Repetition in Programs

Just as the ability to make decisions (if-else selection statements) is an important programming tool, so too is the ability to specify the repetition of a group of operations.

When solving a general problem, it is sometimes helpful to write a solution to a specific case. Once this is done, ask yourself:

- Were there any steps that I repeated? If so, which ones?
- Do I know how many times I will have to repeat the steps?
- If not, how did I know how long to keep repeating the steps?

Counting Loops

A counter-controlled loop (or counting loop) is a loop whose repetition is managed by a loop control variable whose value represents a count. Also called a while loop.

```
1 Set counter to an initial value of 0;
2 while counter < someFinalValue do
3     Block of program code;
4     Increase counter by 1;
5 end
```

Algorithm 1: Counter-Controlled Loop

The C While Loop

This while loop computes and displays the gross pay for seven employees. The loop body is a compound statement (between brackets) The loop repetition condition controls the while loop.

```
1 int count_emp = 0; // Set counter to 0
2 while (count_emp < 7) { //If count_emp < 7, do stuff
3     printf("Hours > ");
4     scanf("%d", &hours);
5     printf("Rate > ");
6     scanf("%lf", &rate);
7     pay = hours * rate;
8     printf("Pay is $ %6.2f\n", pay);
9     count_emp = count_emp + 1; // Increment count_emp */
10 }
11 printf("\nAll employees processed\n");
```
Common Programming Errors

- Skipping crucial steps could lead to an infinite loop
- Common error: forgetting to increment your loop control variable
- Syntax error: misplaced semicolons

```c
int count = 1;
while (count <= 10) {
    printf("Count = %d\n", count);
    count = count + 1;
}
```

General While Loops

Best to generalize code whenever possible.

```c
int numEmployees = 7,
    count_emp = 0;
printf("How many employees > ");
scanf("%d", & numEmployees);
while (count_emp < numEmployees) {
    ...
    count_emp = count_emp + 1;
}
```

Using `numEmployees` instead of the constant 7 allows our code to be more general.

While Loop Exercise

**Exercise**

Write a while loop to compute the sum of natural numbers 1 to 100:

\[
\sum_{i=1}^{100} i = 1 + 2 + \cdots + 100
\]

Generalize the loop so that the sum from 1 to any \(n\) can be computed.

**Steps to design:**

- Identify and define a loop control variable.
- Write the syntax for the loop control structure
- Fill in the code used within the loop to compute the sum

```c
int sum = 0;
int n = 100;
int i = 1; /* our loop control variable */
while (i <= n)
    
    sum = sum + i;
    i = i + 1;
}
printf("Sum 1 to %d is %d\n", n, sum);
```

While Loop Example II

Instead of the sum of integers 1 to \(n\), compute the product:

\[
\prod_{i=1}^{100} i = 1 \times 2 \times \cdots \times 100
\]

What changes need to be made?

- Variable names?
- Initialized variable value?
- Operators?

Note: this is the factorial function, 
\[n! = \prod_{i=1}^{n} i\]
While Loop Example II

```c
int product = 1;
int n = 100; /* general variable, may be changed or read from input */
int i = 1; /* our loop control variable */
while (i <= n)
{
    product *= i;
    i += 1;
}
printf("Product 1 to %d is %d\n", n, product);
```

Program Failed

Run the previous program: it gives an answer of 0—why?

- Debug your code: use a printf statement in the loop to see what intermediate values are computed:
  ```c
  printf("i = %3d product = %d\n", i, product);
  ```
- Check the answers with a calculator
- For what i does this program fail?

