4.1 Loops

PARTICIPATION ACTIVITY 4.1.1: Looping while the condition is true.

Animation captions:

1. Enter loop body if condition true
2. Enter loop body again if condition still true
3. Exit when loop condition false

Some behaviors should be repeated over and over, like a racecar driving around a track. A ***loop*** is a construct that repeatedly executes specific code as long as some condition is true.

PARTICIPATION ACTIVITY 4.1.2: Loop basics.

Which loop condition achieves the given racetrack driving goal?

1) Loop as long as it is sunny.
   - ○ It is sunny.
   - ○ It is not sunny.

2) Loop as long as it is not raining.
   - ○ It is raining.
   - ○ It is not raining.

3) Loop 3 times.
   - ○ Number of completed laps is 0 or greater.
   - ○ Number of completed laps is less than 3.
   - ○ Number of completed laps equals 3.

4) Loop while the car's fuel tank is at least 20% full.
   - ○ Fuel tank is at 20%.
   - ○

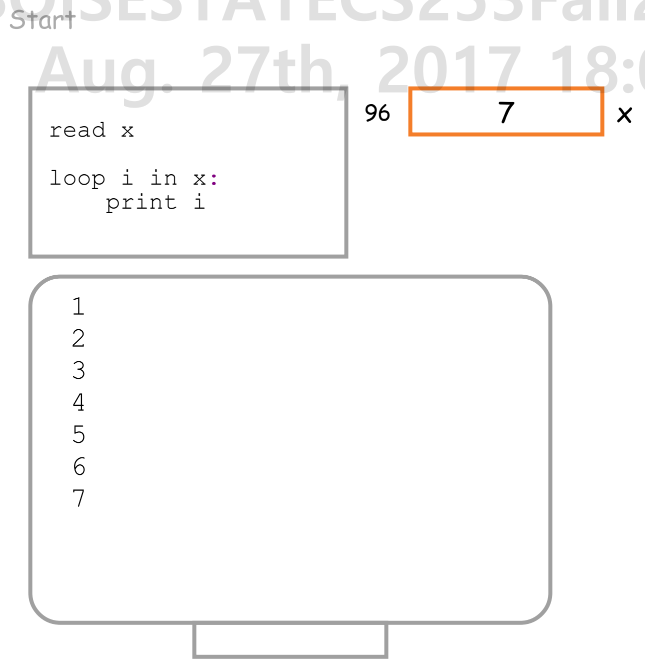
◯ Fuel tank is 20% or more.

◯ Fuel tank is less than 20%.

The above describes a common kind of loop known as a *while* loop.

Below is a loop (in no particular language) that prints a value a specified number of times.

PARTICIPATION ACTIVITY 4.1.3: Loop a given number of times.



```
read x

loop i in x:
   print i
```

```
1
2
3
4
5
6
7
```

4.2 While loops

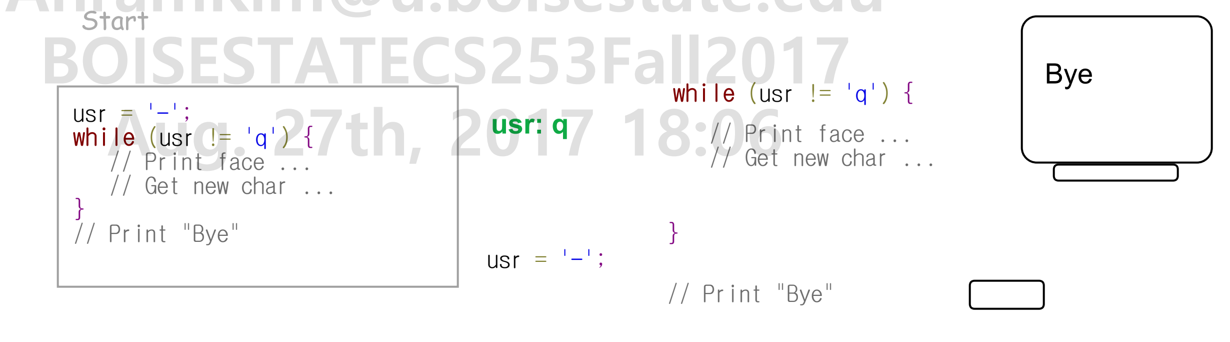
A ***while loop*** is a program construct that executes a list of sub-statements repeatedly as long as the loop's expression evaluates to true.

Construct 4.2.1: While loop statement general form.

```
while (expression) { // Loop expression
   // Loop body: Sub-statements that execute if the
   // expression evaluated to true
}
// Statements that execute after the expression evaluates to false
```

When execution reaches the while loop statement, the expression is evaluated. If true, execution proceeds into the sub-statements inside the braces, known as the ***loop body***. At the loop body's end, execution goes back to the while loop statement start. The expression is again evaluated, and if true, execution again proceeds into the loop body. But if false, execution instead proceeds past the closing brace. Each execution of the loop body is called an ***iteration***, and looping is also called *iterating*.

PARTICIPATION ACTIVITY 4.2.1: While loop.


```
usr = '-';
while (usr != 'q') {
   // Print face ...
   // Get new char ...
}
// Print "Bye"
```

PARTICIPATION ACTIVITY 4.2.2: Basic while loops.

How many times will the loop body execute?

1)
```
x = 3;
while (x >= 1) {
   // Do something
   x = x - 1;
}
```

Check Show answer

2) Assume user would enter 'n', then 'n', then 'y'.
```
// Get userChar from user here
while (userChar != 'n') {
   // Do something
   // Get userChar from user here
}
```

Check Show answer

3) Assume user would enter 'a', then 'b',

then 'n'.

```
// Get userChar from user here
while (userChar != 'n') {
   // Do something
   //Get userChar from user here
}
```

Check Show answer

The following example uses the statement `while (userChar != 'q') { }` to allow a user to end a face-drawing program by entering the character q:

Figure 4.2.1: While loop example: Face-printing program that ends when user enters 'q'.

```
#include <stdio.h>

int main(void) {
   char userChar = '-'; // User-entered char

   while (userChar != 'q') {
      // Print face
      printf("%c %c₩n", userChar, userChar);
      printf(" %c₩n", userChar);
      printf("%c%c%c₩n", userChar, userChar, userChar);

      // Get user character
      printf("₩nEnter a character ('q' to quit): ");
      scanf(" %c", &userChar);
      printf("₩n");
   }

   printf("Goodbye.₩n");

   return 0;
}
```

```
- -
 -
---

Enter a character ('q' to quit): a

a a
 a
aaa

Enter a character ('q' to quit): x

x x
 x
xxx

Enter a character ('q' to quit): q

Goodbye.
```

Notice the space before the %c in the `scanf(" %c", &userChar)` statement above. The space causes scanf to first read and discard any whitespace characters, including spaces (' '), tabs ('\t'), and newline ('\n') characters, in the user input before reading and storing the character indicated by the %c format specifier.

Above, if the user presses key 'a' followed by pressing enter, the system sends two characters to the program: 'a' and then a newline character '\n' (some systems send both '\n' plus '\r'). If the statement was instead `scanf("%c", &userChar)` without the space before %c, the scanf would read the '\n' into usr and try to print the face with '\n' for the user's character.

A common error is to forget to insert a blank space at the beginning of a scanf statement, causing whitespace and newlines to not be skipped, usually resulting in odd program behavior.

Once execution enters the loop body, execution continues to the body's end even if the expression becomes false midway through.

PARTICIPATION ACTIVITY 4.2.3: Loop expressions.

Use a *single operator* in each expression, and the most straightforward translation of the stated goal into an expression.

1) Iterate while x is less-than 100.

```
while (          ) {
    /* Loop body statements go
here */
}
```

Check Show answer

2) Iterate while x is greater than or equal to 0.

```
while (          ) {
    // Loop body
}
```

Check Show answer

3) Iterate while c equals 'g'.

```
while (          ) {
    // Loop body
}
```

Check Show answer

4) Iterate while c is not equal to 'x'.

```
while (          ) {
    // Loop body
}
```

Check Show answer

5) Iterate *until* c equals 'z' (tricky; think

carefully).

```
while (                    ) {
    // Loop body
}
```

Check Show answer

Below is a simple loop example, which separately prints each digit of an integer, showing each iteration.

