CS 430
Spring 2015

Mike Lam, Professor

Subprograms and Activation

Warm-up Activity

- What does the following C++ program print?

```
#include <stdio.h>
int foo(int x, int *y, int &z) {
    x = 4; *y = 5; z = 6;
    printf("%d %d %d\n", x, *y, z);
    return x + *y + z;
}
int main() {
    int a, b, c, d;
    a = 1; b = 2; c = 3;
    d = foo(a, &b, c);
    printf("%d %d %d %d\n", a, b, c, d);
}
```

Subprograms

- General characteristics
  - Single entry point
  - Caller is suspended while subprogram is executing
  - Control returns to caller when subprogram completes
- Procedure vs. function
  - Functions have return values

Subprograms

- New-ish terms
  - Header: signaling syntax for defining a subprogram
  - Parameter profile: number, types, and order of parameters
  - Signature/protocol: parameter types and return type(s)
  - Prototype: declaration without a full definition
  - Referencing environment: variables visible inside a subprogram
  - Call site: location of a subprogram invocation

Parameters

- Formal vs. actual parameters
  - Formal: parameter inside subprogram definition
  - Actual: parameter at call site
- Semantic models: *in*, *out*, *in-out*
- Implementations (key differences are *when* values are copied and exactly *what* is being copied)
  - **Pass-by-value (*in, value*)**
  - Pass-by-result (*out, value*)
  - Pass-by-copy (*in-out, value*)
  - **Pass-by-reference (*in-out, reference*)**
  - **Pass-by-name (*in-out, name*)**

Parameters

- Pass-by-value
  - Pro: simple
  - Con: costs of allocation and copying
  - Often the default
- Pass-by-reference
  - Pro: efficient (only copy 32/64 bits)
  - Con: hard to reason about, extra layer of indirection, aliasing issues
  - Often used in object-oriented languages
- Pass-by-name
  - Pro: powerful
  - Con: expensive to implement, very difficult to reason about
  - **Rarely used!**

Example

- Trace x, y, a, b, c, and d after each numbered line:

```
   foo(a,b,c,d):
1:   a = a + 1                # a is passed by value
2:   b = b + 1                # b is passed by copy
3:   c = c + 1                # c is passed by reference
4:   d = d + 1                # d is passed by name

     x = [1,2,3,4]
     y = 2
5:   foo(x[0],x[1],y,x[y])
```

Example

- Trace x, y, a, b, c, and d after each numbered line:

```
     foo(a,b,c,d):
1:     a = a + 1           # a is passed by value
2:     b = b + 1           # b is passed by copy
3:     c = c + 1           # c is passed by reference
4:     d = d + 1           # d is passed by name

       x = [1,2,3,4]
       y = 2
5:     foo(x[0],x[1],y,x[y])
```

```
       x = [1,2,3,4]  y=2   a=1  b=2  c=&y  d=x[y]
    1: x = [1,2,3,4]  y=2   a=2  b=2  c=&y  d=x[y]
    2: x = [1,2,3,4]  y=2   a=2  b=3  c=&y  d=x[y]
    3: x = [1,2,3,4]  y=3   a=2  b=3  c=&y  d=x[y]
    4: x = [1,2,3,5]  y=3   a=2  b=3  c=&y  d=x[y]
    5: x = [1,3,3,5]  y=3   a=2  b=3  c=&y  d=x[y]
```

Other Design Issues

- How are formal/actual parameters associated?
  - Positionally, by name, or both?
- Are parameter default values allowed?
- Are method parameters type-checked?
  - Statically or dynamically?

Other Design Issues

- Are local variables statically or dynamically allocated?
- Can subprograms be passed as parameters?
  - How is this implemented?
  - Shallow/dynamic, deep/static, or ad-hoc referencing environment?
- Can subprograms be nested?
- Can subprograms be polymorphic?
  - Ad-hoc/manual, subtype, or parametric/generic?
- Are function side effects allowed?
- Can a function return multiple values?