Overflow

- We got the wrong answer for i = 13,
  13! = 6,227,020,800
- We used a 32-bit integer to store product
- Maximum representable value is $2^{31} = 2,147,483,648$
- When a number is too large (or too small!) to be represented by its type, overflow occurs (or underflow)
- More sophisticated solutions are available, but beyond this course

Compound Assignment Operators

- Expressions such as `variable = variable op expression;` (where `op` is a C operator such as `+,-,*,/`) occur frequently
- C provides several syntax shortcuts
  - `x = x + 1;` and `x += 1;` are "equivalent"
  - Can do this with other operators (see table)

```
Expression       Shortcut
---------------------
x = x + 1;        x += 1;
x = x - 1;        x -= 1;
x = x * 5;        x *= 5;
x = x / 2;        x /= 2;
```

Table: Compound Assignment Operators

For Loops

- Program Style
- Increment and Decrement Operators
- Increment and Decrement Other Than 1

Compound Assignment Operators

Example Revisited

```c
int product = 1;
int n = 100; /* general variable, may be changed or read from input */
int i = 1; /* our loop control variable */
while (i <= n)
{
    product *= i;
    i += 1;
}
printf("Product 1 to %d is %d\n", n, product);
```
For Loops

- Any repetition can be implemented using a while loop
- Another way to construct a counting loop is to use a for loop
- C provides for statements as another form for implementing loops.
- As before we need to initialize, test, and update the loop control variable.
- The syntax for a for statement is more rigid: it designates a specific place for the initialization, testing, and update components

For Loop Example

Computing the sum using a for-loop:

```c
int sum = 0;
int n = 100;
int i;
for (i = 0; i <= n; i++)
{
    sum = sum + i;
}
```

- Advantages: more readable, more predictable
- Easier to debug
- Pitfall: note the placement of semicolons!

Increment Operators

- New syntax: i++
- Known as a (postfix) increment
- “Equivalent” to i = i + 1
- Also available: (postfix) decrement: i-- (“equivalent” to i = i - 1)

Program Style

For clarity, the book usually places each expression of the for heading on a separate line. If all three expressions are very short, however, they will be placed on one line.

The body of the for loop is indented just as the if statement.

Increment and Decrement Operators

The counting loops that we have seen have all included assignment expressions of the form

- counter = counter + 1
- counter++
- counter += 1

This will add 1 to the variable counter.

Using -- will subtract one from the counter.

Increment and Decrement Other Than 1

We can use the “shortcut” compound assignment operators with values other than 1

- Increment operations: sum = sum + x or sum += x, will take the value of sum, add x to it, and then assign the new value to sum
- Decrement operations: temp = temp - x or temp -= x, will take the value of temp, subtract x from it and then assign the new value to temp
Increment and Decrement Other Than 1

Example

```c
/* increment by 10 */
int x = 10;
int i;
for (i =0; i <100; i+=x)
    printf("i = %d\n",i);

/* decrement by 5 */
int y = 5;
for (i =25; i >=0; i -=y)
    printf("i = %d\n",i);
```

Conditional Loops

- The exact number of loop repetitions we need to run for a loop will not always be known before loop execution begins.

Initialization step? Test? Update action?

Exercise

Create a program that prompts the user for a value x and multiplies it by the previous value of x storing the result in x until the user enters a 0.

Exercise

Pseudocode

```c
Set x to an initial value of 1 ;
Prompt the user for a value input;
while input is not zero do
    Set x to x multiplied by input;
    Prompt the user for a new input value ;
end
```

Algorithm 2: Prompt Product Loop

Exercise

Translated to C

```c
int x = 1;
int value ;
printf(" Enter a value , (0 to quit )> ");
scanf(" %d", &value );
while(value != 0)
{
    x = x * value;
    printf(" Enter a value , (0 to quit )> ");
    scanf(" %d", &value );
}
printf(" The product is %d", value );
```

Loop Design

To this point, we have been analyzing the actions a loop performs.

Now, we also want to design our own loops:

- Sentinel-Controlled Loops
- Using a for Statement to Implement a Sentinel Loop
Sentinel-Controlled Loops

- Often we don’t know how many data items the loop should process when it begins execution.
- **Sentinel-Controlled Loops** continue to read data until a unique data value is read, called the *sentinel value*.
- The sentinel value should be a value that could not normally occur as data.
- Reading the sentinel value signals the program to stop reading and processing new data and exit the loop.
- Example: Product of a list of numbers, with 

```c
int sentinelValue = -1;
int score = 0;
printf("Enter first score (%d to quit)\n", sentinelValue);
for(score = sentinelValue; score != sentinelValue; score = sentinelValue) {
    scanf("%d", &score);
    if(score < i)
        printf("+");
    else
        printf("*");
    printf("\n");
}
```