PARTICIPATION ACTIVITY 4.2.4: While loop step-by-step.

Animation captions:

1. User enters the number 902. The first iteration prints "2".
2. The second iteration prints "0".
3. The third iteration prints "9". num is then 0, so every digit has been printed.

Below is another loop example. The program asks the user to enter a year, and then prints the approximate number of a person's ancestors who were alive for each generation leading back to that year, with the loop computing powers of 2 along the way.

Figure 4.2.2: While loop example: Ancestors printing program.

```
#include <stdio.h>

int main(void) {
   const int YEARS_PER_GEN = 20; // Approx. years per generation
   int userYear = 0;             // User input
   int consYear = 0;             // Year being considered
   int numAnc = 0;               // Approx. ancestors in considered year

   printf("Enter a past year (neg. for B.C.): ");
   scanf("%d", &userYear);

   consYear = 2020;
   numAnc = 2;

   while (consYear >= userYear) {
      printf("Ancestors in %d: %d\n", consYear, numAnc);

      numAnc = 2 * numAnc;                // Each ancestor had two
parents
      consYear = consYear - YEARS_PER_GEN; // Go back 1 generation
   }

   return 0;
}
```

```
Enter a past year (neg. for B.C.):
1900
Ancestors in 2020: 2
Ancestors in 2000: 4
Ancestors in 1980: 8
Ancestors in 1960: 16
Ancestors in 1940: 32
Ancestors in 1920: 64
Ancestors in 1900: 128

...

Enter a past year (neg. for B.C.):
1600
Ancestors in 2020: 2
Ancestors in 2000: 4
Ancestors in 1980: 8
Ancestors in 1960: 16
Ancestors in 1940: 32
Ancestors in 1920: 64
Ancestors in 1900: 128
Ancestors in 1880: 256
Ancestors in 1860: 512
Ancestors in 1840: 1024
Ancestors in 1820: 2048
Ancestors in 1800: 4096
Ancestors in 1780: 8192
Ancestors in 1760: 16384
Ancestors in 1740: 32768
Ancestors in 1720: 65536
Ancestors in 1700: 131072
Ancestors in 1680: 262144
Ancestors in 1660: 524288
Ancestors in 1640: 1048576
Ancestors in 1620: 2097152
Ancestors in 1600: 4194304
```

Each iteration prints a line with the year and the ancestors in that year. (Note: the numbers are large due to not considering breeding among distant relatives, but nevertheless a person has many ancestors).

The program checks for `consYear >= userYear` rather than for `consYear != userYear`, because consYear might be decreased past userYear without equaling it, causing an infinite loop, printing years well past 1950. An ***infinite loop*** is a loop that will always execute (i.e., execute infinitely) because the loop's expression always evaluates to true. A common error is to accidentally create an infinite loop due to assuming equality will be reached. Good practice is to include greater-than or less-than along with equality in a loop expression.

Another common error is to use the assignment operator = rather than the equality operator == in a loop expression, usually causing an unintended infinite loop.

A program with an infinite loop may print excessively, or just seem to stall. On some systems, the user can halt execution by pressing Control-C on the command prompt, or by selecting Stop (or Pause) from within an IDE.

PARTICIPATION ACTIVITY 4.2.5: While loop iterations.

What will the following code output? (For an infinite loop, type "IL")

1)
```
int x = 0;
while (x > 0) {
   printf("%d ", x);
   x = x - 1;
}
printf("Bye");
```

Check Show answer

2)
```
int x = 5;
int y = 18;
while (y >= x) {
   printf("%d ", y);
   y = y - x;
}
```

Check Show answer

3) (Assume the user always enters 'q')
```
int z = 0;
char c = 'y';
while (c = 'y') {
   printf("%d ", z);
   scanf(" %c", &c);
   z = z + 1;
}
```

Check Show answer

4)
```
int x = 10;
while (x != 3) {
   printf("%d ", x);
   x = x / 2;
}
```

Check Show answer

5)
```
int x = 0;
while (x <= 5) {
   printf("%d ", x);
}
```

Check Show answer

PARTICIPATION ACTIVITY

4.2.6: Range of data types.

Computing in loops can easily exceed a variable's range. Execute the ancestors program below with the given input of 1300. What do you observe around year 1400? Recall that an int variable can usually only represent up to about 2 billion. Try changing the declaration of numAnc from type int to long long (and change the printf's %d to %lld), and then see how distant of a year you can enter before observing incorrect output.

Load default template...

```

#include <stdio.h>

int main(void) {
   const int YEARS_PER_GEN = 20; // Approx. years per gener
   int userYear = 0;             // User input
   int consYear = 0;             // Year being considered
   int numAnc = 0;               // Approx. ancestors in co

   printf("Enter a past year (neg. for B.C.): ");
   scanf("%d", &userYear);

   consYear = 2020;
   numAnc = 2;
   while (consYear >= userYear) {
      printf("Ancestors in %d: %d\n", consYear, numAnc);

      numAnc = 2 * numAnc;                  // Each ancestor
      consYear = consYear - YEARS_PER_GEN; // Go back 1 gen
```

```
1300
```

Run

CHALLENGE ACTIVITY

4.2.1: Enter the output for the while loop.

Start

Type the program's output.

```
#include <stdio.h>
int main(void) {
   int g = 0;

   while (g <= 2) {
      printf("%d", g);
      g = g + 1;
   }

   return 0;
}
```

```
012
```

1 2 3 4 5

Check Next

CHALLENGE ACTIVITY 4.2.2: Basic while loop with user input.

Write an expression that executes the loop while the user enters a number greater than or equal to 0.

Note: These activities may test code with different test values. This activity will perform three tests, with userNum initially 9 and user input of 5, 2, -1, then with userNum initially 0 and user input of -17, then with userNum initially -1. See How to Use zyBooks.

Also note: If the submitted code has an infinite loop, the system will stop running the code after a few seconds, and report "Program end never reached." The system doesn't print the test case that caused the reported message.

```c
#include <stdio.h>

int main(void) {
   int userNum = 0;

   userNum = 9;
   while (/* Your solution goes here  */) {
      printf("Body\n");
      scanf("%d", &userNum);
   }
   printf("Done.\n");

   return 0;
}
```

Run

CHALLENGE ACTIVITY 4.2.3: Basic while loop expression.

Write a while loop that prints userNum divided by 2 (integer division) until reaching 1. Follow each number by a space. Example output for userNum = 20:

```
10 5 2 1
```

Note: These activities may test code with different test values. This activity will perform four tests, with userNum = 20, then with userNum = 1, then with userNum = 0, then with userNum = -1. See How to Use zyBooks.

Also note: If the submitted code has an infinite loop, the system will stop running the code after a few seconds, and report "Program end never reached." The system doesn't print the test case that caused the reported message.

```
#include <stdio.h>

int main(void) {
   int userNum = 0;

   userNum = 20;

   /* Your solution goes here  */

   printf("\n");

   return 0;
}
```

Run

4.3 More while examples

The following is an example of using a loop to compute a mathematical quantity. The program computes the greatest common divisor (GCD) among two user-entered integers numA and numB, using Euclid's algorithm: If numA > numB, set numA to numA - numB, else set numB to numB - numA. These steps are repeated until numA equals numB, at which point numA and numB each equal the GCD.

Figure 4.3.1: While loop example: GCD program.

```
#include <stdio.h>

// Output GCD of user-input numA and numB

int main(void) {
   int numA = 0; // User input
   int numB = 0; // User input

   printf("Enter first positive integer: ");
   scanf("%d", &numA);

   printf("Enter second positive integer: ");
   scanf("%d", &numB);

   while (numA != numB) { // Euclid's algorithm
      if (numB > numA) {
         numB = numB - numA;
      }
      else {
         numA = numA - numB;
      }
   }

   printf("GCD is: %d₩n", numA);

   return 0;
}
```

```
Enter first positive integer: 9
Enter second positive integer: 7
GCD is: 1

...

Enter first positive integer: 15
Enter second positive integer: 10
GCD is: 5

...

Enter first positive integer: 99
Enter second positive integer: 33
GCD is: 33

...

Enter first positive integer: 500
Enter second positive integer: 500
GCD is: 500
```

PARTICIPATION ACTIVITY 4.3.1: GCD program.

Refer to the GCD code provided in the previous figure. Assume user input of numA = 15 and numB = 10.

