CS323 Lecture: Control Structures last revised 3/4/09

Objectives

1. To survey the various decision and loop structures included in most languages, and issues related to them.
2. To discuss the goto controversy and related issues
3. To introduce some less-common control structures
4. To discuss recursion as an alternate way of accomplishing iteration

Materials:

1. Projectable of non-recursive inorder traversal
2. Tree of calls for recursive fibonacci(6)

I. Introduction
-- ------------

A. Any program can be thought of as being composed of two kinds of statements:

1. "Workhorse" statements that perform actual computation.

2. Control statements that govern the order in which the other statements are performed.

B. At the machine-language level, most computers offer but four classes of control statements.

1. The unconditional branch instruction

2. The conditional branch instruction that does or does not branch based on some simple condition like the sign of the last arithmetic operation done or the value of some hardware flag (like overflow)

3. The subroutine call instruction

4. The subroutine return instruction

C. Early languages had control statements that closely mirrored this hardware reality. For example, early FORTRAN had:

1. An ordinary GOTO statement and a computed GOTO statement, both based on the machine language unconditional branch instruction.

2. An arithmetic IF statement, plus IF statements that interogated various program status elements such as DIVIDE OVERFLOW - mirroring the various forms of machine language conditional branch instruction.

3. A subroutine CALL statement, mirroring the machine language subroutine call instruction. (This was not part of FORTRAN I but was added in FORTRAN II)

4. A RETURN statement, mirroring the machine language subroutine return instruction. (Also added to FORTRAN II).

5. The DO loop, which was the only control structure having no direct analogue in the underlying machine language.

D. However, it was realized very early that higher-level languages could (and should) incorporate higher-level control structures. This can be seen as early as Algol-60; but very similar statements are still found in languages derived from Algol including C/C++/Java.

   1. Algol-60 included an if .. then .. else statement which could be controlled by any boolean expression.

   2. It likewise included a powerful for .. do statement similar to the one in C. Because it could include a while clause, it could serve for both definite and indefinite loops.

   3. It included a switch facility that gave something of the effect of an Ada case or a C switch - though the form found in newer languages is quite different.

   4. Of course, it also had facilities for functions and procedures.

   5. Curiously, the first control structure described in the Algol report was the goto statement!

E. Today, there is wide agreement on certain basic control structures that should be included in a language. The following structures are found in some form in almost every language:

   1. If .. then .. else

      Note that the short-circuit handling of and, or arises rather naturally as a consequence of the way ifs involving compound conditions are compiled

      Example: if (i < j && k > l)
                      statement

               may be translated into

               compare i and j
               branch if greater-equal to L1 ; Note "branch if not true"
               compare k and l
               branch if less-equal to L1   ; Note "branch if not true"
               code for statement
          L1:

       a. Historically, some languages allowed the compiler to generate code like this without specifying which test is done first

       b. Languages that support short-circult if basically just dictate that the compiler does the tests in left-to-right order

       c. In fact, it turns out that non-short-circuit constructs (e.g. Ada's and, or and C/C++/Java's &, |) are actually harder to compile!

   2. Some sort of counter-controlled loop - like FORTRAN's DO or C's for ... (The book refers to this as an "enumeration controlled loop"). While straight-forward in principle, there are some issues that arise with this

a. Should the index variable be allowed to be a non-ordinal type (e.g. real)?

b. Is it possible to alter the value of the index variable within the loop body? If so, what impact does this have on the actual sequence of values the index variable assumes? (Note: in some cases there is a real surprise lurking here!)

c. Is the number of iterations of the loop pre-computed when the loop is first encountered, or is the loop test done on every iteration? That is, does altering the index variable or final or step values in the loop body have any impact on the number of times the loop actually iterates?

   e.g.

```
start := 1;
finish := 5;

for index in start .. finish loop
     -- do something
     finish := finish - 1;
end loop
```

   From looking at the loop header, one would infer that the loop body is done 5 times. But since finish is decremented on each time through the loop, it may be that what we have looks like this:

```
index   finish

1       5 -> 4
2       4 -> 3
3       3 -> 2      So finished after 3 iterations?
```

   i. There may be significant efficiency advantages to pre-computing the number of iterations, since the code to decrement and test a counter is quite simple (just 1 instruction on many machines). Of course, in this case altering the index, final, or step value inside the loop body will have no impact on the actual number of times the loop is done.

   ii. It can be argued that there are semantic reasons for pre-computing the number of iterations as well, given that the counter-controlled loop is sometimes referred to as "definite iteration".

   iii. OTOH, sometimes the syntax of the loop statement implies testing every time through the loop - e.g.

