Chapter 13
Control Structures

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

Iteration
- executing code multiple times, ending based on evaluated expression
  - while
  - for
  - do-while

If

```c
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).

Example If Statements

```c
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
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.\n");
- if (x == 2) y = 5;

A common programming error (= instead ==), 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-else

```c
if (condition)
    action_if;
else
    action_else;
```

Else allows choice between two mutually exclusive actions without re-testing condition.

Matching Else with If

Else is always associated with closest unassociated if.

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

is NOT the same as...

```c
if (x != 10) {
    if (y > 3)
        z = z / 2;
    else
        z = z * 2;
    if (x != 10) {
        if (y > 3)
            z = z / 2;
        else
            z = z * 2;
    }
```
### Chaining If’s and Else’s

```c
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

```
while (test)
    loop_body;
```

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

**Note:** Test is evaluated **before** executing loop body.

### Infinite Loops

- The following loop will never terminate:
  ```c
  x = 0;
  while (x < 10)
      printf("%d ", x);
  ```
- Loop body does not change condition, so test never fails.
- This is a common programming error that can be difficult to find.

### For

```
for (init; end-test; re-init)
    statement
```

**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.
Example For Loops

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

/* -- what does this one output? -- */
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++;

Nested Loops

Loop body can (of course) be another loop.

/* print a multiplication table */
for (mpl1 = 0; mpl1 < 10; mpl1++) {
  for (mpl2 = 0; mpl2 < 10; mpl2++) {
    printf("%d\t", mpl1*mpl2);
  }
  printf("\n");
}

Braces aren’t necessary, but they make the code easier to read.

Another Nested Loop

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

for (outer = 1; outer <= input; outer++) {
  for (inner = 0; inner < outer; inner++) {
    sum += 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-While

```c
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.

---

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 π using its series expansion.
  - User inputs number of terms.
  - \( \pi = 4 \left( \frac{4}{3} - \frac{4}{5} + \frac{4}{7} - \cdots + (-1)^{n+1} \frac{4}{2n+1} \right) \)

---

Pi: 1st refinement

- Initialize iteration count
- Get Input
- Evaluate next term
- count = count + 1
- Evaluate next term
- for loop
Pi: 2nd refinement

- Initialize iteration count
- Evaluate next term
- if count is odd subtract term
- else add term
- count = count + 1

Pi: Code for Evaluate Terms

```c
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
}
```

Note: Code in text is slightly different, but this code corresponds to equation.

Pi: Complete Code

```c
#include <stdio.h>

int main() {
    double pi = 0.0;
    int numOfTerms;
    printf("Number of terms (must be 1 or larger): ");
    scanf("%d", &numOfTerms);
    for (int 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("pi is about %f\n", pi);
    return 0;
}
```

Problem 2: Finding Prime Numbers

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

Start

Initialize

Print primes

Stop

Initialize num = 2

num < 100

F

Print num if prime

num = num + 1

Primes: 2nd refinement

Initialize num = 2

num < 100

F

Divide num by 2 through 10

no divisors?

T

Print num

Primes: 3rd refinement

Divide num by 2 through 10

no divisors?

F

Print num

Initialize divisor = 2

divisor <= 10

F

Clear flag if num%d < divisor > 0

divisor = divisor + 1

Primes: Using a Flag Variable

To keep track of whether number was divisible, we use a boolean “flag” variable.

- Set prime = true, assuming that number is prime.
- If a 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 <stdbool.h>, which defines the type bool, and the constants true & false.
Primes: Complete Code

```c
#include <stdio.h>
#include <stdbool.h>

int main() {
    // start with 2 and go up to 100
    for (int num = 2; num < 100; num++) {
        bool prime = true;  // assume prime
        // test whether divisible by 2 through 10
        for (int 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);
    }
    return 0;
}
```

Optimization: Could put a break here to avoid some work. (Section 13.5.2)

Switch

```c
switch (expression) {
    case const1:
        action1; break;
    case const2:
        action2; break;
    default:
        action3;
}
```

Evaluate expression

Switch Example

```c
/* 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:
        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.
  ```c
  case 1: /* illegal if i is a variable */
  ```
- If no break, then next case is also executed.
  ```c
  switch (a) {
      case 1:
          printf("A");
      break;
      case 2:
          printf("B");
      break;
      default:
          printf("C");
  }
  ```
Problem 3: Searching for Substring

- Have user type in a line of text and print the number of occurrences of "the".
- Reading characters one at a time using the `getchar()` function to return 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 back to start state.
  - **Switch statement** is a good match to state machine.

Substring: State machine to flow chart

```plaintext
read char

match = 0
if 't', match=1
if 'h', match=2
if 'e', count++
and match = 0
if 't', match=1
else match=0

T

T

T

T

if Y, match=1
if 't', match=1
if 'h', match=2
if 'e', count++ and match = 0
if 't', match=1
else match=0

other

match = 1
T

T

if Y, match=1
if 't', match=1
if 'h', match=2
if 'e', count++ and match = 0
if 't', match=1
else match=0

other

match = 2
T

T

if Y, match=1
if 't', match=1
if 'h', match=2
if 'e', count++ and match = 0
if 't', match=1
else match=0

other
```

Substring: Code (Part 1)

```c
#include <stdio.h>

int main() {
    char key; /* input character from user */
    int match = 0; /* 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;
```
Substring: Code (Part 3)

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);

Break and Continue

break;
• used only in switch statement or iteration statement
• breaks out of the “smallest” (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 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?
  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?

Looking Ahead: C Pointers

• Pass by value, pass by reference
  float fFloat;
  float *pFloat = &fFloat;
  printf("address: \%p\n", pFloat);
  fFloat = 0.5f;
  printf("value: \%f\n", fFloat);
  *pFloat = 1.0f;
  printf("value: \%f\n", fFloat);
  *(fFloat) = 1.5f;
  printf("value: \%f\n", fFloat);
Looking Ahead: C Functions

- Pass by value, pass by reference

```c
void quadratic(int a, int b, int c,
float *r1, float *r2) {
...
*r1 = (-b + sqrt(b*b + 4*a*c))...
*r2 = (-b - sqrt(b*b + 4*a*c))...
}
```

- Calling

```c
float a, b, c, r1, r2; ...
quadratic(a, b, c, &r1, &r2);
```

Looking Ahead: C Arrays

- Static allocation for string

```c
char string[80];
```

- Dynamic allocation for string

```c
char *string = malloc(80);
strcpy(string, "Hello World");
printf("string: %s
", string);
free(string);
```

Looking Ahead: C Strings

- Functions for manipulating strings:

```c
char *strcpy(char *s1, char *s2);  // copy s2 into s1
int strcmp(char *s1, char *s2);    // compare s2 to s1
char *strcat(char *s1, char *s2);  // append s2 to s1
char *strtok(char *s1, char *delims); // tokenize s1 by delimiters
size_t strlen(char *s1);          // length of s1
```

Looking Ahead: C File I/O

- Read integer (string) from file using streams:

```c
FILE *fp = fopen("data.txt", "r");
if (fp != NULL) {
    fscanf(fp, "%d", &value);
    fclose(fp);
} else ... // error condition
```
Looking Ahead: C File I/O

Write integer (string) to file using streams:
```c
FILE *fp = fopen("data.txt", "w");
if (fp != NULL) {
    fprintf(fp, "%d", value);
    fclose(fp);
} else ... // error condition
```