**Scope**

|  |  |
| --- | --- |
| **Static vs Dynamic Scope** - based on language definition, non-local variables can be associated based on  **dynamic** the most recent association in the dynamic calling chain  **static** static physical structure of the code at translation time  **Static Scoping - File-Based**  Some languages have global variables which are defined outside of functions, but are available to either all functions or all functions within a file. Local variables with the same name take precedence over global variables. C and COBOL use this **file-based static scoping**.  **Static Scoping in C**  In addition to the external global variables described above, C has a concept of private to a file. Unfortunately, this is called **static** in C.  C also has the concept of retaining a value for a variable in a function even when the function returns (unlike automatic variables). Unfortunately, this is also called **static** in C.  **Static takes runners only, if not then go to the top.**  C also allows nesting of blocks of code -- each of which can have their own local variables.  Python has a File-Based Static Scoping which we will discuss later in the semester. | Example 1: static scoping in C  int iX = 100;  int iY = 200;  int iZ = 300;  void funcA()  {  int iZ = 400;  iX += 10;  iY += 10;  printf("funcA: %d %d %d\n", iX, iY, iZ);  funcB();  }  void funcB()  {  iX += 5;  iY += 5;  iZ += 5;  printf("funcB: %d %d %d\n", iX, iY, iZ);  }  Trace:  Globals  iX 100 110 115  iY 200 210 215  iZ 300 305  funcA  iZ 400  funcB  Output:X Y Z  funcA: 110 210 400  funcB: 115 215 305 |
| **Static Scoping - Nested Functions**  Some languages have functions declared within functions. The internal function can reference the surrounding functions' variables as non-locals. Again, local variables have precedence. PL/I, ALGOL, and Pascal use this **nested-function static scoping**. | Example 2: Static Scoping with PL/I nested functions (procedures in PL/I)  A: PROC;  DCL X FIXED BIN,  Y FIXED BIN;  X = 10;  Y = 11;  CALL B;  B: PROC; //B declared in A  DCL Z FIXED BIN,  Y FIXED BIN;  X = 20;  Y = 21;  Z = 22;  CALL C;  C: PROC; //Declared in B  DCL Q FIXED BIN;  X = 30;  Y = 31;  Z = 32;  Q = 33;  CALL D;  END C;  END B;    D: PROC; //Declared in A  DCL W FIXED BIN;  X = 40;  Y = 41;  W = 44;  END D;  END A;  What can procedure C reference?  C: Q  B: Z, Y  D: n/a  A: X, B, D,(A)  Can function C reference variable W in function D?  No |
| **Dynamic Scoping**  Some languages use dynamic scoping where non-local references are resolved based on the most recent calling chain. This is used in SNOBOL, Perl, and some older dialects of LISP.  int iX = 100;  int iY = 200;  int iZ = 300;  void funcA()  {  int iZ = 400;  iX += 10;  iY += 10;  printf("funcA: %d %d %d\n", iX, iY, iZ);  funcB();  }  void funcB()  {  iX += 5;  iY += 5;  iZ += 5;  printf("funcB: %d %d %d\n", iX, iY, iZ);  }  Call A | Example 3: Dynamic Scoping illustrated using C syntax from example 1  What would the output be if using dynamic scoping?  Trace:  Globals  iX 100 110 115  iY 200 210 215  iZ 300  funcA  iZ 400 405  funcB  Output:  funcA: 110 210 400  funcB: 115 215 405 |
