understood by the time Dijkstra wrote his influential article in 1968. Since then, there have been a few more new concepts in control flow for programming languages. These are seen in more modern program languages.

First, consider the problem of looping over all the elements of a collection. For example, maybe we want to write a function sum that adds up all the integers in some data structure. The question is, how can we write this function once and have it work for any data structure?

Iterators

One solution to this is a special data structure called an *iterator*. An iterator is a kind of interface that allows for moving over all the elements in a collection: it needs to allow for getting the *next* element in sequence, and for indicating when there is no next element (when the collection has been exhausted).

C++ solves this problem by making an iterator an *abstraction* of a pointer type, so it supports incrementing with the ++ operator, and dereferencing with the ∗ operator. You have used C++ iterators in your labs, so you should know what these look like; see this page for some more exciting details.

In Java, there is an actual Iterator interface that has a next() method to get the next element, and a hasNext() method to tell whether we are at the end of the collection yet.

Python is fairly similar to Java: any iterator type must implement a special method called ___next___() that returns the next element. Python differs slightly in that the end of a collection is specified by raising a special exception called StopInteration, which doesn't require a separate method call.

For-each loops

Both Java and Python support another relatively new control-flow feature: the *for-each loop*. This is a loop that goes over every element in some collection, assigning each one to the same variable name and executing the body of the loop. For example, here's Python code to sum all the integers in some collection:

```
# Assume c is some collection, like a list or a dictionary
total = 0
for item in c:
    total += item
print(total)
```

The Java version is only slightly more verbose:

```
Collection<Integer> c;
int total = 0;
```

```
for (int item : c) {
  total += item;
}
System.out.println(total);
```

How do you think these work? That's right, iterators! In Python, the special function ___iter___() has to return an iterator object for any collection; in Java it's called iterator(). So this amazing-looking for-each loop is really just calling an iterator, but doing all the annoying work of getting the iterator, assigning each element to the name item, and checking when the iterator is empty, so you don't have to. How nice!

4 Generics

As we have seen, many of the more recent developments in programming languages have to do with programmer convenience: making it easier to write more maintainable code, and to avoid the need to copy the same chunks of code between different places in your program. Another development in this direction is the use of *generic code.*

This term has slightly different meanings depending on who you ask, but by generic code we mean code that can be used in an identical way to handle many different types. A simple example is a "min" function that takes two arguments and returns whichever one is smaller. What generic code helps us avoid is the need to write different versions of "min" for every different type of thing we might want to compare. Let's see how we could do this in a few different languages.

In Scheme, it's pretty darn easy:

```
(define (min a b)
  (if (< a b)
      a
      b))
```

The reason it's so simple in Scheme is that there are no declared types, and no static (compile-time) type checking. This basically makes all code generic code! The min function above can be used to find the smaller of two things, no matter what their types are, as long as the less-than operator works.

Of course it won't always be quite so simple. Here's a generic min in C++ *using templates*:

```
template<class T>
T min(T a, T b) {
  if (a < b) return a;
  else return b;
}
```

C++ definitely *does* have declared types and static-time type checking. So how does that work? What C++ templates do is create *copies* of the code in question. In this case, at compile-time, multiple copies of the min function written above will be made for every type T that min gets called on! So for example, if our code uses min to compare ints sometimes, and to compare strings some other times, then *at compile-time* there will be two min functions created: one for ints, and one for strings. This is why it's called "template" programming: all we are really writing is a *template* for the code that's going to get generated by the compiler.

The C++ template approach to allowing generics by code generation is an extremely powerful feature - in fact, whole books have been written on the subject of template programming in C++! This is a form of *metaprogramming*, because it's telling the compiler to write new code for you, and then to compile that code that it just wrote! The downside is that there is a potential for "code bloat", where the size of the executable could get really large if the C++ template mechanism has to generate many, many copies of different classes and functions for the various template type parameters. Fortunately, this won't happen if you only use templates in the simple way shown above. But believe me, it's possible!

The "Java way" of getting generic code looks very similar to the C++ approach:

```
static <T extends Comparable<T>> T min(T a, T b) {
  if (a.compareTo(b) < 0) return a;
  else return b;
}
```

(Note that the Comparable interface is what specifies the compareTo function. This is necessary because Java doesn't allow you to overload operators like <.)

So Java uses angle brackets and type names just like C++ templates. Are they the same "under the hood"? Somewhat surprisingly, *Java generics are completely different from C++ templates.*

The way Java deals with the issue is much simpler: at compile-time, all the type checking is done according to the template parameters, filling in a name like T for whatever type the function is being called on. This is similar to C++, but where they differ is that Java does *not* make any copies of the generic method or class. Instead, after doing the static type checking, the Java compiler does something called *type erasure*; basically, all of the template types like T revert back to Object, which you know is the super-class of every type in Java. So at run-time, there is only a single copy of any generic class or method, and it has Object types instead of T (or whatever else the generic type was called). This is only possible because Java actually has the class hierarchy where Object is the superclass of everything, unlike C++.

The advantages to Java's approach are that it is simpler and "cleaner" to implement, it makes the compiler run faster, and (importantly) it can never make the size of the compiled code "bloat" up too much. But the fact that no code generation occurs also limits the capabilities. For example, Java generics do not allow one to create new instances of the generic type. The reason is that something like making a new instance requires that type to be known *at run-time*. But with Java generics, the type information is erased at run-time, making that kind of thing impossible!

So in summary, there are 3 options for getting generic code in a language:

- If there are no declared types and no static type checking (like in Scheme), generic code is never any problem, and doesn't require any special syntax.
- In a language like C++, *code generation* (through C++ templates) can make *copies* of the generic code at compile-time. This is extremely powerful and can allow you to do almost anything, but can be somewhat difficult to get right, and in nasty cases can also lead to very large executable files.
- In a language like Java, *type erasure* allows the generic types to be checked at compile-time, but still only have a single copy of the method or class in the compiled file. This results in smaller code and less confusing errors, but lacks some of the advanced functionality of template programming.

In the slides, we also looked at how to make generic classes in C++ and Java, specifically via a simple linked list class. Have a look!