EECS 490 – Lecture 3

Control Flow

Announcements

- Homework 1 due Friday at 8pm
- Project 1 due Wednesday 9/20

Agenda

- Sequencing
- Unstructured Control
- Structured Control
- Exceptions

Short Circuiting

- In many languages, logical operators are *short circuiting*, meaning that later operands are not evaluated if the overall expression value is already determined

```
if (x != 0 && foo(x)) {
    ...
}
```

May be expensive to execute, or require that x `!= 0`

- Ternary conditional operators also only evaluate the required operands

```
int y = (x != 0 ? z / x : 0);
```

Avoid division by zero

Sequencing Operator

- Some languages provide an operator for explicitly sequencing the evaluation of expressions
  - Generally, the result of the overall expression is the result of the last one

- C++ example:

```
int x = (3, 4);
cout << x;
```

Evaluates 3, throws it away; evaluates 4, initializes x to 4

- Scheme example:

```
(define x (begin (+ 1 3) (/ 4 2)))
(display x)
```

Sets x to 2

Compound Assignment

- A compound assignment differs from an "equivalent" assignment expression in that the left-hand side is only evaluated once

```
int array[10] = {};
int i = 0;
array[i++] += 2;
cout << i << endl;
array[i++] = array[i++] + 2;
cout << i << endl;
```

Prints 1

Prints 3

Statement Sequences

- Statements generally have side effects, so they must execute in some well-defined sequence[1]
- *Blocks* and *suites* consist of sequences of statements
- The language syntax determines how statements are separated or terminated
  - Separated by semicolon:

    `S_1; S_2; ... ; S_N`
  - Terminated by semicolon:

    `S_1; S_2; ... ; S_N;`

Trailing semicolon required

[1] Compilers/interpreters can reorder statements if they can guarantee that it won't change the semantics.

Gotos

- Some languages provide a mechanism for direct transfer of control in the form of a *goto*

Label

```
int x = 0;
LOOP: printf("%d\n", x);
x++;
goto LOOP;
```

Go to statement at given label

- Correspond to machine-level direct jumps

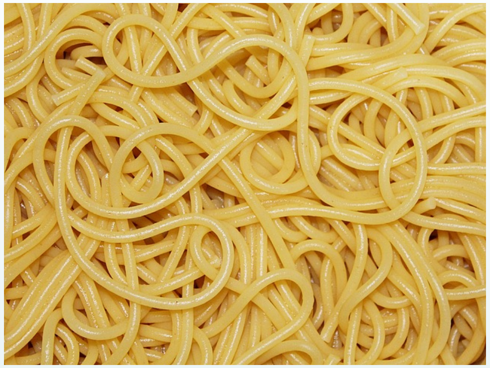
- Some languages provide a variant that can also branch

```
goto (10, 20, 30) i
```

Go to the ith label

Goto Problems

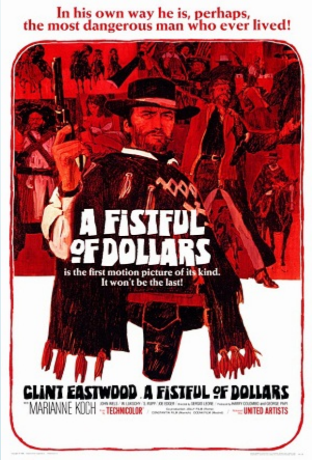
- Gotos are criticized for resulting in *spaghetti code*, code with a complex control structure that is difficult to follow



```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
```

vs.

```
10 FOR i = 1 TO 10
20      PRINT i; " squared = "; i * i
30 NEXT i
40 PRINT "Program Completed."
```



Conditionals

- Compound statement that expresses conditional execution

- General form:

```
if <test> then <statement1> else <statement2>
```

- In most languages, the else branch can be elided

```
if <test> then <statement>
```

Dangling Else

- In many languages, the syntax of conditionals results in a potential ambiguity

  ```
  if <test1> if <test2> then <stmt1> else <stmt2>
  ```

- Which if does the else belong to? This is called a *dangling else*

- The usual resolution is that an else belongs to the innermost possible if

Cascading Conditional

- Nested conditionals can get cumbersome and hard to follow, so languages often provide syntax for *cascading* conditionals where only one branch runs

