Imperative Programming

Vitaly Shmatikov
Reading Assignment

- Mitchell, Chapter 5.1-2
- C Reference Manual, Chapter 8
Imperative Programming

- Oldest and most popular paradigm
  - Fortran, Algol, C, Java ...
- Mirrors computer architecture
  - In a von Neumann machine, memory holds instructions and data
- Key operation: assignment
  - Side effect: updating state (i.e., memory) of the machine
- Control-flow statements
  - Conditional and unconditional (GO TO) branches, loops
Elements of Imperative Programs

- Data type definitions
- Variable declarations (usually typed)
- Expressions and assignment statements
- Control flow statements (usually structured)
- Lexical scopes and blocks
  - Goal: provide locality of reference
- Declarations and definitions of procedures and functions (i.e., parameterized blocks)
Variable Declarations

Typed variable declarations restrict the values that a variable may assume during program execution

- Built-in types (int, char ...) or user-defined
- Initialization: Java integers to 0. What about C?

Variable size

- How much space needed to hold values of this variable?
  - C on a 32-bit machine: sizeof(char) = 1 byte, sizeof(short) = 2 bytes, sizeof(int) = 4 bytes, sizeof(char*) = 4 bytes (why?)
  - What about this user-defined datatype:

```c
typedef struct TreeNode {
    int x;
    TreeNode *front, *back;
} ;
```
Variables: Locations and Values

- When a variable is declared, it is bound to some memory location and becomes its identifier
  - Location could be in global, heap, or stack storage
- l-value: memory location (address)
- r-value: value stored at the memory location identified by l-value
- Assignment: $A$ (target) $=$ $B$ (expression)
  - Destructive update: overwrites the memory location identified by $A$ with a value of expression $B$
    - What if a variable appears on both sides of assignment?
Copy vs. Reference Semantics

- **Copy semantics**: expression is evaluated to a value, which is copied to the target
  - Used by imperative languages
- **Reference semantics**: expression is evaluated to an object, whose pointer is copied to the target
  - Used by object-oriented languages
Variables and Assignment

- On the RHS of an assignment, use the variable’s r-value; on the LHS, use its l-value
  - Example: \( x = x + 1 \)
  - Meaning: “get r-value of \( x \), add 1, store the result into the l-value of \( x \)”

- An expression that does not have an l-value cannot appear on the LHS of an assignment
  - What expressions don’t have l-values?
    - Examples: \( 1 = x + 1 \), \( ++x++ \) (why?)
    - What about \( a[1] = x + 1 \), where \( a \) is an array? Why?
I-Values and r-Values (1)

◆ Any expression or assignment statement in an imperative language can be understood in terms of l-values and r-values of variables involved
  • In C, also helps with complex pointer dereferencing and pointer arithmetic

◆ Literal constants
  • Have r-values, but not l-values

◆ Variables
  • Have both r-values and l-values
  • Example: \( x = x \times y \) means “compute rval(x)*rval(y) and store it in lval(x)”
l-Values and r-Values (2)

◆ Pointer variables
  • Their r-values are l-values of another variable
    – Intuition: the value of a pointer is an address

◆ Overriding r-value and l-value computation in C
  • &x always returns l-value of x
  • *p always return r-value of p
    – If p is a pointer, this is an l-value of another variable

int x = 5; // lval(x) is some (stack) address, rval(x) == 5
int *p = &x // rval(p) == lval(x)
*p = 2 * x; // rval(p) <- rval(2) * rval(x)

What are the values of p and x at this point?
l-Values and r-Values (3)

Declared functions and procedures

- Have l-values, but no r-values

```c
int f(int y); // lval(f) is some global address
typedef int (*IFP)(int); // pointer to an int function that takes an int argument
IFP g = &f; // lval(g) <- lval(f)
(*g)(5); // (rval(g)) == lval(f), so *g invokes f with argument rval(5)
    // the function call operator () has higher precedence than * so
    // we have to write (*g)(5) to defference g to invoke f(5)
```
Turing-Complete Mini-Language

- Integer variables, values, operations
- Assignment
- If
- Go To
Structured Control Flow

Control flow in imperative languages is most often designed to be **sequential**

- Instructions executed in order they are written
- Some also support concurrent execution (Java)

Program is **structured** if control flow is evident from syntactic (static) structure of program text

- **Big idea**: programmers can reason about dynamic execution of a program by just analyzing program text
- Eliminate complexity by creating language constructs for common control-flow “patterns”
  - Iteration, selection, procedures/functions
Fortran Control Structure

10 IF (X .GT. 0.000001) GO TO 20
11 X = -X
   IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
   X = X-Y-Y
30  X = X+Y
...
50 CONTINUE
   X = A
   Y = B-A
   GO TO 11
...

