Computer Systems

Lecture 16

Nested Conditionals

Dr Lito Michala, Dr José Cano
School of Computing Science
University of Glasgow
Spring 2020

Outline

- Blocks
  - Syntax for blocks
  - Enter at beginning, exit at end
- Nested conditionals
- Case and jump tables
- Real world incident: the SSL Bug

Blocks

- You'll need to learn many programming languages, eventually
  - There are concepts that appear in most languages
  - It's useful to focus on the general concept

- One important concept is a block
  - A block or compound statement is a single statement that contains several statements
  - The purpose is to let you have a group of statements in a loop, or controlled by a conditional

- The syntax is the detailed punctuation used to indicate a block, and this varies in different languages
  - There are lots of small syntax differences between languages
  - Some languages use := to mean assign, = to mean equals
  - Some languages use = to mean assign, == to mean equals

Python syntax style for blocks: layout

- The layout (the indentation) determines what is inside the if statement

```
a = 1
if x < y :
    b = 2
    c = 3
d = 4
```

- If you change the indentation, you change the meaning of the program

- In Python, you write : to mean then, and else to mean else

Algol/Pascal style for blocks: begin-end

```
a := 1;
if x<y
   then begin b := 2;
                  c := 3
          end
   else begin d := 4;
                  e := 5
          end;
f := 6;
```

- You write then to mean then, and else to mean else

- Statements must be separated by semicolon ;

C style for blocks: braces

```
a = 1;
if (x < y) b = 2;
else
      { d = 4;
        e = 5; }
f = 6;
```

- In C you don't write then at all, but this means the condition x < y must be enclosed in parentheses so the compiler can tell where the condition ends and the then-statement begins

Block structured style for blocks: matching keywords

```
a := 1;
if x<y
     then b := 2;
          while i<n do
               sum := sum+x[i];
               i := i+1
          endwhile
     else d := 4
          e := 5
endif;
f := 6;
```

- This style introduces a lot of keywords (endif, endwhile, endfor, endrepeat) but you don't need braces around a block
  - It makes code more readable and enables the compiler to produce better error messages

Enter at beginning, exit at end

- A common programming style is to require
  - Each block enters only at the beginning of the block
  - Each block exits only at the end of the block

- This style is helpful in some programming languages, but in some languages it sometimes makes code less readable

- In assembly language, and for compilation patterns, it is necessary to follow this style

- In high level languages this style is sometimes helpful, but not always

Single entrance/exit for compilation patterns

- It is straightforward to translate high level control constructs using the compilation patterns

- These require that the blocks of code always start at the beginning and finish at the end

- It's bad to jump into the middle of a block, or exit out of the middle because
  - The compilation patterns won't work correctly
  - You'll have to duplicate a lot of code
  - Example: returning from a procedure requires restoring the registers, resetting the stack pointer, loading the return address
  - That code should not be duplicated in several places in a procedure

Systematic approach to programming?

- Start by understanding what your program should do
- Express the algorithm using high level language notation (and it's ok to mix in some English too)
- Translate the high level to the low level
  - Assignment statements: x := expression
  - I/O statements: Write string
  - goto L
  - if boolean then goto L
- Translate the low level to assembly language
- Retain the high and low level code as comments
- Do hand execution at every level

Why use this systematic approach to programming?

- It enables you to write correct code at the outset, and minimize debugging
- If there is a bug, it helps you to catch it early (e.g. in translation to goto form)
- If there's a bug in an instruction, the comments enable you to find it quickly
  - A common error is to use a wrong register number: add R9,R3,R4
  - Poor comments don't help: add R9,R3,R4 ; R9 := R3+R4
  - Good comments help a lot: add R9,R3,R4 ; x := alpha + (a[i]*b[i])
  - Look at the register usage comments (oops, x is in R8, not R9, now I know how to x it and I don't have to read the entire program)
- Professional software needs to be maintained
  - The comments make the software easier to read and more valuable

Outline

- Blocks
  - Syntax for blocks
  - Enter at beginning, exit at end
- Nested conditionals
- Case and jump tables
- Real world incident: the SSL Bug

Nested if-then-else

- Conditional statements can be nested deeply

```
if b1
    then S1
        if b2
            then S3
            else S4
        S5
    else  S6
            if b3
                then S7
            S8
```

