**Parameter Passing**

|  |  |
| --- | --- |
| **Parameter Passing**  **Arguments (i.e., actual parameters) -** passed to a function. Arguments might be identifiers, constants or expressions (which might involve function calls)  **Parameters (i.e., formal parameters) -** represent the arguments in the called function  Parameter transmission techniques:  **By Value -** the value of the argument is passed and becomes the value of the formal parameter  **By Reference -** *conceptually* a pointer (usually the location of the argument) is transmitted; function can modify the argument; often called **by address** parameter passing  **By Name -** transmit an unevaluated argument, allowing the called function to evaluate it  Note: many languages use By Value, but pass references to objects (i.e., the pointer's value is copied not the address of the variable). You will find many incorrect sites which state a language has by reference parameter passing; instead, it is better to call this **by value object reference**. | From the Overview:  C   * arrays are passed by reference * by default, everything else is passed by value; however, an address of the argument can be passed by using the & operator   // call  determineMinMax(gradeM, iNumEntries, &dMin, &dMax);  // function declaration  void determineMinMax(double gradeM[], int iNumEntries  , double \*pdMin, double \*pdMax)  {  …  if (gradeM[i] > \*pdMax)  \*pdMax = gradeM[i];  …  }  PL/I   * uses **by reference** * instead of the programmer dereferencing a parameter in the called function, the compiler did that under the covers.   /\* call \*/  determineMinMax(gradeM, iNumEntries, min, max);  /\* function declaration \*/  determineMinMax: PROC (gradeM, iNumEntries, min, max);  DCL gradeM(\*) FLOAT,  iNumEntries FIXED BIN,  min FLOAT,  max FLOAT;  …  IF gradeM(i) > max  max = gradeM(i);  …  END determineMinMax;  Java   * uses by value and by value object reference * when an argument is a reference to an object, a copy of that reference is passed. (This is not by reference parameter passing.) |
| **By Value Parameter Transmission Technique**   * not including **by value object reference** * the value of the argument is transmitted to the called subroutine and becomes the value of the formal parameter * subroutine cannot modify the argument since it only receives the value * side effects cannot occur   With **by value object reference**:   * a copy of the object reference passed; this is not a copy of the object * the referenced object can be modified if not immutable * the corresponding argument cannot be changed to reference a different object | **Example 1: Passing by Value**:   |  |  | | --- | --- | | int iX = 5;  sub(iX);  void sub(int iSX)  {  iSX += 10;  } |  |   **Example 2: Passing by Value Object Reference**:   |  |  | | --- | --- | | Student student = new Student(loginId);  Course course = new Course (courseNr);  …  student.withdraw(course);  // inside Student class:  public ReturnObj  withdraw (Course withdrawCourse)  {  …  withdrawCourse.seatsAvail++;  …  } |  |   Instead of by Value Object Reference, what would be different if course is passed to withdraw using by Reference? |
| **By Reference Parameter Transmission Technique**   * Conceptually the address of the argument is transmitted * subroutine can modify the argument * side effects can occur   **Implementations:**  **true reference:**   * pointer to actual location of the argument is passed * to reference the parameter, it has to be dereferenced (in C programmer does the dereference; other languages do it implicitly) | **Example 3: Using by reference to insert a new value in a binary tree in C and PL/I**  insertT(&pRoot, value);  NodeT \*insertT(NodeT \*\*pp, int value)  {  // If \*pp is null, this is where we want to insert.  