C Control Structures

Chapter 13
Control Structures

**Conditional**
- making a decision about which code to execute, based on evaluated expression
  - *if*
  - *if-else*
  - *switch*

**Iterative**
- executing code multiple times, ending based on evaluated expression
  - *for*
  - *while*
  - *do-while*
if (condition) action;

**Condition** is a C expression, which evaluates to TRUE (non-zero) or FALSE (zero).

**Action** is a C statement, which may be simple or compound (a block).
Examples of if statements

```java
if (x <= 10)
    y = x * x + 5;

if (x <= 10)
    { y = x * x + 5;
      z = (2 * y) / 3;
    }

if (x <= 10)
    y = x * x + 5;
    z = (2 * y) / 3;
```

- **Compound statement:** both executed if `x <= 10`
- **Only first statement is conditional:** second statement is always executed (indentation doesn't help)
About braces indentation...

if (x <= 10)
{
    y = x * x + 5;
    z = (2 * y) / 3;
}

Good style: the braces are aligned (*)

"Different " style: the braces are not aligned (*)

(*) Source: My opinion
LC-3 code for if statement

\[
\text{if} \ (x == 2) \\
y = 5;
\]

```
LDR  R0, R5, #0       ; load x into R0
ADD  R0, R0, #-2      ; subtract 2
BRnP NOT_TRUE          ; if non-zero, x is not 2
AND  R1, R1, #0       ; store 5 to y
ADD  R1, R1, #5
STR  R1, R5, #-1
NOT_TRUE ...          ; next statement
```
LC-3 code for if-else

if(x)
{
    y++;        
z--;        
}
else
{
    y--;        
z++;        
}

LDR  R0, R5, #0
BRz  ELSE

; x is not zero
LDR  R1, R5, #-1  ; incr y
ADD  R1, R1, #1
STR  R1, R5, #-1
LDR  R1, R5, #02  ; decr z
ADD  R1, R1, #1
STR  R1, R5, #-2
JMP  DONE  ; skip else code

; x is zero
ELSE
LDR  R1, R5, #-1  ; decr y
ADD  R1, R1, #-1
STR  R1, R5, #-1
LDR  R1, R5, #-2  ; incr z
ADD  R1, R1, #1
STR  R1, R5, #-2
DONE  ...  ; next statement
More if examples

• if(0 <= age && age <= 11)
  kids += 1;

• if(month == 4 || month == 6 ||
  month == 9 || month == 11)
  printf(“The month has 30 days.
”);

• if(x = 2) // always true,
  y = 5;

This is a common programming error (= instead of ==), not caught by compiler because it’s syntactically correct.
if's can be nested

```c
if (x == 3)
    if (y != 6)
        {
            z = z + 1;
            w = w + 2;
        }
if ((x == 3) && (y != 6))
    {
        z = z + 1;
        w = w + 2;
    }
```

is the same as...

```c
if ((x == 3) && (y != 6))
    {
        z = z + 1;
        w = w + 2;
    }
```
if (condition)
  action_if;
else
  action_else;

else allows choice between two mutually exclusive actions without re-testing condition.
Matching `else` with `if`

This code

```java
if(x != 10)
  if(y > 3)
    z = z/2;
else
  z = z * 2;
```

is the same as...

```java
if(x != 10)
{   if(y > 3)
    z = z/2;
} else
  z = z * 2;
```

Else is always associated with the **closest** unassociated `if`.

and is NOT the same as...

```java
if (x != 10)
{   if (y > 3)
    z = z/2;
} else
  z = z * 2;
```
if (month == 4 || month == 6 || month == 9 || month == 11)
    printf(“Month has 30 days.\n”);
else if (month == 1 || month == 3 || month == 5 || month == 7 || month == 8 || month == 10 || month == 12)
    printf(“Month has 31 days.\n”);
else if (month == 2)
    printf(“Month has 28 or 29 days.\n”);
else
    printf(“Don’t know that month.\n”);
while (test)  
  loop_body;

Executes loop body as long as test evaluates to TRUE (non-zero).

Note: Test is evaluated before executing loop body.
LC-3 code for while

\begin{align*}
x &= 0; \\
\text{while } (x < 10) \\
\{ & \quad \text{printf}(\"%d\", x); \\
& \quad x = x + 1; \\
\}
\end{align*}

\begin{verbatim}
x = 0; \\
while (x < 10) \\
{   printf("%d ", x); \\
   x = x + 1; \\
}
\end{verbatim}
Infinite loops

The following loop will never terminate:

```c
x = 0;
while (x < 10)
    printf("%d ", x);
```

Loop body does not change condition, so the test never fails.