1) For the GCD program, what is the value of numA *before* the first loop iteration?

Check Show answer

2) What is the value of numB *after* the first iteration of the while loop?

Check Show answer

3) What is numB after the second iteration of the while loop?

Check Show answer

4) How many loop iterations will the algorithm execute?

Check Show answer

Figure 4.3.2: While loop example: Conversation program.

```
#include <stdio.h>
#include <string.h>

/* Program that has a conversation with the user. Uses a switch statement
 and a random number (sort of) to mix up the program's responses. */

int main(void) {
   const int TEXT_LIMIT = 1000;     // Size limit for user input
   char userText[TEXT_LIMIT] = ""; // User input
   int rand0_3 = 0;                 // Random number 0 - 3

   printf("Tell me something about yourself. ");
   printf("You can type ₩"Goodbye₩" at anytime to quit.₩n₩n> ");

   fgets(userText, TEXT_LIMIT, stdin);  // Reads a full line of text
   userText[strlen(userText)-1] = '₩0'; // Replaces the newline character

   while (strcmp(userText,"Goodbye") != 0) {
      rand0_3 = strlen(userText) % 4; // "Random" num. %4 ensures 0-3
      switch (rand0_3) {
         case 0:
            printf("₩nPlease explain further.₩n₩n> ");
            break;

         case 1:
            printf("₩nWhy do you say: ₩"%s₩"?₩n₩n> ", userText);
            break;

         case 2:
            printf("₩nI don't think that's right.₩n₩n> ");
            break;

         case 3:
            printf("₩nWhat else can you share?₩n₩n> ");
            break;

         default:
            printf("₩nUh-oh, something went wrong. Try again.₩n₩n");
      }

      fgets(userText, TEXT_LIMIT, stdin);
      userText[strlen(userText)-1] = '₩0';
   }

   printf("₩nIt was nice talking with you. Goodbye.₩n₩n");

   return 0;
}
```

```
Tell me something about yourself. You can type "Goodbye" at anytime to quit.

> I'm 26 years old.

Why do you say: "I'm 26 years old."?

> Well, I was born 26 years ago.

I don't think that's right.

> I am sure it is correct.

Please explain further.

> Goodbye

It was nice talking with you. Goodbye.
```

If you haven't covered fgets, just know that fgets reads an entire line of text. In contrast, scanf would just read a word (stopping when reaching a space).

PARTICIPATION ACTIVITY 4.3.2: Conversation program.

1) What will be printed if the user types "Ouch"?

Check Show answer

2) What will be printed if the user types "Bye"?

Check Show answer

3) Which switch branch will execute if the user types "Goodbye"? Valid answers are branch 0, 1, 2, 3, or none.

Check Show answer

4) How many loop iterations will execute if the user plans to type "I'm hungry", "You are weird", "Goodbye", and "I like you".

Check Show answer

CHALLENGE ACTIVITY 4.3.1: Bidding example.

Write an expression that continues to bid until the user enters 'n'.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   char keepGoing = '-';
   int nextBid = 0;

   srand(5);
   while (/* Your solution goes here  */) {
      nextBid = nextBid + (rand()%10 + 1);
      printf("I'll bid $%d!\n", nextBid);
      printf("Continue bidding? (y/n) ");
      scanf("%c", &keepGoing);
   }
   printf("\n");

   return 0;
}
```

Run

CHALLENGE ACTIVITY 4.3.2: While loop: Insect growth.

Given positive integer numInsects, write a while loop that prints that number doubled without reaching 100. Follow each number with a space. After the loop, print a newline. Ex: If numInsects = 8, print:

```
8 16 32 64
```

```
#include <stdio.h>

int main(void) {
   int numInsects = 0;

   numInsects = 8; // Must be >= 1

   /* Your solution goes here  */

   return 0;
}
```

Run

4.4 Counting

Commonly, a loop should iterate a specific number of times, such as 10 times. A ***loop variable*** counts the number of iterations of a loop. To iterate N times using an integer loop variable i, a while loop[Note_whileloops] with the following form is used:

Construct 4.4.1: Loop variable to iterate N times.

```
// Iterating N times using loop variable i
i = 1;
while (i <= N) {
   // Loop body
   i = i + 1;
}
```

For example, the following program outputs the amount of money in a savings account each year for the user-entered number of years, with $10,000 initial savings and 5% yearly interest:

Figure 4.4.1: While loop that counts iterations: Savings interest program.

```
#include <stdio.h>

int main(void) {
   const int INIT_SAVINGS = 10000;     // Initial savings
   const double INTEREST_RATE = 0.05; // Interest rate
   int userYears = 0;                  // User input of number of years
   int i = 0;                          // Loop variable
   double currSavings = 0.0;           // Savings with interest

   printf("₩nInitial savings of $%d", INIT_SAVINGS);
   printf("₩nat %lf yearly interest.₩n₩n", INTEREST_RATE);

   printf("Enter years: ");
   scanf("%d", &userYears);

   currSavings = INIT_SAVINGS;
   i = 1;
   while(i <= userYears) {
      printf(" Savings in year %i: $%lf₩n", i, currSavings);
      currSavings = currSavings + (currSavings * INTEREST_RATE);

      i = i + 1;
   }

   return 0;
}
```

```
Initial savings of $10000
at 0.050000 yearly interest.

Enter years: 5
 Savings in year 1: $10000.000000
 Savings in year 2: $10500.000000
 Savings in year 3: $11025.000000
 Savings in year 4: $11576.250000
 Savings in year 5: $12155.062500

...

Initial savings of $10000
at 0.050000 yearly interest.

Enter years: 15
 Savings in year 1: $10000.000000
 Savings in year 2: $10500.000000
 Savings in year 3: $11025.000000
 Savings in year 4: $11576.250000
 Savings in year 5: $12155.062500
 Savings in year 6: $12762.815625
 Savings in year 7: $13400.956406
 Savings in year 8: $14071.004227
 Savings in year 9: $14774.554438
 Savings in year 10: $15513.282160
 Savings in year 11: $16288.946268
 Savings in year 12: $17103.393581
 Savings in year 13: $17958.563260
 Savings in year 14: $18856.491423
 Savings in year 15: $19799.315994
```

The statements that cause iteration to occur userYears times are highlighted.

A common error is to forget to include the loop variable update (`i = i + 1`) at the end of the loop, causing an unintended infinite loop.

PARTICIPATION ACTIVITY 4.4.1: Basic while loop parts.

Use <= in each loop expression.

1) Loop iterates 10 times.

```
i = 1;
while (          ) {
   // Loop body
   i = i + 1;
}
```

Check Show answer

2) Loop iterates 2 times.

```
i = 1;
while (          ) {
   // Loop body
   i = i + 1;
}
```

Check Show answer

3) Loop iterates 8 times. NOTE the initial value of i.

```
i = 0;
while (                    ) {
   // Loop body
   i = i + 1;
}
```

Check Show answer

Counting down is also common, such as counting from 5 to 1, as below.

Figure 4.4.2: While loop with variable that counts down.

```
i = 5;
while (i >= 1) {
   // Loop body
   i = i - 1;
}
```

The loop body executes when i is 5, 4, 3, 2, and 1, but does not execute when i reaches 0.

Counting is sometimes done by steps greater than 1, such as a loop that prints even values from 0 to 100 (0, 2, 4, 6, ..., 98, 100), as below.

Figure 4.4.3: Loop variable increased by 2.

```
i = 0;
while (i <= 100) {
   // Loop body
   i = i + 2;
}
```

Note that the loop variable update is `i = i + 2;` rather than `i = i + 1;`

Creating the loop variable initialization, expression, and loop variable update to achieve specific goals is an important skill.

PARTICIPATION ACTIVITY 4.4.2: Loop to print presidential election years.

Modify the program to print the U.S. presidential election years since 1792 to present day, knowing such elections occur every 4 years. Don't forget to use <= rather than == to help avoid an infinite loop.

Load default template... Run

```

#include <stdio.h>

int main(void) {
   int electYear = 0;

   electYear = 1792;
   // FIXME: Put the following in a while loop
   printf("%d\n", electYear);

   return 0;
}

```

PARTICIPATION ACTIVITY 4.4.3: More counting with while loops.

Complete the following.

1) Loop iterates with i being the odd integers from 0 to 9.

