TEACHING PARAMETER PASSING BY EXAMPLE USING THUNKS IN C AND C++

Joseph Bergin
Pace University
New York, NY 10038
(212) 346-1499
BERGINF @ PACEVM.BITNET

Stuart Greenfield
Marist College
Poughkeepsie, NY 12601
(914) 575-3000

INTRODUCTION

Most programming language survey textbooks [2][3][4][5][6][7]discuss parameter passing at length. Historically this has been an important topic of research and a number of different parameter passing mechanisms have been employed by various languages. This paper discusses a method by which students may simulate various mechanisms using currently popular languages which do not support those mechanisms directly. Programming exercises can thus be used to further the understanding of the issues involved in parameter passing. For example, Modula-2, some enhanced versions of Pascal (e.g., Turbo Pascal and Think Pascal), and ANSI C and C++ provide the capability to simulate the "pass-by-value-result" and "pass-by-name" mechanisms. An earlier version of this paper [1] focused on simulating different mechanisms using Modula-2 and Pascal. Here we present a generalization that unifies the mechanisms into a single one. Examples are implemented in C and include an application of Jensen's device [2][6][9]. Note that we shall be employing the "C subset of C++" in the following, rather than ANSI-C itself.

It is assumed that the reader is already familiar with the pass-by-value and pass-by-reference mechanisms. In order to simulate the other mechanisms in C, we employ function types and function parameters.

A study of available parameter passing mechanisms generates insight into several subtleties of language design. This is especially true of the effects of the interactions of language features such as the relationship between block structure and the problem of non-local names, and parameter semantics and the problems of variable access and variable aliasing.

In C, aliasing of a variable may occur through using the same variable for two different formal parameters. Aliasing may also occur if a global variable is passed as an argument in a situation in which the routine called also references that same global as a non-local name. Aliasing may occur even in the case of value parameters when the types represent pointer variables.

Variable access is an important issue because the state of the machine in which a variable is accessed may affect the address and/or the value of the variable to which the access refers. For example, if we make an array subscript reference, as in A[n], the expression may refer to any one of a collection of variables: the cells of the array. In C, when such an expression or a pointer to it is passed as an actual parameter the expression (which is an address in the case of a pointer) is evaluated once, at subroutine invocation time and the result is used throughout the body of the subroutine as the value of the formal parameter. This is the case even though that subroutine may have access to n, and may change the value of n. This fixed access to a variable is not necessarily true for the pass-by-value-result or pass-by-name mechanisms.

THE PASS-BY-NAME MECHANISM

Parameter passing using the pass-by-name mechanism is a means by which the "in-out" passing mode may be implemented. If a formal parameter has been passed using the pass-by-name mechanism, then each time it is referenced, the address of its corresponding actual parameter is recalculated and is used as the target for the reference. The simplest way to implement pass-by-name is by employing macro expansion to replace the names representing the formal parameters by the names representing the corresponding actual parameters everywhere in the body of the called function. Then execute the resulting code. Certainly such a methodology may require recompilation of the body of the function for each call, so that the addresses of non-local names may be recalculated.

An alternative approach involves passing an actual parameter access function to the called routine in place of the actual parameter. This access function provides a method for repeated access to either the value of an actual parameter or to its address. Such an access function has been termed a "thunk" (by Peter Z. Ingerman, an early implementor of Algol)[8]. A thunk is a parameterless function that returns the address of the particular actual parameter for which it is written. In Algol, a language employing pass-by-name, thunks are produced automatically by the compiler. For simulation purposes we may create and pass these functions.

SIMULATING PASS-BY-NAME USING THUNKS

If we are working in a programming language such as C, then we can declare the template for a thunk through the use of a function type declaration, such as

```
typedef float* (*floatThunk)(void);
```

which declares a floatThunk to be a pointer to a function that takes no parameters and returns a pointer to a float, which may be interpreted as the address of a float.

Now, in order to simulate the pass-by-name mechanism, we create a thunk corresponding to the actual parameter that is to be passed by name and pass the thunk instead. Within the body of the called function each access of a parameter passed by name is replaced by a dereference of an application of the corresponding thunk. The effect of this is the recalculation of the address of the actual parameter each time the formal parameter is accessed, rather than once at the beginning of the function, as is done using pass-by-reference.

We offer the following steps to simulate the pass-by-name mechanism in C (Similar steps may be written for other languages that support function types or function parameters.):

step 1. Declare a thunk type for each distinct type that is to be passed by name using the syntax:
typedef <aType> * (* <aName>)(void);
step 2. Code a thunk for each actual name parameter such that it returns the address of that actual parameter. This thunk is created within the name scope of the actual parameter. If the actual parameter is a local then the address of this local will need to be stored in a global temporary or, alternatively, passed to the thunk.
step 3. Construct the called function heading such that the formal name parameters are the corresponding thunks. Use parameter names that are "fresh names", not otherwise used in the context of the function.
step 4. In the body of the called function any reference to a name parameter is replaced by a dereference of a call of the corresponding thunk.