Special case of nested if-then-else

- Often the nesting isn't random, but has this specific structure

```
if b1
  then S1
  else if b2
          then S2
          else if b3
                  then S3
                  else if b4
                          then S4
                          else if b5
                                  then S5
                                  else S6
```

- (Some languages require punctuation to avoid ambiguity)

Another way to write it

- To avoid running o the right side of the window, it's usually indented like this

```
if b1
    then S1
else if b2
    then S2
else if b3
    then S3
else if b4
    then S4
else if b5
    then S5
else S6
```

Some programming languages have elsif or elif

```
if b1 then S1
    elif b2 then S2
    elif b3 then S3
    elif b4 then S4
    eleif b5 then S5
    else S6
```

- It avoids ambiguity
- It signals to the compiler and to a human programmer that this specific construct is being used
- It allows good indentation layout without violating the basic principle of indentation
- Some languages have this, some don't

A common application: numeric code

- Nested if-then-else but the boolean conditions are not arbitrary
  - They are checking the value of a code

```
if code = 0
    then S1
else if code = 1
    then S2
else if code = 2
    then S3
else if code = 3
    then S4
else if code = 4
    then S5
```

Outline

- Blocks
  - Syntax for blocks
  - Enter at beginning, exit at end
- Nested conditionals
- Case and jump tables
- Real world incident: the SSL Bug

The case statement

```
case n of
    0 -> Stmt
    1 -> Stmt
    2 -> Stmt
    3 -> Stmt
    4 -> Stmt
    5 -> Stmt
    else -> Stmt // handle error
```

- This means: execute the statement corresponding to the value of n
- Many programming languages have this; the syntax varies a lot but that isn't what's important

Example: numeric code specifies a command

```
; The input data is an array of records, each specifying an operation
;     Command : record
;           code : Int          ; specify which operation to perform
;           i : Int             ; index into array of lists
;           x : Int             ; value of list element

; The meaning of a command depends on the code:
;     0 terminate the program
;     1 insert x into set[i]
;     2 delete x from set[i]
;     3 return 1 if set[i] contains x, otherwise 0
;     4 print the elements of set[i]
```

Selecting the command with a case statement

```
; Initialize
;      BuildHeap ()

; Execute the commands in the input data
;       finished := 0
;       while InputPtr <= InputEnd && not finished
;              CurrentCode := (*InputPtr).code
;              p := set[*InputPtr]     ; linked list
;              x := (*InputPtr).x                    ; value to insert, delete, search

;               case CurrentCode of
;                   0 : <CmdTerminate>
;                   1 : <CmdInsert>
;                   2 : <CmdDelete>
;                   3 : <CmdSearch>
;                   4 : <CmdPrint>
;                   else : <>
;              InputPtr := InputPtr + sizeof(Command)
; Terminate the program
; halt
```

Finding a numeric code

- It's tedious and inefficient to go through the possible values of a numeric code in sequence
  - If you're looking up Dr Zhivago in the phone book, do you look at Arnold Aardvark, and Anne Anderson, on on and on?
  - You go straight to the end of the book

- We want to find the statement corresponding to a numeric code directly, without checking all the other values

A problem with efficiency

- The problem
  - There are many applications of case statements
  - They are executed frequently
  - The number of cases can be large (not just 4 or 5; can be hundreds)
  - The implementation of the if-then-else requires a separate compare and jump for each condition

- The solution
  - A technique called jump tables

Jump tables: the basic idea

- For each target statement (S1, S2, S3, etc) in the conditional, introduce a jump to it: jump S1[R0], jump S2[R0], etc

- Make an array "jt" of these jump instructions

  jt[0] = jump S0[R0]

  jt[1] = jump S1[R0]

  jt[2] = jump S2[R0]

  jt[3] = jump S3[R0]

  jt[4] = jump S4[R0]

- Given the code, Jump to jt[code]
- Each element of the array is an instruction that requires two words
- So jump to &jt + 2 code

Jump table

```
; Jump to operation specified by code
        add R4,R4,R4                    ; code := 2*code
        lea R5,CmdJumpTable[R0]         ; R5 := pointer to jump table
        add R4,R5,R4                    ; address of jump to operation
        jump 0[R4]                      ; jump to jump to operation
CmdJumpTable
        jump CmdTerminate[R0]           ; code 0: terminate the program
        jump CmdInsert[R0]              ; code 1: insert
        jump CmdDelete[R0]              ; code 2: delete
        jump CmdSearch[R0]              ; code 3: search
        jump CmdPrint[R0]               ; code 4: print

CmdDone
        load R5,InputPtr[R0]
        lea R6,3[R0]
        add R5,R5,R6
        store R5,InputPtr[R0]
        jump CommandLoop[R0]
```

We have to be careful!