Similar structure may occur in assembly code
Historical Debate

◆ Dijkstra, “GO TO Statement Considered Harmful”
  • Letter to Editor, Comm. ACM, March 1968
  • Linked from the course website

◆ Knuth, “Structured Prog. with Go To Statements”
  • You can use goto, but do so in structured way …

◆ Continued discussion
  • Welch, “GOTO (Considered Harmful)^n, n is Odd”

◆ General questions
  • Do syntactic rules force good programming style?
  • Can they help?
Modern Style

- Standard constructs that structure jumps
  - if ... then ... else ... end
  - while ... do ... end
  - for ... { ... }
  - case ...

- Group code in logical blocks

- Avoid explicit jumps (except function return)

- Cannot jump into the middle of a block or function body
Iteration

**Definite**

\[
\text{for (int } i = 0; i < 10; i++) \{
\quad a[i] = 0; \quad // initialize each array element to zero
\}\]

**Indefinite**

- Termination depends on a dynamically computed value

\[
\text{int } m = 0;
\text{while (n > 0) } \{
\quad m = m \ast n;
\quad n = n - 1;
\}\]

How do we know statically (i.e., before we run the program) that **the loop will terminate**, i.e., that \( n \) will eventually become less than or equal to 0?
Iteration Constructs in C

- while (condition) stmt;
  
  while (condition) { stmt; stmt; ...; }

- do stmt while (condition);
  
  do { stmt; stmt; ...; } while (condition);

- for (<initialize>; <test>; <step>) stmt;
  
  - Restricted form of “while” loop - same as
    
    <initialize>; while (<test>) { stmt; <step> }

    for (<initialize>; <test>; <step>) { stmt; stmt; ...; }
“Breaking Out” Of A Loop in C

```c
int y; // y is in the “outer” scope
...
while (cond == true) {
    int x; // x is local to the while blocks scope (its extent and lifetime)
    ...
    if (x < y) {  // special case...
        break; // leave while loop
    }
    ... // normal case
}

while (cond1 == true) {
    while (cond2 == true) {
        if (x < y) // special case
            break; // leave inner loop, but not outer loop
        ...
    }
    ... // control resumes here after a break from the inner loop
```
Forced Loop Re-Entry in C

while (cond-expr == true) {
    ... // do something while cond is true
    if (a == b) {
        ... // do something special
        continue; // transfer to start of while and re-evaluate cond
    }
    ... // remaining statements of while loop
}
Block-Structured Languages

Nested blocks with local variables

- Storage management
  - Enter block: allocate space for variables
  - Exit block: some or all space may be deallocated
Blocks in Common Languages

◆ Examples
  • C, JavaScript * { ... }
  • Algol begin ... end
  • ML let ... in ... end

◆ Two forms of blocks
  • Inline blocks
  • Blocks associated with functions or procedures
    - We’ll talk about these later

* JavaScript functions provides blocks
Simplified Machine Model

Registers

Code

Data

Program counter

Environment pointer

Stack

Heap
Memory Management

- Registers, Code segment, Program counter
  - Ignore registers (for our purposes) and details of instruction set

- Data segment
  - Stack contains data related to block entry/exit
  - Heap contains data of varying lifetime
  - Environment pointer points to current stack position
    - Block entry: add new activation record to stack
    - Block exit: remove most recent activation record
Scope and Lifetime

◆ Scope
  • Region of program text where declaration is visible

◆ Lifetime
  • Period of time when location is allocated to program

```c
{ int x = ..., 
    { int y = ..., 
        { int x = ..., 
            ... 
        },
    },
};
```

• Inner declaration of x hides outer one ("hole in scope")
• Lifetime of outer x includes time when inner block is executed
• Lifetime ≠ scope
**Inline Blocks**

◆ **Activation record**
  - Data structure stored on **run-time stack**
  - Contains space for local variables

```
{ int x=0;
  int y=x+1;
  { int z=(x+y)*(x-y);
  };
};
```

Push record with space for x, y
Set values of x, y
Push record for inner block
Set value of z
Pop record for inner block
Pop record for outer block

May need space for variables and intermediate results like \((x+y), (x-y)\)
Activation Record ForInline Block

Control link
- Pointer to previous record on stack

Push record on stack
- Set new control link to point to old env ptr
- Set env ptr to new record

Pop record off stack
- Follow control link of current record to reset environment pointer

In practice, can be optimized away
Example

```c
{ int x=0;
    int y=x+1;
    {
        int z=(x+y)*(x-y);
    }
};
```

Push record with space for x, y
Set values of x, y
Push record for inner block
Set value of z
Pop record for inner block
Pop record for outer block