      C/C++/Java for (initialization; end test; increment)

d. What kinds of step value are supported?

   - Many languages allow arbitrary values - e.g.

     FORTRAN:

     DO 100, I = START, FINISH, STEP

     C/C++/Java

     for (i = start; i <= finish; i += anything)

- Ada allows only +/- 1 - using the constructs

```
in range
in reverse range
```

e. What is the final value of the index variable when the loop terminates?

(Note: Ada avoids this issue by decreeing that the index variable is strictly local to the loop body, and hence cannot be accessed when the loop body terminates)

You will explore each of these issues on a homework problem

3. Some sort of indefinite loop (The book refers to this as a "logically controlled loop")

a. Many languages support two varieties of this - one of which allows the loop to be done no times at all, and one of which requires the loop to be done at least once

Example: C/C++/Java while ... vs do ... while

b. Some languages support only the "0-or-more times" variety - e.g. Ada

4. Some sort of multi-way decision structure, like C's switch, Ada's case, or LISP's COND.

a. Multi-way selection using if, and LISP's COND, suffer from the problem of doing many tests

Example: Suppose we wanted to write the word "vowel" or "consonant" for a given letter of the alphabet. We might write something like

C++:

```
if (letter == 'A')
    cout << "vowel" << endl;
else if (letter == 'B')
    cout << "consonant" << endl;
...
else // must be Z
    cout << "consonant" << endl;
```

LISP

```
(defun classify (letter)
    (cond
        ((eq letter #\A) "vowel")
        ((eq letter #\B) "consonant")
        ...
        (t "consonant")
    )
)
```

Either version requires potentially 25 tests, and is cumbersome to write/read to boot.

b. FORTRAN includes a computed goto that allowed multiway branching based on an expression. The above could be written

```
GOTO (1, 2, 2, ... 2), LETTER - 64
...
1   WRITE (*, *) "vowel"
    GOTO 3
2   WRITE (*, *) "consonant"
3   ...
```

This has a very efficient translation into machine language via the use of a jump table, which in this case would have 26 entries

```
Address of 1
Address of 2
Address of 2
...
Address of 2
```

The machine code now computes the address of the desired element of this table (just like it does with any array subscript) and then places this address in the program counter.

c. Algol-60 had a switch construct that was even more explicitly a higher-level representation of the machine language code.

d. The switch/case construct found in many newer languages may be compiled along similar lines - i.e. a jump table is created corresponding to the for each of the cases (with multiple entries if multiple values share the same code) and then the expression is used to index into this table. This is what lies behind the common requirement that the selection expression be of an ordinal type.

(If the set of case labels is sparse, a hashtable may be used instead of an array)

e. The LISP COND is not handled this way, of course, since it allows general tests. Common Lisp also has a case statement; but since it allows non-integer case labels, it needs to be implemented by using a hashtable rather than a simple array.

5. Some programming languages generalize the notion of a for loop to allow iteration over constructs other than a simple range of integers.

a. For example, Java supports a generalized for loop that can be used with arrays and any class that implements the Iterable interface (which includes the standard collections)

Example:

```
int [] x = new int [10];
...
for (int element : x)
    System.out.println(element);
```

prints all the elements of the array x

Example:

```
java.util.Set<Employee> employees = new HashSet<Employee>;
...
for (Employee employee : employees)
    employee.pay();     // Assuming Employee has a pay() method
```

b. The C++ for loop is general enough support an idiom like this, given appropriate overloading of the operators ++ and -> (which is provided by the STL templates).

```
typedef set<Employee> EmployeeSet;
EmployeeSet employees;
...
for (EmployeeSet::iterator iter = employees.begin();
        iter != employees.end();
        iter ++)
     iter -> pay();   // Assuming Employee has a pay() const method
```

c. Other languages support an iterator approach that closely resembles coroutines, which we will discuss under procedures.