As an example, suppose we wish to simulate the following "function" and call, where the pair of parameters are passed by name:

```
void p(   PASSBYNAME int x,
          PASSBYNAME float y){
     whatever...  x  ....  y  ...whatever
}; // the function itself

p(n,m); // the call
```

Developing our four steps, the simulation is:

```
/* step 1 */
     typedef  int* (*intThunk)(void);
     typedef float* (* floatThunk)(void);

/* step 2 */
     int* nThunk(void){ return &n; };
     float* mThunk(void){ return &m;  };

/* step 3 */
     void p(  intThunk  xT,
              floatThunk  yT){
/* step 4 */
     whatever...    *(xT())  ....
                    *(yT())  ...whatever
     };
```

The call is then p(nThunk, mThunk);

AN EXAMPLE USING THE PASS-BY-NAME SIMULATION

Pass by name has more than historical interest. While it is complicated and difficult to use without error, it is a very powerful device. Using it, it is possible to write very general code to manipulate data structures, including recursively defined structures. Jensen's device (due to Jorn Jensen) applies a process to an array, permitting great flexibility, including specification of the operation to be performed and even of the structure of the array. In particular the same process may be made to operate on an array of scalars or on an array of vectors. Consider the following code, in which we pass the index to the array by reference and an element of the array by name using vSubiThunk.

```
int i;
int v[5] = {19,4,3,2,6};

// step 1
typedef int* (*intThunk)(void);

// step 2
int* vSubiThunk(void){ return &(v[i]); };

// step 3
int sum(int & i,int L,int U, intThunk A){
     int s=0;
     i = L;          // initialize the index
     while (i <= U) {
// step 4
       s += *A();// add next element
        i++;         // update the index
     };
  return s;
};

void main(void){
     int z = sum(i,0,4,vSubiThunk);
};
```

The above call will produce the sum of the scalars in array v. (Note that here we call and dereference the thunk on only the right side of an assignment. It is also allowable to call and dereference it on the left side.) Now however, suppose that we wish to sum the elements of a two dimensional array. The same sum function will work if we replace the call with one using a more sophisticated thunk:

```
int i;
int v[5][3] = {19,4,3,2,6,4,7,21,8,3,1,9,6,15,2};

typedef int* (*intThunk)(void);

int* vSubijThunk(void){
     return &(v[i][j]);
};
```

```
int sum(int & i,int L,int U, intThunk A){
      int s=0;
      i = L;     // initialize the index
      while (i <= U) {
        s += *A();//add next element
         i++;   // update the index
         };
      return s;
};

int glob;  // a global temporary
int * arrayThunk(void){
      glob = sum((j,0,2,vSubijThunk);
            // Note the “two-level” thunk.  We call a thunk
            // from another.
      return &glob;
};

void main(void){
      int z = sum(i,0,4,arrayThunk);
};
```

THE PASS-BY-VALUE-RESULT MECHANISM

Parameter passing using the pass-by-value-result mechanism (also known by the name copy-in-copy-out) is also a means by which the "in-out" passing mode may be implemented. However, unlike pass-by-name and pass-by-reference, the called function does not have direct access to the actual parameters.

Stating merely that a set of parameters are to be passed by value-result leaves us with a few ambiguities. Although the order in which actual parameters are "copied-in" to the called function does not have an affect on the returned results, the "copy-out" order of those parameters, in general, does. Thus the copy back order should be specified -- left-to-right or right-to-left. Furthermore, the address of the actual parameters may be calculated at copy-in time only or may be calculated twice, at copy-in time and copy-out time. Therefore, we cite four different versions of the pass-by-value-result mechanism, namely,

- address calculated once, copy back left-to-right
- address calculated once, copy back right-to-left
- address calculated twice, copy back left-to-right
- address calculated twice, copy back right-to-left

The above versions will be referred to as **vr1LR**, **vr1RL**, **vr2LR**, and **vr2RL**, respectively, throughout the remainder of this paper.

If addresses are calculated once, copies of the actuals are made and assigned to locally allocated variables (the formal pass-by-value-result parameters) of the called function at the time of invocation. The function then executes using those local variables. At the time of return, the values of the locals are "copied back" in the appropriate order to the actual parameters whose addresses had been determined at subprogram entry time. If addresses are calculated twice, a recalculation of the actual addresses occurs at return time. In this latter case, changing some variable within the subprogram may affect where the final values are copied back to.

