LECTURE 20

Parameter Passing

PARAMETER PASSING

Parameterized subroutines accept arguments which control certain aspects of their behavior or act as data on which the subroutine must operate.

Today we’ll be discussing the most common modes of parameter passing as well as special-purpose parameters and function returns.

SOME DEFINITIONS

- Formal parameters
  - Lisp: (lambda (a b) (/ (+ a b) 2.0))
  - C: float ave(float a, float b) { return (a+b)/2.0; }
- Actual parameters
  - Lisp function arguments: (ave x 10.0)
  - C function arguments: ave(x, 10.0)
- Operands (of operators and special forms)
  - Lisp special forms: (if flag “yes” “no”)
  - C operators: x > 0 && flag

PARAMETER PASSING MECHANISMS

- Parameter passing mechanism: the mechanism by which the actual parameter information is passed to the subroutine.
- Many potential ways -- copying value, copying reference, etc.
- Some common parameter passing mechanisms:
  - Call by value
  - Call by reference
  - Call by sharing (object reference)
  - Call by result
  - Call by name
- Different mechanisms used by C, Fortran, Pascal, C++, Java, Ada and Algol 60.

CALL BY VALUE

- The only parameter passing mechanism in C.
- Actual parameter is evaluated and its value is assigned to the formal parameter.
- A formal parameter in the function body behaves as a local variable.

```
x:integer
procedure foo(y:integer)
    y:=3
    print x
...
x:=2
foo(x)
print x
```

- Passing a pointer value allows the value of the actual parameter to be modified.
  - Pointers are basically explicit references.
- For example:

```
swap(int* a, int* b){
  int t = *a;
  *a = *b;
  *b = t;
}
```

```
swap(&v1, &v2);
```

CALL BY REFERENCE

- The only parameter passing mechanism in Fortran.
- If the actual parameter is an L-value (e.g. a variable), its reference is passed to the subroutine.
- If the actual parameter is an R-value (e.g. an expression), it is assigned to a hidden temporary variable whose reference is passed to the subroutine.
- For example:

```
SUBROUTINE SHIFT(A, B, C)
INTEGER A, B, C
A = B + C
B = 0
C = 6
END
```

For example, SHIFT(X, 2+1, 3) assigns 6 to X and the assignments to B and C in the subroutine have no effect.

```
x:integer
procedure foo(y:integer)
    y:=3
    print x
...
x:=2
foo(x)
print x
```

COMPARISON

- Call-by-value: the value of the actual parameter is copied to the formal parameter.
  - For parameters with large data structures (e.g. a class with big arrays), call-by-value may result in high overheads.
  - The entire class including the arrays has to be copied!
- Call-by-reference: only the reference is copied for each parameter
  - No significant overhead for large data structures.
  - But your original argument is in danger of being modified by the subroutine.
  - In C++, this is fixed by the const qualifier.

EMULATING CALL-BY-REFERENCE WITH CALL-BY-VALUE

- What if we want to call FORTRAN functions from C programs or vice versa?
- We need to deal with the two different parameter passing mechanisms in the two languages.
- We must emulate call-by-reference using call-by-value.

```
SUBROUTINE SHIFT(A, B, C)
INTEGER A, B, C
A = B + C
END
```

```
#include <stdio.h>
extern void shift_(int *A, int *B, int *C);
main(){
   int i, j, k;
   i = j = k = 2;
   shift_(&i, &j, &k);
   printf("i = %d, j = %d, k = %d\n", i, j, k);
}
```

```
diablo:~>g77 shift.f shift.c
diablo:~>./a.out
i = 4, j = 2, k = 2
```

EMULATING CALL-BY-REFERENCE WITH CALL-BY-VALUE

How do we use the following C function in a FORTRAN program?

```
c_shift_real(int a, int b, int *c){
   *c = a + b;
 }
```

Create a wrapper for the FORTRAN program to call.

```
c_shift__(int *a, int *b, int *c){
   c_shift_real(*a, *b, c);
}
```

EMULATING CALL-BY-REFERENCE WITH CALL-BY-VALUE

```
 SUBROUTINE SHIFT(A, B, C)
 INTEGER A, B, C
 CALL C_SHIFT(A, B, C)
 END
```

```
diablo:~>g77 shift2.f shift2.c
diablo:~>./a.out
i = 2, j = 2, k = 4
```

```
#include <stdio.h>
extern void shift_(int *A, int *B, int*C);
void c_shift_real(int A, int B, int *C){
  *C = A + B;
}

void c_shift__(int *A, int *B, int *C){
  c_shift_real(*A, *B, C);
}

int main(){
  int i, j, k;
  i = j = k = 2;
  shift_(&i, &j, &k);
  printf("i = %d, j = %d, k = %d\n", i, j, k);
  return 0;
}
```

PARAMETER PASSING IN PASCAL

- Call by value and call by reference parameter passing.
- Call by value is similar to C.
- Call by reference: indicated with var qualifier in argument list.
- For example

```
procedure swap(var a:integer, var b:integer)
var t;
begin
  t := a;
  a := b;
  b := t
end
```

where the var parameters a and b are passed by reference.

