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 with commas like "`1, 2, 3`".

This solution prints an extra comma!

```
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.


```
 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 `goto`s.

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:

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


Aside: Scripting Languages

bash, Ruby, Python, Pearl, and PHP are examples of *scripting languges*.
They are designed for *small tasks* that involve coordination or communication with other programs.

Common properties:

- Interpreted, with dynamic typing
- Emphasis on *expressivity* and *ease of programming* over efficiency
- Allows multiple paradigms (functional, imperative, object-oriented)
- Built-in string handling, data types
- Extensive "shortcut" syntactic constructs

Scripting example: Prime generation in Python

```
def PrimeGen():
  for p in itertools.count(2):
    if all(p%i != 0 for i in range(2,p)):
      yield p

for p in PrimeGen():
  if p < 100: print(p)
  else: break
```

Generators

Sometimes a function computes multiple values as it goes along.

An iterator created automatically from such a function is called a *generator*

Simpler (related) Python example:

```
def factors(n):
  for i in range(2,n):
    if n % i == 0: yield i
```


The Need for Generic Code

A *function* is an abstraction of similar behavior with *different values*.

Generic code takes this to the next level, by abstracting similar functions (or classes) with *different types*.

Most common usages:

- Basic functions: min/max, sorting
- Collections: vector, linked list, hash table, etc.

Genericity in Scheme

In Scheme and other languages with *run-time type checking*, writing generic functions is (mostly) trivial.

Generic minimum function:

```
(define (minimum a b)
  (if (<= a b) a b))
```

Generic binary tree structure:

```
(define (make-bt ele left right)
  (lambda (command)
    (cond [(symbol=? command 'left) left]
          [(symbol=? command 'right) right]
          [(symbol=? command 'root) ele])))

(define BST (make-bt 4 (make-bt 2 (make-bt 1 null null)
                                  (make-bt 3 null null))
                       (make-bt 6 (make-bt 5 null null)
                                  (make-bt 7 null null))))
```

Genericity in C++

Old School (C style)

- Use *function-like macros* to code-generate every possibility.
- Types to be used in generic functions/classes must be explicitly specified.

Templates (C++ style)

- Built into the language; don't rely on preprocessor
- Compiler does code generation, similar to macros
- Types to be used are determined *implicitly* at compile-time
- *Separate compilation* becomes difficult or impossible.

```
#define WRITE_LL_CLASS(T) \
class Node_ ## T { \
  public: \
    T data; \
    Node_ ## T * next; \
    Node_ ## T (T d, Node_ ## T * n) :data(d), next(n) { } \
\
    T printAndSum() { \
      cout << data << endl; \
      if (next == NULL) return data; \
      else return data + next->printAndSum(); \
    } \
};

WRITE_LL_CLASS(float)
WRITE_LL_CLASS(int)

int main() {
  Node_float* fhead = NULL;
  Node_int* ihead = NULL;

  // ... fill the lists with some input

  cout << "Floating sum: " << fhead->printAndSum() << endl << endl;
  cout << "Int sum: " << ihead->printAndSum() << endl << endl;
}
```

```
template <class T>
class Node {
  public:
    T data;
    Node<T> * next;
    Node<T> (T d, Node<T> * n) :data(d), next(n) { }

    T printAndSum() {
      cout << data << endl;
      if (next == NULL) return data;
      else return data + next->printAndSum();
    }
};

int main() {
  Node<float>* fhead = NULL;
  Node<int>* ihead = NULL;

  // ... fill the lists with some input

  cout << "Floating sum: " << fhead->printAndSum() << endl << endl;
  cout << "Int sum: " << ihead->printAndSum() << endl << endl;
  return 0;
}
```

Genericity in Java

Old School (Java $\leq$ 1.4)

- Use abstract base classes/interfaces like `Object`
- Make extensive use of polymorphism
- Lots of *upcasting* and *downcasting*

Generics (Java $\geq$ 5)

- Similar *syntax* to C++ templates
- Compiler checks type safety then *removes* generic type information
- Up/downcasting still performed, implicitly
- Generics are only *syntactic sugar*

Manual Genericity in Java

```
interface Sum { void add(Number x); }

class FloatSum implements Sum {
  float val = 0;
  public void add(Number x)
  { val += ((Float)x).floatValue(); }
  public String toString() { return String.valueOf(val); }
}

class IntSum implements Sum {
  int val = 0;
  public void add(Number x)
  { val += ((Integer)x).intValue(); }
  public String toString() { return String.valueOf(val); }
}
```



```
class LLOld {
  Number data;
  LLOld next;

  LLOld(Number d, LLOld n) { data = d; next = n; }

  Sum printAndSum(Sum summer) {
    System.out.println(data);
    summer.add(data);
    if (next != null) next.printAndSum(summer);
    return summer;
  }

  public static void main(String[] args) {
    LLOld flist = null;
    LLOld ilist = null;

    // ...  fill the lists with some input

    System.out.println("Floating␣sum:␣" +
      flist.printAndSum(new FloatSum()) + "\n");
    System.out.println("Integer␣sum:␣" +
      ilist.printAndSum(new IntSum()) + "\n");
  }
}
```

Java 5 Generics

```
interface Sum<T> { void add(T x); }

class FloatSum implements Sum<Float> {
  float val = 0;
  public void add(Float x)
  { val += x.floatValue(); }
  public String toString() { return String.valueOf(val); }
}

class IntSum implements Sum<Integer> {
  int val = 0;
  public void add(Integer x)
  { val += x.intValue(); }
  public String toString() { return String.valueOf(val); }
}
```



```
class LLNew<T> {
  T data;
  LLNew<T> next;

  LLNew(T d, LLNew<T> n) { data = d; next = n; }

  Sum<T> printAndSum(Sum<T> summer) {
    System.out.println(data);
    summer.add(data);
    if (next != null) next.printAndSum(summer);
    return summer;
  }

  public static void main(String[] args) {
    LLNew<Float> flist = null;
    LLNew<Integer> ilist = null;

    // ... fill the lists with some input

    System.out.println("Floating sum: " +
      flist.printAndSum(new FloatSum()) + "\n");
    System.out.println("Integer sum: " +
      ilist.printAndSum(new IntSum()) + "\n");
  }
}
```


Trade-Offs in Generics

- **No declared types**
  - No *enforced* notion of “list of integers” etc.
  - Requires dynamic typing; slower
- **Code Generation** (C++ templates)
  - Can result in (combinatorial!) code explosion
  - Very powerful and general, but somewhat unintuitive
- **Code Annotation** (Java 5 generics)
  - Syntactic sugar; extensive run-time casting results
  - Types not known to the program at runtime — eliminates some capabilities

Permissions

What things might you want to do with *software* (binary) or with its *source code*?

Non-Free Licenses

- **Proprietary**
- **Shareware**
- **Freeware**

Free Software Licenses

- **GPL**
- **LGPL**
- **Permissive**

Case Studies

- Template programming and the LGPL
- Linux on the Kindle
- Iceweasel
- Vernor v. Autodesk
- "Badgeware" and CPAL

Class outcomes

You should know:

- What structured vs unstructured programmings is.
- The goods and bads of GOTOs
- What an iterator is, and where/how/why they are used.
- What a for-each loop is, and where/how/why they are used.
- What a scripting language is
- What a generator is
- What a generic class/function is
- How genericity works in C++ and Java
- Major types of software licenses
- What copyleft is and why it matters