F. Some other issues we will discuss

1. One issue that remains fairly controversial is the proper role (if any) of the GOTO statement.

2. Another important issue is the use of recursion as an alternative to traditional iteration.

3. The book also discussed applicative, normal order and lazy evaluation of routine parameters in this chapter, but we will consider these under the procedures topic in a week or so.

II. The GOTO Controversy

A. As the above discussion implies, the history of control structures has moved in the direction of ADDING new structures to the list of "standard structures" found in most languages. There is, however, one exception to this trend - the movement away from the goto statement, which has been a basic control statement since the days of Algol and FORTRAN.

1. In the late 1960's, as the software crisis loomed, computer scientists became aware that the unbridled use of the goto statement was a major source of problems in maintaining software.

2. Edsger Dijkstra's famous letter "Goto statement considered harmful" appeared in the Communications of the ACM in March, 1968. His letter said what many already knew was true, and so served as an impetus for what came to be known as the "structured programming revolution".

a. In the strict sense, structured programming refers to the discipline of building complex programs out of procedures and functions plus three basic building blocks for use within them. (The examples use C notation)

i. The sequence structure:

```
{
    S1;
    S2;
    ...
    Sn
}
```

ii. The decision structure:

```
if C
   S
```

or

```
if C
   S1
else
   S2
```

(or the switch structure if available in the language)

iii. The loop structures:

```
for ...
    S
```

or

```
while C
    S
```

or

```
do
    S
while C
```

(Where, in each case, any of the statements S can itself be another structure)

b. In common paralance, though, the term "structured programming" was often understood to mean "goto-less programming" - i.e. the tendency was to define it negatively (no goto's) rather than positively (use of well-defined and understood structured forms.)

B. The structure theorem

1. In 1964, Bohm and Jacopini proved that every proper program, however complicated, can be transformed into an equivalent program composed only of nested combinations of the three basic structures: sequence, if .. then .. else, and while .. do. This theorem undergirded the structured programming movement, because it showed that adherence to structured programming does not prevent one from writing all possible programs.

2. The theorem was proved constructively, by spelling out a process for transforming an unstructured program into a structured one. Unfortunately, if this transformation is applied mechanically it may result in a program that is even harder to follow than the original one!

C. The development of structured programming led to the question "should modern programming languages include a goto statement?"

1. Those who argue for the retention of the goto statement point to several kinds of situations where its use seems appropriate and desireable.

   a. Mid-exit loops.

   b. Dealing with errors that are discovered within a procedure, necessitating termination of that procedure and perhaps one or more calling levels.

   c. Early return from routines, and multi-level return from routines, under circumstances that are not erroneous (e.g. finding a desired item during a search)

2. Others argue against the retention of the goto by arguing that there is a cleaner way to handle each of these.

   a. Some problems seem to naturally call for a mid-exit loop.

      Example: Reading and processing data from some input stream, with end-of-input marked by end-of-file or some sentinel value.

      e.g.
```
        loop
            read some input
            if end of input goto done
            process the input just read
        forever
    done:
```

      There are quite a number of ways of handling situations like this without using a goto:

      i. A special way of handling end-of-input in the IO statement

         Example: (FORTRAN 95)

```
        DO WHILE (.TRUE.)
            READ (*, *, END=999) some input
            -- Process input
        END WHILE
999
```

         Problems:

         ASK

         - Solves only certain kinds of problems - not a general solution to all kinds of problems

         - Depends on input output being handled in the language, whereas most modern languages use library routines instead

ii. Use of an if

```
do
    read some input
    if (end test not met)
        process the input just read
while end test not met
```

Problems:

