Functions

Terms

• foo is the caller
• bar is the callee
• a, b are the actual parameters to bar
• x, y are the formal parameters of bar
• Shorthand:
  • argument = actual parameter
  • parameter = formal parameter

Different kinds of parameters

• Value parameters
• Reference parameters
• Result parameters
• Value-result parameters
• Read-only parameters

Value parameters

• “Call-by-value”
• Used in C, Java, default in C++
• Passes the value of an argument to the function
• Makes a copy of argument when function is called
• Advantages! Disadvantages?

Value parameters

```c
int x = 1;
void main () {
    foo(x, x);
    print(x);
}
void foo(int y, int z) {
    y = 2;
    z = 3;
    print(x);
}
```

• What do the print statements print?

Value parameters

```c
int x = 1;
void main () {
    foo(x, x);
    print(x);
}
void foo(int y, int z) {
    y = 2;
    z = 3;
    print(x);
}
```
Value parameters

```c
int x = 1;
void main () {
   foo(x, x);
   print(x);
}
void foo(int y, int z) {
    y = 2;
    z = 3;
    print(x);
}
```

• What do the print statements print?
• Answer:
  ```c
  print(x); //prints 1
  print(x); //prints 1
  ```

Reference parameters

```c
int x = 1;
void main () {
   foo(x, x);
   print(x);
}
void foo(int &y, int &z) {
    y = 2;
    z = 3;
    print(x);
}
```

• “Call-by-reference”
• Optional in Pascal (use “var” keyword) and C++ (use “&”)
• Pass the address of the argument to the function
• If an argument is an expression, evaluate it, place it in memory and then pass the address of the memory location
• Advantages? Disadvantages?

Reference parameters

```c
int x = 1;
void main () {
   foo(x, x);
   print(x);
}
void foo(int &y, int &z) {
    y = 2;
    z = 3;
    print(x);
}
```

• What do the print statements print?

Result parameters

```c
int x = 1;
void main () {
   foo(x, x);
   print(x);
}
void foo(int &y, int &z) {
    y = 2;
    z = 3;
    print(x);
}
```

• Return values of a function
• Some languages let you specify other parameters as result parameters – these are un-initialized at the beginning of the function
• Copied at the end of function into the arguments of the caller
• C++ supports “return references”
  ```c
  int& foo( ... )
  ```
  compute return values, store in memory, return address of return value
int x = 1;
void main () {
    foo(x, x);
    print(x);
}

void foo(int y, result int z) {
    y = 2;
    z = 3;
    print(x);
}

Monday, September 26, 2011

What do the print statements print?
Answer:
print(x): // prints 3  
print(x): // prints 1

Value-result parameters

"Copy-in copy-out"
Evaluate argument expression, copy to parameters
After subroutine is done, copy values of parameters back into arguments
Results are often similar to pass-by-reference, but there are some subtle situations where they are different
Value-result parameters

```
int x = 1;
void main () {
    foo(x, x);
    print(x);
}

void foo(int y, value result int z) {
    y = 2;
    z = 3;
    print(x);
}
```

- What do the print statements print?
- Answer: `print(x);` //prints 3
- `print(x);` //prints 1

What about this?

```
int x = 1;
void main () {
    foo(x, x);
    print(x);
}

void foo(value result int y, value result int z) {
    y = 2;
    z = 3;
    print(x);
}
```

- What do the print statements print?
- Answer: `print(x);` //undefined!
- `print(x);` //prints 1

Read only parameters

- Used when callee will not change value of parameters
- Read-only restriction must be enforced by compiler
- This becomes tricky when in the presence of aliasing and control flow

```
void foo(readonly int x, int y) {
    int * p;
    if (...) p = &x else p = &y
    *p = 4
}
```

- Is this legal? Hard to tell!

Esoteric: “name” parameters

- “Call-by-name”
- Usually, we evaluate the arguments before passing them to the function. In call-by-name, the arguments are passed to the function before evaluation
- Not used in many languages, but Haskell uses a variant

```
int x = 2;
void main () {
    foo(x + 2);
}
```

```
int x = 2;
void main () {
    foo(x + 2);
}
```

```
void foo(int y) {
    z = y + 2;
    print(z);
}
```

```
void foo(int y) {
    z = x + 2 + 2;
    print(z);
}
```
Why is this useful?

- Consider the code on the left.
- Normally, we must evaluate `bar()` before calling `foo()`.
- But what if `bar()` has an infinite loop?
- In call by name, this program still terminates.

```c
int x = 2;
void main () {
  Foo(bar());
}

void foo(int y) {
  z = 3;
  print(z);
}
```

Other considerations

- Scalars
  - For call by value, can pass the address of the actual parameter and copy the value into local storage within the procedure.
  - Reduces size of caller code (why is this good?)
  - If scalar is a constrained type (e.g., a Pascal range type), must insert type check for return values.
  - For machines with a lot of registers (e.g., MIPS), compilers will save a few registers for arguments and return types.
  - Less need to manipulate stack.

Other considerations

- Arrays
  - For efficiency reasons, arrays should be passed by reference (why?)
  - Java, C, C++ pass arrays by reference by default (technically, they pass a pointer to the array by value)
  - Pass in a fixed size dope vector as the actual parameter (not the whole array?)
  - Callee can copy array into local storage as needed.

