Lecture Unit 4: Flow of Control
Examples of Statements in C

Conditional Statements
- The if-else Conditional Statement
- The switch-case Conditional Statement
- The (exp) ? x:y Conditional Expression

Iteration or Loop Statements
- The while Loop Statement
- The do-while Loop Statement
- The for Loop Statement

Jump Statements: break, continue, and goto

Example: Primality Testing
Examples of Statements

❖ Every program in C is a sequence of compiler directives, declarations, and **statements**. There are six kinds of statements in C: **expression statement, compound statement, conditional statement, iteration (loop) statement, labeled statement, and jump statement**.

❖ So far, we have seen just one of these --- the **expression statement**. An expression statement is just an expression followed by `;` for example:

```
x = a+b;
x++; a+b; ;
```

❖ Here is one more kind of statements --- the **compound statement** also called a **block**. A compound statement is any sequence of statements and declarations enclosed in `{ . . . }` for example:

```
int main()
{
  t = x;
  x = y;
  y = t;
}
```

```
{ a = x++;
  if (a == x)
  {
    printf("Wow?\n");
    return 1;
  }
}
```
Outline of this Lecture

❖ Examples of Statements in C

❖ Conditional Statements
  › The if-else Conditional Statement
  › The switch-case Conditional Statement
  › The (exp) ? x:y Conditional Expression

❖ Iteration or Loop Statements
  › The while Loop Statement
  › The do-while Loop Statement
  › The for Loop Statement

❖ Jump Statements: break, continue, and goto

❖ Example: Primality Testing
The `if` Conditional Statement

The syntax of the `if` conditional statement is as follows:

```
if (expression) statement
```

- If the logical value of the `expression` is **True** the `statement` is executed.
- If the logical value of the `expression` is **False** the `statement` is skipped, and the program flow continues with the next statement.
- In most cases, but not always, the `statement` is a **compound statement**.

**Example:**

```c
if (b*b < 4*a*c) {
   printf("Quadratic equation has no real roots!");
   return 1;
}
```

Example:
The syntax of the `if-else` conditional statement is as follows:

- If the logical value of the `expression` is `True`, `statement1` is executed, then `statement2` is skipped and the program flow continues.
- If the logical value of the `expression` is `False`, `statement2` is executed, then `statement1` is skipped and the program flow continues.

**Example:**

```c
if (tomorrow_is_rainy == 'y')
   printf("Take your umbrella.\n");
else
   printf("Leave the umbrella at home.\n");
```
What does the following statement do?

```c
if (a==1)
    if (b==2)
        printf("***");
    else
        printf("###");
else printf("###");
```

❖ The rule is this: the `else` always belongs to the last else-less `if`.

❖ Nevertheless, the rule can be modified by explicitly creating a single compound statement enclosed in `{ . . . }` for example like this:
Multi-way if-else-if Chains

The best way to explain this is by using an example:

```c
if (a < 0)
    return -1;
else
    if (a == 0)
        return 0;
    else
        if (a < 11)
            return 1;
        else
            return a;
```

The best way to explain this is by using an example:
Proper indentation is extremely useful in elucidating the structure of an **if-else-if** chain, like this:

```c
if (a < 0)
    return -1;
else
    if (a == 0)
        return 0;
    else
        if (a < 11)
            return 1;
        else
            return a;
```

Both indentation methods are well-established. The second one results in a more compact code, with shorter lines.
The syntax of the `switch-case-default` conditional statement is as follows:

```c
switch (expression) {
  case value1: statements
               break;      // optional
  case value2: statements
               break;      // optional
    ...
  default: statements  // optional
}
```

The value of the `expression` is compared to `value1`, `value2`, and so on sequentially until there is a match. If there is no match, `default` is executed. If there is also no `default` case, nothing is executed. If there is a match, execution starts at this case and goes on until either a `break` or the closing brace `}` is encountered.
int x, y;
char op;
... // read x, y, op
switch (op)
{
  case '+':
    printf("Result is: %d", x + y);
    break;    // what if we remove it?
  case '-':
    printf("Result is: %d", x - y);
    break;
  ... // code for '*' and '/' and '%'
  default:
    printf("Invalid operation!");
}
Conditional Expression (exp) ? x : y

Expressions in C can involve the **ternary conditional operator**. The syntax of this operator ? : is as follows:

```
expression1 ? expression2 : expression3
```

- If the logical value of `expression1` is True, `expression2` is evaluated.
- If the logical value of `expression1` is False, `expression3` is evaluated.