```
i = 1;
while (i <= 9)  {
   // Loop body
   i = ____;
}
```

Check Show answer

2) Loop iterates with i being multiples of 5 from 0 to 1000 (inclusive).

```
i = 0;
while (i <= 1000)  {
   // Loop body
   i = ____;
}
```

Check Show answer

3) Loop iterates from 212 to 32 (inclusive).

```
i = 212;
while (i >= 32)  {
   // Loop body
   i = [          ];
}
```

Check Show answer

4) Loop iterates from -100 to 31 (inclusive).

```
i = -100;
while (i [          ]
32)  {
   /* Loop body statements go
here */
   i = i + 1;
}
```

Check Show answer

PARTICIPATION ACTIVITY 4.4.4: Loop simulator.

The following tool allows you to enter values for a loop's parts, and then executes the loop. Using the tool, try to solve each listed problem individually.

1. 0 to 100,000 by 5000s (so 0, 5000, 10000, ...).
2. -19 to 19 by 1s.
3. 10 to -10 by 1s.
4. Multiples of 3 between 0 and 100
5. Powers of 2 from 1 to 256 (so 1, 2, 4, 8, ...).
6. Come up with your own challenges.

```
int i = [     ];
while (i [     ][     ]) {
   printf("%d ", i);
   i = i [     ][     ];
}
```

Run code

Output is: `Awaiting your input...`

PARTICIPATION ACTIVITY

4.4.5: Calculate a factorial.

Write a program that lets a user enter N and that outputs N! (meaning N*(N-1)*(N-2)*...*2*1). Hint: Initialize a variable totalValue to N, and use a loop variable i that counts from N-1 down to 1.

Load default template...

```

#include <stdio.h>

int main(void) {
   int totalVal = 0;
   int userInt = 0;

   // FIXME: Ask user to input an integer, store in userInt

   totalVal = userInt;
   // FIXME: Add while loop that counts down to 1, updating tc

   printf("%d! is %d\n", userInt, totalVal);

   return 0;
}

```

```
5
```

Run

Because i = i + 1 is so common in programs, the programming language provides a shorthand version ***++i***. The ++ is known as the ***increment operator***. A loop can thus be written as follows.

Construct 4.4.2: Loop with increment operator.

```
i = 1;
while (i <= N) {
   // Loop body
   ++i;
}
```

No space is necessary between the ++ and the i. A common error by new programmers is to use i = ++i instead of just ++i. The former works but is strange and unnecessary.

Likewise, the ***decrement operator***, as in --i, is equivalent to i = i - 1.

Sidenote: C++'s name stems from the ++ operator, suggesting C++ is an increment or improvement over its C language predecessor.

The increment/decrement operators can appear in *prefix* form (++i or--i) or *postfix* form (i++ or i--). The distinction is relevant when used in a larger expression, as in x < i++. The prefix form first increments the variable, then uses the incremented value in the expression. The postfix form first uses the current variable value in the expression, and then increments the variable. We do not recommend use of the increment/decrement operators in larger expressions, and thus only use the prefix form, which some say is safer for beginner programmers in case they accidentally type i = ++i, which works as expected, whereas i = i++ does not.

PARTICIPATION ACTIVITY 4.4.6: Increment/decrement operators.

1) What is the final value of i?

```
i = 0;
++i;
++i;
```

Check Show answer

2) Replace the loop variable update statement by using the decrement operator.

```
i = 9;
while (i > 0) {
   // Loop body
   i = i - 1;
}
```

```
i = 9;

while (i > 0) {

   // Loop body

   ;

}
```

Check Show answer

CHALLENGE ACTIVITY 4.4.1: While loop: Print 1 to N.

Write a while loop that prints 1 to userNum, using the variable i. Follow each number (even the last one) by a space. Assume userNum is positive. Ex: userNum = 4 prints:

```
1 2 3 4
```

```
#include <stdio.h>

int main(void) {
   int userNum = 0;
   int i       = 0;

   userNum = 4;      // Assume positive

   /* Your solution goes here  */

   printf("\n");

   return 0;
}
```

Run

CHALLENGE ACTIVITY 4.4.2: Printing output using a counter.

Retype the following and run, note incorrect behavior. Then fix errors in the code, which should print numStars asterisks.

```
while (numPrinted != numStars) {
   printf("*");
}
```

```
#include <stdio.h>

int main(void) {
   int numStars = 0;
   int numPrinted = 0;

   numStars = 12;
   numPrinted = 1;

   /* Your solution goes here  */

   printf("\n");

   return 0;
}
```

Run

(*Note_whileloops) *(To instructors): Focus is placed on mastering basic looping using while loops, before introducing for loops. Also, looping N times is initially done using 1 to <= N rather than 0 to < N due to being more intuitive to new programmers and less prone to error, the latter being commonplace as a consequence of arrays being numbered starting at 0.*

4.5 For loops

Counting in loops is so common that the language supports a loop type for that purpose. A ***for loop*** statement collects three parts—the loop variable initialization, loop expression, and loop variable update—all at the top of the loop, thus enhancing code readability reducing errors like forgetting to update the loop variable.

Construct 4.5.1: For loop.

```
for (initialExpression; conditionExpression; updateExpression) {
  // Loop body: Sub-statements to execute if the
  // conditionExpression evaluates to true
}
// Statements to execute after the expression evaluates to false
```

A while loop and its equivalent for loop are shown below. Clearly, while loops are sufficient, but a for loop is a widely-used programming convenience.

PARTICIPATION ACTIVITY 4.5.1: While/for loop correspondence.

Start

```
i = 0;
while (i <= 99) {
   // Loop body statements
   ++i;
}
```

```
for (i = 0; i <= 99; ++i) {
   // Loop body statements
}
```

i = 0;
while (i <= 99) {

for (i = 0; i <= 99; ++i) {

```
        // Loop body statements                    // Loop body statements
        ++i;                                    }
    }
```

Note that the for loop's third part (++i above) does *not* end with a semicolon.

PARTICIPATION ACTIVITY 4.5.2: For loops.

Complete the for loop to achieve the goal. Use prefix increment (++i) or decrement (--i) where appropriate.

1) Iterate for i from 0 to 9.

```
for (i = 0; i <= 9;
                    ) {
// Loop body
}
```

Check Show answer

2) Iterate for numCars from 1 to 500. Note the variable is numCars (not i).

```
for (
numCars <= 500; ++numCars) {
   // Loop body
}
```

Check Show answer

3) Iterate for i from 99 down to 0. Compare with 0.

```
for (i = 99;
                     --i) {
   // Loop body
}
```

Check Show answer

4) Iterate for i from 0 to 20 by 2s (0, 2, 4, ...). Use i = ??, NOT ++i.

```
for (i = 0; i <= 20;
            ) {
   // Loop body
}
```

Check Show answer

5) Iterate for i from -10 to 10. Compare with 10.

```
for (            ) {
   // Loop body
}
```

Check Show answer

Table 4.5.1: Choosing between while and for loops: General guidelines (not strict rules though).

for	Use when the number of iterations is computable before entering the loop, as when counting down from X to 0, printing a character N times, etc.
while	Use when the number of iterations is not computable before entering the loop, as when iterating until a user enters a particular character.

PARTICIPATION ACTIVITY 4.5.3: While loops and for loops.

Choose the most appropriate loop type.

1) Iterate as long as user-entered char c is not 'q'.

- ○ while
- ○ for

2) Iterate until the values of x and y are equal, where x and y are changed in the loop body.

- ○ while
- ○ for

3)

Iterate 100 times.

- while
- for

Good practice is to use a for loop's parts to count the necessary loop iterations, with nothing added or omitted. The following loop examples should be avoided, if possible.

Figure 4.5.1: Avoid these for loop variations.

```
// initialExpression not related to counting iterations; move r = rand() before loop
for (i = 0, r = rand(); i < 5; ++i) {
   // Loop body
}

// updateExpression not related to counting iterations; move r = r + 2 into loop body
for (i = 0; i < 5; ++i, r = r + 2) {
   // Loop body
}
```

PARTICIPATION ACTIVITY 4.5.4: For loop variations.

1) Each of the above for loop variations yields a syntax error.
- True
- False

2) Even though the above for loop variations may execute correctly, they are generally considered bad style.
- True
- False

A common error is to also have a ++i; statement in the loop body, causing the loop variable to be updated twice per iteration.

Figure 4.5.2: Common error: loop variable updated twice.