| **1.**  **Globals**  **X**  **Y**  **Z**  **funA**  **x**  **funcB**  **x**  **funcC**  **y**  **z**  **2. Second calling sequence**  **Globals**  **X**  **Y**  **Z**  **funA**  **x**  **funcC**  **y**  **z**  **funcB**  **x** | Example 4: Another Dynamic scoping illustrated using C syntax  int x;  int y;  int z;  void funcC()  {  double y;  int z;  x = 11;  y = 21;  z = 31;  }  void funcB()  {  double x;  x = 12;  y = 22;  z = 32;  }  void funcA()  {  int x;  x = 10;  y = 20;  z = 30;  }  Questions:  1. Suppose funcA invokes funcB which invokes funcC:  1.1 Which x,y,z does funcA reference? funcA: x, Globals: y, z  1.2 Which x,y,z does funcB reference? funcB: x, Globals: y, z  1.3 Which x,y,z does funcC reference? funcB: x, funcC: y, z  2. If funcA invokes funcC which invokes funcB, what is different?  funcC is being called from funcA instead of funcB  for function FuncC: funcA:x, funcC: y, z |
| **Perl Nonlocal Variables**  By default in Perl, variables are global, allowing them to be referenced or set almost anywhere. Usually, references are to the global variables.  Perl (unfortunately) used the term **local** to start a new nonlocal **dynamic** scope:  local *variableName*;  In the calling chain after this declaration, nonlocal references will be to this copy of the variable.  After **local** was already used for defining that new dynamic scope, Perl needed locals. To define a variable as a local (CS term not the Perl term), specify  **my** *variableName*;  That makes that variable **private** to the function instead of using the global. Additionally, a **my** variable isn't available to dynamic scope. | Example 5: Nonlocal references in Perl  #!/usr/bin/perl  sub mess\_with\_it {  $x=$x+10;  }  sub funcA {  my $x=20;  mess\_with\_it();  print "A $x\n";  }  sub funcB {  local $x=20;  mess\_with\_it();  print "B $x\n";  }  $x=100;  funcA();  print "Main $x\n";  funcB();  print "Main $x\n";  Trace:  Globals  x 100 110  funcA  x 20  funcB  x 25 35  Output:  A 20  Main 110  B 35  Main 110 |
|  |  |
| **Disadvantages of Dynamic Scoping**   * Local variables in one function are not protected from modification in any function called by that function. * The data characteristics of a non-local variable are not understood until the non-local is referenced at run-time. |  |
| **Data Control Operations**  Four major data control operations:  **Activating** creating an association between an identifier and a program or data object  **Hiding** making an active association invisible typically due to the precedence of a local reference  **Unhiding** making a hiden association visible  **Deactivating** destroying an association between an identifier and its associated object  In C, we have the following for automatic variables:  Activating   * automatic variables on block/function entry   Hiding   * when a function calls another function, the calling function's local variables are not visible to the called function * when a function is called, its locals with the same name as globals cause those globals to be hidden   Unhiding   * making local variables visible when a call returns   Deactivating   * when a function exits, its automatic variables are destroyed   **Referencing Environment -** the set of currently active associations (includes global, current locals, current parameters, and current non-locals) | Example 6: C static scope   1. // globals 2. int ix = 10; 3. int iy = 20; 4. int iz = 30; 5. void funcC() 6. { 7. int iy = 23; 8. int iz = 33; 9. } 10. void funcB() 11. { 12. int ix = 12; 13. int iq = 52; 15. funcC(); 16. } 17. void main() 18. { 19. int iz = 31; 20. int iw = 41; 22. funcB(); 23. }     Assuming execution begins at main:  1. What is referencable at line 22?  **locals:** iz, iw  **globals:** ix, iy  **hides:** global, iz  2. What is referencable at line 15?  **locals:** funcB: ix, iq  **globals:** iy, iz  **hides:** global: ix  3. What is referencable at line 9?  **locals:** funcC: iy, iz  **globals:** ix  **hides:** globals: iy, iz |
| **Globals**  **ix**  **iy**  **iz**  **Main**  **iz**  **iw**  **funcB**  **ix**  **iq**  **funcC**  **iy**  **iz** | Example 7: Using C syntax, but assuming Dynamic scope   1. // globals 2. int ix = 10; 3. int iy = 20; 4. int iz = 30; 5. void funcC() 6. { 7. int iy = 23; 8. int iz = 33; 9. } 10. void funcB() 11. { 12. int ix = 12; 13. int iq = 52; 15. funcC(); 16. } 17. void main() 18. { 19. int iz = 31; 20. int iw = 41; 22. funcB(); 23. }     Assuming execution begins at main and using a dynamic scope :  1. What is referencable at line 22?  **locals:** main: iz, iw  **nonlocals:** globals: ix, iy  **hides:** globals: iz  2. What is referencable at line 15?  **locals:** funcB: ix, iq  **nonlocals:** main: iz, iw, globals: iy  **hides:** globals: ix  3. What is referencable at line 9?  **locals:** funcC: iy,iz  **nonlocals:** funcB: ix, iq, main: iw  **hides:** main: iz, globals : ix, iy, iz |
| **Implementing Static Scope**  With nesting of functions for static scope, each function needs a mechanism for referencing the non-local variables. This may involve many surrounding functions. For automatic variables, translation requires knowing:   * The offset of the variable within the activation record's based address. * The base address for each referenceable activation record (local and each non-local).   During compilation, we have a symbol table for each variable. It will contain the offset and which base address to use.  BA = activation records Levels OUT  Offset / addy =  BA[0] - Base Address for local, It’s a local  BA[1] - Base Address for surrounding function  BA[2] - Base Address for the function that surrounds the surrounding function  BA[i] - Base Address for activation record that is i static chain links away | From example 2 **EXAM QUESTION**   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | A: PROC;  DCL X FIXED BIN, // 2 bytes  Y FIXED BIN; // 2 bytes  X = 10;  Y = 11;  CALL B;  B: PROC;  DCL Z FIXED BIN,  Y FIXED BIN;  X = 20;  Y = 21;  Z = 22;  CALL C;  C: PROC;  DCL Q FIXED BIN;  X = 30;  Y = 31;  Z = 32;  Q = 33;  CALL D;  END C;  END B;    D: PROC;  DCL W FIXED BIN;  X = 40;  Y = 41;  W = 44;  END D;  END A; | function A   |  |  |  | | --- | --- | --- | | Symbol | Which BA | Offset/Address | | X | 0 | 0 | | Y | 0 | 2 | | B | 0 | @B | | D | 0 | @D |   function B   |  |  |  | | --- | --- | --- | | Symbol | Which BA | Offset/Address | | Z | 0 | 0 | | Y | 0 | 2 | | X | 1 | 0 | | C | 0 | @C |   function C   |  |  |  | | --- | --- | --- | | Symbol | Which BA | Offset/Address | | Q | 0 | 0 | | Z | 1 | 0 | | Y | 1 | 2 | | X | 2 | 0 | | D | 2 | @D |   function D   |  |  |  | | --- | --- | --- | | Symbol | Which BA | Offset/Address | | W | 0 | 0 | | X | 1 | 0 | | Y | 1 | 2 | | |