if (\*pp == NULL)  {  \*pp = allocateNodeT(value);  return \*pp;  }  // does it match  if (value == (\*pp)->key)  return \*pp;  // which side should we follow  if (value < (\*pp)->key)  return insertT(&((\*pp)->pLeft), value);  else  return insertT(&((\*pp)->pRight), value);  }  PL/I - using by reference to insert a new value in a binary tree  INSERT(ROOT, VALUE);  INSERT: PROC(P, VALUE);  DCL P PTR,  VALUE FIXED BIN;  IF P = NULL THEN  DO;  P = ALLOCATE\_NODE(VALUE);  RETURN P;  END;  ELSE IF VALUE = P->KEY THEN  RETURN P;  ELSE IF VALUE < P->KEY THEN  RETURN INSERT (P->PLEFT, VALUE);  ELSE  RETURN INSERT (P->PRIGHT, VALUE);  END INSERT; |
| **An example showing by value and by reference**  **Trace: by value**  **Caller:**  **Q 5**  **R 2**  **A]**  **X 5 6**  **Y 2 7**  **Trace: by ref**  **Caller:**  **Q 5 6**  **R 2 7**  **A]**  **X @Q //address of Q**  **Y @R** | **Example 4: by value and by reference paramter passing using PL/I syntax:**  A: PROC (X, Y);  DCL X, Y;  X = X + 1;  Y = X + 1;  PUT LIST('A', X, Y);  END A;  **By value parameter transmission:**  DCL Q, R;  Q = 5;  R = 2;  CALL A (Q, R);  PUT LIST('CALLER', Q, R);  **Output:**  A 6 7  5 2  **By reference parameter transmission:**  same code.  **Output:**  A 6 7  6 7 |
| **By Reference Parameter Transmission Technique continued**  **Implementations continued:**  **copy & restore**   * just like **by value,** the value is copied to the parameter * after subroutine execution, restore the value of the argument from the parameter   **Note:** suppose a language passes by reference. What should be done if a constant or an expression is the argument? | **Repeat example 4 using Copy and Restore**  **Caller:**  **Q 5 6 RESTORE FROM A:**  **R 2 7**  **A]**  **X 5 6**  **Y 2 7**  **OUTPUT: A 6 7**  **6 7** |
| **Issues with Copy and Restore**   * Passing the same argument more than once * Referencing a nonlocal which is also an argument | **Example 5: True Ref vs Copy/Restore passing same arg more than once. Same function A, but different invoking code.**  A: PROC (X, Y);  DCL X, Y;  X = X + 1;  Y = X + 1;  PUT LIST('A', X, Y);  END A;  DCL Q;  Q = 5;  CALL A (Q, Q);  PUT LIST('CALLER', Q);  **Trace and Output for true reference:**  CALLER:  Q 5 6 7  A]  X @Q  Y @Q  Output:  A 7 7  Caller: 7  **Trace and Output for by reference using copy & restore (restore X before Y):**  Caller:  Q 5 ~~6~~ 7  A]  X 5 6  Y 5 7  Output:  A 6 7  Caller: 7 if restore x first 6 if restore y first  **Trace and Output for by reference using copy & restore (restore Y before X):**  Same as before  Output:  A 6 7  Caller: 7 if restore x first 6 if restore y first |
| **Another example using by Reference** | **Example 6: True Ref vs Copy/Restore referencing a nonlocal which is also an argument**  **Referencing a nonlocal**  A: PROC(X);  DCL X;  Q = Q + 1;  X = X + 1;  PUT LIST('A', Q, X);  END A;  Q = 5;  CALL A(Q);  PUT LIST('CALLER', Q);  **Trace and Output for true ref:**  Caller:  Q 5 6 7  A]  X @Q  Output:  A 7 7  Caller: 7  **Trace and Output for copy & restore:**  Caller”  Q 5 6  A]  X 5 6  Output:  A 6 6  6 |
| **By Name Parameter Transmission**  Pass unevaluated arguments which can be evaluated by the called subroutine.  ALGOL approach - whenever a **by name** parameter is referenced, it is evaluated in the calling environment.   * Simple variables are treated as by reference * constants are treated as by value * expressions involving variables as subscripts are implemented by a simple subroutine called a **thunk**. On each reference, it is reevaluated. We can assign a value to it. * expressions involving variables are implemented by a **thunk.** Assignments to the parameter are not allowed. | **Example 7: ALGOL by name approach using C syntax**  int a[] = {10,20,30,40,50};  int i=0;  sub(a[i]); // assuming ALGOL by name parameter transmission  void sub(int X)  {  for (i = 0; i < 5; i++)  {  printf("%d ", X); // evaluate when it is referenced  }  }  Output:  10 20 30 40 50  Suppose we declared the variable, **i,** also in sub as a local. What would be printed?  int a[] = {10,20,30,40,50};  int i=0;  sub(a[i]); // assuming ALGOL by name parameter transmission  void sub(int X)  {  int i  for (i = 0; i < 5; i++)  {  printf("%d ", X); // evaluate when it is referenced  }  }  10 10 10 10 10 because the I in calling environment didn’t change what change was the I in the local |