The conditional expression ? : is **not** a statement; thus it can appear as part of another expression. For example:

```c
if (n > 1) c = 2*a;
else c = 2*b;
```

What is the value of `x` here?

```c
float x;  x = 1/((n > 1) ? 2.0:2);
```

More Examples:

```c
min = (x < y) ? x:y;
distance = (a > b) ? (a - b):(b - a);
printf("You have %d item%s\n", n, (n == 1) ? ":" : ":s");
```
Outline of this Lecture

❖ Examples of Statements in C

❖ Conditional Statements
  ▶ The \texttt{if-else} Conditional Statement
  ▶ The \texttt{switch-case} Conditional Statement
  ▶ The \texttt{(exp) ? x:y} Conditional Expression

❖ Iteration or Loop Statements
  ▶ The \texttt{while} Loop Statement
  ▶ The \texttt{do-while} Loop Statement
  ▶ The \texttt{for} Loop Statement

❖ Jump Statements: \texttt{break}, \texttt{continue}, and \texttt{goto}

❖ Example: Primality Testing
What Is a Loop?

❖ In general, a **loop** is a piece of code that is executed repeatedly several times, where “several” could range from zero to infinity. Here are some examples of loops:

```
for (i = 1; i <= n; i++)
  factorial *= i;
```

```
while (1)
{
  factorial *= i++;
  if (i > n) break;
}
```

❖ Each time the piece of code is executed is called a **loop iteration**, or simply an iteration. These iterations continue until either:

- The **loop condition** ceases to be satisfied, or
- The program exits from the loop via a *jump statement*, such as **break** or **return**.

❖ The C language provides three different types of loops:

- The **while** statement
- The **do-while** statement
- The **for** statement
The syntax of the `while` loop statement is as follows:

```
while (expression) 
    statement
```

- The `expression` is evaluated. If its logical value is `True` (the `expression` is nonzero) then `statement` is executed.
- After that the `expression` is evaluated again, and the process repeats until the logical value of the `expression` becomes `False`.
- When the logical value of the `expression` is `False`, program continues to next statement.

**Example:**

```c
int n, factorial = 1, i = 1;
scanf("%d", &n);
while(i <= n) factorial *= i++;
printf("n!= %d", factorial);
```
Example: Odd and Even Averages

**Problem:** Prompt the user to enter a positive integer, and keep doing this while the user complies. When the user inputs an integer that is not positive, output:

- The average of all the even positive integers the user entered
- The average of all the odd positive integers the user entered

**Assumptions:** The user input is valid. The user enters only integers, including at least one odd integer and at least one even integer.

**Algorithm:** Read user input in a while loop, and keep the current even_sum and odd_sum. At each iteration, the integer entered by the user will be added to even_sum if it is even and to odd_sum if it is odd. Also count the total number of odd and even integers entered. Then:

```
even_average = even_sum / even_count
odd_average = odd_sum / odd_count
```
Example: Odd and Even Averages

```c
#include <stdio.h>

int main()
{
    int odd_sum = 0, even_sum = 0, odd_count = 0, even_count = 0, next;

    printf("Enter a positive integer: ");
    scanf("%d", &next);

    while (next > 0)
    {
        if (next % 2) {odd_sum += next; odd_count++;}
        else {even_sum += next; even_count++;}

        printf("Enter a positive integer: ");
        scanf("%d", &next);
    }

    printf("\nAverage of even integers is: %6.2f\n"
          " Average of odd integers is: %6.2f\n"
          , (double)even_sum/even_count, (double)odd_sum/odd_count);

    return 0;
}
```
The do-while Loop Statement

- The syntax of the do-while loop statement is as follows:

```c
do statement while (expression);
```

- First, the `statement` is executed, and then `expression` is evaluated.
- If its logical value is `True` (the `expression` is nonzero), the `statement` is executed again, and the process repeats until the logical value of the `expression` becomes `False`.
- When `expression` evaluates to `False`, program flow continues to the next statement.