- C/C++/Java:

```
if (<test1>) <statement1>
else if (<test2>) <statement2>
...
else if (<testN>) <statementN>
else <statementN+1>
```

Branches are checked in order

At most one branch runs

- Python uses "`elif`" instead of "`else if`"

Switch Statements

- A *switch* or *case* statement allows branching based on the value of a non-boolean expression

```
switch <expression>:
  case <value1>: <statement1>
  case <value2>: <statement2>
  ...
  case <valueN>: <statementN>
  default: <statementN+1>
```

Generally must be compile-time constant

- Many differences between languages
  - Can a default case be defined
  - Do the cases have to cover all possible values
  - Does execution "fall through" from one case to another
  - Can a single case cover multiple values

The example is pseudocode and not from a real language.

Loops

- Unbounded repetition is a necessary condition for a language to be Turing complete

- Some languages provide constructs for bounded loops, where the number of iterations is known at compile time or upon entry to the loop

- General form of unbounded loop:

```
while <expression> do <statement>
```

Loop Variants

- Repeat until:

```
do <statement> until <expression>
```

Executes at least once

Complements of each other

- Do while:

```
do <statement> while <expression>
```

- For loop:

```
for (<init>; <test>; <update>) <statement>
```

Foreach Loops

- Iterates over the elements of a sequence
- Also called "range-based for loop"
- Compiler determines initialization, test, and update

```
template <typename Container>
void print_all(const Container &values) {
  for (auto i : values) {
    cout << i << endl;
  }
}

def print_all(values):
    for i in values:
        print(i)
```

Loop Termination

- Sometimes it can be useful to terminate a loop early

```
bool found = false;
for (size_t i = 0; i < size; i++) {
  if (array[i] == value) {
    goto end; break;
  }
}
end: cout << "found? " << found;
```

- `break`: terminate loop and move to code after loop
- `continue`: terminate loop iteration and move to next iteration

Termination in Nested Loops

- What if we want to terminate an outer loop (or iteration) from an inner loop?
- In C or C++, must either use goto or refactor code
- Java has *labelled* `break/continue`

```
outer: for (...) {
  for (...) {
    if (...) break outer;
  }
}
```

Break Time!

- We'll start again in five minutes.

Exceptions

- Separate job of *detecting* errors from task of *handling* errors
  - May not have enough context at detection point to be able to recover

- Provide a *structured* mechanism for handling errors
  - Make it apparent in code what code an error handler covers and what kinds of errors it can handle

Overview of Exceptions

- Language provides:
  - Syntax for specifying what region of code a set of error handlers covers
  - Syntax for defining the error handlers for a region of code, and the kinds of exceptions each one can handle
  - A mechanism for *throwing* or *raising* an exception
- Optional: a means of defining new kinds of exceptions
  - Java: must subclass `Throwable`
  - Python: must be a subtype of `BaseException`
  - C++: can be any type

Raising an Exception

- An exception may be raised by library code or by the runtime
  - Example: divide by zero, file could not be opened

- An exception may also be manually raised

```
throw Exception();
```

raise in Python

Exception object (may be class rather than instance in Python)

Scoping of Exception Handlers

- Exception handlers are **dynamically** scoped

```
def foo():
    try:
        bar()
    except NotImplementedError:
        print('caught exception')

def bar():
    baz()

def baz():
    raise NotImplementedError('baz')
```

- If exception reaches top level, program terminates

Python Example

```
def average_input():
    while True:
        try:
            data = input('Enter some values: ')
            mean = average(list(map(float,
                                    data.split())))
        except EOFError:
            return
        except ValueError:
            print('Bad values, try again!')
        else:
            return mean

def average(values):
    count = len(values)
    if count == 0:
        raise ValueError('No values to average!')
    return sum(values) / count  average_input()
```

Exception Clauses

- `try`: dynamic region of code for which exceptions are handled
- `catch/except`: exception handler
  - Handlers checked in order until an appropriate one is found
- `finally`: run whether or not an exception occurs
- `else`: run if no exception occurs
- Not all languages provide every kind of clause