Sequence Control

|  |  |
| --- | --- |
| **Sequence Control**  In most languages, flow is linearly downward statement by statement. We have statements which can alter that flow:  Conditional - if, if-then-else, case constructs  Iteration - loop, repetition (e.g., for)  Subprogram - macros, functions, recursion, interrupts, event-based |  |
| **Conditional Statements**  Programming languages support conditional constructs with several styles of if-then-else and case constructs.  if-then construct:    if-then-else construct: | C:  // if-then  if (grade >= 60)  printf("you passed\n");  // if-then-else with nested if-then-else  if (strcmp(transaction.command, "WITHDRAWAL") == 0)  if (account.dBalance > 0)  {  account.dBalance -= transaction.dAmount;  printf("New balance: %10.2lf\n", acccount.dBalance);  }  else  printf("Insufficient funds: %10.2lf"\n  , transaction.dAmount;  else if (strcmp(transaction.command, "DEPOSIT") == 0)  account.dBalance += transaction.dAmount;  Python:  # if-then  if grade >= 60:  print("you passed")//**indentation represents the if statement**  # if-then-else with nested if-then-else  if command == "WITHDRAWAL":  if dBalance > 0:  dBalance -= dAmount  print("New balance:", dBalance)  else:  print("Insufficient funds:", dAmount)  elif command == "DEPOSIT":  dBalance += dAmount |
| **Conditional Statements - Case**  Case constructs allow the selection of particular cases based on the value of an expression. An optional default case is provided when the expression's value doesn't match one of the case values. | C: // the syntax in C that allows multiple values for  // a single process caused C to require a break statement  **// No case statement in Python**  switch(cLetterGrade)  {  case 'A':  case 'B':  case 'C':  printf("You are doing well\n");  break;  case 'D':  printf("borderline passing\n");  break;  case 'F':  printf("failing\n");  break;  default:  printf("unknown grade: '%c'\n", cLetterGrade);  } |
|  | PL/I case statement examples:  /\* Example 1 \*/  SELECT (LETTER\_GRADE);  WHEN('A', 'B', 'C')  PUT SKIP LIST('You are doing well');  WHEN('D')  PUT SKIP LIST('borderline passing');  WHEN('F')  PUT SKIP LIST('failing');  OTHERWISE  PUT SKIP EDIT('Unknown grade: ', LETTER\_GRADE)  (A, A);  END;  /\* Example 2 \*/  SELECT (TRANSACTION.COMMAND);  WHEN ('DEPOSIT') CALL DEPOSIT(TRANSACTION, ACCOUNT);  WHEN ('WITHDRAWAL') CALL WITHDRAWAL(TRANSACTION, ACCOUNT);  WHEN ('INTEREST') CALL ADD\_INTEREST(TRANSACTION, ACCOUNT);  OTHERWISE  DO;  PUT SKIP EDIT('INVALID COMMAND: ', TRANSACTION.COMMAND)  (A, A);  END;  END;  /\* Example 2 \*/  SELECT;  WHEN (TRANSACTION.COMMAND = 'WITHDRAWAL'  & ACCOUNT.BALANCE < 0) CALL OVERDRAWN(TRANSACTION, ACCOUNT);  WHEN (TRANSACTION.COMMAND = 'DEPOSIT') CALL DEPOSIT(TRANSACTION, ACCOUNT);  OTHERWISE CALL WITHDRAWAL(TRANSACTION, ACCOUNT);  END; |
| In functional languages such as LISP, conditional expressions can appear anywhere that expressions can appear. | LISP  (defun MAX2 (x y) //max of two values  (cond ( (> x y) x) //cond(if) x greater than y, return x  ( (= x y) z) // if x == y, return z  ( T y) //if T = true , return y  )  ) |
| **Iteration**  Looping constructs have many interesting features in programming languages. Languages have simple loops (e.g., while, do while) and loops with control variables (e.g., for). Additionally, there are statements for exiting the loop (e.g., break) or continuing the loop ignoring the rest of the statements (e.g., continue).  while-loop construct: | C:  while (fgets(szInput, MAX\_INPUT, stdin) == 0)  {  iScanfCnt = sscanf("%30[^,], %lf",  , student.szName  , student.dGradeTotal);  if (iScanfCnt < 2)  {  printf("Bad Data: %s\n", szInput);  continue;  }  printf("%30s %6.2lf\n");  dTotal += student.dGradeTotal;  iStudentCount++;  }  Java:  OuterFor:  for (int i = 0; i < rowCount; i++)  {  for (int j = 0; j < colCount; j+)  {  if (array2D[i][j] < 0)  break OuterFor; // Java provides this // Not C  sum += array2D[i][j];  }  } |
| **Iteration - do while loop**  If it is necessary to do the first iteration, use this style of loop. | C:  do  {  printf("\nEnter a number (enter 0 to end): ");  scanf("%d", &iNum);  iSum += iNum;  }  while (iNum != 0);  printf("\nsum=%d\n", iSum); |
| **Iteration With Control Variables**  There are many types of iteration statements that use control variables to influence the number of iterations.  **counter-controlled construct**   * counts the number of iterations * can changing the counter variable within the loop affect the execution? * can changing a condition variable within the loop affect the execution? | C:  for (i = 0; i < iEnd; i++)  {  // loop body  }   * changing i inside the loop affects number of iterations * changing iEnd inside the loop affects number of iterations   FORTRAN:  do 100 cv=start,end,incr //100 is a label, not do this 100 times  body  100 continue   * The body should not change cv. * Prior to executing the loop, FORTRAN accesses the value of start, end, and incr. This means changing them inside the loop has no impact. |