**Example:**

```c
do
{   printf("Enter a positive integer: ");
    scanf("%d", &x);
} while (x <= 0);
```

The statement of a do-while loop is executed at least once!
The syntax of the for loop statement is as follows:

```
for (expression1;
     expression2;
        expression3)
statement
```

- First, `expression1` is executed. This serves as the initialization of the loop.
- Next, `expression2` is evaluated. If its logical value is True (`expression2` is nonzero), the `statement` and then `expression3` are executed. They keep being executed, in this order repeatedly, as long as `expression2` evaluates to True.
- When `expression2` evaluates to False, program flow continues to next statement.

Each of the three expressions can be empty; empty `expression2` is True!
The for Loop: Examples

- Here is an example of a for loop that computes the sum of the first \( n \) squares:

  ```c
  sum = 0;
  for (i = 1; i <= n; i++) sum += i*i;
  ```

- Another example:

  ```c
  factorial = 1;
  for (i = 2; i <= n; i++) factorial *= i;
  ```

**Note:** The for and the while loops are equivalent!
The for Loop: Comma Operator

What is the comma operator?

- An expression like this `exp1, exp2` is evaluated by first evaluating `exp1` then evaluating `exp2`. Similarly for `exp1, exp2, exp3`, etc.
- The type of a comma expression is the type of the last `exp` evaluated.
- Such expressions can be used in the initialization/increment of for-loop.

Example. We have seen this before:

```c
sum = 0;
for (i = 1; i <= n; i++)
   sum += i*i;
```

This can be also written as:

```c
for (sum = 0, i = 1; i <= n; i++)
   sum += i*i;
```

Or even as:

```c
for (sum = 0, i = 1; i <= n; sum += i*i, i++);
```
Example: Primality Testing

**Problem:** Given a positive integer \( n \), decide whether it’s prime.

**Definition:** A positive integer \( n \) is prime if and only if \( n > 1 \) and it has no divisors in the set \( \{2, 3, \ldots, n-1\} \). Equivalently, \( n \) is not prime if either \( n = 1 \) or \( n \) has a divisor in the set \( \{2, 3, \ldots, n-1\} \).

**Solution:**

```c
unsigned int is_prime = 1, n, i;
... \// read n

if(n == 1) is_prime = 0;
else 
   for (i = 2; i < n; i++)
      if (n % i == 0) is_prime = 0;

if (is_prime)
   printf("%d is a prime\n", n);
else
   printf("%d is not a prime\n", n);
```

Problem: Given a positive integer \( n \), decide whether it’s prime.
Outline of this Lecture

❖ Examples of Statements in C

❖ Conditional Statements
  ‣ The if-else Conditional Statement
  ‣ The switch-case Conditional Statement
  ‣ The (exp) ? x:y Conditional Expression

❖ Iteration or Loop Statements
  ‣ The while Loop Statement
  ‣ The do-while Loop Statement
  ‣ The for Loop Statement

❖ Jump Statements: break, continue, and goto

❖ Example: Primality Testing
The break and continue Statements

These are special *jump statements* that change the default behavior of the *while*, *do-while*, and *for* loops:

**break** Causes **immediate exit from the loop**. The rest of the code in the loop is skipped, and the program flow proceeds to the *next statement after the end of the loop*.

Recall that **break** works in exactly the same way also for the *switch-case-default* statement. The following, though, works only for loops:

**continue** Causes the **rest of the loop to be skipped**. The program flow proceeds to the *next iteration of the same loop*.

**Example:** What will get printed?

```c
for (k = i = 0; i < 8; i++)
    for (j = 0; j < 8; j++)
        if (j == 4) break;
        k++;
printf("%d,%d,%d", i,j,k);
```

```c
for (k = i = 0; i < 8; i++)
    for (j = 0; j < 8; j++)
        if (j == 4) continue;
        k++;
printf("%d,%d,%d", i,j,k);
```

```
8, 4, 32
```

```
8, 8, 56
```
Examples of continue Usage

❖ The following code reads 100 integers and computes the sum of those that are nonnegative:

```c
for (i = 0; i < 100; i++)
{
    scanf("%d",&num);
    if(num < 0) continue;
    sum += num;
}
```

❖ What is the difference between the code above and the code to the left? Are they equivalent?

❖ The for loop will always read exactly 100 integers. The while loop will read as many integers as it needs (perhaps 1,000,000) to get exactly 100 nonnegative integers.