| **Jensen's Device**  The sum function uses by Name parameter passing and simply varies an index from a beginning value to an ending value.  // ALGOL by name approach using C syntax  int sum(int X, int Index, int beginVal, int endVal)  {  int retSum = 0;  for (Index = beginVal; Index <= endVal; Index += 1)  {  retSum += X;  }  return retSum;  } | **Example 8: Jensen's device**  **// Jensen's Device Example 8-1:**  int i;  printf("Example 8-1:", sum(i, i, 1, 5));  **// Jensen's Device Example 2:**  int i;  printf("Example 8-2:", sum(2\*i, i, 1, 5));  **// Jensen's Device Example 3:**  int i;  int A[] = {10, 5, 20}  printf("Example 8-3:", sum(A[i], i, 0, 2));  **// Jensen's Device Example 8-4:**  How would you sum along the main diagonal in a two-dim array?  int i;  int Two[5][5];  printf("Example 4:", sum( Two[i][i],i,0,4));  **// Jensen's Device Example 8-5:**  How would you sum along the entire contents of a two-dim array?  Hint: sum can call sum  int i;  int Two[5][5];  printf("Example 5:", sum( ??); |
| **Subroutines as Parameters**  When might it be useful to pass a subroutine as a parameter (i.e., not an evaluation of a function call, but the actual subroutine)?   * Plotting software - the function that returns Y values * Sorting software - provide a function that returns whether a value is less than another value * Area under a curve   C programming language:   * From the calling function (e.g., main), we pass the name of the function (e.g., ployfunc1) as the argument without any parameters (and without parentheses). * The invoked function (e.g., plot) which receives that argument must define its parameter as **a function pointer**. The declaration must also include the data type returned by the argument and the datatype of each of that argument's parameters. This aids the compiler in type checking parameters. | **Example 9:** **simple plotting software in C**  // Suppose you are plotting the coordinates of some function. We could  // pass a function when given an X value and returns a Y value to the  // plotting software.  double polyFunc1(double x);  double polyFunc2(double x);  // Notice the declaration of the 1st parameter, func, in this prototype  // declaration which is highlighted in green. The first double is the  // return type. The second double is the datatype of its only parameter.  void plot(double(\*func)(double), char \*pszTitle, double dLow, double dHi, double dInc);  void main()  {  plot(polyFunc1, "Plot 1: ", 0.0, 30.0, 1.0);  plot(polyFunc2, "Plot 2: ", 0.0, 30.0, 4.0);  }  double polyFunc1(double x)  {  return 4 \* cos(0.5\*x) + 6;  }  double polyFunc2(double x)  {  return 0.25\*x + 1.0;  } |
| * To call that argument as a function inside the invoked function, we must **dereference** the parameter. | **Example 9 continued:** **simple plotting software in C**  void plot(double(\*func)(double), char \*pszTitle, double dLow, double dHi, double dInc)  {  int i;  int j;  double x;  double y;  char cGridM[30][30];  memset(cGridM, '.', 900); // initialize the grid to dots  // include the x-axis where y = 0  for (j = 0; j < 30; j++)  cGridM[0][j] = '\_';  // include the y-axis where x = 0  for (i = 0; i < 30; i++)  cGridM[i][0] = '|';  // include the points  for (x = dLow; x < dHi; x += dInc)  {  y = (\*func)(x); // invoke the parameter which is a function  i = (int) (x);  j = (int) (y+0.499);  // check range  if (i >= 0 && i < 30 && j >= 0 && j < 30)  cGridM[j][i] = '\*';  }  printf("\n\n%30s\n", pszTitle);  for (i = 29; i >= 0; i--)  {  // print the Y axis label (every 4th value)  if (i % 4 == 0)  printf("\n%2d ", i);  else  printf("\n ");  for (j = 0; j < 30; j++)  printf("%c", cGridM[i][j]); // putc would be more  // efficient  }  // print the x-axis labels  printf("\n 0");  for (j = 4; j < 30; j += 4)  printf(" %2d", j);  printf("\n");  } |