PARAMETER PASSING IN C++

- Call by value and call by reference parameter passing.
- Call by value is similar to C.
- Call by reference: use reference (&) operator.
- For example:

```
swap(int &a, int &b)
{ int t = a; a = b; b = t; }
```

where the reference parameters a and b are passed by reference.

- Arrays are automatically passed “by reference” (like in C).
- Big objects should be passed by reference instead of by value.
- To protect data from modification when passed by reference, use const:

```
        store_record_in_file(const huge_record &r) { ... }
```

PARAMETER PASSING IN ADA

- *In-mode* parameters can be read but not written in the subroutine.
  - Call-by-value; writes to the parameter are prohibited in the subroutine.
- *Out-mode* parameters can be written but not read in the subroutine (Ada 95 allows read).
  - Call by result, which uses a local variable to which the writes are made.
  - The resulting value is copied to the actual parameter to pass the value out when the subroutine returns.
- *In-out-mode* parameters can be read and written in the subroutine.
  - Call by value/result uses a local variable that is initialized by assigning the actual parameter's value to it.
  - The resulting value is copied to the actual parameter to pass the value out when the subroutine returns.

PARAMETER PASSING IN ADA: EXAMPLE

For example:

```
procedure shift(a:out integer,
                b:in out integer,
                c:in integer) is
begin
  a := b;
  b := c;
end shift;
```

Here, a is passed out, b is passed in and out, and c is passed in.

PARAMETER PASSING IN ADA (CONT'D)

The Ada compiler generates specific code for the example shift procedure for in, out, and in/out parameters to implement call by value (in), call by result (out), and call by value/result (in/out), which is similar to the following C function:

```
void shift(int *a, int *b, int c)
{ int tmpa, tmpb = *b, tmpc = c;     // copy input values at begin
  tmpa = tmpb;
  tmpb = tmpc;                       // perform operations on temps
  *a = tmpa;
  *b = tmpb;                         // copy result values out before
return
}
```

- Temps are initialized, operated on, and copied to out mode parameters.
- This is more efficient than pass by reference, because it avoids repeated pointer indirection to access parameter values.
- The Ada compiler may decide to use call by reference for passing non-scalars (e.g. records and arrays) for memory access optimization.

PARAMETER PASSING AND ALIASING

An *alias* is a variable or formal parameter that refers to the same value as another variable or formal parameter.

Example variable aliases in C++:

```
int i, &j = i; // j refers to i (is an alias for i)
i = 2;
j = 3;
cout << i;     // prints 3
```

Example parameter aliases in C++. The result of shift(x, y, x) is that x is set to y but y is unchanged:

```
shift(int &a, int &b, const int &c){
     a = b;
     b = c;
}
```

Example mixing of variable and parameter aliases in C++. The result of score(sum, 7) is that sum is incremented by 14:

```
int sum = 0;
score(int &total, int val) {
     sum += val;
     total += val;
}
```

PARAMETER PASSING AND ALIASING (CONT'D)

- Ada forbids parameter aliases.
  - Allows compiler to choose call by reference with the same effect as call by result.
  - But most compilers don't check and the resulting program behavior is undefined.

```
procedure rand(a:out integer,
               b:out integer,
               c:in integer) is
begin
  a := c*2;
  b := c*3;
end shift;
```

What is the value of x after rand(x,x,2)?

PARAMETER PASSING ISSUE

- Call by value/result behaves differently compared to call by reference in the presence of aliases (that's why Ada forbids it).

  For example:

```
      procedure shift(a:out integer,
                      b:in out integer,
                      c:in integer) is
      begin
        a := b;
        b := c;
      end shift;
```

- When shift(x,x,0) is called by reference the resulting value of x is 0.
- When shift(x,x,0) is called by value/result the resulting value of x is either unchanged or 0 (because the order of copying out mode parameters is unspecified).

CALL-BY-NAME

- Passes actual arguments such as expressions into the subroutine body for (re)evaluation.
- C/C++ macros (also called defines) adopt a form of call by name.
- For example
```
    #define max(a,b) ( (a)>(b) ? (a) : (b) )
```
- Macro expansion is applied to the program source text and amounts to the substitution of the formal parameters with the actual parameters in the macro
- For example max(n+1, m) is replaced by ((n+1)>(m)?(n+1):(m)). Note: formal parameters are often parenthesized to avoid syntax problems when expanding.
- Similar to call by name, actual parameters are re-evaluated each time the formal parameter is used.
- Watch out for re-evaluation of function calls in actual parameters, for example max(somefunc(),0) results in the evaluation of somefunc() twice if it returns a value > 0.

