Subroutines and Control Abstraction

CSE 307 – Principles of Programming Languages
Stony Brook University

http://www.cs.stonybrook.edu/~cse307
Subroutines

- Why use subroutines?
  - Give a name to a task.
  - We no longer care how the task is done.
- The subroutine call is an expression:
  - Subroutines take arguments (in the formal parameters)
  - Values are placed into variables (actual parameters/arguments)
  - A value is (usually) returned.
Review Of Stack Layout

- Allocation strategies:
  - Static
    - Code
    - Globals
    - Explicit constants (including strings, sets, other aggregates)
    - Small scalars may be stored in the instructions themselves
  - Stack
    - parameters
    - local variables
    - temporaries
    - bookkeeping information
  - Heap
    - dynamic allocation
Review Of Stack Layout

(c) Paul Fodor (CS Stony Brook) and Elsevier
Review Of Stack Layout

- Contents of a stack frame:
  - bookkeeping
    - return Program Counter (PC)
  - saved registers
  - line number
  - static link
  - arguments and returns
  - local variables
  - temporaries
Calling Sequences

• Maintenance of stack is responsibility of calling sequence and subroutine prolog and epilog

• Space is saved by putting as much in the prolog and epilog as possible

• Time may be saved by putting stuff in the caller instead, where more information may be known
Calling Sequences

• Common strategy is to divide registers into *caller-saves* and *callee-saves* sets

• Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

• Some storage layouts use a separate *arguments pointer*
Calling Sequences (LLVM on ARM)

- **Caller:**
  - saves into the “local variable and temporaries” area any caller-saves registers whose values are still needed
  - puts up to 4 small arguments into registers r0-r3
  - puts the rest of the arguments into the argument build area at the top of the current frame
  - puts return address into register lr, jumps to target address, and (optionally) changes instruction set coding
Calling Sequences (LLVM on ARM)

- In prolog, Callee
  - pushes necessary registers onto stack
  - initializes frame pointer by adding small constant to the sp placing result in r7
  - subtracts from sp to make space for local variables, temporaries, and arg build area at top of stack
- In epilog, Callee
  - puts return value into r0-r3 or memory (as appropriate)
  - subtracts small constant from r7, puts result in sp (effectively deallocates most of frame)
  - pops saved registers from stack, pc takes place of lr from prologue (branches to caller as side effect)
Calling Sequences (LLVM on ARM)

- After call, Caller
  - moves return value to wherever it's needed
  - restores caller-saves registers lazily over time, as their values are needed
- All arguments have space in the stack, whether passed in registers or not
- The subroutine just begins with some of the arguments already cached in registers, and 'stale' values in memory
- Optimizing compilers keep things in registers whenever possible, flushing to memory only when they run out of registers, or when code may attempt to access the data through a pointer or from an inner scope
Calling Sequences (LLVM on ARM)

- Many parts of the calling sequence, prologue, and/or epilogue can be omitted in common cases
- Leaving things out saves time
- Simple leaf routines don't use the stack - don't even use memory – and are exceptionally fast
Parameter Passing

- Modes of passing parameters:
  - Call by value: make a copy of the parameter.
  - Call by reference (aliasing): allows the function to change the parameter
    - out-parameters
  - Call by sharing: requires parameter to be a reference itself.
    - Makes copy of reference that initially refers to the same object.
    - E.g., Python, Java Objects.
def f(a):
    a += 1
x = 0
f(x)
print(x)

• value: 0
• reference: 1
• sharing: 0
def f(a):
    a.foo = 1
x = object()
x.foo = 0
f(x)
print x.foo

• value: 0
• reference: 1
• sharing: 1
Parameter Passing

```python
z = object()
z.foo = 1
def f(a):
    a = z
x = object()
x.foo = 0
f(x)
print x.foo
```

- value: 0
- reference: 1
- sharing: 0
Parameter Passing

- **Call-by-value:**
  - Can't have aliasing between parameters.
  - Can be expensive to implement (e.g., copying large objects).
  - Can't change a parameter, except by returning a new copy.