|  | Plot 1:  |.............................  28 |.............................  |.............................  |.............................  |.............................  24 |.............................  |.............................  |.............................  |.............................  20 |.............................  |.............................  |.............................  |.............................  16 |.............................  |.............................  |.............................  |.............................  12 |.............................  |.............................  \*\*..........\*\*...........\*\*...  |..........\*..\*.........\*.....  8 |.\*....................\*...\*..  |.........\*....\*............\*.  |..\*..................\*.......  |........\*......\*............\*  4 |...\*............\*...\*........  |....\*..\*...........\*.........  |.....\*\*..........\*\*..........  |.............................  0 |\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  0 4 8 12 16 20 24 28 |
| **Subroutines as Parameters continued**  In languages which use static scoping and have nested functions, we have to pass more than just the address of the function. The compiler must generate code that also passes the **static scope pointer**. | **Example 10: Subroutine as a Parameter with nested static scope**  Using C-like syntax, but with nested static scope   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | void main()  {  int w, z= 0;  a(q); // pass function q to a  void a(void (\*fn)())  {  int r = 5;  z += 1;  (\*fn)();  if (z == 1)  a(q); // pass a diff q  print r;  void q()  {  int z = 0;  r += 10;  }  }  void q();  {  int y = 1;  w = 2;  }  } | function main   |  |  |  | | --- | --- | --- | | Symbol | Which BA | Offset/Addr | | w | 0 | 0 | | z | 0 | 4 | | a | 0 | @a | | q | 0 | @q outer |   function a   |  |  |  | | --- | --- | --- | | Symbol | Which BA | Offset/Addr | | fn | 0 | 0 parm | | r | 0 | 8 | | z | 1 | 4 | | a | 1 | @a | | q | 0 | @q inner |   function q inner (in a)   |  |  |  | | --- | --- | --- | | Symbol | Which BA | Offset/Addr | | z | 0 | 0 | | r | 1 | 8 |   function q outer (in main)   |  |  |  | | --- | --- | --- | | Symbol | Which BA | Offset/Addr | | y | 0 | 0 | | w | 1 | 0 | | |
| void main()  {  int w, z= 0;  a(q); // pass function q to a  void a(void (\*fn)())  {  int r = 5;  z += 1;  (\*fn)();  if (z == 1)  a(q); // pass a diff q  print r;  void q()  {  int z = 0;  r += 10;  }  }  void q();  {  int y = 1;  w = 2;  }  } | **Example 10 continued: Subroutine as a Parameter with nested static scope**  Tracing the code  Invoke main   |  |  |  | | --- | --- | --- | | α main | w  z 0 | α |   Invoke a passing q outer   |  |  |  | | --- | --- | --- | | α main | w  z 0 1 | α | | β a | fn @q outer, α  r 5 | β α |   Invoke fn which is q outer   |  |  |  | | --- | --- | --- | | α main | w 2  z 1 | α | | β a | fn @q outer, α  r 5 | β α | | θ q outer | y 1 | θ α |   Return to a and invoke a passing q inner   |  |  |  | | --- | --- | --- | | α main | w 2  z 1 2 | α | | β a | fn @q outer, α  r 5 | β α | | € a | fn @q inner, β α  r 5 | € α |   Invoke fn which is q inner   |  |  |  | | --- | --- | --- | | α main | w 2  z 1 2 | α | | β a | fn @q outer, α  r 5 15 | β α | | € a | fn @q inner, β α  r 5 | € α | | ∑ q  inner | z 0 | ∑ β α |   which r gets incremented by 10?  ?the one at beta since it was inside the surrounding scope for the last call? |
