Structured Programming

- Unstructured programming
  - Jumps
  - Famous discussions about the GOTO statement

- Boehm-Jacopini-Theorem:
  - The control structure of every algorithm for a computable function can be expressed by a combination of the following elements:
    - Sequence
      - Sequentially execute two subprograms A, B
    - Selection
      - Intuitively: IF a then B else C
    - Repetition
      - Intuitively: WHILE a DO B

Functions and Procedures

- Subroutines (or subprograms) emerged as result of structured programming
- Named subprograms

- Functions are subroutines which return a result
- Procedures are subroutines which do not return a result

- But:
  - In C both are called functions
  - In Scheme both are called procedures

Scopes

- In Lambda-Calculus:
  - $\lambda x. t \Rightarrow t$ is the scope of $x$

- In many programming languages:

```plaintext
{  var x:=1
    --
   }
```

- Can be standalone (Block Scope), part of an expression (e.g., if (...) then {...} else {...}) , or be a function body.
Scopes

- Function Scopes
  ```javascript
  function f(x) {
    var y := 10;
    ...
  }
  ```

- (Lexically) Nested Functions
  ```javascript
  function outer() {
    ...
    function inner() {
    }
  }
  ```

  Supported in some languages, e.g., Pascal, Scala, (modern) Lisp, ...
  C as a language does not support it but GCC does as an extension

Scopes

- Classic example:
  ```javascript
  var x = 1;
  bar();
  function foo() { print x; }
  function bar() { var x = 100; foo(); }
  ```

- Static scoping:
  - output is 1
- Dynamic scoping:
  - output is 100

Scopes

- Scope: visibility of variable
- Lifetime: period where variable exists in memory
- Scope ≠ lifetime!

```javascript
{  
  var x := 1;
  {  
    var y := 2;
    {  
      var x := 3;
      var z := x + y;
    }
  }
  var a := x + 1;
}
```
Globals
- In many languages: variable declared in “scope 0”, outside of any scope
- static in C and C++
- Universally visible (unless shadowed by local variable)
- Has implication for the storage model.
  - Where to keep static variables
- In the general case: variable outside of current scope

Function Parameter
- Formal Parameter:
  - Statically defined in the function definition
  - function foo(a, b, c)
- Arguments
  - Passed at runtime
  - foo(1, true, f(x))
- Binding of formal to actual is performed at runtime according to the evaluation strategy.

Evaluation Strategy
- Call by value:
  - Formal is bound to the value of actual expression by evaluating it and assigning the result to the function variable.
  - A copy of the actual value is created.
- Call by reference:
  - Formal is bound to location of actual expression.
  - The function can change the value of the arguments and thereby change values which are outside of its own scope.
- Call by name:
  - “Textual replacement” = Substitution

Example:
```
x = 5;
function foo(in) {
    in = 10;
}
foo(x);
print(x);
```
- prints 5
- Call by value
- prints 10
- Call by reference
Evaluation Strategy in C

- **Call by value:**
  ```
  void foo(int i) {
    i++;
  }
  ```

- **Call by reference:**
  ```
  void foo(int *i) {
    i++;
  }
  ```

- **Call by name?**
  ```
  #define foo(i) i++
  ```

Activation Record

- Remember that we talked about lifetime of variables...
- The variables local to the scope need to be kept somewhere, e.g., in memory

Activation Record per function invocation

- Can contain arguments, local variables, return address, etc.
- Precise format depends on the calling conventions

Stack

- LIFO, grows from top of memory to the bottom
- Compiler generates code for allocating memory on the stack where needed

Stack Frame

- Example: x86_64 calling conventions for C:
- First 6 integers (long, uint64_t) or pointers are passed through registers
- Other arguments on stack
- Local variables reside in the memory area below the base pointer
  - Statically allocated through compiler-generated code

Function Prologue/Epilogue in C

- **Prologue**
  ```
  push rbp
  mov rbp, rsp
  sub rsp, 0x64
  ...
  ```
  - push the old base pointer to the stack
  - set new base pointer to old stack pointer (remember, we grow towards the bottom)
  - span new stack frame by decrementing the stack pointer
  - typically save callee-saved registers to stack if needed

- **Epilogue**
  ```
  add rsp, 0x64
  pop rbp
  ret
  ```
  - "restore" stack pointer
  - pop old base pointer from stack
  - return (jump to return address read from stack)

- With optimizations enabled, the compiler plays tricks to avoid some or all of these instructions.
Stack Frame Example

```c
function main() {
    foo();
}

function foo() {
    bar();
}

function bar() {
    mov rbp, rsp
    sub rbp, 0x...
}
```

Activation Records

- Modern CPUs are optimized to deal with function calls and stack manipulations
- However, the static allocation is inflexible
- Some languages, especially functional languages, allocate their activation records on the heap
  - These languages typically allow dynamic creation of functions
  - Heap supports dynamic allocation at any time (think of `malloc`)

Activation Records

- Allocation on the heap is slower than using the stack frame
- Creating a new stack frame is still overhead
  - At least two extra instructions
  - Optimizing compilers try to avoid it wherever possible
- Two common optimizations:
  - Leaf Functions
    - If a function does not call any further functions (leaf function), it is a candidate for running without its own stack frame
    - Works if all arguments to the leaf function can be passed in registers and the return value can be passed through register
  - Function Inlining
    - Eliminates the called function by copying the code into the calling function so that no separate activation record is needed
    - Must do it in a "scope-preserving" way!

Globals Revisited

- In C, we can only access local variables, arguments, and globals.
  - No nested functions in the language
- Where are the global (static) variables allocated?
  - In the Data Segment, a portion of the virtual address space of a program, again generated by the compiler as part of the binary
  - Global variable can be statically referenced relative to the data segment, the compiler keeps track of them
439 Flashback

- Runtime memory image
- Created by the loader
- Copies code and data from storage to memory
- Jumps into the program code at a well-known entry point
  - `C: \&_start`, defined in `crt1.o` (or `crt0.o`)
  - Sets up an initial stack frame
  - Calls main
  - (After main terminates)
  - Returns control to OS

Disclaimer: Simplification (missing VirtMem)

Nested Functions Revisited

- How do activation records deal with nested functions?
  - Usually each activation record contains a link to the lexically enclosing activation record ("dynamic link")