This is a common programming error that can be difficult to find.
for loops

for (init; test; re-init)
  loop_body

Executes loop body as long as test evaluates to TRUE (non-zero). Initialization and re-initialization code included in loop statement.

Note: Test is evaluated before executing loop body.
**LC-3 code for for**

```c
for(i = 0; i < 10; i++)
    printf("%d ", i);
```

(NOTE: This is the same as the **while** example!)
Examples of \texttt{for} loops

\begin{verbatim}
/* -- what is the output of this loop? -- */
for(i = 0; i <= 10; i ++)
    printf("%d ", i);

/* -- what is the output of this loop? -- */
letter = 'a';
for(c = 0; c < 26; c++)
    printf("%c ", letter+c);

/* -- what does this loop do? -- */
numberOfOnes = 0;
for(bitNum = 0; bitNum < 16; bitNum++)
{   if(inputValue & (1 << bitNum))
    numberOfOnes++;
}
\end{verbatim}
Nested loops

The loop body can (of course) be another loop.

/* print a multiplication table */
for(mp1 = 0; mp1 < 10; mp1++)
{
    for(mp2 = 0; mp2 < 10; mp2++)
    {
        printf("%d\t");
    }
    printf("\n");
}
Another nested loop

The test for the inner loop depends on the counter variable of the outer loop.

```c
for(outer = 1; outer <= input; outer++)
{
    for(inner = 0; inner < outer; inner++)
    {
        printf("%d ", inner);
    }
}
```
for vs. while

In general:

for loop is preferred for counter-based loops.
  - Explicit counter variable
  - Easy to see how counter is modified each loop

while loop is preferred for sentinel-based loops.
  - Test checks for sentinel value.

Either kind of loop can be expressed as the other, so it’s really a matter of style and readability.
do

    loop_body;

while(test);

Executes loop body as long as test evaluates to TRUE (non-zero).

Note: Test is evaluated after executing loop body.
if(month == 4 || month == 6 || month == 9 || month == 11)
    printf("Month has 30 days.\n");
else if(month == 1 || month == 3 || month == 5 || month == 7 || month == 8 || month == 10 ||
    month == 12)
    printf("Month has 31 days.\n");
else if(month == 2)
    printf("Month has 28 or 29 days.\n");
else
    printf("Don’t know that month.\n");
switch (expression) {
    case const1:
        action1; break;
    case const2:
        action2; break;
    default:
        action3;
}

Alternative to long if-else chain.
If break is not used, then case "falls through" to the next action.
/* same as month example for if-else */

switch (month) {
    case 4:
    case 6:
    case 9:
    case 11:
        printf("Month has 30 days.\n");
        break;
    case 1:
    case 3:
        /* some cases omitted for brevity...*/
        printf("Month has 31 days.\n");
        break;
    case 2:
        printf("Month has 28 or 29 days.\n");
        break;
    default:
        printf("Don’t know that month.\n");
}
More about switch

**case** expressions must be constant.

```
case i:  /* illegal if i is a variable */
```

If no **break**, then next case is also executed.

```
switch (a)
{
    case 1:
        printf(“A”);
    case 2:
        printf(“B”);
    default:
        printf(“C”);
}
```

What does it do?
break and continue

**break**

- used *only* in switch statement or iteration statement
- passes control out of the innermost loop or switch statement containing it to the statement immediately following
- usually used to exit a loop before terminating condition occurs (or to exit switch statement when case is done)

**continue**

- used only in iteration statement
- terminates the execution of the loop body *for this iteration*
- loop expression is evaluated to see whether another iteration should be performed
- if *for* loop, also executes the re-initializer
Example

What does the following loop do?

```c
for(i = 0; i <= 20; i++)
{
    if(i%2 == 0)
    
        continue;

    printf("%d ", i);
}
```

What would be an easier way to write this?

What happens if `break` instead of `continue`?
Problem Solving in C

• Stepwise Refinement
  - as covered in Chapter 6
• ...but can stop refining at a higher level of abstraction.

• Same basic constructs
  - Sequential -- C statements
  - Conditional -- if-else, switch
  - Iterative -- while, for, do-while
Problem 1: Calculating Pi

Calculate $\pi$ using its series expansion.
User inputs number of terms.