- **Call-by-reference:**
  - No copying objects.
  - Out-parameters (i.e., the procedure returns values through its parameters).
  - Good: More flexibility.
  - Bad: Can be confusing when arguments change.
Parameter Passing

- C/C++: functions
  - parameters passed by value (C)
  - parameters passed by reference can be simulated with pointers (C)
    
    ```c
    void proc(int* x, int y) {*x = *x + y }
    ...
    proc(&a, b);
    ```

  - or directly passed by reference (C++)
    
    ```c
    void proc(int& x, int y) {x = x + y }
    proc(a, b);
    ```
Parameter Passing

- Call-by-sharing.
  - No copying of large objects.
  - No implicit out parameters.
    - Can change objects, but not arguments.
Other fun tricks with parameters:

**Named parameters (pass-by-name):** the values are passed by *associating* each one with a *parameter name*. E.g., in Objective-C:

```objective-c
[window addNewControlWithTitle:@"Title"
    xPosition:20
    yPosition:50
    width:100
    height:50
    drawingNow:YES];
```
Parameter Passing

- **Default parameters**: default values are provided to the function

  - C++ example:
    ```cpp
    void PrintValues(int nValue1, int nValue2=10) {
        using namespace std;
        cout << "1st value: " << nValue1 << endl;
        cout << "2nd value: " << nValue2 << endl;
    }
    ```

    ```cpp
    int main() {
        PrintValues(1); // nValue2 will use default parameter of 10
        PrintValues(3, 4); // override default value for nValue2
    }
    ```
Parameter Passing

- **Variadic functions**: functions of indefinite arities

  - C, Objective-C and C++:

    ```c
    double average(int count, ...){
        va_list ap;  int j;  double tot = 0;
        va_start(ap, count);  //Requires the last fixed parameter (to get the address)
        for(j=0; j<count; j++)
            tot+=va_arg(ap, double);  //Requires the type to cast to.
        va_end(ap);
        return tot/count;
    }
    ```
Parameter Passing

- **Pass-by-name in ALGOL 60:**
  - The body of a function is interpreted at call time after textually substituting the actual parameters into the function body.
  - In this sense the evaluation method is similar to that of C preprocessor macros.
  - By substituting the actual parameters into the function body, the function body can both read and write the given parameters. In this sense the evaluation method is similar to pass-by-reference.
  - The difference is that since with pass-by-name the parameter is evaluated inside the function, a parameter such as \( a[i] \) depends on the current value of \( i \) inside the function, rather than referring to the value of \( a[i] \) before the function was called.
- **Pass-By-Name Security Problem** (see next slide)
Pass-by-name in ALGOL 60:

Pass-By-Name Security Problem:

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

Call swap(i, x[i]): temp := i; i := x[i]; x[i] := temp

After call: i = 5 x[5] = 5

Swap doesn't work!
Parameter Passing

- **Pass by Value-Returned (or value-result):** pass a valuereturned parameter by address (just like pass by reference parameters), but, upon entry, the procedure makes a temporary copy of this parameter and uses the copy while the procedure is executing.
- When the procedure finishes, it copies the temporary copy back to the original parameter.
- In some instances, pass by value-returned is more efficient than pass by reference, in others it is less efficient:
  - If a procedure only references the parameter a couple of times, copying the parameter's data is expensive.
  - If the procedure uses this parameter often, the procedure amortizes the fixed cost of copying the data over many inexpensive accesses to the local copy.
Parameter Passing

• **Pass by Result**: almost identical to pass by value-returned: the procedure uses a local copy of the variable and then stores the result through the pointer when returning.

• The difference between pass by value-returned and pass by result is that when passing parameters by result you do not copy the data upon entering the procedure.

• Pass by result parameters are for returning values, not passing data to the procedure. Therefore, pass by result is slightly more efficient than pass by value-returned since you save the cost of copying the data into the local variable.
<table>
<thead>
<tr>
<th>Parameter mode</th>
<th>Representative languages</th>
<th>Implementation mechanism</th>
<th>Permissible operations</th>
<th>Change to actual?</th>
<th>Alias?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>C/C++, Pascal, Java/C#</td>
<td>value</td>
<td>read, write</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>in, const</td>
<td>Ada, C/C++, Modula-3</td>
<td>value or reference</td>
<td>read only</td>
<td>no</td>
<td>maybe</td>
</tr>
<tr>
<td>out</td>
<td>Ada</td>
<td>value or reference</td>
<td>write only</td>
<td>yes</td>
<td>maybe</td>
</tr>
<tr>
<td>value/result</td>
<td>Algol W</td>
<td>value</td>
<td>read, write</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>var, ref</td>
<td>Fortran, Pascal, C++</td>
<td>reference</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>sharing</td>
<td>Lisp/Scheme, ML, Java/C#</td>
<td>value or reference</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>r-value ref</td>
<td>C++11</td>
<td>reference</td>
<td>read, write</td>
<td>yes*</td>
<td>no*</td>
</tr>
<tr>
<td>in, out</td>
<td>Ada, Swift</td>
<td>value or reference</td>
<td>read, write</td>
<td>yes</td>
<td>maybe</td>
</tr>
<tr>
<td>name</td>
<td>Algol 60, Simula</td>
<td>closure (thunk)</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>need</td>
<td>Haskell, R</td>
<td>closure (thunk) with</td>
<td>read, write†</td>
<td>yes†</td>
<td>yes†</td>
</tr>
</tbody>
</table>