| **Implementing Static Scope - approach 1: follow static chains**   * Within an activation record, include the base address of the surrounding function (BA[1]). * When a non-local is referenced, follow i static chains.   A: PROC;  DCL X FIXED BIN,  Y FIXED BIN;  X = 10;  Y = 11;  CALL B;  B: PROC;  DCL Z FIXED BIN,  Y FIXED BIN;  X = 20;  Y = 21;  Z = 22;  CALL C;  C: PROC;  DCL Q FIXED BIN;  X = 30;  Y = 31;  Z = 32;  Q = 33;  CALL D;  END C;  END B;    D: PROC;  DCL W FIXED BIN;  X = 40;  Y = 41;  W = 44;  END D;  END A; | Invoke A   |  |  |  | | --- | --- | --- | | A | X 10  Y 11 | null |   invoke B   |  |  |  | | --- | --- | --- | | A | X ~~10~~ 20  Y 11 | null | | B | Z 22  Y 21 |  |   invoke C   |  |  |  | | --- | --- | --- | | A | X ~~10~~ ~~20~~ 30  Y 11 | null | | B | Z ~~22~~ 32  Y ~~21~~ 31 |  | | C | Q 33 |  |   invoke D   |  |  |  | | --- | --- | --- | | A | X ~~10~~ ~~20~~ ~~30~~ 40  Y ~~11~~ 41 | null | | B | Z ~~22~~ 32  Y ~~21~~ 31 |  | | C | Q 33 |  | | D | W 44 |  |   How did the system know which activation record surrrounded D? where in the symbol table did it need that knowledge?  Can we improve on following the chain? (note that it isn't a search) |
| **Implementing Static Scope - approach 2: environment vector**   * Store an array of the base addresses of all referenceable activation records in each activation record. That is called an **environment vector**. * The length of the environment vector for each function is known at translation time. | Invoke A   |  |  |  | | --- | --- | --- | | α A | X 10  Y 11 | α |   invoke B   |  |  |  | | --- | --- | --- | | α A | X ~~10~~ 20  Y 11 | α | | β B | Z 22  Y 21 | β α |   Alpha becomes surrounding scope  invoke C   |  |  |  | | --- | --- | --- | | α A | X ~~10~~ ~~20~~ 30  Y 11 | α | | β B | Z ~~22~~ 32  Y ~~21~~ 31 | β α | | θ C | Q 33 | θ β α |   invoke D   |  |  |  | | --- | --- | --- | | α A | X ~~10~~ ~~20~~ ~~30~~ 40  Y ~~11~~ 41 | α | | β B | Z ~~22~~ 32  Y ~~21~~ 31 | β α | | θ C | Q 33 | θ β α | | € D | W 44 | € α | |
| **Is recursion an issue?**  A: PROC;  DCL X FIXED BIN,  Y FIXED BIN;  X = 10;  Y = 11;  CALL B;  B: PROC;  DCL Z FIXED BIN,  Y FIXED BIN;  X = X + 10;  Y = 21;  Z = 22;  CALL C;  C: PROC;  DCL Q FIXED BIN;  Y = Y + 30;  Z = Z + 30;  Q = 33;  IF X = 20 THEN  DO;  X = 30;  CALL B;  END;  END C;  END B;  END A;  In the symbol table for C, what additional entry is needed?  function C   |  |  |  | | --- | --- | --- | | Symbol | Which BA | Offset/Address | | Q | 0 | 0 | | Z | 1 | 0 | | Y | 1 | 2 | | X | 2 | 0 | | **B** | 2 | @B |   B declared in A which is 2 out | Invoke A   |  |  |  | | --- | --- | --- | | α A | X 10  Y 11 | α |   invoke B   |  |  |  | | --- | --- | --- | | α A | X ~~10~~ 20  Y 11 | α | | β B | Z 22  Y 21 | β α |   invoke C   |  |  |  | | --- | --- | --- | | α A | X ~~10~~ ~~20~~ 30  Y 11 | α | | β B | Z ~~22~~ 52  Y ~~21~~ 51 | β α | | θ C | Q 33 | θ β α |   invoke B   |  |  |  | | --- | --- | --- | | α A | X ~~10~~ ~~20~~ 30 40  Y 11 41 | α | | β B | Z ~~22~~ 52  Y ~~21~~ 51 | β α | | θ C | Q 33 | θ β α | | € B | Z 22  Y 21 | € α |   Invoke C   |  |  |  | | --- | --- | --- | | α A | X ~~10~~ ~~20~~ 30 40  Y 11 | α | | β B | Z ~~22~~ 52  Y ~~21~~ 51 | β α | | θ C | Q 33 | θ β α | | € B | Z ~~22~~ 52  Y ~~21~~ 51 | € α | | ∑ C | Q 33 | ∑ € α | |

**©2020 Larry W. Clark,** UTSA CS students may make copies for their personal use