| **Closures**  In the previous example, the nonlocals referenced by the static chain are easily available since the activation record is still available. Another approach is to make the nonlocals available even after the activation record for the surrounding function is no longer active.  Closures provide the ability to reference surrounding statically-scoped data even when the surrounding function is no longer active.  Closures are available in C#, Ruby, JavaScript, some LISP implementations, and Python. | **Example 11 Part 1: Closure example in Python**   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | def begin(x):  # sum is nested inside the  # begin function  def sum(y):  nonlocal x  return x + y  return sum //this return the @ and the surrounding code  c1 = begin(1)  c2 = begin(5)  print(c1(3), c2(3)) | function begin   |  |  |  | | --- | --- | --- | | Symbol | Which BA | Offset /Addr | | x | 0 | 0 parm | | sum | 0 | @sum |   function sum   |  |  |  | | --- | --- | --- | | Symbol | Which BA | Offset /Addr | | y | 0 | 0 parm | | x | 1 | 0 | |   **Note:** the green highlighted code is all part of the function sum. Python uses indentation to syntactically indicate which code is within functions, loops, and conditionals. The statement "return sum" is code for the begin function. Since the sum function is nested within begin, sum's static pointer is to begin.  Trace is shown below. |
| # repeated the code here to avoid scrolling up  def begin(x):  # sum is nested inside begin function  def sum(y):  nonlocal x  return x + y  return sum  c1 = begin(1)  c2 = begin(5)  print(c1(3), c2(3)) | **Example 11 Part 2: Closure example in Python**  Invoke begin passing 1   |  |  |  | | --- | --- | --- | | α begin | x 1 | α |   Returns the function sum, but not an actual invocation of sum. This means that **begin** returns a **closure** which requires the environment of begin which is at α. Note that the activation record for begin is popped, but the global c1 exists referencing whatever it contained.  Globals   |  |  |  |  | | --- | --- | --- | --- | | Symbol | Type | BA | Offset /Addr | | c1 | closure | copy of α  x 1 | @sum |   Invoke begin passing 5   |  |  |  | | --- | --- | --- | | β begin | x 5 | β |   Returns the function sum, but not an actual invocation of sum. This means that **begin** returns a **closure** which requires the environment of begin which is at α. Note that the activation record for begin is popped, but the global c2 exists referencing whatever it contained.  Globals   |  |  |  |  | | --- | --- | --- | --- | | Symbol | Type | BA | Offset /Addr | | c1 | closure | copy of α  x 1 | @sum | | c2 | closure | copy of β  x 5 | @sum | |
| # repeated the code here to avoid scrolling up  def begin(x):  # sum is nested inside begin function  def sum(y):  nonlocal x  return x + y  return sum  c1 = begin(1)  c2 = begin(5)  print(c1(3), c2(3)) | **Example 11 Part 3: Closure example in Python**  Invoke c1(3) which is the sum function with the saved environment of α.  This establishes an activation record for the surrounding environment.   |  |  |  | | --- | --- | --- | | α c1 closure | x 1 | α | | € sum | y 3 | € α |   Returns x+y which is **4** and pops both activation records  Invoke c2(3) which is the sum function with the saved environment of β.  This establishes an activation record for the surrounding environment.   |  |  |  | | --- | --- | --- | | β c1 closure | x 5 | β | | ∑ sum | y 3 | ∑ β |   Returns x+y which is **8** and pops both activation records    Output:  4  8 |

**©2018 Larry W. Clark,** UTSA CS students may make copies for their personal use