Dope vectors

- Remember: store additional information about an array.
  - Where it is in memory.
  - Size of array.
  - # of dimensions.
  - Storage order.
  - Can sometimes eliminate dope vectors with compile-time analysis.

Strings

- Requires a descriptor.
  - Like a dope vector, provides information about string.
  - May just need to pass a pointer (if string contains information about its length).
  - May also need to pass information about length.

Calling a function

- What should happen when a function is called?
  - Set the frame pointer (sets the base of the activation record).
  - Allocate space for local variables (use the function’s symbol table for this).
  - What about registers?
  - Callee might want to use registers that the caller is using.
Saving registers

- Two options: caller saves and callee saves
  - Caller saves
    - Caller pushes all the registers it is using on to the stack before calling function, restores the registers after the function returns
  - Callee saves
    - Callee pushes all the registers it is going to use on the stack immediately after being called, restores the registers just before it returns
  - Why use one vs. the other?
  - Simple optimizations are good here: don’t save registers if the caller/callee doesn’t use any

Activation records

- Return value
- Actual parameters
- Caller's return address
- Caller's frame pointer
- Static links (other FPs)
- Register save area
- Local variables

The frame pointer

- Manipulate with instructions like link and unlink
  - Link: push current value of FP on to stack, set FP to top of stack
  - Unlink: read value at current address pointed to by FP, set FP to point to that value
  - In other words: link pushes a new frame onto the stack, unlink pops it off

Example Subroutine Call and Stack Frame

```
int SubOne(int a, int b) {
    int l1, l2;
    l1 = a;
    l2 = b;
    return l1+l2;
};
```

```
z = SubOne(x,2*y);
```
Example Subroutine Call and Stack Frame

```
Example2

int SubOne(int & a, Class1 b) {
  int1, i2;
  i1 = a;
  i2 = b.f4;
  return i1+i2;
};
```

Example2 assembly code:

```
push push &x
push &y
jsr SubOne
pop pop R1
store R1 z
```

return value &x &y
return address saved frame ptr
l1 b.f100 ...
b.f2 b.f1...
stack
R6
push push &x
push &y
jsr SubOne
pop pop R1
store R1 z
```

Example Subroutine Call and Stack Frame

```
Example

int SubOne(int a, int b) {
  int l1, l2;
  l1 = a;
  l2 = b;
  return l1+l2;
};
```

Example assembly code:

```
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

return value
x
2*y
return address
saved frame ptr
l1 l2...
stack
R6
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

Example Subroutine Call and Stack Frame

```
Example

int SubOne(int a, Class1 b) {
  int1, i2;
  i1 = a;
  i2 = b.f4;
  return i1+i2;
};
```

Example assembly code:

```
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

return value
&x
&y
return address
saved frame ptr
l1 l2...
stack
R6
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

Example Subroutine Call and Stack Frame

```
Example

int SubOne(int a, int b) {
  int l1, l2;
  l1 = a;
  l2 = b;
  return l1+l2;
};
```

Example assembly code:

```
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

return value
x
2*y
return address
saved frame ptr
l1 l2...
stack
R6
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

Example Subroutine Call and Stack Frame

```
Example

int SubOne(int a, Class1 b) {
  int l1, l2;
  l1 = a;
  l2 = b.f4;
  return i1+i2;
};
```

Example assembly code:

```
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

return value
x
2*y
return address
saved frame ptr
l1 l2...
stack
R6
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

Example Subroutine Call and Stack Frame

```
Example

int SubOne(int a, int b) {
  int l1, l2;
  l1 = a;
  l2 = b;
  return l1+l2;
};
```

Example assembly code:

```
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

return value
x
2*y
return address
saved frame ptr
l1 l2...
stack
R6
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

Example Subroutine Call and Stack Frame

```
Example

int SubOne(int a, int b) {
  int l1, l2;
  l1 = a;
  l2 = b;
  return l1+l2;
};
```

Example assembly code:

```
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

return value
x
2*y
return address
saved frame ptr
l1 l2...
stack
R6
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

Example Subroutine Call and Stack Frame

```
Example

int SubOne(int a, int b) {
  int l1, l2;
  l1 = a;
  l2 = b;
  return l1+l2;
};
```

Example assembly code:

```
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

return value
x
2*y
return address
saved frame ptr
l1 l2...
stack
R6
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

Example Subroutine Call and Stack Frame

```
Example

int SubOne(int a, int b) {
  int l1, l2;
  l1 = a;
  l2 = b;
  return l1+l2;
};
```

Example assembly code:

```
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

return value
x
2*y
return address
saved frame ptr
l1 l2...
stack
R6
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

Example Subroutine Call and Stack Frame

```
Example

int SubOne(int a, int b) {
  int l1, l2;
  l1 = a;
  l2 = b;
  return l1+l2;
};
```

Example assembly code:

```
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

return value
x
2*y
return address
saved frame ptr
l1 l2...
stack
R6
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

Example Subroutine Call and Stack Frame

```
Example

int SubOne(int a, int b) {
  int l1, l2;
  l1 = a;
  l2 = b;
  return l1+l2;
};
```

Example assembly code:

```
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```

return value
x
2*y
return address
saved frame ptr
l1 l2...
stack
R6
push push x
mul 2 y t1
push t1
jsr SubOne
pop pop R1
store R1 z
```