Particular forms of this mechanism have been used in some implementations of FORTRAN[2][3] and are employed automatically by some versions of Ada[2][4] to implement the "in-out" passing mode in at least some cases, depending on the structure of the actuals being passed. One advantage of this scheme is that it is possible to implement it in such a way that actual parameters need not be limited to variables but may be expressions. In this latter case the "copy-back" to the actual would be omitted by the compiler. FORTRAN implementations employing the pass-by-value-result mechanism are able to simulate both "in-out" mode and strictly "in" mode passing with but one mechanism in this manner[2]. This leads to a bit more flexibility for the programmer in those cases in which a particular parameter need not be returned to the calling environment.

SIMULATING PASS-BY-VALUE-RESULT USING THUNKS

In our earlier paper [1] there appear a number of ad-hoc methods for simulating the different versions of this mechanism. Here we unify the technique with that for pass-by-name, by employing thunks. However, if the address is to be computed only once we shall need to remember the address of the parameters from the beginning of the invoked function until its end. We shall find it instructive to have an additional address type, which in C is just:

```
typedef int* address;
```

We also use thunks as formal (physical) parameters, creating new names which do not clash with other names in the context of the called function. Then we use local variables as "pseudo-parameters" (logical parameters). The physical parameters are used to set the logical parameters (and in the case of **vr1LR** and **vr1RL**, their addresses) on function entry and copied back in the appropriate order upon exit. For **vr2LR** and **vr2RL,** the address of each simulated value-result actual parameter is recalculated just prior to the return, in the appropriate copy-back order.

We offer the following steps to simulate the pass-by-value-result mechanism in C.

For **vr1LR** and **vr1RL**: Perform steps 1, 2, and 3 of pass-by-name and then:

step 4. Declare a local “logical parameter” for each such passed parameter and an address corresponding to each logical parameter.

step 5. Prior to the subprogram code, "copy" the parameter values "in" to, the corresponding locally declared addresses and logical parameters.

step 6. Let the subprogram code operate on the logical parameters. The body of the function is unchanged.

step 7. After completion of the function code, "copy" the locals "back" to the previously computed addresses, left-to-right or right-to-left, as appropriate.

For **vr2LR** and **vr2RL**:
Replace step 4, 5, and 7 in the above by:
step 4. Declare a local "logical parameter" for each such passed parameter.
step 5. Prior to the subprogram code, "copy" the parameter values "in" to, the corresponding logical parameters by dereferencing a call to its thunk.
step 7. After completion of the function code, "copy" the locals "back" to the destinations by again dereferencing a call to the thunk. Do this left-to-right or right-to-left, as appropriate.

As an example, suppose we wish to simulate the following where the parameters are passed by value-result:

```
void p(VALRES int x, VALRES int y){
...whatever ... x ... y ...whatever
};
```

The simulation for **vr1LR** (or **vr1RL**) is:

```
/* step 1 */
    typedef int*  address;
    typedef address (*intThunk)(void);
/* step 2 */
address aT(void){ return &a; };
address bT(void){ return &b; };
/* step 3 */
    void p(intThunk xP,  intThunk yP){
/* step 4 */
    int x,y; // logical parameters
    address xA, yA; // the addresses
/* step 5 */
    xA = xP();  yA = yP();  // get  addresses
    x = *xA; y = *yA;   // get values
/* step 6 */
    same ...whatever ... x ... y ...whatever
/* step 7 */
    *xA = x; *yA = y;
    //  or *yA = y; *xA = x;  copy back
    };
```

And the simulation for **vr2LR** (or **vr2RL**) is:

```
/* step 1 */
    typedef int*  address;
    typedef address (*intThunk)(void);
/* step 2 */
address aT(void){ return &a; };
address bT(void){ return &b; };
/* step 3 */
    void p(intThunk xP, intThunk yP){
/* step 4 */
    int x,y;  // logical parameters
/* step 5 */
    x = *xP();  y = *yP();  // copy in
/* step 6 */
    same ...whatever ... x ... y ...whatever
/* step 7 */
    *xP() = x; *yP() = y;
        // or  *yP() = y;  *xP() = x;   copy back
    };
```

To simulate the call p(a,b) we use p(aT,bT) instead.

