Alright, let's cover the essential concepts of loops in C for a 1st-year undergraduate. Loops are fundamental control flow structures that allow you to execute a block of code repeatedly until a certain condition is met.

Loops in C: Notes for a 1st Year Undergraduate

Loops are used to automate repetitive tasks in programming. C provides three main types of loops: for, while, and do-while.

1. for Loop

The for loop is often used when you know in advance how many times you need to execute a block of code.

```
for (initialization; condition; increment/decrement) {
    // Code to be executed repeatedly
}
```

- **initialization**: This statement is executed only once at the beginning of the loop. It's typically used to initialize a loop counter variable.
- **condition**: This expression is evaluated before each iteration of the loop. If the condition is true, the loop body is executed. If it's false, the loop terminates.
- **increment/decrement**: This statement is executed after each iteration of the loop. It's usually used to update the loop counter variable.

Example:

```
#include <stdio.h>

int main() {
    int i;

    for (i = 0; i < 5; i++) {
        printf("Iteration: %d\n", i);
    }

    return 0;
}
```

Output:

```
Iteration: 0
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
```

Key Points about for Loop:

- The initialization, condition, and increment/decrement parts are optional. You can have an empty for loop (;;) which might run indefinitely (infinite loop) if no break condition is provided inside the loop body.

- You can initialize multiple variables in the initialization part (separated by commas), and you can have multiple increment/decrement statements (also separated by commas).
- The condition can be any valid C expression.

2. while Loop

The while loop executes a block of code as long as a specified condition is true. The condition is checked *before* each iteration.

```
while (condition) {
    // Code to be executed repeatedly as long as the condition is true
}
```

- **condition**: This expression is evaluated before each iteration. If it's true, the loop body is executed. If it's false, the loop terminates.

Example:

```
#include <stdio.h>

int main() {
    int count = 0;

    while (count < 3) {
        printf("Count is: %d\n", count);
        count++;
    }

    return 0;
}
```

Output:

```
Count is: 0
Count is: 1
Count is: 2
```

Key Points about while Loop:

- It's possible that the loop body might not execute at all if the condition is false initially.
- You need to ensure that the condition eventually becomes false within the loop body (usually by modifying a variable involved in the condition) to avoid an infinite loop.

3. do-while Loop

The do-while loop is similar to the while loop, but it checks the condition *after* executing the loop body. This guarantees that the loop body will execute at least once.

```
do {
    // Code to be executed repeatedly
} while (condition); // Note the semicolon at the end
```

- **condition**: This expression is evaluated *after* each iteration. If it's true, the loop continues. If it's false, the loop terminates.

Example:

```
#include <stdio.h>

int main() {
    int num;

    do {
        printf("Enter a positive number: ");
        scanf("%d", &num);
    } while (num <= 0);

    printf("You entered: %d\n", num);

    return 0;
}
```

Key Points about do-while Loop:

- The loop body executes at least once, even if the condition is initially false.
- Remember the semicolon ; at the end of the while condition in a do-while loop.

4. Loop Control Statements

C provides statements to alter the normal flow of loops:

- **break Statement:** The break statement is used to immediately terminate the innermost loop (either for, while, or do-while) it is enclosed in. Execution continues with the statement immediately following the loop.

  ```
  #include <stdio.h>

  int main() {
      for (int i = 0; i < 10; i++) {
          if (i == 5) {
              break; // Exit the loop when i is 5
          }
          printf("i = %d\n", i);
      }
      printf("Loop finished.\n");
      return 0;
  }
  ```

  Output:

  ```
  i = 0
  i = 1
  i = 2
  i = 3
  i = 4
  Loop finished.
  ```

- **continue Statement:** The continue statement skips the rest of the current iteration of the loop and proceeds to the next iteration. For for loops, it goes to the increment/decrement

part; for while and do-while loops, it goes to the condition evaluation.

```
#include <stdio.h>

int main() {
    for (int i = 0; i < 5; i++) {
        if (i == 2) {
            continue; // Skip the iteration when i is 2
        }
        printf("i = %d\n", i);
    }
    return 0;
}
```

Output:

```
i = 0
i = 1
i = 3
i = 4
```

5. Nested Loops

You can have one loop inside another loop. This is called nested looping. The inner loop will execute completely for each iteration of the outer loop.

```
#include <stdio.h>

int main() {
    int i, j;

    for (i = 1; i <= 3; i++) {
        printf("Outer loop (i = %d):\n", i);
        for (j = 1; j <= 2; j++) {
            printf("  Inner loop (j = %d)\n", j);
        }
    }

    return 0;
}
```

Output:

```
Outer loop (i = 1):
  Inner loop (j = 1)
  Inner loop (j = 2)
Outer loop (i = 2):
  Inner loop (j = 1)
  Inner loop (j = 2)
Outer loop (i = 3):
  Inner loop (j = 1)
  Inner loop (j = 2)
```

Choosing the Right Loop

- Use a for loop when you know the number of iterations in advance or when you have a clear initialization, condition, and increment/decrement.
- Use a while loop when you need to repeat a block of code as long as a condition remains true, and you might not know the number of iterations beforehand. The condition is checked before the loop body.
- Use a do-while loop when you need to execute a block of code at least once, and then continue as long as a condition is true. The condition is checked after the loop body.

Loops are fundamental for creating programs that can perform repetitive tasks efficiently. Understanding how to use each type of loop and the loop control statements is crucial for writing effective C code.