```
// Loop variable updated twice per iteration
for (i = 0; i < 5; ++i) {
   // Loop body
   ++i; // Oops
}
```

PARTICIPATION ACTIVITY 4.5.5: For loop double increment.

1) Putting ++i at the end of a for loop body, in addition to in the updateExpression part, yields a syntax error.

- True
- False

CHALLENGE ACTIVITY 4.5.1: Enter the output for the for loop.

Start

Type the program's output.

```
#include <stdio.h>
int main(void) {
   int i = 0;

   for (i = 0; i <= 4; ++i) {
      printf("%d", i);
   }

   return 0;
}
```

```
01234
```

1 2 3 4 5

Check Next

CHALLENGE ACTIVITY 4.5.2: For loop: Print 1 to N.

Write a for loop that prints: 1 2 .. userNum. Print a space after each number, including after the last number. Ex: userNum = 4 prints:

```
1 2 3 4
```

```
#include <stdio.h>

int main(void) {
   int userNum = 0;
   int i = 0;

   userNum = 4;

   /* Your solution goes here  */

   printf("\n");

   return 0;
}
```

Run

CHALLENGE ACTIVITY 4.5.3: For loop: Print N to 0.

Write code that prints: userNum ... 2 1 Blastoff! Your code should contain a for loop. Print a newline after each number and after Blastoff!. Ex: userNum = 3 outputs:

```
3
2
1
Blastoff!
```

```
#include <stdio.h>

int main(void) {
   int userNum = 0;
   int i = 0;

   userNum = 3;

   /* Your solution goes here  */

   return 0;
}
```

Run

4.6 Nested loops

A ***nested loop*** is a loop that appears in the body of another loop. The nested loops are commonly referred to as the ***inner loop*** and ***outer loop***.

Nested loops have various uses. One use is to generate all combinations of some items. For example, the following program generates all two-letter .com Internet domain names.

Figure 4.6.1: Nested loops example: Two-letter domain name printing program.

```
#include <stdio.h>

/* Output all two-letter .com Internet domain names */

int main(void) {
   char usrInput = '?';
   char letter1 = '?';
   char letter2 = '?';

   printf("Enter any key to begin: ");
   scanf("%c", &usrInput); // Unused; just to start the printing

   printf("₩nTwo-letter domain names:₩n");

   letter1 = 'a';
   while (letter1 <= 'z') {
      letter2 = 'a';
      while (letter2 <= 'z') {
         printf("%c%c.com₩n", letter1, letter2);
         ++letter2;
      }
      ++letter1;
   }

   return 0;
}
```

```
Enter any key to begin: a

Two-letter domain names:
aa.com
ab.com
ac.com
ad.com
ae.com
af.com
ag.com
ah.com
ai.com
aj.com
ak.com
al.com
am.com
an.com
ao.com
ap.com
aq.com
ar.com
as.com
at.com
au.com
av.com
aw.com
ax.com
ay.com
az.com
ba.com
bb.com
bc.com
bd.com
be.com

...

zw.com
zx.com
zy.com
zz.com
```

Note that the program makes use of ascending characters being encoded as ascending numbers, e.g., 'a' is 97, 'b' is 98, etc., so assigning 'a' to letter1 and then incrementing yields 'b'.

(Forget about buying a two-letter domain name: They are all taken, and each sells for several hundred thousand or millions of dollars. Source: dnjournal.com, 2012).

PARTICIPATION ACTIVITY 4.6.1: Two character dotcom domain names.

Modify the program to include two-character .com names where the second character can be a letter or a number, as in a2.com. Hint: Add a second loop, following the `while (letter2 <= 'z')` loop, to handle numbers.

Load default template...

Run

```

#include <stdio.h>

/* Output all two-letter .com Internet domain names */
```

```
int main(void) {
   char letter1 = '?';
   char letter2 = '?';

   printf("\nTwo-letter domain names:\n");

   letter1 = 'a';
   while (letter1 <= 'z') {
      letter2 = 'a';
      while (letter2 <= 'z') {
         printf("%c%c.com\n", letter1, letter2);
         ++letter2;
      }
      ++letter1;
   }
```

Below is a nested loop example that graphically depicts an integer's magnitude by using asterisks, creating a "histogram." The inner loop is a for loop that handles the printing of the asterisks. The outer loop is a while loop that handles executing until a negative number is entered.

Figure 4.6.2: Nested loop example: Histogram.

```
#include <stdio.h>

int main(void) {
   int numAsterisk = 0;  // Number of asterisks to print
   int i = 0;            // Loop counter

   numAsterisk = 0;
   while (numAsterisk >= 0) {
      printf("Enter an integer (negative to quit): ");
      scanf("%d", &numAsterisk);

      if (numAsterisk >= 0) {
         printf("Depicted graphically:₩n");
         for (i = 1; i <= numAsterisk; ++i) {
            printf("*");
         }
         printf("₩n₩n");
      }
   }

   printf("Goodbye.₩n");
   return 0;
}
```

```
nter an integer (negative to quit): 9
Depicted graphically:
*********

Enter an integer (negative to quit): 23
Depicted graphically:
***********************

Enter an integer (negative to quit): 25
Depicted graphically:
*************************

Enter an integer (negative to quit): -1
Goodbye.
```

PARTICIPATION ACTIVITY 4.6.2: Nested loops: Inner loop execution.

1) Given the following code, how many times will the inner loop body execute?

```
int row = 0;
int col = 0;
for(row = 0; row < 2; row = row + 1) {
   for(col = 0; col < 3; col = col + 1) {
      // Inner loop body
   }
}
```

Check Show answer

2) Given the following code, how many times will the inner loop body execute?

```
char letter1 = '?';
char letter2 = '?';

letter1 = 'a';
while (letter1 <= 'f') {
   letter2 = 'c';
   while (letter2 <= 'f') {
      // Inner loop body
      ++letter2;
   }
   ++letter1;
}
```

Check Show answer

PARTICIPATION ACTIVITY 4.6.3: Nested loops: What is the output.

1) What is output by the following code?

```
int row = 0;
int col = 0;
for(row = 2; row <= 3; row = row + 1) {
   for(col = 0; col <= 1; col = col + 1) {
      printf("%d%d ", row, col);
   }
}
```

Check Show answer

2) What is output by the following code?

```
char letter1 = '?';
char letter2 = '?';

letter1 = 'y';
while (letter1 <= 'z') {
   letter2 = 'a';
   while (letter2 <= 'c') {
      printf("%c%c ", letter1, letter2);
      ++letter2;
   }
   ++letter1;
}
```

Check Show answer

CHALLENGE ACTIVITY 4.6.1: Nested loops: Indent text.

Print numbers 0, 1, 2, ..., userNum as shown, with each number indented by that number of spaces. For each printed line, print the leading spaces, then the number, and then a newline. Hint: Use i and j as loop variables (initialize i and j explicitly). Note: Avoid any other spaces like spaces after the printed number. Ex: userNum = 3 prints:

```
0
 1
  2
   3
```

```
#include <stdio.h>

int main(void) {
   int userNum  = 0;
   int i = 0;
   int j = 0;

   /* Your solution goes here  */

   return 0;
}
```

Run

CHALLENGE ACTIVITY 4.6.2: Nested loops: Print seats.

Given numRows and numCols, print a list of all seats in a theater. Rows are numbered, columns lettered, as in 1A or 3E. Print a space after each seat, including after the last. Ex: numRows = 2 and numCols = 3 prints:

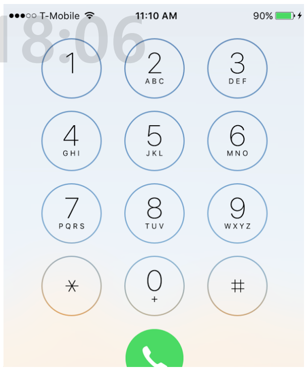
```
1A 1B 1C 2A 2B 2C 
```

```c
#include <stdio.h>

int main(void) {
   int numRows = 2;
   int numCols = 3;

   // Note: You'll need to declare more variables

   /* Your solution goes here  */

   printf("\n");

   return 0;
}
```

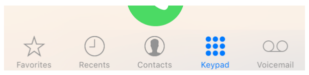
Run

4.7 Developing programs incrementally

Creating correct programs can be hard. Following a good programming process helps. What many new programmers do, but shouldn't, is write the entire program, compile it, and run it—hoping it works. Debugging such a program can be difficult because there may be many distinct bugs.



Experienced programmers develop programs ***incrementally***, meaning they create a simple program version, and then growing the program little-by-little into successively more-complete versions.



The following program allows the user to enter a phone number that includes letters. Such letters appear on phone keypads along with numbers, enabling phone numbers like 1-555-HOLIDAY. The program converts a phone number having numbers/letters into one having numbers only.

The first program version simply prints each string element, to ensure the loop iterates properly.

Figure 4.7.1: Incremental program development.

