Function Calling Conventions
Passing arguments to functions

- Terminology:
  - Callee has **parameters** that are passed into it (also called **formal parameters**)
  - Caller passes **arguments** to callee (also called **actual parameters**)
  - But this transfer of data between caller and callee can take many forms!

```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?
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);
    print(y);
}
```

• What do the print statements print?

• Answer:

• print(x); //prints 3
  print(x); //prints 3
  print(x); //prints 3
  print(y); //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
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?
• 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

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

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

• What do the print statements print?

• Answer:

  • print(x) //prints 3
  
  print(w) //prints 2
  
  print(x) //prints 1
  
  print(w) //prints 2
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 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
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()` runs for a long time?

- In call by name, we only evaluate `bar()` if we need to use it
## Parameter Passing

<table>
<thead>
<tr>
<th>Parameter Type</th>
<th>Copy-in?</th>
<th>Copy-out?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value parameter</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Reference parameter</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Result parameter</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Value-result parameter</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

(Evaluate before call)

(Evaluate after call)

<table>
<thead>
<tr>
<th>Parameter Type</th>
<th>Copy-in?</th>
<th>Copy-out?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name parameter</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Question:**

How to implement swap(x, y)?
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
How to pass values

- Pass arguments in registers (value, if call by value; address of data if call by reference)
  - Advantage: fast, no memory operations to retrieve values
  - Disadvantage: limited space, need to be careful about more complicated data, uses more registers
- Pass arguments on the stack (through memory)
  - Advantage: unlimited space for passing arguments, saves registers for other use
  - Disadvantage: requires more memory, adds instruction overhead

- Architectures with lots of registers (e.g., RISC-V) prefer to pass arguments in registers, but all architectures default to stack if needed
  - In project, we will pass arguments on the stack to simplify code generation; passing in registers is a good optimization!