ASK

- Makes the code a bit more complex to read, due to extra if and indentation

- Does the "end test" twice on every iteration of the loop, which may be inefficient if extensive computation is involved, or even cause a problem if the test involves side effects

iii. The requirement for two complete tests can be eliminated by introducing a boolean variable

```
boolean exit = false;
do
    read some input
    if (end test not met)
        process the input just read
    else
        exit = true;
while not exit
```

- There are still two tests needed, but one is quite simple; and matters have been complicated by introducing a new variable.

iv. The "priming the pump" pattern

```
read some input
while not at end of input
    process the input just read
    read some input
```

Problem:

ASK

- Need to repeat the "read some input" code. Note that though the code _appears_ twice, it is only _executed_ once for each loop iteration. (If this is just a single statement, this is probably not a problem, but what if it's more complex?)

v. Special mid-loop exit constructs in the language (C/C++/Java break; ada exit when)

e.g. (Ada)

```
loop
     read some input
     exit when end of input
     process input just read
end loop;
```

Problem:

ASK

- What happens when one wants to exit nested loops?

  - C/C++/Java break cannot handle this - break only exits one level of loop

  - Ada has a labelled loop facility that allows for this

```
Main_Loop: loop
    loop
        loop
            exit Main_Loop when ...
        end loop;
    end loop;
end loop Main_Loop;
```

b. Many languages now incorporate some sort of exception-handling mechanism for "bailing out" when an error is discovered.

c. Some problems call for exiting immediately from a routine when some (non-erroneous) condition arises. Most languages now incorporate an explicit return statement that handles this cleanly in the case of exiting a single routine; but the problem becomes much more difficult if one wants to exit several levels of routines - e.g.

```
procedure a is

    procedure b is

        procedure c is
        begin
            ...
            if exit condition met exit a
            ...
        end c;

    begin -- b
        ...
        c;
        ...
    end b;

 begin -- a
    ...
    b;
    ...
 end a;
```

- When c detects the exit condition, the stack looks like this

```
---------------
| frame for c |
---------------
| frame for b |
---------------
| frame for a |
---------------
| frame for   |
| caller of a |
---------------
```

- and after the exit we want it to look like this

```
---------------
| frame for   |
| caller of a |
---------------
```

i. Block-structured languages may allow a non-local goto, provided the goto is enclosed in the block containing the target - e.g.

```
          if exit condition met goto done
          ...

      begin -- a
          ...
          b;
          ...
 done:
      end a;
```

Part of the execution of the goto is unwinding the stack, which is done by the language implemntation.

ii. Some languages (e.g. LISP) support a return-from construct. (The return-from must specify a block that lexically encloses it.) Again, the language imnplementation handles unwinding the stack.

iii. Algol allowed procedures to have labels as parameters, allowing the caller to specify a location where control should be transferred if an early exit were needed. (But this has pretty much disappeared from the programming language landscape)

iv. Some operating systems provide a mechanism for stack unwinding to produce the effect of a non-local goto, with no requirement for lexical enclosure. (Example: unix setjmp, longjmp)

v. Sometimes the exception-handling mechanism is used for this, even though the reason for the exit is not really an error

3. Nonetheless, few language designers have eliminated the goto statement altogether. (Java is an exception - not only does Java not have a goto statement, but it makes goto a reserved word to catch accidental use when translating code written in another language.)

IV. Recursion as an Alternative Way of Doing Iteration

A. It can be shown that recursion and iteration using loops are equivalent. Any problem that can be solved using recursion can be solved without recursion using loop(s); and any problem that can be solved using loop(s) can be solved using recursion without using loop(s).