```
#include <stdio.h>

int main(void) {
   char phoneChar = '_';   // char in phone number string

   printf("₩nEnter phone number: ");

   scanf("%c", &phoneChar);  // Reads first char of user input
   printf("Phone number is: ");

   while (phoneChar != '₩n') {
      printf("%c", phoneChar);  // Print element as is
      scanf("%c", &phoneChar);  // Read next char of user input
   }

   printf("₩n");

   return 0;
}
```

```
Enter phone number: 1-555-HOLIDAY
Phone number is: 1-555-HOLIDAY
```

The second program version outputs any number elements, outputing '?' for non-number elements. A ***FIXME comment*** is commonly used to indicate program parts to be fixed or added, as above. Some editor tools automatically highlight the FIXME comment to attract the programmer's attention.

Figure 4.7.2: Second version echoes numbers, and has FIXME comment.

```
Enter phone number: 1-555-HOLIDAY
Numbers only: 1?555????????
```

```
#include <stdio.h>

int main(void) {
   char    phoneChar = '_';    // Current char in phone number string

   printf("₩nEnter phone number: ");

   scanf("%c", &phoneChar);  // Reads first char of user input
   printf("Numbers only: ");

   while (phoneChar != '₩n') {

      if ((phoneChar >= '0') && (phoneChar <= '9')) {
         printf("%c", phoneChar); // Print element as is
      }
      // FIXME: Add else-if branches for letters and hyphen
      else {
         printf("?");
      }

      scanf("%c", &phoneChar);  // Read next char of user input
   }

   printf("₩n");

   return 0;
}
```

The third version completes the else-if branch for the letters A-C (lowercase and uppercase), per a standard phone keypad. The program also modifies the if branch to echo a hyphen in addition to numbers.

Figure 4.7.3: Third version echoes hyphens too, and handles first three letters.

```
Enter phone number: 1-555-HOLIDAY
Numbers only: 1-555-??????2?
```

```
#include <stdio.h>

int main(void) {
   char phoneChar = '_';    // Current char in phone number string

   printf("\nEnter phone number: ");

   scanf("%c", &phoneChar);  // Reads first char of user input
   printf("Numbers only: ");

   while (phoneChar != '\n') {

      if (((phoneChar >= '0') && (phoneChar <= '9')) || (phoneChar == '-')) {
         printf("%c", phoneChar); // Print element as is
      }
      else if ( ((phoneChar >= 'a') && (phoneChar <= 'c')) ||
                ((phoneChar >= 'A') && (phoneChar <= 'C')) ) {
         printf("2");
      }
      // FIXME: Add remaining else-if branches
      else {
         printf("?");
      }

      scanf("%c", &phoneChar);  // Read next char of user input
   }

   printf("\n");

   return 0;
}
```

The fourth version can be created by filling in the if-else branches similarly for other letters. We added more instructions too. Code is not shown below, but sample input/output is provided.

Figure 4.7.4: Fourth and final version sample input/output.

```
Enter phone number (letters/- OK, no spaces): 1-555-HOLIDAY
Numbers only: 1-555-4654329

...

Enter phone number (letters/- OK, no spaces): 1-555-holiday
Numbers only: 1-555-4654329

...

Enter phone number (letters/- OK, no spaces): 999-9999
Numbers only: 999-9999

...

Enter phone number (letters/- OK, no spaces): 9876zywx%$#@
Numbers only: 98769999????
```

PARTICIPATION ACTIVITY 4.7.1: Incremental programming.

Complete the program by providing the additional if-else branches for decoding other letters in a phone number. Try incrementally writing the program by adding one "else if" branch at a time, testing that each added branch works as intended.

Load default template...

```

#include <stdio.h>

int main(void) {
   char   phoneChar = '_';   // Current char in phone number

   printf("\nEnter phone number: \n");

   scanf("%c", &phoneChar);  // Reads first char of user inp
   printf("Numbers only: ");

   while (phoneChar != '\n') {

      if (((phoneChar >= '0') && (phoneChar <= '9')) || (pho
         printf("%c", phoneChar); // Print element as is
      }
      else if ( ((phoneChar >= 'a') && (phoneChar <= 'c')) |
                ((phoneChar >= 'A') && (phoneChar <= 'C')) )
         printf("2");
      }
```

1-800-555-HOLIDAY

Run

PARTICIPATION ACTIVITY 4.7.2: Incremental programming.

1) A good programming process is to write the entire program, then incrementally remove bugs one at a time.

- True
- False

2) Expert programmers need not develop programs incrementally.

- True
- False

3) Incremental programming may help reduce the number of errors in a program.

- True
- False

4) FIXME comments provide a way for a programmer to remember what needs to be added.

- ○ True
- ○ False

5) Once a program is complete, one would expect to see several FIXME comments.

- ○ True
- ○ False

4.8 Break and continue

A ***break statement*** in a loop causes an immediate exit of the loop. A break statement can sometimes yield a loop that is easier to understand.

Figure 4.8.1: Break statement: Meal finder program.

```
#include <stdio.h>

int main(void) {
   const int EMPANADA_COST = 3;
   const int TACO_COST     = 4;

   int userMoney    = 0;
   int numTacos     = 0;
   int numEmpanadas = 0;
   int mealCost     = 0;
   int maxEmpanadas = 0;
   int maxTacos     = 0;

   printf("Enter money for meal: ");
   scanf("%d", &userMoney);

   maxEmpanadas = userMoney / EMPANADA_COST;
   maxTacos     = userMoney / TACO_COST;

   for (numTacos = 0; numTacos <= maxTacos; ++numTacos) {
      for (numEmpanadas = 0; numEmpanadas <= maxEmpanadas; ++numEmpanadas) {

         mealCost = (numEmpanadas * EMPANADA_COST) + (numTacos * TACO_COST);

         // Find first meal option that exactly matches user money
         if (mealCost == userMoney) {
            break;
         }
      }

      // Find first meal option that exactly matches user money
      if (mealCost == userMoney) {
         break;
      }
   }

   if (mealCost == userMoney) {
      printf("$%d buys %d empanadas and %d tacos without change.₩n",
             mealCost, numEmpanadas, numTacos);
   }
   else {
      printf("You cannot buy a meal without having change left over.₩n");
   }

   return 0;
}
```

```
Enter money for meal: 20
$20 buys 4 empanadas and 2 tacos without change.

...

Enter money for meal: 31
$31 buys 9 empanadas and 1 tacos without change.
```

The nested for loops generate all possible meal options for the number of empanadas and tacos that can be purchased. The inner loop body calculates the cost of the current meal option. If equal to the user's money, the search is over, so the break statement immediately exits the inner loop. The outer loop body also checks if equal, and if so that break statement exits the outer loop.

The program could be written without break statements, but the loops' condition expressions would be more complex and the program would require additional code, perhaps being harder to understand.

PARTICIPATION ACTIVITY 4.8.1: Break statements.

Given the following while loop, what is the value assigned to variable z for the given values of variables a, b and c?

```
mult = 0;
while (a < 10) {
   mult = b * a;
   if (mult > c) {
      break;
   }
   a = a + 1;
}
z = a;
```

1) a = 1, b = 1, c = 0

Check Show answer

2) a = 4, b = 5, c = 20

Check Show answer

A ***continue statement*** in a loop causes an immediate jump to the loop condition check. A continue statement can sometimes improve the readability of a loop. The example below extends the previous meal finder program to find meal options for which the total number of items purchased is evenly divisible by the number of diners. The program also outputs all possible meal options, instead of just reporting the first meal option found.

Figure 4.8.2: Continue statement: Meal finder program that ensures items purchased is evenly divisible by the number of diners.

