Crude Video Game Simulator Algorithm

The following program will simulate free games at an arcade. The player will enter his/her score for 5 levels of play. The sum of these is their game score. If this exceeds 100,000, then the player gets a free game. The player continues to get free games if they exceed 100,000, and their total score is the sum of all of their game scores. This algorithm simulates this process and prints out the player's total score at the end of the game.

```
#include <stdio.h>
int main() {

    int temp_score, game_score, total_score;
    int level, done = 0;

    total_score = 0; //Initialize player's total score.

    //Continue playing games until user fails to earn a free game
    while (done == 0) {

        // Initialize player's game score and level.
        game_score = 0;

        // Tabulate game score for all 5 levels
        for (level=1; level<=5; level++) {
            printf("What was your score on level %d?\n",level);
            scanf("%d", &temp_score);
            game_score = game_score + temp_score;
        }

        // Compute new total score and stop if player has failed
        // to earn a free game.
        total_score = total_score + game_score;
        if (game_score < 100000)
            done = 1;
        else
             printf("Congratulations, you have won a free game!\n");
    }

    printf("Game Over! Your total score is %d\n", total_score);
    return 0;
}
```

The Comma Operator

The comma operator allows multiple statements to execute on a single line. To do this, simply separate the two statements with a comma:

```
stmt1, stmt2;
```

An example of this would be:

```
i = 0, j = 0;
```

All this does is execute the leftmost statement first, followed by the rightmost statement. In most situations, you can just as easily write out two separate statements:

```
i = 0;
j = 0;
```

The only situation where using the comma operator may be desirable is in a for loop. If you want to initialize two variables or increment two variables, you could do something like:

```
for (i=0, j=0; i+j<20; i+=2, j--) {
   ...
}
```

My advice would be that unless you find a compelling reason to use the comma operator, don't use it. The example of page 98 of the text shows a potential problem of doing so.

Do-While Loop

This is the third and last loop you will see. Here is the general syntax:

```
do
  stmt;
while (<bool exp>);
```

Typically, the statement will be a block, so usually, we have:

```
do {
  stmt1;
  stmt2;
  ...
  stmtn;
} while (<bool exp>);
stmtA;
```

Here is how this construct is evaluated when run:

1) Execute statements 1 through n.
2) Check the boolean expression.
3) If it is true, go back to step 1, otherwise continue execution with statement A.

The key difference between this construct and the while construct is that the body of the loop must be executed at least once here. If you have this type of situation and it is easier for you to check the boolean condition AFTER you have executed a block of statements than before, then this loop may be a suitable choice.

Our favorite sum program...using a do-while

```
/* Arup Guha   9/17/03
   This program sums up the odd numbers
   in between 1 and 100, inclusive and
   prints this sum to the screen.        */

#include <stdio.h>
int main() {

    // Initialize variables.
    int val = 1;
    int sum = 0;

    // Add successive odd numbers.
    do {
         sum = sum + val;
         val = val + 2;
    } while (val < 100);

    // Print out the result.
    printf("1+3+5+...+99=%d\n",sum);
    return 0;
}
```

Guessing Game!

```
/* Arup Guha   9/17/03
   This program allows the user to guess a
   secret number from 1 to 100. The user is
   told if their guess is too low or high
   and is allowed to guess again.        */
#include <stdio.h>
int main() {

    int secret_number;
    int guess;

    // Create secret number randomly.
    secret_number = 1+rand()%100;

    // Prompt user for a guess.
    printf("Guess a number(1-100)\n");
    do {

       // Read in guess and print out
       // appropriate response.
       scanf("%d", &guess)

       if (guess < secret_number)
         printf("Your guess is too low!\n");
       else if (guess > secret_number)
         printf("Your guess is too high\n");

    } while (guess != secret_number);

    // Print out winning message.
    printf("You guessed correctly!\n");
    return 0;
}
```

The goto statement

As the book says, generally speaking, this statement should not be used. There are a few instances where it makes code a bit more efficient and reduces the number of nested structures, but in my experience, when most introductory programmers use gotos, they don't do so efficiently and their code generally becomes confusing to them.