| **Iteration with Control Variables - Based on other Objects**  Many languages support iteration based on the contents of other objects. This helps simplify what a programmer must do.   * The control variable represents one item from the contents of the other object. For example, one item in a list. * How is the loop impacted if we add items to the other object?   + Be careful, adding items can cause an infinite loop.   + In some cases, an item can be added, but the iteration is already past that item in the contents of the other object (e.g., adding an item in a hash table which happens to fall into a location that the iteration has already past) | Python:  # example 1  fruitM = ["apple", "orange", "banana"]  for fruit in fruitM:  print (fruit)  Output:  apple  orange  banana  # example 2  i = 0  fruitM = ["apple", "orange", "banana"]  fruitM.append("tangerine")  for fruit in fruitM:  print (fruit)  i += 1  if i < 3:  fruitM.append("grape")    print (fruitM)  Output:  apple  orange  banana  tangerine  grape  grape  ['apple', 'orange', 'banana', 'tangerine', 'grape', 'grape'] |
|  | C++:  #include <string>  #include <unordered\_map>  using namespace std;  // Defaults hash function and initial hash table number of buckets  typedef std::unordered\_map<std::string, std::string> ContactHT;  ContactHT contactM;  // output the contents of the unordered\_map  for (auto contact : contactM)  {  printf("%s:'%s'\n", contact.first.c\_str(), contact.second.c\_str());  }  Java:  for (Map.Entry<String, String> entry : contactMap.entrySet())  System.out.println(entry.getKey() + ":" + entry.getValue()); |
| **Subprogram Control**  Categories of subprograms:  **Simple (aka, macro)**  **Recursive**  **Interrupts**  **Coroutines**  **Scheduled Routines**  The other categories involve relaxing 1 or more of the assumptions. | Simple subroutines (macros)   * Conceptually, these can be implemented by copying the code in place. The effect of calling the subroutine would be obtained if the call was replaced by a copy of the body of the called subroutine before execution. * **5 Assumptions for simple subroutines**:  1. No recursion.  We couldn't copy the code into a recursive call. 2. All calls must be explicit.  Otherwise, the translation could not "copy" the subroutine. 3. Complete execution at each call.    * When a subroutine is invoked, an activation record is created.    * When it returns, the activation record is destroyed. 4. Single execution sequence.    * At any point during execution, exactly one subroutine has control.    * Others (calling subroutines) may be suspended.    * Other subroutines may be inactive if they were never called. 5. Immediate transfer of control at point of call. Immediately after a call, the called subroutine is invoked.   Prior to Fortran 90, FORTRAN was implemented as simple subroutines. |
| **Subprogram Control - Simple Subroutines**   * Need a simple activation record which records the return address and has memory for local variables. * The memory for local variables can be static memory which is loaded at program load. |  |
| **Subprogram Control - Recursive**  Relax assumption #1. The real issue is that each recursive call requires its own activation record supporting return address, parameters, and locals.  We can stack activation records giving a different activation for each call. | Simple activation record doesn't support multiple return addresses, separate copy of locals, and different parameters.  Issue with return addresses:  subA calls subB which calls subC    subC calls subB, causing subB to lose the subA return address |
| **Subprogram Control - Interrupts Causing Handlers to be Invoked**  Relax assumption #2. In some languages, programs can specify what to do when an interrupt happens. This can include specification of a subroutine to handle particular interrupts.  When the interrupt happens, the handler is invoked implicitly. | PL/I:  /\* specification of the interrupt handler \*/  ON STRINGRANGE CALL errorTooshort; //doesn’t happen immediately set up for later call  ON STACKOVERFLOW CALL handleOverflow;  ON OVERFLOW  BEGIN;  PUT SKIP LIST('Values too large');  END;  /\* we can enable handling of the interrupt \*/  (OVERFLOW): CALC: PROC( …); //turns on overflow checking  …  X = Y + Z\*W;  END CALC; |
| **Subprogram Control - Coroutines**  Relax assumption #3 (complete execution on each call). With coroutines, functions maintain the point of execution when resumed by other functions.  Not all algorithms are hierarchical. Hierarchical:    Some algorithms are more naturally coroutines:   * Each function in an interactive game may more easily be managed as a coroutine.   With coroutines:   * subroutines maintain the current execution point * only one coroutine is executing at a time (the others are suspended) * coroutines are **resume**d instead of just called from the top * still just 1 thread * perfect for parallel processing(game programs | Coroutines:    subC resumes subB    several resume sequences |
| **Subprogram Control - Scheduled Routines**  Relax assumption #5. Do not transfer to a subroutine when is it called; instead, advance based on events.  Simulation languages need to schedule calls or continuations based on an event (time-based, condition-based).   * Many widgets (e.g., people, telephone calls, cars) are in the system at once. * Logically, the widgets are executing in parallel. * Actually, only one widget has control at any given time.   The simulation software requires:   * time-based event list ordered by time * waiting condition list   Scheduler control subroutine:  Continue each widget that has its condition satisfied.  Advance clock to next scheduled event. | GPSS-like:  GENERATE UNIFORM(50,20)  ENTER QUEUE  SEIZE SERVER // waits until server isn't busy  LEAVE QUEUE  ADVANCE NORMAL(100,10) // advance clock  RELEASE SERVER  TERMINATE |