Implementing a Sentinel Loop

Because the `for` statement combines the initialization, test, and update in one place, some programmers prefer to use it to implement sentinel-controlled loops.

```c
for(i = 1; i <= 10; i++) {
    for(j = 1; j <= 10; j++) {
        if(j < i)
            printf("+");
        else
            printf("*");
    }
    printf("\n");
}
```

Nested Loops

Like `if` statements, loops can also be nested.
- Nested loops consist of an outer loop with or more inner loops.
- Each time the outer loop is repeated, the inner loops are reentered.
- The inner loop control expressions are reevaluated, and all required iterations are performed.
The do-while Statement and Flag-Controlled Loops

- do-while statement
- flag-controlled loops

Do-While Statement

- The for statement and the while statement evaluate conditions before the first execution of the loop body.
- In most cases, this pretest is desirable:
  - Prevents the loop from executing when there are no data items to process
  - Prevents execution when the intimal value of the loop control variable is outside the expected range.
- Situations involving interactive input, when we know that a loop must execute at least one time, often use a do-while loop.

Do While

```
char letter_choice;
do {
  printf("Enter a letter from A through E> ");
  scanf("%c", &letter_choice);
} while (letter_choice < 'A' || letter_choice > 'E');
```

Flag-Controlled Loops

- Sometimes a loop repetition condition becomes so complex that placing the full expression in its usual spot is awkward.
- In many cases, the condition may be simplified by using a flag.
  - A flag is a type int variable used to represent whether or not a certain event has occurred.
  - A flag has one of two values: 1 (true) and 0 (false).
Flag
Example

```c
char letter_choice;
int isDone = 0;
while (!isDone)
{
    printf("Enter a letter from A through E> ");
    scanf(" %c", &letter_choice);
    isDone = (letter_choice >= 'A' && letter_choice <= 'E');
}
```

How to Debug and Test Programs

- Debugging using `printf`
- Off-by-One Loop Errors
- Testing

Debugging using `printf`

- Use several `printf` statements to output the values of your variables to make sure they have the correct value in them as your program executes.
- It is often helpful to print out the value of your loop control variable to make sure you are incrementing it and will not enter an infinite loop.

Off-by-One Loop Errors

**Loop boundaries** - the initial and final values of the loop control variable.

- A fairly common logic error in programs with loops is a loop that executes one more time or one less time than required.
  - If a sentinel-controlled loop performs an extra repetition, it may erroneously process the sentinel value along with the regular data.
  - If a loop performs a counting operation, make sure that the initial and final values of the loop control variable are correct and that the loop repetition condition is right.
    - The sum of 1...100, is not `for(i = 1; i < 100; i++) sum += i;`
    - Instead, `i <= 100` should be used.

Testing

After all errors have been corrected and the program appears to execute as expected, the program should be tested thoroughly to make sure it works.

For a simple program, make enough test runs to verify that the program works properly for representative samples of all possible data combinations.

Common Programming Errors I

- `if` and `while` statement can be confused, since they have similar appearance.
- Remember to initialize loop control variable as to prevent infinite loops.
- Infinite loops are bad: kill your program using control-C
- Remember to use brackets `{ ... }` around the code of the loop statements.
Be careful about the loop conditions, if we only want positive results then `if(result != 0)` would not work since result might become negative without ever being 0.

- `do-while` loops always execute once and then tests the condition.
- With the compound assignment operators, the parentheses are assumed to be around any expression that is the second operand.

### Real World Example

**What happened?**

```java
while (days > 365)
{
  if (IsLeapYear(year))
    {
      if (days > 366)
        {
          days -= 366;
          year += 1;
        }
    }
  else
    {
      days -= 365;
      year += 1;
    }
}
```

### Other Examples

- Zune Bug: December 31st, 2008
- 2008 was a leap year: 366 days
- Thousands of Zunes froze for 24 hours
- An embedded module in the Zune contained the following (actual) code

### Questions

Questions?