**Figure 9.3 Parameter-passing modes.** Column 1 indicates common names for modes. Column 2 indicates prominent languages that use the modes, or that introduced them. Column 3 indicates implementation via passing of values, references, or closures. Column 4 indicates whether the callee can read or write the formal parameter. Column 5 indicates whether changes to the formal parameter affect the actual parameter. Column 6 indicates whether changes to the formal or actual parameter, during the execution of the subroutine, may be visible through the other: *Behavior is undefined if the program attempts to use an r-value argument after the call. †Changes to arguments passed by need in R will happen only on the first use; changes in Haskell are not permitted.
Returning from a Function

- Different ways of returning a value from a function.
  - Return statement
  - Assigning to the function name (Pascal, Fortran, Algol)
    - This interacts poorly w/ scoping and recursion
  - Special return location
    - Eiffel calls it Result
    - Means we don't have to allocate a variable to store the result in
Generic Subroutines and Modules

• Generic modules or classes are particularly valuable for creating containers: data abstractions that hold a collection of objects
• When defining a function, we don't need to give all the types
• Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right
Implementation of generic programming:

- One approach is implicit *parametric polymorphism*:
  - Dynamic typing.
  - Just try running the code.
    - No checking at compile time - not type safe.
- **Python approach**.

An alternative is to have a function that has parameterized types

- Generic classes and methods
- Can be static typed checked

**Java approach**

```java
boolean <T> allEqual(T a, T b, T c) {
    return a.equals(b) && b.equals(c);
}
```
Generic Subroutines and Modules

- Parameterized types: Two implementation approaches:
  - C++:
    - generates new code for each type:
      - linker can help with that
      - allows specialization
      - can make the code bigger
      - can use types in the function: new T();
      - Templates can cause horrible error messages
  - Java
    - type erasure:
      - Replace all type parameters in generic types with their bounds,
      - Only one instance of the code,
      - Can't do operations involving the type.
Generic Subroutines and Modules

class DefaultDict <T> {
    T get(k) {
        if (! this.hasKey(k)) {
            this.put(k, new T());
        }
        return super.get(k);
    }
}

• We need to specify which operations a type parameter must support.
• C++:
  • look at the operations used, derive it from that.
    • example above: needs to be creatable, needs to be insertable into it
• Java:
  • specify which class inherits from (Object by default).
Generic Subroutines and Modules

- Generics are better than macros:
  - E.g., take the macro:
    ```c
    #define min(a, b) (a < b) ? a : b
    ```
  - Problem: `min(a++, b++)`
    - Variables `a++` or `b++` evaluated more than once
  - C++ generic:
    ```c
    template <class T>
    T min(T a, T b) {
        return (a < b) ? a : b;
    }
    ```
    - Far fewer problems: variables evaluated only once.
Exception Handling

• What is an exception?
  • a hardware-detected run-time error or unusual condition detected by software

• Examples
  • arithmetic overflow
  • end-of-file on input
  • wrong type for input data
  • user-defined conditions, not necessarily errors
Exception Handling

- What is an exception handler?
  - code executed when exception occurs
  - may need a different handler for each type of exception

- Why design in exception handling facilities?
  - allow user to explicitly handle errors in a uniform manner
  - allow user to handle errors without having to check these conditions
  - explicitly in the program everywhere they might occur
Coroutines

- Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name.
- Coroutines can be used to implement:
  - iterators
  - threads
- Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack.
Coroutines

var q := new queue
coroutine produce
  loop
    while q is not full
      create some new items
      add the items to q
    yield to consume
coroutine consume
  loop
    while q is not empty
      remove some items from q
      use the items
    yield to produce
Coroutines

Figure 8.6 A cactus stack. Each branch to the side represents the creation of a coroutine (A, B, C, and D). The static nesting of blocks is shown at right. Static links are shown with arrows. Dynamic links are indicated simply by vertical arrangement: each routine has called the one above it. (Coroutine B, for example, was created by the main program, M. B in turn called subroutine S and created coroutine D.)
Summary

- Functional Abstraction:
  - Functions help us abstract the code:
    - by being able to give parts of the program meaningful name
    - by creating scopes in which data and control flow is controlled.
  - Learned about stack layouts:
    - Static link.
    - Dynamic link.
  - 3 main calling conventions.
    - Pass by value.
    - Pass by reference.
    - Pass by sharing.
- Generics
  - Java
  - C++