```c
#include <stdio.h>

int main(void) {
   const int EMPANADA_COST = 3;
   const int TACO_COST     = 4;

   int userMoney    = 0;
   int numTacos     = 0;
   int numEmpanadas = 0;
   int mealCost     = 0;
   int maxEmpanadas = 0;
   int maxTacos     = 0;
   int numOptions   = 0;
   int numDiners    = 0;

   printf("Enter money for meal: ");
   scanf("%d", &userMoney);

   printf("How many people are eating: ");
   scanf("%d", &numDiners);

   maxEmpanadas = userMoney / EMPANADA_COST;
   maxTacos     = userMoney / TACO_COST;

   numOptions = 0;
   for (numTacos = 0; numTacos <= maxTacos; ++numTacos) {
      for (numEmpanadas = 0; numEmpanadas <= maxEmpanadas; ++numEmpanadas) {

         // Total items purchased must be equally divisible by number of diners
         if ( ((numTacos + numEmpanadas) % numDiners) != 0) {
            continue;
         }

         mealCost = (numEmpanadas * EMPANADA_COST) + (numTacos * TACO_COST);

         if (mealCost == userMoney) {
            printf("$%d buys %d empanadas and %d tacos without change.\n",
                   mealCost, numEmpanadas, numTacos);
            numOptions += 1;
         }
      }
   }

   if (numOptions == 0) {
      printf("You cannot buy a meal without having change left over.\n");
   }

   return 0;
}
```

```
Enter money for meal: 60
How many people are eating: 3
$60 buys 12 empanadas and 6 tacos without change.
$60 buys 0 empanadas and 15 tacos without change.

...

Enter money for meal: 54
How many people are eating: 2
$54 buys 18 empanadas and 0 tacos without change.
$54 buys 10 empanadas and 6 tacos without change.
$54 buys 2 empanadas and 12 tacos without change.
```

The nested loops generate all possible combinations of tacos and empanadas. If the total number of tacos and empanadas is not exactly divisible by the number of diners (e.g., `(( numTacos + numEmpanadas) % numDiners ) != 0 )`, the continue statement proceeds to the next iteration, thus causing incrementing of numEmpanadas and checking of the loop condition.

Break and continue statements can avoid excessive indenting/nesting within a loop. But they could be easily overlooked, and should be used sparingly, when their use is clear to the reader.

PARTICIPATION ACTIVITY 4.8.2: Continue.

Given:

```
for (i = 0; i < 5; ++i) {
   if (i < 10) {
      continue;
   }
   <Print i>
}
```

1) The loop will print at least some output.

- ○ True
- ○ False

2) The loop will iterate only once.

- ○ True
- ○ False

CHALLENGE ACTIVITY 4.8.1: Simon says.

"Simon Says" is a memory game where "Simon" outputs a sequence of 10 characters (R, G, B, Y) and the user must repeat the sequence. Create a for loop that compares the two strings starting from index 0. For each match, add one point to userScore. Upon a mismatch, exit the loop using a break statement. Ex: The following patterns yield a userScore of 4:

```
simonPattern: RRGBRYYBGY
userPattern:  RRGBBRYBGY
```

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char simonPattern[50] = "";
   char userPattern[50] = "";
   int userScore = 0;
   int i = 0;
   userScore = 0;
   strcpy(simonPattern, "RRGBRYYBGY");
   strcpy(userPattern, "RRGBBRYBGY");

   /* Your solution goes here  */

   printf("userScore: %d\n", userScore);

   return 0;
}
```

Run

4.9 Enumerations

Some variables only need store a small set of named values. For example, a variable representing a traffic light need only store values named GREEN, YELLOW, or RED. An ***enumeration type*** declares a name for a new type and possible values for that type.

Construct 4.9.1: Enumeration type.

```
enum identifier {enumerator1, enumerator2, ...};
```

The items within the braces ("enumerators") are integer constants automatically assigned an integer value, with the first item being 0, the second 1, and so on. An enumeration declares a new data type that can be used like the built-in types int, char, etc.

Figure 4.9.1: Enumeration example.

```
User commands: n (next), r (red), q (quit).

Red light  n
Green light  n
Yellow light  n
Red light  n
Green light  r
Red light  n
Green light  n
Yellow light  n
Red light  q
Quit program.
```

```
#include <stdio.h>

/* Manual controller for traffic light */
int main(void) {

   enum LightState {LS_RED, LS_GREEN, LS_YELLOW, LS_DONE};
   enum LightState lightVal = LS_RED;
   char userCmd = '-';

   printf("User commands: n (next), r (red), q (quit).\n\n");

   lightVal = LS_RED;
   while (lightVal != LS_DONE) {

      if (lightVal == LS_GREEN) {
         printf("Green light  ");
         scanf(" %c", &userCmd);
         if (userCmd == 'n') { // Next
            lightVal = LS_YELLOW;
         }
      }
      else if (lightVal == LS_YELLOW) {
         printf("Yellow light  ");
         scanf(" %c", &userCmd);
         if (userCmd == 'n') { // Next
            lightVal = LS_RED;
         }
      }
      else if (lightVal == LS_RED) {
         printf("Red light  ");
         scanf(" %c", &userCmd);
         if (userCmd == 'n') { // Next
            lightVal = LS_GREEN;
         }
      }

      if (userCmd == 'r') { // Force immediate red
         lightVal = LS_RED;
      }
      else if (userCmd == 'q') { // Quit
         lightVal = LS_DONE;
      }
   }

   printf("Quit program.\n");
   return 0;
}
```

The program declares a new enumeration type named LightState. The program then declares a new variable lightVal of that type. The loop updates lightVal based on the user's input.

The example illustrates the idea of a ***state machine*** that is sometimes used in programs, especially programs that interact with physical objects, wherein the program moves among particular situations ("states") depending on input; see Wikipedia: State machine.

Because different enumerated types might use some of the same names, e.g., `enum Colors {RED, PURPLE, BLUE, GREEN};` might also appear in the same program, the program above follows the practice of prepending a distinguishing prefix, in this case "LS" (for Light State).

One might ask why the light variable wasn't simply declared as a string, and then compared with strings "GREEN", "RED", and "YELLOW". Enumerations are safer. If using a string, an assignment like `light = "ORANGE"` would not yield a compiler error, even though ORANGE is not a valid light color. Likewise, `light == "YELOW"` would not yield a compiler error, even though YELLOW is misspelled.

One could instead declare constant strings like `const string LS_GREEN = "GREEN";` or even integer values like `const int LS_GREEN = 0;` and then use those constants in the code, but an enumeration is clearer, requires less code, and is less prone to error.

Note: Each enumerator by default is assigned an integer value of 0, 1, 2, etc. However, a programmer can assign a specific value to any enumerator. Ex:
`enum TvChannels {TC_CBS = 2, TC_NBC = 5, TC_ABC = 7};`

PARTICIPATION ACTIVITY 4.9.1: Enumerations.

1) Declare a new enumeration type named HvacStatus with three named values HVAC_OFF, AC_ON, FURNACE_ON, in that order.

Check Show answer

2) Declare a variable of the enumeration type HvacStatus named systemStatus.

Check Show answer

3) Assign AC_ON to the variable systemStatus.

Check Show answer

4) What is the integer value of systemStatus after the following?
`systemStatus = FURNACE_ON;`

Check Show answer

5) Given `enum TvChannels {TC_CBS = 2, TC_NBC = 5, TC_ABC = 7};`, what does `printf("%d", TC_ABC);` output?

Check Show answer

CHALLENGE ACTIVITY 4.9.1: Enumerations: Grocery items.

Print either "Fruit", "Drink", or "Unknown" (followed by a newline) depending on the value of userItem. Print "Unknown" (followed by a newline) if the value of userItem does not match any of the defined options. For example, if userItem is GR_APPLES, output should be:

```
Fruit
```

```
#include <stdio.h>

int main(void) {
   enum GroceryItem {GR_APPLES, GR_BANANAS, GR_JUICE, GR_WATER};

   enum GroceryItem userItem = GR_APPLES;

   /* Your solution goes here  */

   return 0;
}
```

Run

CHALLENGE ACTIVITY 4.9.2: Soda machine with enums.

