**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
- 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?

```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**

- What do the print statements print?

```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

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

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

- What do the print statements print?
- Answer:

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

Reference parameters

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

- What do the print statements print?
- Answer:

  ```
  print(x); //prints 3
  print(x); //prints 3
  ```

Reference parameters

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

- What do the print statements print?
- Answer:

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

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
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);
}
```

Wednesday, October 14, 15

Read only parameters

- Used when callee will not change value of parameters
- Read-only restriction must be enforced by compiler
- This can be tricky when in the presence of aliasing and control flow
  ```c
  void foo(const int x, int y) {
      int * p;
      if (...) p = &x else p = &y
      *p = 4
  }
  ```
- Is this legal? Hard to tell!
- gcc will not let the assignment happen

Wednesday, October 14, 15

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);
}
```

Why is this useful?

- Consider the code on the left
- Normally, we must evaluate bar() before calling foo()
- But what if bar() runs for a long time?
- In call by name, we only evaluate bar() if we need to use it

```c
int x = 2;
void main () {
    foo(bar());
}
void foo(int y) {
    if (...) {
        z = y;
    } else {
        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

- 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

Function call behavior

```
main()

fp -> sp

main() {
  foo();
  ...
}
```

```
foo()

fp -> sp

foo() {
  bar();
  ...
  baz();
}
```
Function call behavior

```
main() {
    foo();
    ...
}
foo() {
    bar();
    ...
    baz();
}
```

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 i1, i2;
    i1 = a;
    i2 = b;
    return i1+i2;
};
```

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

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

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

Example Subroutine Call and Stack Frame

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

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

Example Subroutine Call and Stack Frame

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

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