What could we change to make the two loops equivalent?
Infinite Loops and **break** Usage

- Infinite loops could arise in a C program either **by design** or **by mistake**. If a program goes into an infinite loop by mistake, it will usually “hang” producing no output. In most cases, you can terminate such a program by pressing **CTRL-C** and then debug.

- However, infinite loops are often introduced into a C program by design. For example:

```c
sum = 0;
do{
   scanf("%d", &num);
   sum += num;
} while (num >= 0)
```

The program can be **designed** to exit such infinite loops via one of the following statements: **break**, **return**, or **goto**.

**Example:**
The following loop is designed to read from the input nonnegative integers and compute their sum, exiting as soon as the user inputs the first negative integer.

```c
sum = 0;
do{
   scanf("%d", &num);
   sum += num;
} while (num >= 0)
```

**What’s wrong with this do-while loop?**
**Problem:** Compute the logical OR of \( n \) input values.

**Solution:** Two different approaches, both using `break`.

```c
enum boolean {FALSE,TRUE};
int i, value;
boolean or;

or = FALSE;
for (i = 0; i < n; i++)
{
    scanf("%d", &value);
    if (value)
    {
        or = TRUE;
        break;
    }
}
```

```c
for (i = 0; i < n; i++)
{
    scanf("%d", &value);
    if (value) break;
    or = (i < n);
}
```

What happens if we delete the `break`?
The goto Jump Statement

To use the goto statement, you first need to create a labeled statement, which is any statement prefixed by a label:, like this:

```
sum: x = a+b;
myloop: while(1) x++;
error: return 1;
```

Any identifier can serve as the label in a labeled statement. Thereafter, you can use `goto sum`, `goto myloop`, etc. at any point in the program.

**Example of Usage:**

```
for (i = 0; i < n; i++)
{
    ...
    // do something
    for (j = 0; j < m; j++)
    {
        ...
        // do something else
        if (disaster) goto error;
        ...
    }
}
...
error: printf("Please wait while I clean-up the mess!");
    ...
    // clean-up the mess
```
Outline of this Lecture

❖ Examples of Statements in C

❖ Conditional Statements
  ‣ The if-else Conditional Statement
  ‣ The switch-case Conditional Statement
  ‣ The (exp) ? x:y Conditional Expression

❖ Iteration or Loop Statements
  ‣ The while Loop Statement
  ‣ The do-while Loop Statement
  ‣ The for Loop Statement

❖ Jump Statements: break, continue, and goto

❖ Example: Primality Testing
Recall: Primality Testing

**Problem:** Given a positive integer \( n \), decide whether it’s prime.

**Definition:** A positive integer \( n \) is **prime** if and only if \( n > 1 \) and it has no divisors in the set \( \{2,3,\ldots,n-1\} \). Equivalently, \( n \) is **not prime** if either \( n = 1 \) or \( n \) has a divisor in the set \( \{2,3,\ldots,n-1\} \).

**Solution:**

```c
unsigned int is_prime = 1, n, i;
... \ read n

if(n == 1) is_prime = 0;
else 
   for (i = 2; i < n; i++)
      if (n % i == 0) is_prime = 0;
if (is_prime)
   printf("%d is a prime\n", n);
else
   printf("%d is not a prime\n", n);
```

```c
unsigned int is_prime = 1, n, i;
... \ read n
if(n == 1) is_prime = 0;
else 
   for (i = 2; i < n; i++)
      if (n % i == 0) is_prime = 0;
if (is_prime)
   printf("%d is a prime\n", n);
else
   printf("%d is not a prime\n", n);
```
If \( n \) is not 2, but is divisible by 2, then \( n \) is **not** prime.

If \( n \) is not divisible by anything up to the square root of itself, then it must be prime.

It is enough to check divisibility up to square root of \( n \).

Since \( n \) is odd under `else`, we can skip testing divisibility by all even integers \( i \).

When a divisor \( i \) is found, \( n \) is **not** prime, and there is no need to keep testing.

```c
#include <math.h>
...
int is_prime = 1, n, i, sqrt_n;
...
if (n == 1 || (n != 2 && n%2 == 0))
   is_prime = 0;
else
{
   sqrt_n = (int) sqrt(n);
   for (i = 3; i <= sqrt_n; i += 2)
      if (n%i == 0) {is_prime = 0; break;}
   }
if (is_prime)
   printf("%d is a prime\n", n);
else
   printf("%d is not a prime\n", n);
...
```