



The University of North Carolina at Chapel Hill

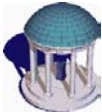
---

COMP 144 Programming Language Concepts  
Spring 2003

## Variables, Structured Flow, Sequencing and Selection

Stotts, Hernandez-Campos

1

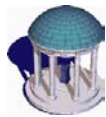


## Control Flow

---

- Control flow refers to the **order in which a program executes**
- This is fundamental in the imperative programming paradigm
  - E.g. Java or Python
- In other programming paradigms, the compilers or the interpreters take care of the ordering
  - E.g. functional and logic programming

2

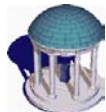


## Control Flow Mechanisms

---

- Sequencing
  - Textual order, Precedence in Expression
- Selection
- Iteration
- Procedural abstraction
- Recursion
- Concurrency
- Nondeterminacy

3



## Assignment

---

- The basic operation in imperative language is assignment
  - The *side effect* of this operation is a change in memory
  - Assignments affect the whole state of the program
- Purely functional language do not have assignment
  - Side effects are not possible
  - Expression in purely functional languages depend only in their referencing environment
- *Expressions* produce values
- *Statements* do not return values, but they have side effects

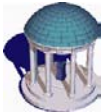
4



## Variables

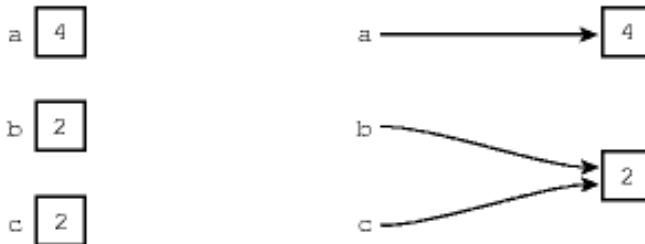
- Value model of variables
  - E.g. Pascal's `A := 3 ;`
  - `A` is an *l-value*, and it denotes a position in memory
  - `3` is a *r-value*, and it denotes a value with no explicit location
  - Consider `A := A + B ;`
    - r-val* (points to `B`)
    - l-val* (points to `A`)
    - same text, two different compiler handlings
- Reference model of variables
  - E.g. Java's `A = new Integer(3)`
  - Variables are references to values

5



## Variables

- Value and Reference model of variables



- Variables in Java
  - <http://java.sun.com/docs/books/tutorial/java/nutsandbolts/variables.html>

6



## Expressions Other Issues

- *Initialization* may be implicit or explicit (at declaration or execution)
  - E.g. All variables have a default value in Perl, while Python requires all variables to be initialized
    - » As a consequence, a simple typo in the name of variable in a Perl program may become a nasty bug
- *Orthogonality* means that features can be used in any combination and their meaning is consistent regardless of the surrounding features
  - E.g. Assignment within conditional expressions in C++
  - This is powerful but problematic, since typos may be legal
    - » E.g. `if (a==b) { ... }` and `if (a = b) { ... }`

7

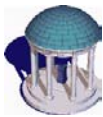


## Expressions Other Issues

- Execution ordering within expressions is complicated by *side effects* (and *code improvements*)
  - E.g. In Java,

```
b = 1
int increment(int a) {
    b += 1;
    return a + 1;
}
c = (3 * b) * increment(b)
```
  - If the function call is evaluated before  $(3 * b)$  the final value of  $c$  is 12. If the product is evaluated before the function call, the value of  $c$  is 6.
  - But **side effects within functions are a bad idea!**

8



## Expressions Other Issues

---

- Expressions may be executed using *short-circuit evaluation*
  - E.g. In C,

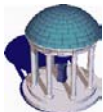
```
p = my_list;
while (p && p->key != val)
    p = p->next;
```

    - » Field `key` is not accessed if the pointer `p` is null
  - This is not available in Pascal, so an additional variable is required (e.g. `keep_searching`)

```
p := my_list;
while (p<>nil) and (p^.key <> val) do (*ouch*)
    p := p^.next
```

    - » Access to `key` causes a run-time error at end of the search

9

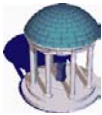


## Expressions Examples

---

- Expressions in Java
  - <http://java.sun.com/docs/books/tutorial/java/nutsandbolts/expressions.html>
- Expressions in Python
  - <http://www.python.org/doc/current/ref/expressions.html>

10



## Unstructured Flow

### The GOTO Statement

---

- Control flow in assembly languages is achieved by means of conditional jumps and unconditional jumps (or branches)
  - *E.g.*

