CSE 307: Principles of Programming Languages

Procedures and Parameter Passing

R. Sekar

Topics

Control Statements (Continued)

- Procedure calls:
  - Communication between the calling and the called procedures takes place via parameters.
- Semantics:
  - substitute formal parameters with actual parameters
  - rename local variables so that they are unique in the program
    - In an actual implementation, we will simply look up the local variables in a different environment (callee's environment)
    - Renaming captures this semantics without having to model environments.
  - replace procedure call with the body of called procedure

Section 1

Parameter Passing Mechanisms

Parameter-passing semantics

- Call-by-value
- Call-by-reference
- Call-by-value-result
- Call-by-name
- Call-by-need
- Macros

Call-by-value

- Evaluate the actual parameters
- Assign them to corresponding formal parameters
- Execute the body of the procedure.

```
int p(int x) {
   x =x +1 ;
   return x ;
}
```

- An expression y = p(5+3) is executed as follows:
  - evaluate 5+3 = 8, call p with 8, assign 8 to x, increment x, return x which is assigned to y.

Call-by-value (Continued)

- Preprocessing
  - create a block whose body is that of the procedure being called
  - introduce declarations for each formal parameter, and initialize them with the values of the actual parameters
- Inline procedure body
  - Substitute the block in the place of procedure invocation statement.

Call-by-value (Continued)

- Example:

```
int z;
void p(int x){
   z = 2*x;
}
main(){
   int y;
   p(y);
}
```

- Replacing the invocation p(y) as described yields:

```
int z;
main(){
   int y;
   {
      int x1=y;
      z = 2*x1;
   }
}
```

“Name Capture”

- Same names may denote different entities in the called and calling procedures
- To avoid name clashes, need to rename local variables of called procedure
  - Otherwise, local variables in called procedure may be confused with local variables of calling procedure or global variables

Call-by-value (Continued)

- Example:

```
int z;
void p(int x){
   int y = 2;
   z = y*x;
}
main(){
   int y;
   p(y);
}
```

- After replacement:

```
int z;
main(){
   int y;
   {
      int x1=y;
      int y1=2;
      z = y1*x1;
   }
}
```

Call-by-reference

- Evaluate actual parameters (must have l-values)
- Assign these l-values to formal parameters
- Execute the body.

```
int z = 8;
y=p(z);
```

- After the call, y and z will both have value 9.
- Call-by-reference supported in C++, but not in C
  - Effect realized by explicitly passing l-values of parameters using “&” operator

Call-by-reference (Continued)

- Explicit simulation in C provides a clearer understanding of the semantics of call-by-reference:

```
int p(int *x){
   *x = *x + 1;
   return *x;
}
...
int z;
y= p(&z);
```

Call-By-Reference (Continued)

- Example:

```
int z;
void p(int x){
   int y = 2;
   z = y*x;
}
main(){
   int y;
   p(y);
}
```

- After replacement:

```
int z;
main(){
   int y;
   {
      int& x1=y;
      int y1=2;
      z = y1*x1;
   }
}
```

Call-by-value-result

- Works like call by value but in addition, formal parameters are assigned to actual parameters at the end of procedure.

```
void p (int x, int y) {
   x = x +1;
   y = y+ 1;
}
...
int a = 3;
p(a, a) ;
```

- After the call, a will have the value 4, whereas with call-by- reference, a will have the value 5.

Call-by-value-result (Continued)

- The following is the equivalent of call-by-value-result call above:

```
x = a; y =a ;
x = x +1 ;
y =y +1 ;
a =x ; a =y ;
```

- thus, at the end, a = 4.

