CS113: Lecture 2

Topics:

- Decision and Control statements (e.g. if-else, switch, while, etc.)
- Relational, Equality, and Logical operators
if statement

- Basic form:

```c
if( condition )
    statement;

(Statement executed if, and only if, the condition is “true”)
```

- Example (fragment):

```c
if( 5 > 3 )
    printf( "5 is strictly greater than 3.\n" );
```

- The statement can be a block of code containing more than one statement - enclosed in curly braces:

```c
if( a > 0 )
{
    printf( "a is positive.\n" );
    printf( "In case you didn’t hear me,
            I said that a is positive.\n" )
}
```

- Be careful! What happens here?

```c
a = -5;
if( a > 0 )
    printf( "a is positive.\n" );
    printf( "In case you didn’t hear me,
            I said that a is positive.\n" )
```
Relational and Equality operators

• In actuality, expressions like "5 > 3" are evaluated to integer values: 1 for true, 0 for false. Thus the program

```c
void main()
{
    printf( "Result of 1 > 2: %d\n", 1 > 2 );
    printf( "Result of 6 < 8: %d\n", 6 < 8 );
}
```

gives as output:

Result of 1 > 2: 0
Result of 6 < 8: 1

• Relational operators: >, >=, <, <=

• Equality operators: ==, !=

   - IMPORTANT! == (two equals) versus = (one equal) is an extremely common source of programmer errors in C. One equal, =, is an assignment operator.
More on our friend if

- if executes the statement (or statement block) after it when the specified condition is non-zero.

- Thus, the following fragment prints: Hi!

```c
if( 18 )
    printf( "Hi!\n" );
if( 0 )
    printf( "Bye.\n" );
```

- What does the following fragment do?

```c
int a;
printf( "Enter a number:" );
scanf( "%d", &a );
if( a = 3 )
    printf( "You typed 3.\n" );
```

- Notice that there is no semicolon after the condition of an if statement.
**Logical Operators**

- Enter the three logical operators: `&&`, `||`, `!`

- `&&`, `||` (logical AND, logical OR) are binary operators: two arguments.

- `expression1 && expression2` evaluates to 1 ("true") if both expressions are non-zero, otherwise evaluates to 0 ("false").

- `expression1 || expression2` evaluates to 1 ("true") if either or both expressions are non-zero, otherwise evaluates to 0 ("false").

- `!expression` evaluates to 1 ("true") if the expression is zero, otherwise evaluates to 0 ("false").

- Example.

```c
if(( 3 >= 5 ) || !(2 > 4))
{
    printf( "The OR is true.\n" );
}
if(( 3 >= 5 ) && !(2 > 4))
{
    printf( "The AND is true.\n" );
}
```

- "Short-circuit evaluation" used.
  (The `!(2 > 4)` in second `if` not evaluated.)
if-else

- Basic form:

```c
if( condition )
    statement1;
else
    statement2;
```

- As before, each statement can be either a single command (terminated with a semicolon), or a block of commands delimited by curly braces.

- Example.

```c
if(( year % 4 == 0 && year % 100 != 0 ) ||
    ( year % 400 == 0 ))
{
    printf( "%d is a leap year\n", year );
}
elser
{
    printf( "%d is not a leap year\n", year );
}
```
More on if-else

• Is there a difference between

```java
if( condition )
    statement1;
else
    statement2;

and

if( !condition )
    statement2;
else
    statement1;
```

• Common usage for a series of if-elses:

```java
if( expression1 )
    statement1;
else if( expression2 )
    statement2;
else if( expression3 )
    statement3;
...
else
    statement;
```

The temptation is to continually indent.
Under what conditions is statement3 executed?
An example

- Example.

```c
void main()
{
    int num;
    printf( "Please enter a positive integer:\n" );
    scanf( "%d", &num );

    if( num % 3 == 0 )
        printf( "%d is divisible by 3.\n", num );
    else if( num % 2 == 0 )
        printf( "%d is divisible by 2, but not 3.\n", num );
    else
        printf( "%d is not divisible by 3 nor 2.\n", num );
}
```
The "dangling else problem"

• Beware this type of ambiguous situation:

```c
if( a == 3 )
if( a == 5 )
    printf( "a is 5.\n" );
else
    printf( "Doh!\n" );
```

• Instead, use braces:

```c
if( a == 3 )
{
    if( a == 5 )
        printf( "a is 5.\n" );
    else
        printf( "Doh!\n" );
}
```
switch statement

• Similar to a chain of if/else statements, but more restricted in terms of functionality.

• Useful when one wants to branch based on the value of an expression.

• General form:

```java
switch( expression )
{
    case constant1:
        statement1;
        [break;]
    case constant2:
        statement2;
        [break;]
    ...
    default:
        statement;
        [break;]
}
```
The fall-through property

- Use breaks! What happens if the breaks are removed?

```c
switch( num )
{
    case 1:
        printf( "Behind Door 1 is nothing.\n" );
        break;
    case 2:
        printf( "Behind Door 2 is a goat.\n" );
        break;
    case 3:
        printf( "Behind Door 3 is a pot of gold.\n" );
        break;
}
```

- Sometimes we can exploit the fall-through property:

```c
switch( month )
{
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
        printf( "31 days.\n" );
        break;
    case 2:
        printf( "28 or 29 days.\n" );
        break;
    default:
        printf( "30 days.\n" );
}
```
while statement

- Nice and simple:
  
  ```
  while( condition )
  statement;
  ```

- A `break` statement inside the statement block causes the loop to be stopped.

- A variant:
  
  ```
  do
    statement;
  while( expression );
  ```

- The statement is always executed at least once. Equivalent to:
  
  ```
  statement;
  while( expression )
    statement;
  ```
while example

• Keeping a running sum.

    void main()
    {
        int sum = 0, number = 0;
        while( number != -1 )
        {
            sum += number;
            printf( "The running sum is: %d
", sum );
            printf( "Enter a pos. integer (-1 quits):
" );
            scanf( "%d", &number );
        }
    }

• Another way to do it.

    void main()
    {
        int sum = 0, number;
        while( 1 )
        {
            printf( "The running sum is: %d
", sum );
            printf( "Enter a pos. integer (-1 quits):
" );
            scanf( "%d", &number );
            if( number == -1 ) break;
            sum += number;
        }
    }

Note: while( 1 ) is conventional for "infinite" loops
for statement

- General form:

```c
for( initial-stmt; condition; iteration-stmt )
    body-stmt;
```

- Equivalent to:

```c
initial-stmt;
while( condition )
{
    body-stmt;
    iteration-stmt;
}
```

- `break` can also be used, within the `body-stmt`.

- `break` in general applies to innermost loop (while, do/while, for) or switch statement.

- `continue` statement (not frequently used) causes the next iteration to be executed - jumps to condition-test of innermost loop (while, do/while) or next increment statement (for).
for example

- Summing the first ten positive even numbers (2, 4, 6, ..., 20).

```c
void main()
{
    int i, sum = 0;
    for( i = 1; i <= 10; i++ )
        sum += 2 * i;
    printf( "The sum is %d\n", sum );
}
```

- Another way to do it.

```c
void main()
{
    int i, sum = 0;
    for( i = 2; i <= 20; i += 2 )
        sum += i;
    printf( "The sum is %d\n", sum );
}
```

- Notice: no semicolon after the condition of the for.