Control Flow

The control flow of a program is the way an execution moves from statement to statement.

The textbook breaks it down into:
- Sequencing (do the next thing)
- Selection (pick something to do, e.g. if, switch)
- Iteration (repeat something, e.g. while, for)
- Recursion
- Unstructured (e.g. goto)

Unstructured flow: GOTO

In the beginning, there was GOTO. And GOTO was good.
- Directly jumps from one place (the goto) to another (the label)
- Corresponds exactly to machine code
- Very efficient
- Can cause some problems...

Good Use of Goto?

Say we want to print a vector, comma-separated, like “1, 2, 3”.

This solution prints an extra comma!

```cpp
vector<int> v;
// ...
int i = 0;
while (i < v.size()) {
    cout << v[i] << ",";
    ++i;
}
cout << endl;
```
Goto Problems

- They don’t play well with scopes.
  (Restricting to local gotos avoids this.)

- Can be used to cook up “spaghetti code” — hard to follow.

- Hard to know where we are in the program,
  i.e., hard to reason about the program’s correctness/performance.

```c
int x = 0;
char c;
goto rs;

fns :
  if (c != '1' && c != '0') goto er;
goto ns;
rd:
  c = getchar();
ns:
  if (c == '1') { x = x*2 + 1; goto rd; }
  if (c == '0') { x = x*2; goto rd; }
es:
  if (c == '_')
    { c = getchar();
      goto es;
    }
  if (c == '\n') goto done;
er:
  printf("Error!\n");
  return 1;
rs:
  c = getchar();
  if (c == '_') goto rs;
  else goto fns;
done:
  printf("%i\n",x);
```

Structured Programming

*Structured programming* is probably all you have ever known.

Championed by Dijkstra in the pioneering paper “GOTO Statement Considered Harmful” (1968).

Structured programming uses control structures such as functions, *if*, *while*, *for*, etc., even though these are mostly compiled into *gotos*.

Allows us to reason about programs, enforce modularity, write bigger and better programs.
Looping over a Collection

How would you write C++ code to loop over the elements of
- an array?
- a linked list?
- a binary search tree?

How can we separate interface from implementation?

Iterators

An iterator needs to be able to:
- Get initialized over a collection.
- Move forward (maybe backwards?) through a collection.
- Fetch the current element
- Know when it’s done.

In C++, an iterator overrides ++ and * to become an abstract pointer.
In most other languages (e.g., Java), an iterator has to extend an abstract base type with next() and hasNext() methods.

For-Each Loops

A for-each loop provides an even easier way to loop over the elements of a collection.

Java example:
```
HashSet<String> hs;
// ...
for (String s : hs) {
    System.out.println(s);
    // This prints out all the strings in the HashSet.
}
```

This construct is supported by most modern languages.
Often there is a direct connection with iterators.
In some languages (e.g., Python), this is the only for loop.
Dirty Switches

**switch** statements blur the line between structured and unstructured programming.

Here’s my favorite solution to the “print with commas” problem:

```cpp
vector<int> v;
// ...
int i = 0;
switch(v.empty()) {
    for (; i < v.size(); ++i) {
        cout << " , ";
        case false:
            cout << v[i];
        }
    }
    cout << endl;
```

Advanced Topics

There’s a lot more we could talk about!

- **Unwinding** (for inside-out **goto**s)
- **Jumping out of a loop** (**break**, **continue**)
- **Labeled breaks**
- **Generators**

Class outcomes

You should know:

- What structured vs unstructured programmings is.
- Structured programming constructs:
  - sequencing, selection, iteration, recursion
- Why **GOTO**s might be “considered harmful”
- Why **GOTO**s are useful sometimes
- What an iterator is, and where/how/why they are used.
- What a for-each loop is, and where/how/why they are used.