Call-By-Value-Result (Continued)

- Example:

```
void p(int x, y){
   x = x + 1;
   y = y + 1;
}
main(){
   int u = 3;
   p(u,u);
}
```

- After replacement:

```
main(){
   int u = 3;
   {
      int x1 = u;
      int y1 = u;
      x1 = x1 + 1;
      y1 = y1 + 1;
      u = x1; u = y1;
   }
}
```

Call-by-Name

- Instead of assigning l-values or r-values, CBN works by substituting actual parameter expressions in place of formal parameters in the body of callee
- Preprocessing:
  - Substitute formal parameters in procedure body by actual parameter expressions.
  - Rename as needed to avoid “name capture”
- Inline:
  - Substitute the invocation expression with the modified procedure body.

Call-By-Name (Continued)

- Example:

```
void p(int x, y){
   int u;
   if (x==0)
       then return 1;
   else{
      return y;
   }
}
main(){
   int u=5; int v=0;
   p(v,u/v);
}
```

- After replacement:

```
main(){
   int u;
   {
      if (0==0)
         then return 1;
      else{
         return y;
      }
   }
}
```

Call-By-Need

- Similar to call-by-name, but the actual parameter is evaluated at most once
  - Has same semantics as call-by-name in functional languages
    - This is because the value of expressions does not change with time
  - Has different semantics in imperative languages, as variables involved in the actual parameter expression may have different values each time the expression is evaluated in C-B-Name

Macros

- Macros work like CBN, with one important difference:
  - No renaming of “local” variables
- This means that possible name clashes between actual parameters and variables in the body of the macro will lead to unexpected results.

Macros (Continued)

- given

```
#define sixtimes(y) {int z=0; z = 2*y; y = 3*z;}
main() {
   int x=5, z=3;
   sixtimes(z);
}
```

- After macro substitution, we get the program:

```
main(){
   int x=5,z=3;
   {int z=0; z = 2*z; y = 3*z;}
}
```

Macros (Continued)

- It is different from what we would have got with CBN parameter passing.
- In particular, the name confusion between the local variable z and the actual parameter z would have been avoided, leading to the following result:

```
main() {
   int x = 5, z = 3;
   {
      int z1=0;  // z renamed as z1
      z1 = 2*z;  // y replaced by z without
      z = 3*z1;  // confusion with original z
   }
}
```

Difficulties in Using Parameter Passing Mechanisms

- CBV: Easiest to understand, no difficulties or unexpected results.
- CBVR:
  - When the same parameter is passed in twice, the end result can differ depending on the order.
  - The order of values assigning back to actual parameters.
  - Otherwise, relatively easy to understand.

Difficulties With CBVR (Continued)

- Example:

```
int f(int x, int y) {
   x=4;
   y=5;
}
void g() {
   int z;
   f(z, z);
}
```

- If assignment of formal parameter values to actual parameters were performed left to right, then z would have a value of 5.
- If they were performed right to left, then z will be 4.

Difficulties in Using CBR

- Aliasing can create problems.

```
int rev(int a[], int b[], int size) {
   for (int i = 0; i < size; i++)
      a[i] = b[size-i-1];
}
```

- The above procedure will normally copy b into a, while reversing the order of elements in b.
- However, if a and b are the same, as in an invocation rev(c,c,4), the result is quite different.
- If c is 1,2,3,4 at the point of call, then its value on exit from rev will be 4,3,3,4.

Difficulties in Using CBN

- CBN is complicated, and can be confusing in the presence of side-effects.
  - actual parameter expression with side-effects:

```
void f(int x) {
   int y = x;
   int z = x;
}
main() {
   int y = 0;
   f(y++);
}
```

- Note that after a call to f, y's value will be 2 rather than 1.

Difficulties in Using CBN (Continued)

- If the same variable is used in multiple parameters.

```
void swap(int x, int y) {
   int tp = x;
   x = y;
   y = tp;
}

main() {
   int a[] = {1, 1, 0};
   int i = 2;
   swap(i, a[i]);
}
```

- When using CBN, by replacing the call to swap by the body of swap: i will be 0, and a will be 2, 1, 0.

Difficulties in Using Macro

- Macros share all of the problems associated with CBN.
- In addition, macro substitution does not perform renaming of local variables, leading to additional problems.