B. Nonetheless, there are often reasons for preferring to use one rather than the other for a given problem.

1. For some problems, a recursive solution is much more natural.

   Example: traversal of a binary tree is a natural for recursion; and a non-recursive solution using a loop, though possible, is non-intuitive and hard to read

   Recursive inorder traversal of a tree:

```
void inorder(Node root)
{
   inorder(Root.lchild);
   visit(Root.data);
   inorder(Root.rchild);
}
```

   Non-recursive inorder traversal: (PROJECT)

```
void inorder(Node root)
{
   Stack<Node> toTraverse = new Stack<Node>();
   Node p = root;
   while (p.lchild != null)
   {
       toTraverse.push(p);
       p = p.lchild;
   }
   while (! toTraverse.empty())
   {
       Node p = toTraverse.pop();
       visit(p.data);
       if (p.rchild != null)
       {
           p = p.rchild;
           while (p.lchild != null)
           {
               toTraverse.push(p);
               p = p.lchild;
           }
           toTraverse.push(p);
       }
   }
}
```

2. Pure functional programming relies on recursion rather than iteration, largely because iteration and assignment of values to variables (side effects) are inseparable. (Functional languages often include looping constructs, but their use is discouraged.)

C. However, there are certain inherent inefficiencies to the use of recursion that can be avoided if one is careful.

1. One inefficiency arises because the obvious recursive solution to a problem may do a lot more work than necessary.

a. Example: the Nth Fibonacci number (Fib(N)) is defined by

if N <= 2 Fib(N) = 1 else Fib(N) = Fib(N-1) + Fib(N-2)

which leads to an obvious but quite inefficience recursive solution:

```
int fib(int n)
{
    return (n <= 2) ? 1 : fib(n-1) + fib(n-2);
}
```

To see why this is inefficient, consider the calls involved in calculating the sixth Fibonacci number this way:

PROJECT tree - note repetitions of various steps

(In general, computing Fib(N) involves Fib(N) calls to the base case, plus recursive cases - so it has time complexity a multiple of O(Fib(N))! Since Fib(N) is O(2^N), this is problematic!)

However, there is a nice O(N) iterative solution

```
int fib(int n)
{
    int i = 1;
    int fibI = 1;
    int fibIMinus1 = 0;
    while (i < n)
    {
        i ++;
        int next = fibI+fibIMinus1;
        fibIMinus1 = fibI;
        fibI = next;
    }
    return fibI;
}
```

b. In cases like this, it is often possible to come up with a recursive solution that is just as efficient as an iterative one.

i. One approach is to make use of a recursive auxilliary with extra parameters:

```
public int fib(int n)
{
     return fibAux(n, 1, 1, 0);
}

private int fibAux(int n, int i, int fibI, int fibIMinus1)
{
     return (i == n) ? fibI : fibAux(n,i+1,fibI+fibIMinus1,fibI);
}
```

ii. Another approach is to use memoization

```
int [] fib = new int [ Max potential arg ]; // All initially 0

int fib(int n)
{
    if (fib[n] == 0)
        fib[n] = fib(n-1) + fib(n-2);
    return fib[n];
}
```

The key here is that each element of the fib[] array is calculated just once and then saved for future use.

2. Another inefficiency arises from the overhead associated with function calls.

   a. As we have previously noted, a call to a routine requires the creation of a stack frame that holds its return point, parameters, and local variables. This frame is created when the function is called and destroyed when it terminates.

      (With high speed memory being quite large on modern computers, this is less of an overhead issue than it once was)

   b. Languages that rely heavily on recursion (e.g. functional languages) sometimes incorporate tail-recursion optimization in their implementation.

      i. A recursive call is said to be tail-recursive if

      - It is the last step in the computation.
      - No computation is done after the recursive call completes.
      - The result of the recursive call becomes the result of the original call.

      Example: The following is tail recursive

      ```
      int gcd(int a, int b)
      {
           return b == 0 ? a : gcd(b, a % b);
      }
      ```

      Example: The following is _not_ tail recursive

      ```
      int fact(int n)
      {
           return n <= 1 ? 1 : n * fact(n-1);
      }
      ```

      Why?

      ASK

ii. When a function does a tail-recursive call, there is no need to create a new stack frame on top of the original one. Instead, the parameters of the original frame can simply be reset to the values required for the recursive call and execution can be restarted at the beginning of the code.

Example: Consider the computation of gcd(20, 12) - which involves recursive calls to gcd(12, 8), gcd(8, 4), and gcd(4, 0)

The single stack frame would be initially

```
return point: main
a: 20
b: 12
```

The first recursive call would transform this to

```
return point: main
a: 12
b: 8

etc.
```