Complete the code provided to add the appropriate amount to totalDeposit.

```
#include <stdio.h>

int main(void) {
   enum AcceptedCoins {ADD_QUARTER, ADD_DIME, ADD_NICKEL, ADD_UNKNOWN};
   enum AcceptedCoins amountDeposited = ADD_UNKNOWN;

   int totalDeposit = 0;
   int usrInput = 0;

   printf("Add coin: 0 (add 25), 1 (add 10), 2 (add 5).  ");
   scanf("%d", &usrInput);

   if (usrInput == ADD_QUARTER) {
      totalDeposit = totalDeposit + 25;
   }

   /* Your solution goes here  */

   else {
      printf("Invalid coin selection.\n");
   }
```

Run

4.10 C example: Salary calculation with loops

PARTICIPATION ACTIVITY 4.10.1: Calculate adjusted salary and tax with deductions: Using loops.

A program may execute the same computations repeatedly.

The program below repeatedly asks the user to enter an annual salary, stopping when the user enters 0 or less. For each annual salary, the program determines the tax rate and computes the tax to pay.

1. Run the program below with annual salaries of 40000, 90000, and then 0.
2. Modify the program to use a while loop inside the given while loop. The new inner loop should repeatedly ask the user to enter a salary deduction, stopping when the user enters a 0 or less. The deductions are summed and then subtracted from the annual income, giving an adjusted gross income. The tax rate is then calculated from the adjusted gross income.
3. Run the program with the following input: 40000, 7000, 2000, 0, and 0. Note that the 7000 and 2000 are deductions.

```
#include <stdio.h>

int main(void) {
   const char SALARY_PROMPT[] = "\nEnter annual salary (0 to exit): ";
   int    annualSalary        = 0;
   int    deduction           = 0;
   int    totalDeductions     = 0;
   double taxRate             = 0.0;
   int    taxToPay            = 0;

   printf("%s", SALARY_PROMPT);
   scanf("%d", &annualSalary);

   while (annualSalary > 0) {
      // FIXME: Add a while loop to gather deductions. Use the variables
      // deduction and totalDeduction for deduction handling.
      // End the inner while loop when a deduction <= 0 is entered.

      // Determine the tax rate from the annual salary
      if (annualSalary <= 20000) {
         taxRate = 0.10;          // 0.10 is 10% written as a decimal
```

```
40000
90000
0
```

Run

A solution to the above problem follows. The input consists of three sets of annual salaries and deductions.

PARTICIPATION ACTIVITY 4.10.2: Calculate adjusted salary and tax with deductions: Using loops (solution).

```
#include <stdio.h>

int main(void) {
   const char SALARY_PROMPT[] = "\nEnter annual salary (0 to exit): ";
   const char PROMPT_DEDUCTION[] = "Enter a deduction (0 to end deductions): ";
   int    annualSalary        = 0;
   int    oneDeduction        = 0;
   int    adjustedSalary      = 0;
   int    totalDeductions     = 0;
   double taxRate             = 0.0;
   int    taxToPay            = 0;

   printf("%s\n", SALARY_PROMPT);
   scanf("%d", &annualSalary);

   while (annualSalary > 0) {
      totalDeductions = 0;     // Start with 0 for each annual salary
      printf("%s\n", PROMPT_DEDUCTION);
      scanf("%d", &oneDeduction);

      while (oneDeduction > 0) {
```

```
40000 3000 6000 0
90000 5000 0
60000 2000 1000 1450 0
```

Run

PARTICIPATION ACTIVITY 4.10.3: Create an annual income and tax table.

A tax table shows three columns: an annual salary, the tax rate, and the tax amount to pay. The program below shows most of the code needed to calculate a tax table.

1. Run the program below and note the results.
2. Alter the program to use a for loop to print a tax table of annual income, tax rate, and tax to pay. Use starting and ending annual salaries of 40000 and 60000, respectively, and a salary increment of 5000.
3. Run the program again and note the results. You should have five rows in the tax table.
4. Alter the program to add user prompts and read the starting and ending annual incomes from user input.
5. Run the program again using 40000 and 60000, respectively, and the same salary increment of 5000. You should have the same results as before.
6. Alter the program to ask the user for the increment to use in addition to the starting and ending annual salaries.
7. Run the program again using an increment of 2500. Are the entries for 40000, 45000, 50000, 55000 and 60000 the same as before?

```c
#include <stdio.h>

int main(void) {
   const int INCOME_INCREMENT  = 5000;
   int    annualSalary         = 0;
   double taxRate              = 0.0;
   int    taxToPay             = 0;
   int    startingAnnualSalary = 0;  // FIXME: Change the starting salary to 40000
   int    endingAnnualSalary   = 0;  // FIXME: Change the ending salary to 60000

   // FIXME: Use a for loop to calculate the tax for each entry in the table.
   // Hint: the initialization clause is annualSalary = startingAnnualSalary

      // Determine the tax rate from the annual salary
      if (annualSalary <= 0) {
         taxRate = 0.0;
      }
      else if (annualSalary <= 20000) {
         taxRate = 0.10;   // 0.10 is 10% written as a decimal
      }
```

```
40000 60000 5000
```

Run

A solution to the above problem follows.

PARTICIPATION ACTIVITY 4.10.4: Create an annual income and tax table (solution).

```
#include <stdio.h>

int main(void) {
   int    annualSalary         = 0;
   double taxRate              = 0.0;
   int    taxToPay             = 0;
   int    startingAnnualSalary = 0;
   int    endingAnnualSalary   = 0;
   int    incomeIncrement      = 0;

   printf("Enter first annual salary for the table: \n");
   scanf("%d", &startingAnnualSalary);
   printf("Enter last annual salary for the table: \n");
   scanf("%d", &endingAnnualSalary);
   printf("Enter the increment for the table: \n");
   scanf("%d", &incomeIncrement);

   for (annualSalary = startingAnnualSalary; annualSalary <= endingAnnualSalary;
        annualSalary += incomeIncrement) {

      // Determine the tax rate from the annual salary
```

```
40000 60000 2500
```

Run

4.11 C example: Domain name validation with loops

PARTICIPATION ACTIVITY 4.11.1: Validate domain names.

A ***top-level domain*** (TLD) name is the last part of an Internet domain name like .com in example.com. A ***core generic top-level domain*** (core gTLD) is a TLD that is either .com, .net, .org, or .info. A ***second-level domain*** is a single name that precedes a TLD as in apple in apple.com

The following program uses a loop to repeatedly prompt for a domain name, and indicates whether that domain name consists of a second-level domain followed by a core gTLD. An example of a valid domain name for this program is apple.com. An invalid domain name for this program is support.apple.com because the name contains two periods. The program ends when the user presses just the Enter key in response to a prompt.

1. Run the program and enter domain names to validate. Note that even valid input is flagged as invalid.
2. Change the program to validate a domain name. A valid domain name for this program has a second-level domain followed by a core gTLD. Run the program again.

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

int main(void) {
   char inputName[50]  = "";
   char searchName[50] = "";
   char coreGtld1[50]  = ".com";
   char coreGtld2[50]  = ".net";
   char coreGtld3[50]  = ".org";
   char coreGtld4[50]  = ".info";
   char theTld[50]     = "";
   bool isCoreGtld     = false;
   // FIXME: Add variable periodCounter to count periods in a domain name
   int periodPosition = 0; // Position of the period in the domain name

   int j = 0;

   printf("\nEnter the next domain name (<Enter> to exit): \n");
   strcpy(inputName, "");
```

```
apple.com
APPLE.COM
apple.comm
```

Run

A solution for the above problem follows.

PARTICIPATION ACTIVITY 4.11.2: Validate domain names (solution).

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

int main(void) {
   char inputName[50]  = "";
   char searchName[50] = "";
   char coreGtld1[50]  = ".com";
   char coreGtld2[50]  = ".net";
   char coreGtld3[50]  = ".org";
   char coreGtld4[50]  = ".info";
   char theTld[50]     = "";
   bool isCoreGtld     = false;
   int periodCounter   = 0;
   int periodPosition  = 0;

   int j = 0;
   int i = 0;

   printf("\nEnter the next domain name (<Enter> to exit): \n");
```

```
apple.com
APPLE.COM
apple.comm
```

Run