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots + (-1)^{n-1} \frac{4}{2n+1} + \cdots$$

Start

Evaluate Series

Initialize

Get Input

Output Results

Stop
Pi: 1st refinement

Start
Initialize
Get Input
Evaluate Series
Output Results
Stop

Initialize iteration count

count < terms

Evaluate next term

count = count + 1

for loop
Pi: 2nd refinement

Initialize iteration count

count < terms

count = count + 1

Evaluate next term

if count is odd

subtract term

else

add term

add term
Pi: Code for Evaluate Terms

```c
for (count=0; count < numOfTerms; count++) {
    if (count % 2) {
        /* odd term -- subtract */
        pi -= 4.0 / (2 * count + 1);
    } else {
        /* even term -- add */
        pi += 4.0 / (2 * count + 1);
    }
}
```

Note: Code in text is slightly different, but this code corresponds to equation.
#include <stdio.h>

main() {
    double pi = 0.0;
    int numOfTerms, count;

    printf("Number of terms (must be 1 or larger) : ");
    scanf("%d", &numOfTerms);

    for (count=0; count < numOfTerms; count++) {
        if (count % 2) {
            pi -= 4.0 / (2 * count + 1); /* odd term -- subtract */
        } else {
            pi += 4.0 / (2 * count + 1); /* even term -- add */
        }
    }
    printf("The approximate value of pi is %f\n", pi);
}
Problem 2: Finding Prime Numbers

Print all prime numbers less than 100.

- A number is prime if its only divisors are 1 and itself.
- All non-prime numbers less than 100 will have a divisor between 2 and 10.
Primes: 1st refinement

Start

Initialize

Print primes

Stop

Initialize
num = 2

num < 100

Print num if prime

num = num + 1

F

T
Primes: 2nd refinement

Initialize
num = 2

num < 100

Print num if prime

num = num + 1

Divide num by 2 through 10

no divisors?

Print num

F

T
Primes: 3rd refinement

Divide num by 2 through 10

no divisors?

Print num

Initialize divisor = 2

divisor <= 10

Clear flag if num % divisor > 0

divisor = divisor + 1
Primes: Using a Flag Variable

To keep track of whether number was divisible, we use a "flag" variable.

- Set prime = TRUE, assuming that this number is prime.
- If any divisor divides number evenly, set prime = FALSE.
  • Once it is set to FALSE, it stays FALSE.
- After all divisors are checked, number is prime if the flag variable is still TRUE.

Use macros to help readability.

#define TRUE 1
#define FALSE 0
Primes: Complete Code

#include <stdio.h>
#define TRUE 1
#define FALSE 0

main () {
    int num, divisor, prime;

    /* start with 2 and go up to 100 */
    for (num = 2; num < 100; num ++ ) {

        prime = TRUE; /* assume num is prime */
        /* test whether divisible by 2 through 10 */
        for (divisor = 2; divisor <= 10; divisor++)
            if (((num % divisor) == 0) && (num != divisor))
                prime = FALSE; /* not prime */

        if (prime) /* if prime, print it */
            printf("The number %d is prime\n", num);
    }
}
Problem 3: Searching for Substring

- Have user type in a line of text (ending with linefeed) and print the number of occurrences of "the".

- Reading characters one at a time
  - Use the `getchar()` function -- returns a single character.

- Don't need to store input string; look for substring as characters are being typed.
  - Similar to state machine:
    - based on characters seen, move toward success state or move back to start state.
  - Switch statement is a good match to state machine.
Substring: State machine to flow chart

no match

matched 't'

matched 'th'

matched 'the'

other

match = 0

match = 1

match = 2

read char

if 't', match=1

if 'h', match=2

if 'e', count++ and match = 0

other

other

other

increment count

if 't', match=1

if 't', match=0

if 't', match=0

if 't', match=0

if 'h', match=2

if 't', match=1

if 'e', count++ and match = 0

if 't', match=1

if 't', match=0

if 't', match=0

if 't', match=0

if 'h', match=2

if 't', match=1

if 'e', count++ and match = 0

if 't', match=1

if 't', match=0

if 't', match=0

if 't', match=0
```c
#include <stdio.h>

main() {
    char key;    /* input character from user */
    int match = 0; /* keep track of characters matched */
    int count = 0; /* number of substring matches */

    /* Read character until newline is typed */
    while ((key = getchar()) != '\n') {

        /* Action depends on number of matches so far */
        switch (match) {

            case 0:  /* starting - no matches yet */
                if (key == 't')
                    match = 1;
                break;
```
case 1: /* 't' has been matched */
    if (key == 'h')
        match = 2;
    else if (key == 't')
        match = 1;
    else
        match = 0;
    break;
case 2: /* 'th' has been matched */
    if (key == 'e') {
        count++;    /* increment count */
        match = 0; /* go to starting point */
    }
    else if (key == 't') {
        match = 1;
    } else 
        match = 0;
    break;
}

printf("Number of matches = %d\n", count);