Lecture Notes 11

Control Structures

- Control Statement – Statement allowing for different execution paths, or for repeated execution of a path
- Control Structure – Control statement with collection of statements (often block) in which it controls

Selection Statements

- Selection Statement – Provides means of choosing between two or more execution paths
- Two-Way Selection – An If-Then-Else selection statement
- *if-statement* **→ if (** *expression* **)** *statement* [**else** *statement*]
  - Dangling else
    - Disambiguating rule (most closely nested)
    - Bracketing keyword (end if or fi)
- elsif or elif – Provided in languages such as Python or C preprocessor to allow for an else if statement in order to support multiple-selection
- Selector Expression – Selector that results in a value, often exists in functional languages like Lisp
- Multiple-Selection Statement – Allows for selection of any number of statements or statement groups
- Switch (or Case) Statement – A multiple-selection statement that allows for selection based upon value (not just Boolean expression)
  - Fall through – Control transfers to next case (without break) or after statement if no matching “case”
  - `default` case – Catch all if no others match
  - `break` statement – Exits the case/switch statement transferring control to after statement
  - Wildcard pattern – Similar to default case

Iterative Statements

- Iterative Statement – Allows for collection of statements to be executed zero or more times
- Body – Collection of statements controlled by the iterative statement
- Pretest – Repetition completion is tested prior to body execution
- Posttest – Repetition complete is tested after body execution
- Logically Controlled Loops – Loop is controlled by a Boolean expression
- Counter-Controlled Loops – Loop that has a count value maintained to determine completion

- While Loop – A pretest logically controlled loop
  - General form – `e` is the pretest expression and `S` is the collection of statements
    ```
    while(e) S
    ```
- Do-While Loop – A posttest logically controlled loop
  - General form – `e` is the pretest expression and `S` is the collection of statements
    ```
    do S while(e)
    ```
  - Equivalent While
    ```
    S
    while(e) S
    ```
- For Loop – A counter-controlled loop
  - C Style General form – `e1` is the initializer, `e2` is the pretest expression, `e3` is the post loop update, and `S` is the collection of statements
    ```
    for(e1; e2; e3) S
    ```
  - Equivalent While
    ```
    e1
    while(e2){
         S
         e3
    }
    ```
- Foreach Loop – A loop that iterators over items in a container
  - Range Based General form – `v` is the loop variable that is from a range `R` and `S` is the collection of statements
    ```
    foreach(v : R) S
    ```
  - Equivalent While – `first` is a function that returns first element of range, `next` provides the next element in the range, and `end` is marks the end of the range
    ```
    v = first(R)
    while(v ≠ end(R)){
         S
         v = next(R,v)
    }
    ```
- `continue` statement – Continue to the next iteration of the loop
- `break` (or `exit`) statement – Break out of the loop
- `yield` statement – Statement in python that acts like a return but allows continuation of method when invoked subsequent times

Unconditional Branching

- Unconditional Branch statement (or goto) – Transfers execution control to a specific location unconditionally
- Spaghetti Code – Code that is difficult to follow the control flow, often attributed to overuse of goto statements
- Arguments against gotos

  - Completely Unnecessary – Bohm and Jacopini argued that can always be replaced with other structures
  - Damaging? – Dijkstra argued that it is harmful
- When to use gotos
  - Goto Error Handling Nested Loops

```
    if(ok){
        ...
        while(more){
            ...
            while(!found){
                ...
                if(disaster) goto HandleError;
                ...
            }
            ...
        }
        ...
    }
HandleError:...
```

  - Labeled break (Java)

```
OKBlock:
    if(ok){
        ...
        while(more){
            ...
            while(!found){
                ...
                if(disaster) break OKBlock;
                ...
            }
            ...
        }
        ...
    }
```

- Nested If vs Goto for allocation

  ```
  a = allocate();
  if(a){
      b = allocate();
      if(b){
          c = allocate();
          if(c){
              d = allocate();
              if(d){
                  e = allocate();
                  if(e){
                      return 0;
                  }
                  deallocate(d);
              }
              deallocate(c);
          }
          deallocate(b);
      }
      deallocate(a);
  }
  return -1;
  ```

  vs

  ```
  a = allocate();
  if(!a) goto AErr;
  b = allocate();
  if(!b) goto BErr;
  c = allocate();
  if(!c) goto CErr;
  d = allocate();
  if(!d) goto DErr;
  e = allocate();
  if(!e) goto EErr;
  return 0;
  EErr: deallocate(d);
  DErr: deallocate(c);
  CErr: deallocate(b);
  BErr: deallocate(a);
  AErr: return -1;
  ```

- Loop and a half problem – Structured loops may require part of loop appear before loop

Guarded Commands

- Guarded If (Dijkstra) – All boolean expressions `Bx` are evaluated and one of the true ones is nondeterministically chosen to execute associated `Sx` statement.

  ```
  if B1 → S1
  |  B2 → S2
  |  B3 → S3
  ...
  |  Bn → Sn
  fi
  ```

- Guarded Do (Dijkstra) – All boolean expressions `Bx` are evaluated and one of the true ones is nondeterministically chosen to execute associated `Sx` statement. If none are true the loop terminates.

  ```
  do B1 → S1
  |  B2 → S2
  |  B3 → S3
  ...
  |  Bn → Sn
  do
  ```