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

```c
void foo() {
    int a, b;
    ...
    bar(a, b);
}

void bar(int x, int y) {
    ...
}
```
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);
}
```
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);
}

• What do the print statements print?
• Answer:

    print(x); //prints 1
    print(x); //prints 1
```
Reference parameters

- “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);
}
```
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?
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?
• Answer:
  ```c
  print(x); //prints 3
  print(x); //prints 3
  ```
Result parameters

• 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”

    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);
}
Result parameters

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

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

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

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

```c
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?
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?

```c
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 about this?

```c
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?
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

```c
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

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

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

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

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?)

• 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

Caller's responsibility

Callee's responsibility

Stack Growth

FP register

Is this record generated for callee-saves or caller-saves? How would the other record look?
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);
```

Lower addr
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);
```

```
int SubOne(int a, int b) {
    int l1, l2;
    l1 = a;
    l2 = b;
    return l1+l2;
};
```
Example Subroutine Call and Stack Frame

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

z = SubOne(x, 2*y);
```

3-address code:

```
push x
push y
t1 = 2 * y
push t1
jsr SubOne
pop
pop
pop z
```

```
link 3
move $P1 $L1
move $P2 $L2
add $L1 $L2 t2
move t2 $R
unlink
ret
```
Example Subroutine Call and Stack Frame

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

```
z = SubOne(x, 2*y);
```

```
stack

R6

Lower addr

3-address code:

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

```
assembly code:

```
push
push x
load y R1
muli 2 R1
push R1
jsr SubOne
pop
pop
pop R1
store R1 z
```

```
link 3
move $P1 $L1
move $P2 $L2
add $L1 $L2 t2
move t2 $R
unlink
ret
```

```
link R6 3
load 3(R6) R1
store R1 -1(R6)
load 2(R6) R2
store R2 -2(R6)
load -1(R6) R1
add -2(R6) R1
store R1 4(R6)
unlink
ret
```