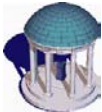
```
JMP 30
...
30: ADD r1, #3
```

    - » 30 is an *assembly-level label*
  - Higher level languages had similar statement: `goto`
    - » *E.g.* In FORTRAN,

```
If A .lt. B goto 10
...
10: ...
```

      - » 10 is a *statement label*

11



## Unstructured Flow

### The GOTO Statement

---

- Goto is considered evil since the 70s
  - It potentially makes programs extremely convoluted
    - » *Spaghetti code*
    - » Code may be difficult to debug and even more difficult to read
  - In 1967, Dijkstra's published an classic article, *GoTo Considered Harmful*, that pointed out the dangers of the `goto` statement in large programs
    - » The larger the program, the worse the impact of `goto` statements

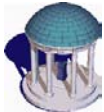
12



## Structured Flow

- Structured flow eliminates goto
  - Nested constructs (blocks) provide the same expressive power
- Any program can be expressed using **sequencing**, **selection** and **iteration**
  - This was proved in a 1964 paper by Bohm & Jacopini
- However, there is a small number of cases in which unstructured flow is still more convenient
  - Modern structured languages like Java, and Perl, have addressed these cases in an structured manner

13



## Structured Flow Special Cases

### • Next, Last (Perl)

```
while ( $d++ ) {  
    if ($d >= 37) { $res = "bingo"; last ; }  
    $sum += $d ;  
}  
# last jumps to here  
  
while ($d < 37) {  
    $d++ ;  
    if ( ($d%5)==1 ) { next } ;  
    $sum += $d ;  
    # next jumps to here  
}
```

14



## Structured Flow Special Cases

---

- Redo (Perl)

```
while ( $d++ ) {  
    # redo jumps to here  
    $r = random($d);  
    if ($r >100) { redo; }  
    $sum += $r*$d ;  
}
```

15



## Structured Flow Special Cases

---

- Continue (Perl)

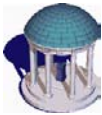
last, next, or redo may appear within a continue block.

last and redo will behave as if they had been executed within the main block.

So will next, but since it will execute a continue block, it may be more entertaining.

```
while (EXPR) {  
    ### redo always comes here  
    do_something;  
} continue {  
    ### next always comes here  
    do_something_else;  
    # then back the top to re-check EXPR  
}  
### last always comes here
```

16

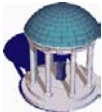


## Structured Flow Special Cases

- Redo (Perl)

```
# a simpleminded Pascal comment stripper
# (warning: assumes no { or } in strings)
LINE: while (<STDIN>) {
    while (s/({.*}.*){.*/$1 /) {}
    s/{.*}/ /;
    if (s/{.*/ /) {
        $front = $_;
        while (<STDIN>) {
            if (/)/) { # end of comment?
                s/^{.$front}{/;
                redo LINE;
            }
        }
    }
    print;
}
```

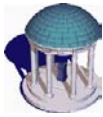
17



## Structured Flow Special Cases

- Break and continue
  - Java's branching statements
    - » <http://java.sun.com/docs/books/tutorial/java/nutsandbolts/branch.html>
- Early subroutine returns
  - Java's return statements
    - » <http://java.sun.com/docs/books/tutorial/java/nutsandbolts/branch.html>
- Exceptions and Errors
  - Java's exception handling
    - » <http://java.sun.com/docs/books/tutorial/java/nutsandbolts/exception.html>

18

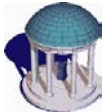


## Sequencing

---

- Sequencing is central to imperative programming languages
- Sequencing of statements is usually defined by textual orders
- Enclosed sequences of statements are called *compound statements* or *blocks*
  - E.g. `begin ... end`, `{ ... }`, Python's indentation
  - Declarations (e.g. variable types) may also be part of a code block

19



## Selection

---

- *If/Then/Else* statement
- *Case/Switch* statement
  - The motivation for this statement is not purely esthetical
  - In some cases, switch statements are faster than nested *if/then/else*
  - The argument of the conditional must be a discrete value, so the target of the selection can be using an array indexed by the values (rather than checking the cases sequentially)
- The *break* statement in C/C++/Java makes code generation even more efficient in some case

20





## Reading Assignment

---

- Scott's chapter 6
  - Subsection 6.1.2-4
  - Section 6.2
  - Section 6.3
  - Section 6.4
- If interested in the history of programming, you can read these two classics
  - Edsger Dijkstra, *Goto Considered Harmful*, CACM 1968.
  - C. Bohm and G. Jacopini. Flow diagrams, *Turing machines and languages with only two formation rules*. Communications of the ACM, 9(5):366-- 371, May 1966.