MACRO AND INLINE FUNCTIONS

- Macro and inline functions in C++ have similar functionality. Are there any differences?
- `#define mymult(a, b) a*b`
- `inline int mymult1(int a, int b) {return a*b;}`

MACRO AND INLINE FUNCTIONS

Parameter passing mechanisms are not the same!

```
carnahan@diablo:~>g++ try.cpp
carnahan@diablo:~>./a.out
mymult(2+2, 3+3) = 11
mymult1(2+2, 3+3) = 24
```

mymult1 essentially uses call-by-value while mymult emulates call-by-name.

```
#include <iostream>
using namespace std;

#define mymult(a, b) a*b
inline int mymult1(int a, int b) {return a*b;}

int main() {
  int a = 3;
  cout <<  "mymult(2+2, 3+3) = " <<  mymult(2+2, 3+3) << "\n";
  cout << "mymult1(2+2, 3+3) = " << mymult1(2+2, 3+3) << "\n";
}
```

PARAMETER PASSING ISSUES

- Call-by-name problem: Hard to write a “swap” routine that works:

```
procedure swap(a, b)
integer a, b, t;
begin
  t := a;
  a := b;
  b := t
end swap
```

- Consider swap(i, a[i]), which executes:

```
t := i
i := a[i]  // this changes i
a[i] := t  // assigns t to wrong array element
```

DEFAULT PARAMETERS

- Ada, C++, Common Lisp, and Fortran 90 support default parameters.
- A default parameter is a formal parameter with a default value.
- When the actual parameter value is omitted in a subroutine call, the user-specified default value is used. Example in C++:

```
void print_num(int n, int base = 10)
    ...
```

- A call to print_num(35) uses default value 10 for base as if print_num(35,10) was called.

POSITIONAL VERSUS NAMED PARAMETERS

- *Positional parameters*: the order of formal and actual arguments is fixed.
- All programming languages adopt this natural convention.
- *Named parameter* (also called keyword parameter): explicitly binds the actual parameter to the formal parameter.
- Ada, Modula-3, Common Lisp, and Fortran 90. For example in Ada:

```
put(item => 35, base => 8);
// this "assigns" 35 to item and 8 to base, which is the same as:
put(base => 8, item => 35);
// and we can mix positional and name parameters as well:
put(35, base => 8);
```

- Pro: documentation of parameter purpose.
- Pro: allows default parameters anywhere in formal parameter list, whereas with positional parameters the use of default parameters is restricted to the last parameter(s) only, because the compiler cannot tell which parameter is optional in a subroutine invocation.

VARIABLE ARGUMENT LISTS

- C, C++, and Common Lisp are unusual in that they allow defining subroutines that take a variable number of arguments. Example in C:

```
#include <stdarg.h>
int plus(int num, ...)
{ int sum;
  va_list args;                // declare list of arguments
  va_start(args, num);         // initialize list of arguments
  for (int i=0; i<num; i++)
    sum += va_arg(args, int); // get next argument (assumed to be int)
  va_end(args);                // clean up list of arguments
  return sum;
}
```

- Function plus adds a set of integers, where the number of integers is the first parameter to the function: plus(4,3,2,1,4) is 10 .
- Used in the printf and scanf text formatting functions in C.
- Variable number of arguments in C and C++ is not type safe as parameter types are not checked.
- In Common Lisp, one can write (+ 3 2 1 4) to add the integers.

FUNCTION RETURNS

- Some programming languages allow a function to return any type of data structure, except maybe a subroutine (requires first-class subroutines).
- Modula-3 and Ada allow a function to return a subroutine as a closure.
- C and C++ allow functions to return pointers to functions (no closures).
- Some languages use a special variable to hold function return value. Example in Pascal:

```
function max(a : integer; b : integer) : integer;
begin
  if a>b then max := a else max := b
end
```

- There is no return statement, instead the function returns with the value of max when it reaches the end.
- The subroutine frame reserves a slot for the return value anyway, so these languages make the slot explicitly available.

FUNCTION RETURNS (CONT'D)

- Ada, C, C++, Java, and other more modern languages typically use an explicit return statement to return a value from a function. Example in C:

```
int max(int a, int b)
{ if (a>b) return a;
  else return b;
}
```

Programmers may need a temporary variable for incremental operations on the return value. For example:

```
int fac(int n)
{ int ret = 1;
  for (i = 2; i <= n; i++)
    ret *= i;
  return ret;
}
```

Wastes time copying the temporary variable value into return slot of subroutine frame on the stack.

GCC CALLING SEQUENCE FOR THE X86 ARCHITECTURE

- cdecl (C declaration) calling sequence.
  - Subroutine arguments are passed on the stack, in the reverse order.
  - Stack is aligned to a 16 byte boundary.
  - Calling function cleans the stack after the function call returns.
    - This allows for variable number of parameters.
  - Integer values and memory address are returned in EAX.
  - EAX, ECX, EDX are caller-saved, the rest are callee saved.

See callingsequence.cpp and callingsequence.s if you’re interested in the details.