- What if code is negative, or larger than the number of cases?
- The jump to the jump table could go anywhere!
- It might not even go to an instruction
- But whatever is in memory at the place it goes to, will be interpreted as an instruction that will be executed
- The program will go haywire
- Debugging it will be hard: the only way to catch the error is to read the code and/or to single step
- Solution: before jumping into the jump table, check to see if code is invalid (too big or too small)

Checking whether the code is invalid

```
CommandLoop
      load R5,InputPtr[R0]          ; R1 := InputPtr
      load R4,0[R5]                 ; R4 := *InputPtr.code
; Check for invalid code
      cmp R4,R0                     ; compare (*InputPtr).code, 0
      jumplt CmdDone[R0]            ; skip invalid code (negative)
      lea R6,4[R0]                  ; maximum valid code
      cmp R4,R6                     ; compare code with max valid code
      jumpgt CmdDone[R0]            ; skip invalid code (too large)
...

CmdDone
      load R5,InputPtr[R0]
      lea R6,3[R0]
      add R5,R5,R6
      store R5,InputPtr[R0]
      jump CommandLoop[R0]
```

Outline

- Blocks
  - Syntax for blocks
  - Enter at beginning, exit at end
- Nested conditionals
- Case and jump tables
- Real world incident: the SSL Bug

A real world incident!

- The moral of the following true story: be meticulous!
- On 21 February 2014, Apple announced a bug in core security software
- It affected iPhone, iPad, Macintosh, Safari
- The bug is serious: it skips a crucial security check in SSL
- A security update was made available quickly
  - That's why you should always apply your security updates!
- It was fixed quickly but we can learn some important lessons from it

What is SSL?

- Secure Sockets Layer
- It arranges that communication between your device and a server is encrypted
- When you see a “padlock icon” on a browser, it means the SSL protocol is being used

Source le: sslKeyExchange.c

- 1,969 lines of code in C
- You can see the code: click on this link, or type it in:
  - http://opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c
- The bug made it possible to break open “secure” Internet trac
- It appears somewhere on the next three slides: can you spot it?

The function SSLVerifySignedServerKeyExchange (1)

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    SSLBuffer       hashOut, hashCtx, clientRandom, serverRandom;
    uint8_t         hashes[SSL_SHA1_DIGEST_LEN + SSL_MD5_DIGEST_LEN];
    SSLBuffer       signedHashes;
    uint8_t                     *dataToSign;
        size_t                  dataToSignLen;

        signedHashes.data = 0;
    hashCtx.data = 0;

    clientRandom.data = ctx->clientRandom;
    clientRandom.length = SSL_CLIENT_SRVR_RAND_SIZE;
    serverRandom.data = ctx->serverRandom;
    serverRandom.length = SSL_CLIENT_SRVR_RAND_SIZE;
```

The function continued (2)

```
    if(isRsa) {
            /* skip this if signing with DSA */
            dataToSign = hashes;
            dataToSignLen = SSL_SHA1_DIGEST_LEN + SSL_MD5_DIGEST_LEN;
            hashOut.data = hashes;
            hashOut.length = SSL_MD5_DIGEST_LEN;

            if ((err = ReadyHash(&SSLHashMD5, &hashCtx)) != 0)
                    goto fail;
            if ((err = SSLHashMD5.update(&hashCtx, &clientRandom)) != 0)
                    goto fail;
            if ((err = SSLHashMD5.update(&hashCtx, &serverRandom)) != 0)
                    goto fail;
            if ((err = SSLHashMD5.update(&hashCtx, &signedParams)) != 0)
                    goto fail;
            if ((err = SSLHashMD5.final(&hashCtx, &hashOut)) != 0)
                    goto fail;
    }
    else {
            /* DSA, ECDSA - just use the SHA1 hash */
            dataToSign = &hashes[SSL_MD5_DIGEST_LEN];
            dataToSignLen = SSL_SHA1_DIGEST_LEN;
    }
    hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
```

The function continued (3)