AN EXAMPLE USING THE PASS BY-VALUE-RESULT SIMULATION

Although, for the bulk of applications, the use of any of the four versions of the pass-by-value-result mechanism (as well as the pass-by-reference and pass-by-name mechanisms) leaves the computing machine in identical states, we can easily find those that do not. As one such application we offer the program shown below, which performs its indicated task illustrating **vr1LR** and **vr2RL** modes of "in-out" for the simple "function":

```
void trial (INOUT int i, INOUT int j, INOUT int k){
    j = 1;
    k = i + j;
};
```

The state in which the memory is left at the return from each call is given in Table 1. Note that only **vr1LR** leaves the memory in the same state as the pass-by-reference mechanism and the results of employing the four different versions of pass-by-value-result are distinct as the reader may easily verify. The code for two of the four cases is shown below.

```
int i=2;  // input values for the function calls
int v[2] = {1,2,3}; // reset for each call

typedef int* (*intThunk)(void);
typedef int* address;

int* vSubiThunk(void){ return &(v[i]); };

int* iThunk(void){ return &i; };

void vr1LR ( intThunk ip,
             intThunk  jp,
             intThunk kp){
    int  i,j,k;
    address ia,ja,ka;
    ia = ip(); ja = jp(); ka = kp();
    i = *ia; j = *ja; k = *ka;
    j = 1;
    k = i + j;
    *ia = i; *ja = j; *ka = k;
}

void vr2RL ( intThunk ip,
             intThunk  jp,
             intThunk kp){
    int  i,j,k;
    i = *ip(); j = *jp(); k = *kp();
    j = 1;
    k = i + j;
    *kp() = k; *jp() = j; *ip() = i;
}
```

The main routine, in outline form, is:

```
void main(void){
. . .
i = 2;  v[0] = 1; v[1] = 2; v[2] = 3;
vr1LR(vSubiThunk,iThunk,vSubiThunk);
cout <<“vr1LR      “<<n<<v[0]<<v[1]<<v[2]<<“\n”;
. . .
i = 2;  v[0] = 1; v[1] = 2; v[2] = 3;
vr2LR(vSubiThunk,iThunk,vSubiThunk);
cout <<“vr2LR“<<n<<v[0]<<v[1]<<v[2]<<“\n”;
. . .
};
```

mechanism	n	v[0]	v[1]	v[2]
value	2	1	2	3
reference	1	1	2	4
vr1LR	1	1	2	4
vr1RL	1	1	2	3
vr2LR	1	1	4	3
vr2RL	1	1	3	4
name	1	1	3	3

Table 1.

SIMULATING PASS-BY-VALUE AND PASS-BY-REFERENCE

In closing, we note that thunks may also be used to simulate value parameters by employing this technique without the copy back phase, and also reference parameters by employing only the addresses used in vr1LR and dereferencing them for each use of the logical parameters. The output of these function calls with the same inputs as before, is shown as part of table 1.

```
void value ( intThunk ip,
             intThunk  jp,
             intThunk kp){
    int  i,j,k;
    i = *ip(); j = *jp(); k = *kp();
    j = 1;
    k = i + j;
};

void reference ( intThunk ip,
             intThunk  jp,
             intThunk kp){
    address i,j,k;
    i = ip(); j = jp(); k = kp();
    *j = 1;
    *k = *i + *j;
};
```

For completeness, we also show the code for the pass-by-name simulation for this example. Its output is included in table 1 as well.

```
void name {intThunk ip,
           intThunk  jp,
           intThunk kp){
*ip() = 1;
*kp() = *ip() + * jp();
};
```

An exercise that gets students thinking hard about these mechanisms, is to construct a single function body that behaves differently for each of the seven mechanisms discussed here, and to simulate the seven mechanisms, proving their solution. The function body shown here has six different results. The seventh can be achieved by generating an alias within the function by setting a global variable that happens to also be passed as an argument. This can make reference parameters behave differently from value-result parameters. Sethi [4] has a similar exercise, though without the variations on value-result parameters.

REFERENCES

[1] Bergin, J. & Greenfield, S, “Programming Experience with Early Parameter Passing Mechanisms using Modula-2 and Pascal,” Proceedings of the Seventh Annual Eastern Small College Computing Conference, Nov. 1991. May be obtained from Pace University as Technical Report # 46, November 1991.

[2] MacLennan, B., "Principles of Programming Languages", 2nd Ed., Holt, Rinehart and Winston, 1987

[3] Sebesta, R., "Concepts of Programming Languages", Benjamin/Cummings, 1989

[4] Sethi, R., "Programming Languages: Concepts and Constructs", Addison- Wesley, 1989

[5] Ghezzi, C. & M. Jazayeri, "Programming Language Concepts", 2nd Ed., Wiley, 1987

[6] Ledgard, H. & M. Marcotty, "The Programming Language Landscape", 2nd Ed., SRA, 1986

[7] Pratt, T., "Programming Languages: Design and Implementation", 2nd Ed., Prentice-Hall, 1984

[8] Ingerman, P., "Thunks", Communications of the ACM, Vol.4 No.1, 1961

[9] Wexelblat, R. (editor), "History of Programming Languages", Academic Press, 1981