Misc. Topics

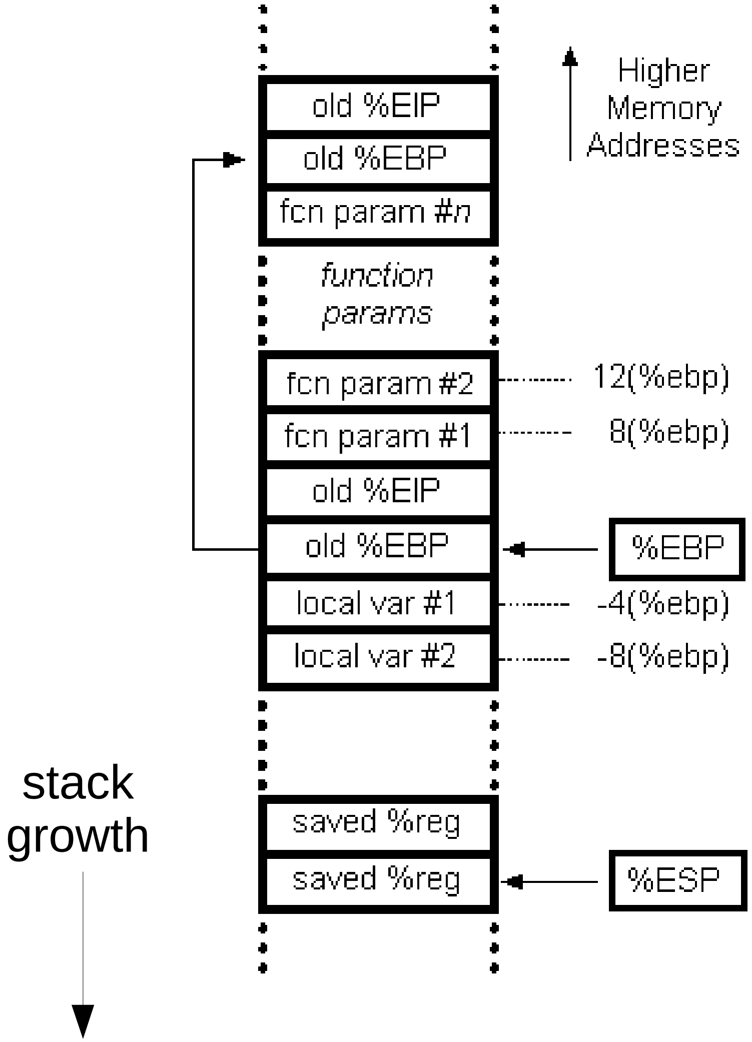
- Macros
  - Call-by-name, “executed” at compile time
- Closures
  - A nested subprogram and its referencing environment
- Coroutines
  - Co-operating procedures

Subprogram Activation

- Call semantics:
  - Save caller status
  - Compute and save parameters
  - Save return address
  - Transfer control to callee
- Return semantics:
  - Save return value(s) and out parameters
  - Restore caller status
  - Transfer control back to the caller
- *Activation record:* data for a single subprogram execution
  - Local variables
  - Parameters
  - Return address
  - Dynamic link

x86 Stack Layout

- Address space
  - Code
  - Static
  - Stack
  - Heap
- Instruction Pointer (IP)
  - Current instruction
- Stack pointer (SP)
  - Top of stack
- Base pointer (BP)
  - Start of current frame



x86 Calling Conventions

Prologue:

```
      push %ebp               ; save old base pointer
      mov  %esp, %ebp         ; save top of stack as base pointer
      sub  X, %esp            ; reserve X bytes for local vars
```

Within function:

```
      +OFFSET(%ebp)           ; function parameter
      -OFFSET(%ebp)           ; local variable
```

Epilogue:

```
      <optional: save return value in %eax>
      leave                   ; mov %ebp, %esp
                              ; pop %ebp
      ret                     ; pop stack and jump to popped address
```

Function calling:

```
      <push parameters>
      <push return address>
      <jump to fname>
      <pop parameters>
```

x86_64 "red zone" (128 bytes reserved below SP)

- optimization: do not explicitly build frame (no SP manipulation)