The break and continue statements

Both of these statements can be used to aid flow control in loops. In particular, here is what they do:

break: Will break you out of the inner-most loop in which the break statement resides.

continue: Will skip the following code within the inner-most loop in which the continue statement resides, returning the flow of control to the top of that loop.

Break Statement Example

In the following code segment we will attempt to pick 10 random numbers. Each of the number picked must be greater than all the previous values picked. Each value will simply be printed to the screen.

```
value = rand();
max = value;
printf("Number 1 = %d\n", value);

for (i=2; i<=10; i++) {

  while (1) {

    if (value > max) {
      max = value;
      break;
    }
    value = rand();

  }

  printf("Number %d = %d\n", i, value);
}
```

Switch Statement

This statement, just like the if statement, allows for conditional execution. The general construct is as follows:

```
switch (<integer expression>) {
    case <value1>:
        <stmts1>
        break;
    case <value2>:
        <stmts2>
        break;
    ...
    default:
        <stmtsn>
}
stmtA
```

The manner in which this expression is evaluated is as follows:

1) The integer expression is evaluated.
2) If this expression is equal to value1, then <stmts1> are executed and the break statement carries the execution to stmtA.
3) Otherwise, each value is compared th the integer expression value until one is found to match. At that point, the corresponding set of statements is executed and then the break carries execution to stmtA.
4) If no listed value is equal to the value of the expression, then the default case is executed.

Couple notes: The lists of values may not include ranges of values, but only single values. Also, the break statements are not required, but without them, then several cases may get executed.

This generally limits the use of case statements to situations where you know an expression will equal one of a few integer values and based on that want to execute a certain segment of code. You can emulate a case statement with an appropriate if statement.

Here is an example of a case statement:

```
switch (answer) {
    case 1:
        printf("You have selected #1.\n");
        break;
    case 2:
        printf("You have selected #2.\n");
        break;
    case 3:
        printf("You have selected #3.\n");
        break;
    default:
        printf("Invalid selection.\n");
}
```

Example using a continue statement

In this segment of code, we add up n values entered by the user that are valid test scores in between 0 and 100, inclusive:

```
count = 0;
while (count < n) {

  scanf("%d", &score);
  if (score < 0 || score > 100)
    continue;

  count++;
  sum = sum + score;
}
```

The Conditional Operator

The general syntax of an expression using the conditional operator is as follows:

```
<boolean expr1> ? <arith expr2> :
                  <arith expr3>
```

This expression is evaluated as follows:

1) The boolean expression is evaluated.
2) If it is true, the entire expression evaluates to <arith expr2>
3) If it is false, the entire expression evaluates to <arith expr3>

Basically, this is a short-hand notation to calculate an expression that could be calculated in an if statement. Here is an example of a segment of code that assigns x to the minimum of x and y, and y to the maximum of x and y. (Assume max and min are declared variables also.)

```
max = (x < y) ? x : y;
min = (x > y) ? x : y;
```

Although this type of expression can shorten code, I believe it makes code more difficult to read at the expense of saving a couple lines of writing. Based on that observation, I do not recommend its use. (Though, you should understand the syntax of it in case you see it used in other peoples' code.)

Common Programming Errors w/loops

1) Creating infinite loops

2) putting a ; right at the end of a loop structure.

(e.g. for (i=0;i<10;i++);)

3) Forgetting to put {} around the loop body.

4) Creating an incorrect boolean expression. (e.g. using an && instead of an ||, or using the boolean expression with an opposite value of the one you should use.)

5) Off by one error - loop runs one too many or too few times.

6) Not using the loop index effectively inside of the body of the loop.

7) Forgetting to print a menu in a loop, or some other task that should be repeated each loop iteration.