```
    if ((err = SSLFreeBuffer(&hashCtx)) != 0)
        goto fail;

    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

        err = sslRawVerify(ctx,
                           ctx->peerPubKey,
                           dataToSign,                          /* plaintext */
                           dataToSignLen,                  /* plaintext length */
                           signature,
                           signatureLen);
        if(err) {
                sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
                goto fail;
        }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

A bit of C

- Syntax of the if-statement

```
if (condition) statement;
```

- Examples

```
if (x<y) n=n+1;

if (n==5)
     { p = 3;
       q = 4;
       printf ("hello p = %3d\n", p);
}
```

A closer look

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

- The second goto fail is unconditional so the check in the last if statement is never executed

What can we learn from this?

- To be a good programmer you need to
  - Know your programming language
  - Be meticulous
  - Read your code carefully
  - Do thorough testing

- It was very helpful that Apple published the source code for this bug

A mystery: The Attempted Linux Trapdoor Hack of 2003

- A trapdoor is a aw (bug? hack?) in software that allows a hostile user to gain abilities they should not have
  - It's like giving the key to the safe in a bank to a burglar
- The master copy of Linux was maintained on Bitkeeper
  - No longer, now git is used
- A secondary copy was on cvs
- Someone (unknown to this day) made a small edit to an obscure piece of code on the cvs version

The offending code: can you spot the trapdoor?

- A change was made to a function that a user could call to wait for an event
- System functions typically take options, and they need to check to see if the options are valid
- If not, a return code *retval* is set to a non-zero value
- So, here's the code

```
if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
        retval = -EINVAL;
```

What's going on?

- In C, == is the comparison operator = is the assignment operator (bad notation, but we're stuck with it)

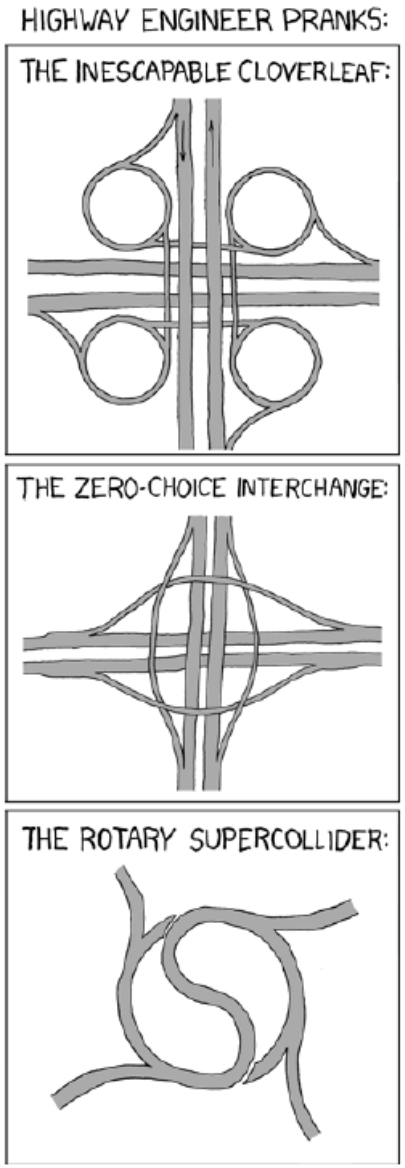
- This code looks like it is checking:
  - If either the WCLONE or WALL option ag is set and the current user is root, then return an error condition

- What it actually says is
  - Check the irrelevant ags, set the current user to be root, get the irrelevant result 0 and do nothing else

```
if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
        retval = -EINVAL;
]
```

What's the significance of root?

- Many processes are running in the computer
- Each has its own set of privileges
- There is one special user “root” which has all privileges: root can do anything at all
- The effect of this change to the source code (a change of one character! is
  - If a user program calls this obscure function with just the right set of obscure options, it suddenly gains full control over the machine
- Don't worry, this faulty code never made it into the master copy of Linux



https://xkcd.com/253/